Engineering Formal Security Policies for Proof-Carrying Code

Andrew Bernard

April 13, 2004

CMU-CS-04-124

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:
Peter Lee, Chair
Karl Crary
Frank Pfenning
Fred B. Schneider, Cornell University

This work has been partially supported by the National Science Foundation under grant CCR-0121633.

The author wishes to acknowledge support through the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298.

Keywords: proof-carrying code, temporal logic, formal verification, proof engineering, security policies

Abstract

Thesis statement: It is practical to engineer a system for proof-carrying code (PCC) in which policy is separated from mechanism. In particular, I exhibit a generic implementation of the PCC infrastructure that accepts a wide variety of security properties encoded in a formal specification language. I approach the problem by addressing two distinct subproblems: *enforcement* (checking programs and proofs) and *certification* (constructing programs and proofs).

for Lisa

Acknowledgements

This dissertation would not have been possible without the assistance, guidance, and support of many different people.

I would like to thank my advisor, Peter Lee, for his unwavering confidence in my abilities and for his thoughtful attention to the progress of my graduate education. Peter's guidance and insights have been an invaluable asset to my research, and I sincerely hope that this work reflects his usual high standard for quality.

I would like to thank my thesis committee for their careful attention to the content of my dissertation, and to my research work as a whole. Any individual characterization of their contributions must necessarily be incomplete, but I will attempt to describe them nonetheless. Frank Pfenning pushed me to delve more deeply into mathematical logic and introduced me to the profound effect that seemingly minor changes can have on the properties of a deductive system. Fred B. Schneider challenged many of my unfounded assumptions about how security policies can be expressed, employed, and understood—the work is much stronger as a result. Karl Crary uncovered several subtle ambiguities in the original formal system and enabled me to develop the more careful account that is presented here.

Bob Harper had a deep influence on my thinking early on in my graduate studies. John C. Reynolds had many helpful suggestions on the formalization of procedure specifications in the derived program logic. And without Peter B. Andrews' excellent class on mathematical logic, I would not have been able to develop the formal semantics for my logical framework.

Michael Donohue and Stephen Magill contributed proofs of many of the derived inference rules that make up the prelude implementation.

I would like to thank Brigitte Pientka, Kevin Watkins, and all the members of the ConCert reading group for many lively discussions of various topics in programming language research. I also had many productive discussions with Josh Berdine and Aaron Stump during the later stages of my dissertation research. My officemates Rob O'Callahan and Yang-hua Chu helped me to refine the initial ideas that led to my dissertation research.

I would like to thank my fiancé Lisa Price for her love and support, for her tireless proof reading, and for her many sacrifices during the later stages of my graduate study. Finally, I would like to thank my parents, Nancy and Rudy Bernard, for their love and support throughout my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We use an increasing number of programs in our daily lives. And the number of sources from which we obtain these programs is also increasing. But when we install a new program on our computer, we should ask ourselves, "is this program safe for me to run?" *i.e.*, "will it preserve the integrity of my operating system and my data?" Technologies for *code safety* enable us to answer "yes" to this question before we run a new program on our computer.

There are several technologies for code safety that are currently in widespread use:

- A *byte-code interpreter* interprets a program written in a language that is inherently safe—the program essentially has no opportunity to execute an unsafe operation. But because the byte-code language is interpreted, any given program takes longer to execute than does an equivalent compiled program. Not only does this make the user wait longer than he or she should have to for the program to run, but this technology can also result in an unacceptable drain on the battery of a small device such as a cell phone or a personal digital assistant (PDA).

- A *just-in-time compiler* overcomes some of the performance limitations associated with the byte-code interpreter by compiling the byte-code language to machine code just before it is run. Thus, the program is able to run almost as fast as an equivalent compiled program, but the just-in-time compiler is itself a large and complex piece of software. Because of this complexity, it is more likely that a serious vulnerability will be introduced into the compiler implementation, and thus compromise the safety of the host computer. Additionally, such vulnerabilities can be very expensive to fix if the just-in-time compiler is widely deployed.

- A *digitally-signed program* is an ordinary machine-code program that contains a digital signature that attests to the safety of the code. Unfortunately, the digital signature does not in and of itself guarantee that the code is safe—it only guarantees that the person who signed the code *believes* that the code is safe. The digital signature infrastructure also introduces a new vulnerability:

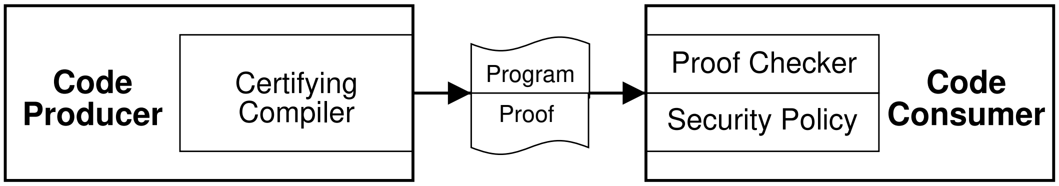

Figure 1.1: PCC Vision

if the private key of the code signer is ever compromised, then anyone who trusts the corresponding public key is potentially subject to having malicious code loaded onto their system.[1]

1.1 Proof-Carrying Code

I believe that *proof-carrying code* (PCC) is the right architecture for code safety because it addresses the aforementioned limitations directly:

- PCC enables *good performance* because source code is compiled to machine code in advance with all compiler optimizations enabled. In addition, some compilers can remove run-time checks that are needed by alternative technologies because the compiler can construct a proof that the run-time check is unnecessary.
- PCC has a *small trusted-computing base* (TCB) because the checking software is relatively easy to implement and the size of the checking program is small. Additionally, some PCC implementations are simple enough to verify for correctness according to a rigorous mathematical standard.
- Finally, PCC introduces *no trusted third parties* into the program checking infrastructure. Each host need trust only its own implementation of a relatively simple proof checker.

I outline the original vision for the PCC architecture in Figure 1.1. Many readers will already be familiar with PCC according to a similar mental model. In this figure, a *code producer* sends a program and its attached safety proof to a *code consumer* who eventually wants to run the code. The code producer uses a *certifying compiler* to construct the machine-code program and its safety proof from an corresponding source-code program. The code consumer uses a *proof checker* to check that the proof corresponds to the program and that the proof is a proof of a particular *security policy*. The security policy is chosen by the code consumer, and it defines precisely what it means for a given program to be "safe to run."

[1] This is not just a hypothetical scenario. In 2002, a serious buffer-overflow vulnerability was discovered in a widely-distributed ActiveX control signed by Microsoft. Because the control is signed by a trusted source, any web page can simply embed the flawed control to reintroduce the vulnerability into the Internet Explorer web browser. The company's response did not instill confidence: "Without a hint of irony, the company recommends removing 'Microsoft' from IE's Trusted Publisher list ..." [Com02]

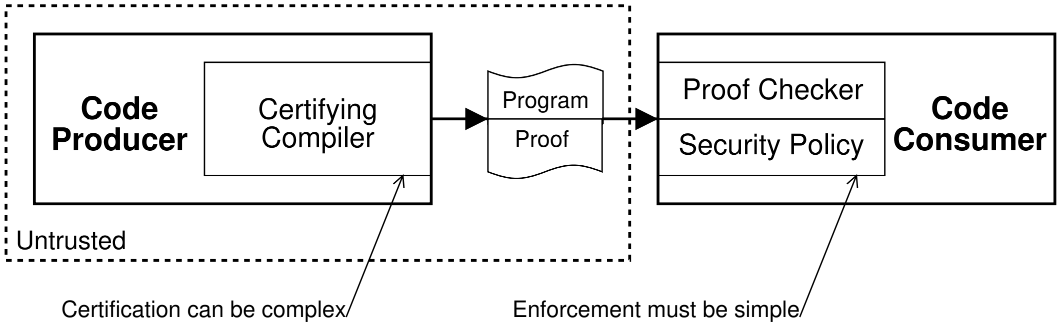

Figure 1.2: PCC Principles

I illustrate the principles behind the PCC architecture in Figure 1.2. In a PCC system, every action the code producer takes is *untrusted*. In particular, any claim the code producer makes about the program (*i.e.*, that it is safe) is checked explicitly by the code consumer according to a rigorous set of inference rules. In order to make the system trustworthy, "enforcement" is designed to be as simple as possible, even if this entails making "certification" more complex. By *enforcement*, I mean every action that the code consumer takes to check the program and the proof. By *certification*, I mean every action that the code producer takes to construct the program and the proof. It is a salient and essential feature of PCC that certification is not part of the TCB.

1.2 SpecialJ

A particularly successful implementation of the PCC architecture revolves around the *SpecialJ Certifying Compiler* [CLN$^+$00], the successor of the *Touchstone Certifying Compiler*[Nec98] which was a cornerstone of the original PCC research. SpecialJ was developed by Cedilla Systems as a production-quality certifying compiler for the Java Programming Language [GJS96]. SpecialJ translates Java source code into certified, optimized IA-32 [Int01] machine code—unlike other implementations of the Java language, SpecialJ does not rely on a byte-code interpreter. SpecialJ enforces a security policy that is based on Java type safety. Essentially, the typing invariants that already exist in type-correct source code are translated to the compiled machine code.[2] SpecialJ is successful in the sense that it can compile large programs efficiently. For example, it can compile the entire Java Development Kit, version 1.3, as well as the HotJava web browser. The HotJava web browser alone consists of over 150,000 lines of source code.

Unfortunately, the actual implementation of the SpecialJ infrastructure is not quite as simple as Figure 1.1 would suggest. As I illustrate in Figure 1.3, when the

[2]Note that although SpecialJ incidentally relies on an automatic theorem prover to synthesize typing derivations, the theorem prover does not "discover" any new properties of the code. The function of the theorem prover is simply to recover any information that is not cost effective to propagate explicitly through the compiler.

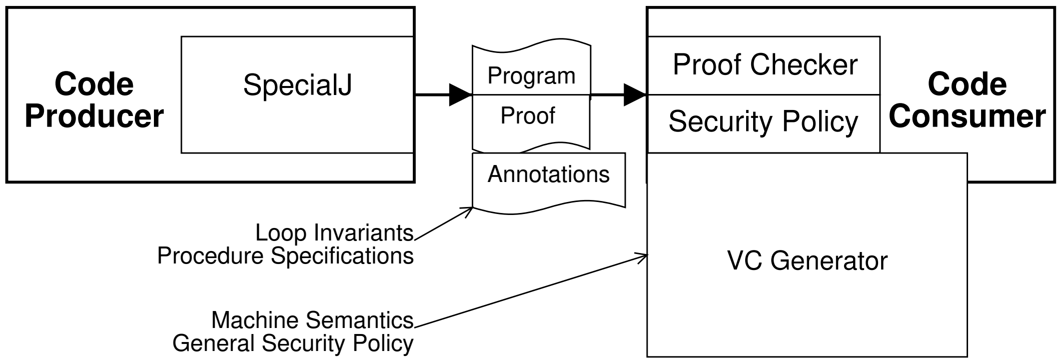

Figure 1.3: PCC Reality

code producer uses SpecialJ as a certifying compiler, the program must be accompanied by an additional set of *code annotations* that consist of loop invariants and procedure specifications. The code annotations are needed because the code consumer will use a software component called a *VC generator* to analyze the program before the proof is checked. The annotations enable the VC generator to interpret loops and procedure calls in the code efficiently. The VC generator embodies the processor semantics of the host computer as well as key aspects of the general security policy that the code consumer will enforce.

The *verification condition* (VC) is a formal proposition that is true only if the program is safe to execute. The VC is essentially an "intermediate" proposition generated by the code consumer that entails safety. The logical justification for the statement "VC entails safety" is normally established by informally verifying the implementation of the VC generator—it is essentially the correctness of the VC-generator code that determines the soundness of this argument. Thus, the safety of the code consumer rests upon a relatively fragile base if the VC generator is only poorly understood or incompletely verified. The VC itself is derived by an algorithm known as symbolic evaluation [Nec97] that essentially simulates the operation of the program on an abstract representation of the machine state.

The key feature of the VC-generator approach that makes it attractive for PCC is that a typical proof of a VC is relatively scalable, in the sense that the size of the proof is close to the size of the program itself.[3] However, the VC-generator approach also suffers from a number of limitations:

- The VC generator is a relatively *complex* piece of software. For example, in the SpecialJ implementation, it consists of over 16,000 lines of particularly dense C code.

- The VC generator is *machine specific* in the sense that it must be extended or re-implemented for each new processor architecture.

[3]The proof size can also be significantly smaller, depending on the representation [NR01], although the code annotations will still occupy a significant amount of space.

- The VC generator is *compiler specific* in the sense that it depends upon the idiosyncrasies of the data-structure formats and calling conventions of the SpecialJ compiler.

- The VC generator is *source-language specific* in the sense that the control flow of the program is restricted to correspond to high-level control structures such as method invocations, case statements, and exception handlers.

- Finally, The VC generator is *security-policy specific* in the sense that key aspects of the code consumer's security policy are built into the implementation.

Several new approaches to PCC, such as *Foundational PCC* [App01] are designed to overcome the aforementioned limitations of the VC-generator approach. In my research, I am particularly interested in overcoming the last limitation that ties the security policy to the VC-generator implementation. I shall show, however, that my approach to PCC also overcomes some of the other limitations as an additional benefit.

How exactly is a PCC security policy represented? In SpecialJ, it is a combination of C code (part of the VC generator), along with a set of typing rules (encoded in the LF Logical Framework [HHP93]). Typing rules are relatively trustworthy, because they are specified formally and can be proven sound individually with respect to a formal semantics. C code, by contrast, is much more obscure and error prone. Essentially, if one wants to verify that a particular security policy is enforced by the VC generator, then the entire implementation must be verified for correctness. Additionally, any change to the code will require that the *entire* implementation be re-verified to preserve its correct status. This lack of modularity makes the VC generator an inherently brittle component that is either untrustworthy or inflexible, according to the diligence of the implementor.

1.3 Towards a Security-Policy Language

Furthermore, in the SpecialJ implementation,

- if the code consumer wants to enforce an *expressive security policy* beyond Java type safety (such as resource bounds or information flow),

- or if the code consumer wants to *manage* a set of security policies,

- or if the code consumer wants to *manipulate* security policies,

then the VC-generator implementation must be updated at each new turn. And, of course, once the implementation is updated, any effort put into verification is potentially lost.

A better enforcement mechanism for PCC would be based on a *universal* program checker for a broad spectrum of security policies. Such an enforcement mechanism would let the code consumer distinguish *policy* from *mechanism* in the PCC infrastructure. What I would like to see, then, is a formal *language* of security policies that can be enforced by a universal program checker. Once such a formal language is enforceable, it might even be feasible to encode the security policy itself in a certificate, and thus provide the code consumer with additional flexibility in terms of which particular security policy is enforced.

In my research, I have focused on using *temporal logic* [MP91, CGP99] as the basis for a formal security-policy language. I believe that temporal logic is an attractive notation for security-policies for the following reasons:

- Temporal logic enables *direct* security-policy specifications, especially when compared to a security policy that is embedded in the code of an enforcement-mechanism implementation. For example, the following specification enforces a simple form of control-flow safety in which the program counter is restricted to a limited range of addresses:

$$\Box(\mathtt{pc} \geq 0 \wedge \mathtt{pc} < 1000)$$

- Temporal logic has a *well-understood semantics* that has been developed through decades of rigorous study.
- Temporal logic can express a *wide variety of security properties*—probably more, in fact, than any foreseeable PCC system would try to enforce.
- Finally, temporal logic enables the reuse of *type-safety specifications* from existing PCC systems such as SpecialJ, provided that a sufficiently expressive variant of temporal logic is selected.

Therefore, I alter the obligation of the code producer to show *directly* that a particular temporal-logic security property holds for any possible execution of the untrusted program. This proof obligation subsumes the existing SpecialJ proof obligation that requires a proof of an intermediate VC. In this approach, the code consumer need no longer incorporate a complex and inflexible VC generator into the TCB. This approach provides a trustworthiness benefit in addition to a flexibility benefit: a complex piece of code is removed from the TCB, and the implementation of the enforcement mechanism is no longer machine specific, compiler specific, source-language specific, or security-policy specific.

In the remainder of this dissertation, I examine whether or not the above approach can be adopted as a *practical* solution to the code safety problem.

1.4 Dissertation Synopsis

In Chapter 2, I review background material that will serve as a foundation for the remainder of the dissertation. The body of this dissertation is structured as in Figure 1.4.

In Chapter 3, I define the variant of temporal logic that I will use as a framework in later chapters. In Chapter 4, I formalize an abstract subset of the Intel IA-32 instruction set architecture that I will use as a target machine model. The primary

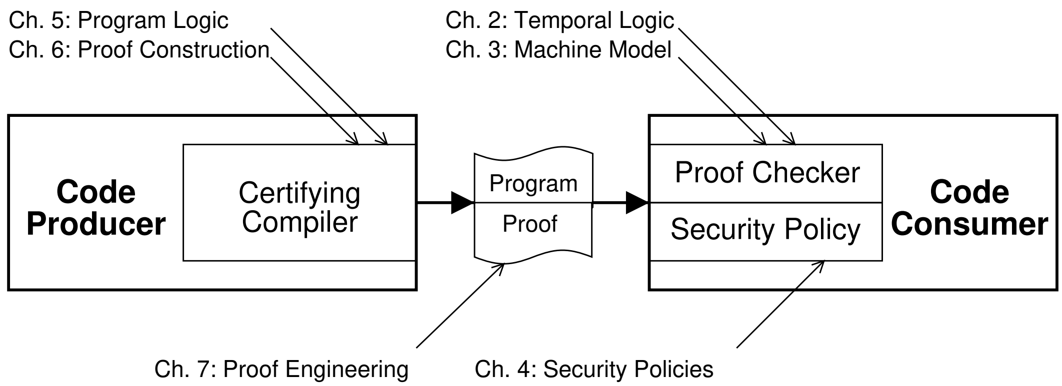

Figure 1.4: Chapter Synopsis

purpose of these two chapters is to provide a sound basis for proof checking. In Chapter 5, I define the precise memory safety policy that is enforced by the code consumer, and is likewise certified by the code producer.

In Chapter 6, I introduce a derived logic of programs that forms the basis for my approach to certification and proof reconstruction. In Chapter 7, I show how proofs are automatically constructed from an annotated object-code program and certificate produced by the SpecialJ compiler. In Chapter 8, I survey how proofs are encoded, and how complete safety proofs are reconstructed from minimal outlines at proof-checking time.

In Chapter 9, I present the results of my experiments with a prototype implementation. Finally, in Chapter 10, I review the contributions of this research, survey further related work, and suggest future improvements.

Note that Appendix A contains a glossary of the formal notation I use in this dissertation—I suggest that the reader refer to this appendix when an unfamiliar symbol is encountered, although I do take care to define the role of each symbol as it is introduced. Appendix B consists of the complete trusted type signature that I use for proof checking, and Appendix C contains the source code for my benchmark programs.

Chapter 2

Background

This chapter contains background material that serves as a foundation for subsequent chapters. In Section 2.1, I review how security policies are treated in existing approaches to certified code, whereas in Section 2.2, I present an overview of my approach to proof-carrying code.

2.1 Security Policies and Certified Code

In this section, I examine security policies and how they relate to certified code, a generalization of the concept of proof-carrying code. First, in Section 2.1.1, I present a general framework for classifying security policies. In Section 2.1.2, I define the concept of certified code, and in Section 2.1.3, I show how security policies are treated by current approaches to certified code. Finally, in Section 2.1.4 I point out the limitations of these approaches with regard to security policies, and in Section 2.1.5, I suggest goals for a more systematic treatment of security policies in certified code.

2.1.1 Security Policies

Let an *execution* be a state sequence of some system: for example, the trace of a single program run. Following Alpern and Schneider [Sch99, AS85, AS86], a *security policy* is a predicate on sets of executions. Each program has a set of possible executions. A program *satisfies* a security policy if the security policy holds for its execution set. A program *violates* a security policy if it does not satisfy the security policy. Security policies allow one to characterize prescribed and proscribed behavior for a given program.

A *security property*, on the other hand, is a predicate on executions—all security properties can be regarded as security policies. A program satisfies a security property if the property holds for each of its possible executions. Equivalently, a program satisfies a security property if its execution set is a subset of the executions for which the property holds.

Some security properties are safety properties, others are liveness properties, and some are neither. Informally, a *safety property* asserts that a specific “bad thing” does not occur in an execution: if a safety property holds for an execution, then it holds for each prefix of the execution. A *liveness property* asserts that a specific "good thing" will occur: if a liveness property does not hold for an execution, then the execution is a prefix of an execution for which the property holds. For example, "each lock can be acquired only once" is a safety property, while "all acquired locks must be released" is a liveness property. Alpern and Schneider [AS85, Sch87] showed that all security properties are the conjunction of a safety property and a liveness property.

Henceforth, I will discuss only security *properties* in this dissertation. In particular, the variant of temporal logic I use is only capable of expressing security properties. All safety properties are expressible in principle, as are many interesting liveness properties.[1] Note that in some cases I still use the term "security policy" when the distinction between a policy and a property is unimportant. Because all security properties are also security policies, these are correct uses of the term.

2.1.2 Certified Code

A system for *certified code* ensures that an untrusted, certified program will not violate a particular security policy. I use the term "certified code" to encompass proof-carrying code (PCC) [Nec97], as well as typed assembly language (TAL) [MWCG98], and some other code-safety technologies.

A certified program is packaged with a *certificate* that attests that the program satisfies the security policy. The host checks the program and certificate without external trust relationships—in particular, the host does not trust the provider of the program. The certificate is encoded in a formal language and it contains enough information to make it possible to efficiently check the program against the security policy. The checker is normally conservative in the sense that it will only accept certified programs, even if an uncertified program would not violate the security policy during an actual run.

An *enforcement mechanism* prevents a program from violating a security policy. Enforcement can be accomplished by certification or by other means, such as run-time monitoring or program instrumentation. For certified code, the enforcement mechanism checks the program and its certificate. An enforcement mechanism is *sound* with respect to a security policy if it rejects all programs that violate the security policy. An enforcement mechanism is *complete* with respect to a security policy if it accepts all programs that satisfy the security policy.

Note that certified code does not presume a trust relationship between code producer and code consumer, or with any third party—contrast this scheme with cryptographic code certification (*e.g.*, signed applets [GMPS97]), in which the code consumer is presumed to trust the code producer and only doubts the *authenticity* of the program. Certified code is compatible with cryptographic certification (*e.g.*, the signature might attest that the program is correct as well as harmless), but certified code protects the code consumer from defective programs in addition to

[1] Because there are an uncountable number of liveness properties, no recursively enumerable language can hope to distinguish them all.

malicious ones. Cryptographic certification requires the code consumer to trust the competence of the code producer, in addition to his or her intentions.

Refer to Kozen [Koz99] for further elaboration on the principles behind certified code.

2.1.3 Approaches to Security-Policy Specification

In this section, I review current approaches to security-policy specification for various enforcement mechanisms.

2.1.3.1 Proof-Carrying Code

Proof-carrying code (PCC) [Nec97, Nec98, NL96, NL98c, NL98a] is form of certified code in which the code producer packages the program with a formal proof that demonstrates that the program satisfies a specific security policy. The code producer constructs the proof and encodes it in a formal logic. The enforcement mechanism checks that the proof is valid, and that it matches the program and the security policy. PCC deliberately places a heavier burden on the code producer than on the code consumer (proof checking is usually easier than proof discovery). PCC enables the code consumer to enforce security policies that are not decidable—the code producer must combine programs and proof generators such that valid proofs are possible. The code producer can even construct proofs manually, but this approach is feasible only for small programs.

Conventional PCC implementations invoke a verification-condition (VC) generator [Kin71] to derive a proposition from the program and a built-in security policy. The code producer is obliged to show that the VC is valid under a set of axioms and inference rules. The code consumer need not trust the code producer, because the enforcement mechanism independently validates the proof.

Necula [Nec98] defines a security policy according to the valid transitions of a *safe interpreter* for an abstract assembly language. The VC generator is based on the operational semantics of the safe interpreter, and he proves its soundness in his dissertation. This approach anticipates limited parameterization, because the preconditions of dangerous instructions are uninterpreted predicates on states. The corresponding predicate symbols are axiomatized to encode security policies for instruction safety and memory safety. Necula developed sample safety properties for proscribing memory access, for sand-boxing memory, and for abstract types.

In foundational proof-carrying code (F-PCC) [AF00], the safety property is built into the machine model. The semantics of the abstract machine is not specified for unsafe transitions, and the code producer is obliged to show that the program can always make some transition.

2.1.3.2 Typed Assembly Language

In *typed assembly language* (TAL) [MWCG98, MCG^{+}99, CM99], the enforcement mechanism is a type checker that does not accept programs that violate a security policy; type annotations accompany program instructions. A TAL compiler translates a well-typed source program into a well-typed assembly program. TAL has a potential efficiency advantage over PCC because safety proofs are not present—however, a TAL type checker must reconstruct type derivations from type annotations. Because PCC transmits complete type derivations, it is practical even when reconstruction is prohibitively expensive.

The original TAL [MWCG98] defines a security policy implicitly by the type system of an assembly language. Walker [Wal00] developed a TAL type system based on an arbitrary security automaton. Alpern and Schneider [AS86, Sch99] invented security automata to encode safety properties as formal automata. The approach Walker takes to TAL is novel because it separates the security policy from the enforcement mechanism, and because security automata are not tailored to the type checker.

Crary and Weirich [CW00] developed a TAL type system that enforces resource bounds. The compiler for this type system is automatic, but it must be given a resource-bound annotation for each function. Crary, Walker, and Morrisett [CWM99] developed a TAL type system to enforce security policies based on a capability [DvH66] calculus. This calculus can ensure the safety of explicit deallocation. This enables an enforcement mechanism without a trusted garbage collector—the inclusion of a garbage collector in the TCB is a drawback of many current enforcement mechanisms.

2.1.3.3 Safe Interpreters

For a *safe interpreter*, the execution language of the program ensures that no security-policy violations can occur. On the one hand, the enforcement mechanism can check the program dynamically, in which case each operation is tested during execution. On the other hand, it can check the program statically, in which case all tests occur before execution. It is common to see a hybrid of both techniques, because static enforcement is typically both more efficient and less automatic than dynamic enforcement.

The Java virtual machine (JVM) [LY99] is a well-known enforcement mechanism for Java class files, which I consider to be a form of certified code. Java class files contain instructions in the Java byte code language; the *byte-code verifier* checks Java byte code statically. The JVM specification [LY99] documents the type-safety policy of the byte-code verifier, in addition to other run-time security checks—this specification thus fixes the security policy for Java byte code. Determining precisely what security policy the JVM enforces is a challenge, because of its informal prose specification. Original implementations of the JVM interpreted byte codes directly, but modern implementations translate byte code into machine code.

The Java Language Specification [GJS96] documents the Java Security Manager, a trusted system that enforces access control. Permissions determine the operations that a process can perform; the JVM manages permissions transparently. Although this approach is convenient in some respects, it prevents optimizing compilers from inlining method calls because of stack introspection [WBDF97].

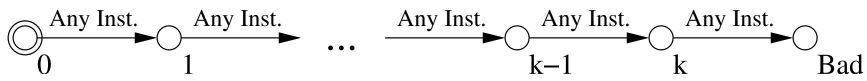

Figure 2.1: Instruction-Bound Security Automaton

For the Java Development Kit (JDK) 1.2 Security Model [GMPS97], the security policy is partially specified through configuration files. A *policy file* specifies which permissions an program receives based on predefined attributes (*e.g.*, its origin or digital signature). Other researchers (*e.g.*, PoET [ES00], J-Kernel [HCC+97], Naccio [ET99]) have developed extensions for more expressive security policies.

2.1.3.4 Software Fault Isolation

Software fault isolation (SFI) [WLAG93, ALLW96] instruments the program so that it cannot violate a built-in security policy. I do not consider SFI certified code, because it does not rely on a certificate. SFI enforces a memory safety policy that the instrumentation tool implicitly defines. SFI is fully automatic because it can instrument any program, regardless of code producer. Unfortunately, SFI relies on run-time checks that entail run-time overhead and preclude fine-grain confidentiality policies [ML97].

Security automata SFI implementation (SASI) is an SFI-based tool developed by Erlingsson and Schneider [ES99, ES00] for enforcing safety properties encoded in a security-automata language. Like security automata for TAL, I consider SASI an important contribution, because it disentangles the security policy from the enforcement mechanism.

A security automaton is a state machine that responds to the actions of a target system. The security automaton enters a "bad" state when the target violates its safety property. In fact, concrete safety properties can be interpreted as security automata. Security automata are an attractive representation because they are enforceable with different mechanisms, and because they encompass all safety properties [Sch99]. For example, to represent the instruction-bound security policy, one constructs a sequence of automaton states of the same length as the bound k (see Figure 2.1). The automaton transitions to a successor state after executing an instruction; the successor of the kth state is the bad state. Note that more concise representations of resource bounds are possible when using security-automata notation (see Schneider [Sch99] for examples).

2.1.4 Limitations of Current Approaches

Unfortunately, enforcement mechanisms often determine security policies, rather than vice-versa. Such security policies (*e.g.*, SFI [WLAG93]) are difficult to document independently. Witness attempts to formalize the Java byte code verifier [SA99, FM98, O'C99]. In the absence of precise definitions, it is impossible to establish rigorously that a security policy prohibits malicious behavior.

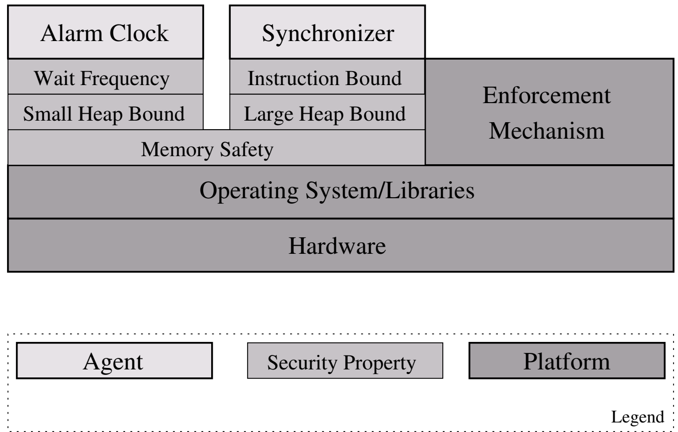

Figure 2.2: Secure PDA

PCC and TAL are based on formal models, but it is often impossible to change a security policy without modifying the enforcement mechanism. For example, Necula [Nec98] parameterizes his PCC by predicates for a memory safety policy, but a resource bound or confidentiality policy requires a change to the code of the implementation.

Because of this rigidity, any work put into verifying an enforcement mechanism is lost when the security policy is changed, because changes may introduce bugs that are not in the formal model. Additionally, enforcing a set of security policies requires a corresponding set of distinct implementations, each of which must be verified. Adding enforcement mechanisms increases the size of the TCB and makes the entire system harder to trust.

To motivate the problems addressed by this research, consider how one might design a PCC-based personal digital assistant (PDA). The PDA can be enhanced by new programs, with the proviso that each such program is checked by PCC before it is installed, thereby ensuring that the PDA (a code consumer) will not cease to work because of faulty or malicious software. Untrusted extensions are provided by a code producer. I will focus on two such extensions for the moment:

- The *alarm clock* runs continuously, but only for brief intervals. It updates the display once per second and emits a special sound, when appropriate.

- The *synchronizer* runs only when the user "docks" the PDA. The synchronizer ensures that the PDA is consistent with a desktop computer.

Figure 2.2 contains a diagram of this design. A trusted *enforcement mechanism* checks each program against several distinct security policies before it is allowed to run. A *memory-safety* policy protects the operating system and libraries from corruption. Additional *resource-bound* policies place limits on the system resources that programs can consume. The memory-safety policy is common to all programs, but the resource-bound policies are tailored to individual programs.

The alarm clock needs little memory to run, but runs continuously for an unlimited period of time; it is usually waiting in between clock ticks. The alarm clock is thus assigned the *wait-frequency* policy that limits it to a small number of instructions before invoking the `wait` system call. The *small-heap-bound* policy constrains the alarm clock to only a small amount of dynamic memory. The *instruction-bound* policy requires the synchronizer to terminate after executing a number of instructions proportional to the size of the PDA address book. The *large-heap-bound* policy constrains the synchronizer to a large amount of dynamic memory (also proportional to the address-book size). Because the synchronizer will terminate in a limited time frame, its dynamic memory will be released promptly.

A typical implementation of this design would require a separate enforcement mechanism for each distinct security policy. This approach has drawbacks, however, because all enforcement mechanisms are in the TCB. Unfortunately, it is relatively difficult to tailor an enforcement mechanism to a new security policy (in general), especially if one expects to change the policy over time or if one wants to vary it for different programs. On the one hand, one can try to incorporate all the security policies at once into a single mechanism, but this leaves a complex mass of code that is difficult to reuse in new situations. On the other hand, one can implement a separate mechanism for each security policy, but this is potentially inefficient because each mechanism must examine the program and its proof. In addition, implementations are relatively difficult to understand and change, and also tend to depart from a specification over time. The system designer is thus interested in developing *security policies* as opposed to *enforcement mechanisms*. Ideally, the security policy is a parameter of the enforcement mechanism.

A better approach would instead treat security policies as parameters of a single universal enforcement mechanism. One could then develop policy and mechanism independently, and reuse a single implementation for an unlimited number of applications. I first attempted to address this problem by extending a standard enforcement mechanism with a security-policy interpreter [BL01]—unfortunately, this approach entails considerable complexity. In this dissertation, I present an alternative approach that uses a simpler enforcement mechanism, but at the expense of longer proof-checking times.

2.1.5 Goals for a Security-Policy Language

To address the problems uncovered in Section 2.1.4, I have developed a *security-policy language* based on temporal logic. A security-policy language encodes security policies in a concrete notation—each "program" of this language denotes a set of executions. For any given concrete security policy, one wants to do the following:

- Understand it in terms of an abstract model
- Reason about it with respect to the language semantics
- Enforce it with a generic implementation
- Enable (or disable) it based on a particular program

- Modify it independently of an implementation

The security-policy language thus has the following goals:

Abstraction The language should be defined mathematically so that it specifies precisely what actions a security policy permits. A developer should not need detailed knowledge of an enforcement mechanism.

Abstraction is a benefit because the language semantics can be defined without reference to an enforcement mechanism. Any enforcement mechanism that implements the language semantics will enforce the correct execution set, so a developer need check only that the security policy has the desired characteristics. Most of the work put into verifying an enforcement mechanism can thus be leveraged by many security policies. Thus, the language semantics becomes the common interface between reasoning about security policies and reasoning about the enforcement mechanism.

In this dissertation, I show the soundness of a particular enforcement mechanism. To ensure that this mechanism enforces a given security policy, one need show only that the security policy has the desired characteristics. For example, for an instruction-bound security policy, one shows that all executions permitted by the security policy do not exceed the intended length.

Expressiveness The security-policy language should encompass specialized security policies (*i.e.*, beyond programming language type safety). For example,

Resource bounds A resource bound limits consumption of system resources (*e.g.*, processor, memory). It is important for critical resources (*e.g.*, mutexes, input–output devices) to be released promptly once acquired. Other applications (*e.g.*, networking) must limit the rate at which a resource is used.

Protection An operating-system protection mechanism prevents unauthorized access to resources [SG98, Lam71, WCC$^+$74]. The JDK 1.2 Security Model [GMPS97] specifies an access control mechanism. It should be possible to encode the JDK 1.2 Security Model in the security-policy language.

Confidentiality A *confidentiality policy* [DD77] is a conservative approximation of an information flow policy that partitions the program into distinct security classes. Some approaches to confidentiality [ML97] classify communication channels according to the highest security class to which a message may belong. An enforcement mechanism prevents higher-security information from flowing to lower-security areas.

Integrity An *integrity policy* helps to guarantee that information is authentic and complete. There has been little work to date on using proof-carrying code to enforce integrity policies, but it is not difficult to imagine a security policy that requires the untrusted program to only supply answers that are based on the results of some trusted set of cryptographic primitives, for example.

Modularity A modular security-policy language supports independently developed security policies, and enforces them without interference. Specifically, if several security properties are enforced in combination, then the permitted executions should be the intersection of the independent execution sets. Thus, the soundness of an individual security policy should not depend on whether it is enforced in combination with other security policies. Schneider [Sch99] identifies similar goals for security automata.

See Chapter 5 for further discussion of how temporal logic can be employed as a security-policy language.

2.2 Proof Engineering and Temporal-Logic PCC

Since its emergence [NL96], a considerable amount of research has been devoted to the engineering aspects of PCC. Much of this research has focused on representation issues, with the three goals of making certificates that (1) are very small, (2) allow the code to be verified quickly, and (3) require only a small and simple program to do the verification. One early approach involved the use of a variant of the LF Logical Framework (LF) [HHP93] to represent certificates as formal proofs [NL98b]. This approach gave encouraging results, by showing that certificate sizes and verification times could, in typical practical situations, scale linearly with code size. Furthermore, the use of LF allowed a key component of the verifier to be generic (simply a type checker for LF), parameterized by an LF signature that formally specifies the inference system for the proof certificates. Later, Necula and Rahul developed the concept of "oracle"-based checking [NR01], which made further reductions in certificate sizes without unduly increasing either the verification time or the complexity of the verification process. Indeed, using oracle-based checking, Colby and others were able to implement a system called SpecialJ that compiled a large suite of production Java programs into certified Intel x86 object programs [CLN$^+$00].

While these early results clearly showed that certified code, and in particular PCC, could be practical in terms of certificate size and verification time, it became clear that the process of verification was more complicated than it needed to be and thus less trustworthy than it ought to be. To this end, three of the most fundamental problems have been addressed by later research:

- Appel and Felty observed that the logical system used by a PCC system made use of constants and axioms based only on an informal understanding of the semantics of the type system and the machine instructions. In making this observation, they proposed the concept of foundational proof-carrying code (F-PCC) [AF00] and have embarked on the construction of a PCC system in which the entire safety policy, all the way down to the semantics of the machine instructions, is formally defined and available for use in certificates and certificate checking.

- Hamid, Shao, *et al.*proposed an approach to F-PCC in which the syntactic meta theory of the machine instructions and type system is proved and represented in LF. Crary later proposed a similar approach for TAL, called TALT [Cra03].

- Finally, Peter Lee and I recognized that, in practice, temporal logic could be used as the basis for a practical, formal specification language for PCC. This led us to propose the concept of temporal-logic proof-carrying code (TL-PCC) [BL02a], which allows safety policies to be specified directly, while also achieving the benefit of eliminating the VC generator from the TCB.

Besides increasing our understanding of the nature of certified code, F-PCC improves the engineering of proof-carrying code by eliminating the need for the VC generator. TL-PCC extends this benefit by allowing safety policies to be specified concisely.

A key practical problem, however, is the impact of F-PCC and TL-PCC on the size of the certificates. Indeed, F-PCC and TL-PCC proofs contain more information than their conventional PCC counterparts, and thus a naïve encoding leads to certificates that are so large as to be completely impractical. Even using a compact binary encoding, my previous experience indicates that certificates are likely to be many times larger than the corresponding programs [BL02a]. While I do not have data on certificate sizes for F-PCC and TALT, I believe they are likely to be similar to what I have observed for TL-PCC.

Fortunately, the situation may not be as dire as it appears. In this dissertation, I present an approach to PCC engineering in which certificates are represented as logic programs that are derived from a sound program logic. Intuitively, these certificates, when executed by a small trusted interpreter, perform the verification-condition generation and proof search needed for safety verification. A key property of the interpretation process is that it can itself be trusted, even if the certificate itself is not trusted. Using this approach, I have achieved, for a suite of relatively simple test cases, certificate sizes on par with those that have been observed for previous conventional PCC systems. While I present this new approach and its results with the specific purpose of reducing certificate sizes for TL-PCC, the approach is general and can, in principle, be adapted for use in F-PCC or TALT.

The cost of this approach comes in the amount of time required for the verification. My current experimental results show that, while certificate sizes are relatively small, verification requires significantly more time, sometimes as much as three orders of magnitude more. While it is not yet known how much room for improvement there is in verification time, it is clear that the new approach is essentially "proving more" than conventional approaches, so in reducing proof sizes, space must have been effectively traded for time.

In Section 2.2.1, I present my approach to PCC. Next, in Section 2.2.2, I discuss the proof engineering problem for my approach to PCC in greater detail, and in Section 2.2.3, I show how I address this problem.

2.2.1 Temporal-Logic PCC

Until now, practical PCC implementations have encoded security proofs in first-order logic, and the enforcement mechanism included a trusted VC generator that essentially encoded the security policy in a C implementation (*e.g.*, Necula [Nec97]). I will argue here that *temporal logic* [MP91, Eme90, CGP99] has certain advantages over first-order logic for PCC. Using temporal logic, one can reclassify the VC generator as an *untrusted* component and thereby allow the security policy to be separated from the enforcement mechanism. This also provides the crucial advantage of reducing the amount of software in the TCB, though as I shall show, this advantage appears to come at the cost of larger proofs. In this respect, my approach resembles *foundational PCC* [App01, AF00], although, unlike foundational PCC, my code producer and consumer must agree on a shared notion of type safety.

A temporal logic is characterized by its temporal operators. Such operators enable one to distinguish the different times at which a proposition is true. In this dissertation, I will identify time with the CPU clock and regard propositions as statements about machine states. For example, the proposition

$$\mathtt{pc} = 0 \supset \bigcirc(\mathtt{pc} = 1)$$

asserts that "if the program counter is 0 now, then it will be 1 in the next state." One can also specify security policies in temporal logic. For example, the proposition

$$\Box(\mathtt{pc} \geq 0 \wedge \mathtt{pc} < 100)$$

asserts that "the program counter is always between zero and 100," but one can also interpret this as the requirement "the program counter must always be between zero and 100"—a specification for a simple form of control-flow safety [Koz98]. I will exploit this duality to reap a practical benefit.

For a PCC system based on first-order logic, the enforcement mechanism generates a proposition from the program and the security policy together—the security proof is a proof of this proposition. For temporal-logic PCC, the enforcement mechanism recognizes the program as a formal term, and the operational semantics of the host machine is encoded as a set of trusted inference rules or as a trusted temporal logic formula. One can then encode the security policy directly—the security proof shows that the security policy is a consequence of running the program from a set of initial conditions. Notice that the security policy is independent of the enforcement mechanism, but no additional mechanism is required to interpret it.

One also wants to be confident that the security policy is correct. This confidence is difficult to obtain for a security policy written in C code. In contrast, temporal logic has a clear semantics, and security policies are comparatively compact.

Temporal logic can express a wide variety of security policies [MP90], including type-safety, resource-bound, and liveness policies. For example,

$$(\mathtt{n} = 0 \wedge \Box(\bigcirc(\mathtt{n}) = \mathtt{n} + 1)) \supset \Box(\mathtt{n} \geq 1000 \supset \Phi_{\mathtt{pc}} = \mathtt{halt})$$

is an encoding of an instruction bound. Read this proposition as "for any n such that n is initially zero and increases by one at each cycle,[2] the program must halt by the time `n` reaches 1000."

As I shall show, a simple enforcement mechanism for temporal-logic PCC can be implemented at the cost of increasing proof-checking times. This can be a favorable trade-off, because it is shifting work from a trusted component to an untrusted one.

2.2.2 Proof Engineering

TL-PCC offers several engineering advantages over conventional PCC:

- The suite of trusted software is easier to verify informally because there is no VC generator. For example, if one wants to use the SpecialJ compiler [CLN+00], one no longer needs to trust over 15,000 lines of C code.
- Programs can be checked against a wide variety of interesting security policies (*e.g.*, type safety, resource bounds, access control) without adding any new trusted code.
- A broader selection of code generation strategies can be supported because there is no built-in program analysis or any need for program annotations.

As I discuss earlier in this section, however, TL-PCC safety proofs must "prove more" than their conventional counterparts. To overcome this difficulty and still obtain reasonably compact certificates, I outline an approach to proof engineering in which an untrusted logic program essentially extracts and proves its own VCs when executed.

A proof engineering technique that has been shown to be extremely useful in practice is based on equipping a logic interpreter with an "oracle string" [NR01]. In this approach, the trusted collection of logical inference rules (the *signature*) is treated as a nondeterministic logic program. If one starts this program with the VC proof obligations as an initial goal, any trace of a successful run is a proof of safety. The oracle string enables the interpreter to make "don't-know" nondeterministic choices correctly every time.

While oracle-based interpretation has proven to be very efficient in practice for conventional PCC, I am not optimistic about the prospect of applying it to TL-PCC without significant adjustments. Based on my experience, a TL-PCC safety proof (in normal form) contains at least an order of magnitude more inferences than a conventional PCC safety proof. To make proof generation even feasible, one must factor common sequences of inferences into derived rules that are treated as primitives by the proof checker.

A second difficulty arises due to the number of rules whose conclusions match any goal in a TL-PCC signature, but are only actually used to justify derived rules. Because these former rules are always possible candidates, the oracle string must

[2] Here I use $\bigcirc()$ as an abbreviation for a more complex expression (see Section 4.4.7 for examples of incrementing parameters).

spend five or six bits just to discount them for each inference. A rough calculation predicts that a proof for a typical program will be many times larger than the program itself. One can address this second problem by considering *only* derived rules during search, and if their conclusions are carefully tailored, one can minimize the number of choices for each inference.

Thus, one is led to conclude that the *entire* logic program should be included in the untrusted proof for TL-PCC. Essentially, this reflects a more foundational viewpoint by not trusting the logic program itself.

But I can take my proposal even further by constraining the logic program to the point where it becomes deterministic. By adopting an idea originally suggested by Pfenning [Pfe01], one can embed the oracle string in the search goal and interpret it explicitly within a logic program. Thus, one can dispense with the external oracle entirely and simply use a deterministic interpreter. A deterministic interpreter yields an additional benefit by not charging for "don't-care" nondeterminism, among others.

At this point the astute reader will object to running an untrusted logic program, because there is no assurance that it is sound, or that it will terminate. The soundness concern is easily addressed, however, because each clause of the logic program must have an associated derivation. Once these derivations are checked, one is assured that any successful run of the logic program will have a derivation in the original signature. The termination concern is less easily dispensed with, but many *ad-hoc* solutions exist for bounding the time spent interpretation one can decrement a counter as the logic program is executed, or, in the case of an interactive download, one can bound the interpretation time by the patience of the user by providing an explicit "reject" button. To a certain extent, the correct solution to the nontermination problem depends on the particular application. In effect, the program obligation is changed from "there must be a proof of safety" to "a proof of safety must be constructed in time t" (the certificate might even supply t). Finally, I should point out that in any PCC system based on LF [HHP93], type checking is already vulnerable to a denial-of-service attack based on effective nontermination, simply because testing LF terms for equivalence is not elementary recursive [Sta77]—in practical terms, nontermination is not a new vulnerability.

2.2.3 Proof Reconstruction

In order to show that a program is safe in TL-PCC, one must prove that a particular proposition (a *security policy*) holds based on ordinary logical inference rules as well as another set of inference rules that encode the machine semantics. The latter rules typically enable one to infer a value of the machine state at the next time from a value of the machine state at the current time.

For this dissertation, I am primarily interested in security policies in the following form:

$$p_{\mathsf{pre}} \supset p_{\mathtt{safe}}\,\mathcal{U}\,p_{\mathsf{post}}$$

p_{pre} is a *precondition* that holds when the program starts, $p_{\mathtt{safe}}$ is an *invariance property* that must hold at each time step, and p_{post} is a *postcondition* that must hold for the program to exit successfully. The above proposition can be read as "precondition implies safe unless postcondition." The crux of the problem is to show that $p_{\mathtt{safe}}$ holds at any given time.

Note that because one does not know the precise initial state that the program will be started from, one cannot simply simulate a complete execution from the machine semantics and then show that $p_{\mathtt{safe}}$ holds at each time. Even if the precise initial state was known, such a proof would be unreasonably large or infinite for any program with loops. Thus, a more sophisticated approach is needed.

2.2.3.1 Symbolic Evaluation

In previous work by Necula and Lee [Nec98], VCs are generated by a technique known as *symbolic evaluation*. While a symbolic evaluator produces the same results as a VC generator, the symbolic evaluator is structurally more similar to the operational semantics of the machine. In effect, the operational semantics is applied to expressions that are an abstraction of a machine state. The machine state is abstract because it is only partially instantiated: it represents many possible run-time states. In practice, one can use an automatic theorem prover to discover a proof that $p_{\mathtt{safe}}$ holds for a given abstract state when $p_{\mathtt{safe}}$ is restricted to relatively simple safety properties.

In previous work [BL02a], I developed a proof generation strategy for PCC that is based on deriving inference rules that mimic the action of a symbolic evaluator. A typical such rule shows that the invariance property holds from the current time whenever a single potentially unsafe instruction is executed. An infinite tower of derived rules is avoided by deferring to a *loop invariant* rule once inside each program loop.

In this dissertation, I generalize this approach by constructing a *derived logic of programs* that is used by the code *producer* as a foundation for proofs of invariance properties. A *logic of programs* [Flo67, Hoa69] consists of a set of formal inference rules that can be used to prove properties of programs in a given programming language. A *derived* logic of programs is a logic of programs in which the formal inference rules are derived from standard logical rules. Because the logic of programs is derived formally, it is an *untrusted* component of the PCC infrastructure, and thus adds no complexity to the TCB. In my PCC system, the derived logic of programs provides the essential formal foundation for the logic program that implements proof reconstruction.

2.2.3.2 Proof Outlines

A logical framework provides a representation of proofs and terms that is independent of any particular logic. LF [HHP93] is one such framework that has been particularly successful for PCC applications. It is not feasible to simply send a proof of safety encoded directly in LF, because embedded type information grows non-linearly with the size of the proof [NL98b]. Significant effort has been devoted to automatically reconstructing type information for LF terms [Nec98]. Unfortunately, my experience indicates that even with a powerful type reconstruction algorithm [PS99], explicit proofs are still many times the size of the program [BL02a].

As I discussed in Section 2.2.2, the safety proof will instead be reconstructed by interpreting derived rules as a logic program—these derived rules will be based on the derived logic of programs. Although the interpreter is actually based on a search over the constant declarations of a LF signature, in the interest of brevity, I will usually pretend that it is a logic interpreter that performs a bottom-up search over temporal-logic inference rules.

One cannot simply search over the same derived rules as would be used in an explicit proof. For example, the conclusion of a the most common transition rule also matches six other rules in my signature. Most reasonable search strategies will require exponential time to execute this program, if they terminate at all. The derived rules will instead be constrained so that they become a deterministic algorithm, given an appropriate initial goal.

A proof checker can support such a strategy with only minimal extensions. As I observed in Section 2.2.2, the search space for derived rules must be isolated from the search space for trusted rules. This is accomplished by introducing, for any p, a new goal

$$\langle\!\langle p \rangle\!\rangle$$

with the following introduction and elimination rules:

$$\frac{p}{\langle\!\langle p \rangle\!\rangle}\ \langle\!\langle\rangle\!\rangle\mathsf{i} \qquad \frac{\langle\!\langle p \rangle\!\rangle}{p}\ \langle\!\langle\rangle\!\rangle\mathsf{e}$$

Obviously, the logic interpreter must ignore $\langle\!\langle\rangle\!\rangle\mathsf{i}$ during proof search—its sole purpose is to define derived rules.

The only thing left to add is a special marker for omitted derivations ($\star$): this will trigger a proof search when they are encountered by the proof checker. It is the responsibility of the code producer to ensure that only decidable derivations are omitted. Thus, the code consumer expects to see an explicit safety proof with some derivations omitted. In practice, the code producer will omit almost the entire proof, but this approach gives the code producer the flexibility to choose any agreeable balance between explicit and omitted proofs.

Now, the strategy of the code producer is to provide a derivation of safety that is constructed as follows:

$$\cfrac{\cfrac{\star}{\langle\!\langle p_{\mathsf{po}} \wedge (p_{\mathsf{pre}} \supset p_{\mathtt{safe}}\,\mathcal{U}\,p_{\mathsf{post}}) \rangle\!\rangle}}{p_{\mathsf{pre}} \supset p_{\mathtt{safe}}\,\mathcal{U}\,p_{\mathsf{post}}}$$

p_{po} is an encoding of a *proof outline* as a proposition. The proof outline functions as an artificial constraint on proof search. The proof outline is structured so as to make only one derived rule applicable for any given goal, though it also contains instantiations for existential variables at certain key junctions. Given the above derivation, proof search starts off with a constrained judgment that will result in a deterministic strategy.

Any explicit proof is in direct correspondence with some proof outline, so it is not difficult to extract a proof outline once the code producer finds an explicit proof. However, because the proof outline is much smaller than the explicit proof, the code producer will send it in place of the proof to the code consumer.

For my purposes, a proof outline can be represented as a binary tree with some token from a countable set (and occasionally a proposition) at each node. Although this representation of a proof outline is not particularly conservative with respect to memory usage, by using a simple customized binary encoding it can be compressed to the point where it consumes much less space in a certificate.

2.3 Dissertation Scope

Many of the techniques presented in this dissertation *cannot* be applied to arbitrary temporal-logic security properties. In particular, the techniques for automatic proof construction are specialized to invariance properties, and Java type safety in particular. In the following table, I illustrate how the scope of the dissertation evolves in terms of what class of security properties are considered:

	Security Properties	Notation
Chapter 3: Temporal Logic	Reactivity	p
Chapter 4: Machine Model	Reactivity	p
Section 5.2: Enforcement	Reactivity	p
Section 5.3: Soundness of Enf.	Reactivity	p
Section 5.4: Security Automata	Safety	$\Box p_{\texttt{safe}}$
Section 5.5: Memory Access	Memory Safety	$p_{\mathsf{pre}} \supset \texttt{safe}\,\mathcal{U}\, p_{\mathsf{post}}$
Section 5.6: Memory Safety	Memory Safety	$p_{\mathsf{pre}} \supset \texttt{safe}\,\mathcal{U}\, p_{\mathsf{post}}$
Section 5.7: Java Types	Java Type Safety	$p_{\mathsf{pre}} \supset \texttt{safe}\,\mathcal{U}\, p_{\mathsf{post}}$
Section 5.8: Java Type Safety	Java Type Safety	$p_{\mathsf{pre}} \supset \texttt{safe}\,\mathcal{U}\, p_{\mathsf{post}}$
Chapter 6: Program Logic	Invariance	$p_{\mathsf{pre}} \supset p_{\texttt{safe}}\,\mathcal{U}\, p_{\mathsf{post}}$
Chapter 7: Proof Construction	Java Type Safety	$p_{\mathsf{pre}} \supset \texttt{safe}\,\mathcal{U}\, p_{\mathsf{post}}$
Chapter 8: Proof Engineering	Java Type Safety	$p_{\mathsf{pre}} \supset \texttt{safe}\,\mathcal{U}\, p_{\mathsf{post}}$

The *reactivity properties* [MP90] are precisely those security properties that can be encoded in temporal-logic and include all safety properties as well as the most familiar liveness properties. In this table, p_{pre}, $p_{\texttt{safe}}$, and p_{post} are arbitrary state specifications that do not contain temporal operators. `safe` is a particular memory safety specification that is defined in Section 5.6.

Chapter 3

Temporal Logic

I choose to use temporal logic [MP91, CGP99] as a notation because it provides a concise encoding for many kinds of security policies. Most of the techniques I present later, however, do not depend in an essential way on the choice of notation and can also be applied to systems based on first-order and higher-order logics.

I use a discrete linear-time first-order temporal logic that resembles classical temporal logic [MP91]. However, instead of developing an axiomatization of this logic, I follow Davies [Dav96] and Simpson [Sim94] and construct a natural-deduction system based on explicit times [BPW02]. The extension of Davies' system to additional temporal operators is straightforward. The extension to first-order quantifiers requires more care to accommodate both rigid and flexible variables.

I use a natural-deduction system to enable compatibility with other PCC systems, and because the orthogonal treatment of connectives facilitates incremental extensions and restrictions. A natural deduction system establishes a clear relationship between the set of connectives and the set of inference rules, because the inference rules are intended to define the meaning of the connectives. In a classical axiomatic system, this relationship is less clear and a separate proof is needed to show that axioms are independent. It is also possible to treat connectives orthogonally in a classical sequent system [Gen69], but it is difficult to represent proof terms compactly in such systems.

I will present things at the level of temporal logic for most of my dissertation. However, keep in mind that the logic is itself encoded as a formal system of the LF logical framework [HHP93]. I reproduce the complete LF representation of my logic as Appendix B.1.

This chapter is structured as follows: in Section 3.1, I present the syntax of the logic, along with the informal meaning of each construct. Section 3.2 is devoted to establishing a precise model-theoretic semantics for the logic that shares the mathematical domain of the operational semantics of the abstract machine. In Section 3.3, I introduce the inference rules of the logic, developed according to the principles of natural deduction [Pfe99]. In Section 3.4, I present generic theories for algebra, equality, pairs, and lists. Finally, in Section 3.5, I prove that the natural deduction system is sound with respect to the model-theoretic semantics.

Times	t	$::= b \mid 0 \mid t_1 + 1$
Rigidities	ρ	$::= \mathsf{ri} \mid \mathsf{fl}$
Parameter lists	α	$::= \cdot \mid \alpha_1, a$
Expressions	e^τ	$::= a^\tau \mid x^\tau \mid f^{\tau_1 \times \cdots \times \tau_k \to \tau}(e_1^{\tau_1}, \ldots, e_k^{\tau_k})$
Propositions	p	$::= R^{\tau_1 \times \cdots \times \tau_k \to o}(e_1^{\tau_1}, \ldots, e_k^{\tau_k})$ $\mid p_1 \wedge p_2 \mid p_1 \vee p_2 \mid p_1 \supset p_2 \mid \forall x^\tau : \rho.\, p_1 \mid \exists x^\tau : \rho.\, p_1$ $\mid \bigcirc p_1 \mid \Box p_1 \mid p_1 \, \mathcal{U}^\Diamond \, p_2$
Judgments	J	$::= t_1 \leq t_2 \mid p : \mathsf{lo}\ (a) \mid e : \rho\ (\alpha) \mid p : \rho\ (\alpha)$ $\mid e \Longrightarrow^0 e' \mid e \Longrightarrow e' \mid e \Longrightarrow^{**} e' \mid e \Longrightarrow^* e'$ $\mid p \Longrightarrow^0 p' \mid p \Longrightarrow p' \mid p \Longrightarrow^{**} p' \mid p \Longrightarrow^* p'$ $\mid p \,@\, t \mid p \,@[t_1, t_2) \mid p \,\underline{@}\, t$
Contexts	Γ	$::= \cdot \mid \Gamma, J$

Figure 3.1: Abstract Syntax

3.1 Syntax

The syntax of my logic (see Figure 3.1) is based on disjoint countably infinite sets of parameters and variables. A *parameter* a is always free in a proposition, whereas a *variable* x is normally bound.[1] This is a many-sorted logic, so each parameter and variable is annotated with an explicit type τ (as a superscript), of which there are countably many. Types have no internal structure. I almost always omit type annotations when they can be inferred or are immaterial.

Primitive functions and relations are named by a countable set of *constants* (f and R, respectively). Constants are also annotated with types: $\tau_1 \times \cdots \times \tau_k \to \tau$ is the annotation of a function from k parameters to a value of type τ, whereas $\tau_1 \times \cdots \times \tau_k \to o$ is the annotation of a relation on k parameters. When k is zero, I use the meta variable c^τ instead of the more general meta variable $f^{\cdot \to \tau}$. Thus, constant values c^τ are nullary functions, whereas constant propositions (*i.e.*, $\top$, $\bot$) are nullary relations. There is a binary equality relation for each type. Each constant relation also has an associated complement relation of the same type: the complement of R is denoted by $\neg R$. $\neg =$ is written $\neq$, and $\neg\top$ is written $\bot$. This is a first-order logic, so functions and relations appear only as constants.

Expressions e^τ are constructed from parameters, variables, and applications of constant functions; τ is the type of e. The simple type system for my logic is essentially built into the syntax.

Following Manna and Pnueli [MP91], some expressions are *rigid*. It is syntactically evident that a rigid expression has the same value at all times. A *flexible* expression may (but need not) have different values at different times. For exam-

[1] The syntactic distinction between parameters and variables simplifies the notation for inference rules.

ple, a constant such as 5 is rigid, whereas the contents of a stack pointer register is flexible. Variables also have rigidity—rigidities must agree when a bound variable is instantiated. The rigidity ρ of a variable is declared when the variable is bound: ri denotes a rigid variable, whereas fl denotes a flexible variable. A rigid expression contains only rigid variables and parameters.

Propositions p encompass the usual connectives and quantifiers of first-order logic, plus the following temporal operators:

- $\bigcirc p$ holds iff p holds at the next future time.
- $\Box p$ holds iff p holds at all future times.
- $\Diamond p$ holds iff p holds at some future time.
- $p_1 \,\mathcal{U}^{\Diamond}\, p_2$ holds iff p_2 holds at some future time, and p_1 holds until then.
- $p_1 \,\mathcal{U}\, p_2$ holds iff p_1 holds until the first future time at which p_2 holds, but p_2 need never hold.

The equivalence connective $\equiv$ is an abbreviation for mutual implication:[2]

$$p_1 \equiv p_1 \stackrel{\mathsf{def}}{\equiv} (p_1 \supset p_2) \wedge (p_2 \supset p_1)$$

Two temporal operators are also defined by abbreviation:

$$\begin{aligned} \Diamond p &\stackrel{\mathsf{def}}{\equiv} \top \,\mathcal{U}^{\Diamond}\, p \\ p_1 \,\mathcal{U}\, p_2 &\stackrel{\mathsf{def}}{\equiv} \Box p_1 \vee (p_1 \,\mathcal{U}^{\Diamond}\, p_2) \end{aligned}$$

I use the following customary precedence conventions (higher to lower) when writing concrete propositions:

$$\begin{array}{l} R(e_1, \ldots, e_k) \quad \bigcirc p \quad \Box p \quad \Diamond p \\ p_1 \,\mathcal{U}^{\Diamond}\, p_2 \quad p_1 \,\mathcal{U}\, p_2 \\ p_1 \wedge p_2 \\ p_1 \vee p_2 \\ p_1 \supset p_2 \\ p_1 \equiv p_2 \\ \forall x\!:\!\rho.\, p \quad \exists x\!:\!\rho.\, p \end{array}$$

As is usual, the scope of a universal or existential quantifier extends as far to the right as possible.

A proposition is *local* in a parameter if it is sensitive only to the current-time value of the parameter (*e.g.*, the value of the parameter can be changed at any other time without affecting the truth of the proposition). A rigid proposition has only rigid parameters (bound variables may be flexible). Propositions differing only in the names of their bound variables are identified, as is customary.

[2] $p_1 \stackrel{\mathsf{def}}{\equiv} p_2$ means that proposition p_1 is a syntactic abbreviation for proposition p_2. $e_1 \stackrel{\mathsf{def}}{=} e_2$ means that expression e_1 is a syntactic abbreviation for expression e_2.

$t_1 \leq t_2$	t_1 denotes the same time as t_2 or an earlier time than t_2.
$p : \mathsf{lo}\,(a)$	p is local in a.
$e : \rho\,(\alpha)$	e denotes a sequence with rigidity ρ. Each parameter in α is assumed rigid.
$p : \rho\,(\alpha)$	p is a proposition with rigidity ρ. Each parameter in α is assumed rigid.
$e \Longrightarrow^0 e'$	e reduces to normal form e' (in one step) when all subterms of e are normal.
$e \Longrightarrow e'$	e reduces to e' (in one step) when all subterms of e are normal.
$e \Longrightarrow^{**} e'$	e reduces to normal form e' when all subterms of e are normal.
$e \Longrightarrow^{*} e'$	e reduces to normal form e'.
$p \Longrightarrow^0 p'$	p reduces to normal form p' (in one step) when all subterms of p are normal.
$p \Longrightarrow p'$	p reduces to p' (in one step) when all subterms of p are normal.
$p \Longrightarrow^{**} p'$	p reduces to normal form p' when all subterms of p are normal.
$p \Longrightarrow^{*} p'$	p reduces to normal form p'.
$p \,@\, t$	p is true at time t.
$p \,@[t_1, t_2)$	p is true at all times in the interval $[t_1, t_2)$ ("p holds *over* t_1 to t_2").
$p \,\underline{@}\, t$	p is true at time t according to a restricted formal system.

Table 3.1: Judgments

A *time expression* t denotes a specific instant in time. Time is counted in unary notation: 0 denotes the earliest possible time (*e.g.*, the start of execution), and $t+1$ denotes the time immediately following time t. A parameter b is a time parameter, designating an arbitrary time.

Following Martin-Löf [ML85], a *judgment* J (see Figure 3.1) is an object of knowledge (*i.e.*, a statement which can be established with evidence). An informal interpretation for each judgment is suggested in Table 3.1. Note that "over" is a derived concept that exist primarily to simplify the encoding of inference rules. In Section 2.2.3.2, I explained the need for a restricted truth judgment for which the code producer can develop a decidable formal system. $\underline{@}$ is the notation for this judgment in the formal system of this chapter and the remainder of this dissertation.

Term rewriting [BN98] plays an important role in proof reconstruction by allowing the code producer to elide the derivations of many decidable equivalence problems. I treat term rewriting as an explicit formal system instead of building it into my logical framework to allow the code *producer* to customize the rewriting strategy. For the purposes of this section, however, term rewriting can be viewed simply as an alternative notion of equality between expressions or equivalence between propositions.

3.1.1 Substitution

$[e_1^\tau / x^\tau]\, e$ is the usual *substitution* of the expression e_1 for free occurrences of the variable x in expression e. For substitution to be well formed, e_1 must have the

$$
\begin{array}{ll}
[e/x]\,a & = a \\
[e/x]\,x & = e \\
[e/x]\,x_1 & = x_1 \text{ if } x \neq x_1 \\
[e/x]\,f(e_1,\ldots,e_k) & = f([e/x]\,e_1,\ldots,[e/x]\,e_k) \\[4pt]
[e/x]\,R(e_1,\ldots,e_k) & = R([e/x]\,e_1,\ldots,[e/x]\,e_k) \\
[e/x]\,(p_1 \wedge p_2) & = [e/x]\,p_1 \wedge [e/x]\,p_2 \\
[e/x]\,(p_1 \vee p_2) & = [e/x]\,p_1 \vee [e/x]\,p_2 \\
[e/x]\,(p_1 \supset p_2) & = [e/x]\,p_1 \supset [e/x]\,p_2 \\
[e/x]\,\forall x.\,p & = \forall x.\,p \\
[e/x]\,\forall x_1.\,p & = \forall x_1.\,[e/x]\,p \text{ if } x \neq x_1 \\
[e/x]\,\exists x.\,p & = \exists x.\,p \\
[e/x]\,\exists x_1.\,p & = \exists x_1.\,[e/x]\,p \text{ if } x \neq x_1 \\
[e/x]\,\bigcirc p & = \bigcirc [e/x]\,p \\
[e/x]\,\Box p & = \Box [e/x]\,p \\
[e/x]\,(p_1\,\mathcal{U}^{\diamond}\,p_2) & = [e/x]\,p_1\,\mathcal{U}^{\diamond}[e/x]\,p_2
\end{array}
$$

Figure 3.2: Substitution

same type as x, and e_1 must be closed (*i.e.*, it must not contain free variables); e need not be closed. $[e^\tau/x^\tau]\,p$ is the usual extension to propositions where e must be closed, but p need not be. See Figure 3.2 for a precise definition of substitution. I write $\mathcal{A}(p)$ for the set of parameters appearing in the proposition p. Thus, $a \notin \mathcal{A}(p)$ asserts that a does not appear in p. Similarly, I write $\mathcal{B}(t)$ for the set of parameters appearing in the time expression t.

Substitution has the following properties:

Proposition 3.1.1 (Absence) $[e_1/x]\,e = e$ *if* x *does not appear free in* e

PROOF: by induction on the structure of e □

Proposition 3.1.2 (Elimination) x *does not appear free in* $[e_1/x]\,e$

PROOF: by induction on the structure of e □

Proposition 3.1.3 (Exchange) $[e_1/x_1]\,[e_2/x_2]\,e = [e_2/x_2]\,[e_1/x_1]\,e$ *if* $x_1 \neq x_2$

PROOF: by induction on the structure of e □

Proposition 3.1.4 (Idempotency) $[e_1/x]\,[e_2/x]\,e = [e_2/x]\,e$

PROOF: by Absence and Elimination □

Proposition 3.1.5 (Idempotency) $[e_1/x]\,[e_2/x]\,p = [e_2/x]\,p$

PROOF: by induction on the structure of p □

Proposition 3.1.6 (Exchange) $[e_1/x_1]\,[e_2/x_2]\,p = [e_2/x_2]\,[e_1/x_1]\,p$ *if* $x_1 \neq x_2$

PROOF: by induction on the structure of p □

3.2 Model-Theoretic Semantics

In this section, I define a formal model for my temporal logic based on environments of infinite sequences of values. Each expression is associated with an infinite sequence of values that represent the values that the expression takes over time. A satisfaction relation determines whether a given judgment holds for a given environment. This model is similar to the usual models of temporal logic [MP91, CGP99] and follows a standard semantic development [And86].

The purpose of the formal model is to relate the logic precisely to a mathematical machine model. The machine model itself is developed in Chapter 4. A rigorous mathematical semantics is needed to given each statement of a security policy a precise, unambiguous meaning.

In this section, I first introduce supporting definitions for the model-theoretic semantics, then I define satisfaction and valuation for each element of the formal syntax. Finally, I derive some properties of the formal model that will be used to support later meta-theoretic proofs.

In Section 3.3, I develop a proof system that is sound with respect to the formal semantics.

3.2.1 Definitions

Each type τ is associated with a set that contains the mathematical values that belong to that type. Val^τ is the set of values v^τ of type τ. A *sequence* π^τ is a mapping from natural numbers (representing times) to values of type τ. An *environment* ϕ maps each parameter to a sequence of the same type. A *time environment* η maps each time parameter to a natural number.

I assume an *interpretation function* $\mathcal{J}$ mapping each constant to a mathematical value of the same type, which may be an ordinary value such as a number (nullary functions), a total function (other functions), or a set of tuples (relations). I assume that $\mathcal{J}$ is defined as follows for the universal constants:

$$\mathcal{J}(\top) = \{\langle\,\rangle\}$$
$$\mathcal{J}(\bot) = \emptyset$$

where $\langle\,\rangle$ is the "unit" or nullary tuple.

Additionally, when $\mathcal{J}$ can be inferred from context, I often write $\underline{f}$ for $\mathcal{J}(f)$ and $\overline{v}$ for the constant c such that $\mathcal{J}(c) = v$, when such a c exists.[3] I also write $\underline{R}(v_1, \ldots, v_k)$ when $\langle v_1, \ldots, v_k\rangle \in \underline{R}$. $\mathcal{J}$ is only partially specified in this chapter—I will incrementally refine the definition as new constants are introduced.

[3]The notation $\underline{c}$ is intended to suggest that the constant c is "pushed downwards," from syntax into semantics, whereas the notation $\overline{v}$ is intended to suggest that the mathematical value v is "lifted upwards," from semantics to syntax.

3.2.2 Valuation

A *valuation function* assigns values to expressions. $\mathcal{V}_\eta(t)$ is the value of time expression t as a natural number in environment η:

$$\begin{aligned}\mathcal{V}_\eta(b) &= \eta(b)\\ \mathcal{V}_\eta(0) &= 0\\ \mathcal{V}_\eta(t+1) &= \mathcal{V}_\eta(t)+1\end{aligned}$$

$\eta(b)$ is always defined because environments are total functions.

$\mathcal{V}^{\mathcal{J}}_\phi$ evaluates expressions of type τ to sequences of values of type τ in the environment ϕ; e must be closed for $\mathcal{V}^{\mathcal{J}}_\phi(e)$ to be defined:

$$\begin{aligned}\mathcal{V}^{\mathcal{J}}_\phi(a) &= \phi(a)\\ \mathcal{V}^{\mathcal{J}}_\phi(f(e_1,\ldots,e_k)) &= j \mapsto \mathcal{J}(f)(\mathcal{V}^{\mathcal{J}}_\phi(e_1)(j),\ldots,\mathcal{V}^{\mathcal{J}}_\phi(e_k)(j))\end{aligned}$$

The notation $j \mapsto v$ constructs an "anonymous" mapping ψ such that $\psi(j) = v$.

The notation $\psi[v_1 \mapsto v_2]$ is the redefinition of the mapping ψ such that v_1 is mapped to v_2:

$$\mathrm{dom}(\psi[v_1 \mapsto v_2]) = \mathrm{dom}\,\psi \cup \{v_1\}$$

$$(\psi[v_1 \mapsto v_2])(v_3) = \begin{cases} v_2 & \text{if } v_1 = v_3\\ \psi(v_3) & \text{otherwise}\end{cases}$$

Let Seq^τ be the set of all sequences of values of type τ. Valuation has the following properties:

Proposition 3.2.1 (Type Preservation) $\mathcal{V}^{\mathcal{J}}_\phi(e^\tau) \in \mathit{Seq}^\tau$ *if e is closed*

PROOF: by induction on the structure of e^τ □

Proposition 3.2.2 (Renaming) $\mathcal{V}_{\eta[b_1 \mapsto j]}(b_1) = \mathcal{V}_{\eta[b_2 \mapsto j]}(b_2)$

PROOF: by definition of $\mathcal{V}_\eta$ □

Proposition 3.2.3 (Renaming) $\mathcal{V}^{\mathcal{J}}_{\phi[a_1 \mapsto \pi]}([a_1/x]\,e) = \mathcal{V}^{\mathcal{J}}_{\phi[a_2 \mapsto \pi]}([a_2/x]\,e)$
if a_1 and a_2 do not appear in e and $[a_1/x]\,e$ is closed

PROOF: by induction on the structure of e □

Proposition 3.2.4 (Independence) $\mathcal{V}_{\eta[b \mapsto j]}(t) = \mathcal{V}_\eta(t)$
if b does not appear in t

PROOF: by induction on the structure of t □

Proposition 3.2.5 (Independence) $\mathcal{V}^{\mathcal{J}}_{\phi[a \mapsto \pi]}(e) = \mathcal{V}^{\mathcal{J}}_\phi(e)$
if a does not appear in e and e is closed

PROOF: by induction on the structure of e □

Proposition 3.2.6 (Past/Future Independence) $\mathcal{V}^{\mathcal{J}}_{\phi[a \mapsto \pi]}(e)(j) = \mathcal{V}^{\mathcal{J}}_{\phi}(e)(j)$ *if* $\pi(j) = \phi(a)(j)$ *and* e *is closed*

PROOF: by induction on the structure of e □

Proposition 3.2.7 (Extraction) $\mathcal{V}_\eta(t) = \mathcal{V}_{\eta[b \mapsto \mathcal{V}_\eta(t)]}(b)$

PROOF: by definition of $\mathcal{V}_\eta$ □

Proposition 3.2.8 (Extraction) $\mathcal{V}^{\mathcal{J}}_{\phi}([e_1/x]\,e) = \mathcal{V}^{\mathcal{J}}_{\phi[a \mapsto \mathcal{V}^{\mathcal{J}}_{\phi}(e_1)]}([a/x]\,e)$ *if* a *does not appear in* e *and* $[e_1/x]\,e$ *is closed*

PROOF: by induction on the structure of e □

3.2.3 Satisfaction

A sequence is *rigid* if it has the same value at all times. The value of a rigid expression is always a rigid sequence, but an expression is not necessarily rigid if its value is a rigid sequence. I write $\pi : \rho$ when π has rigidity ρ:

$$\pi : \rho \quad \text{iff} \quad \rho = \mathsf{ri} \text{ implies } \pi(j_1) = \pi(j_2) \text{ for all } j_1, j_2$$

When viewed semantically, a judgment J is a property of an environment ϕ and a time environment η. The *satisfaction* relation

$$\phi, \eta \models^{\mathcal{J}} J$$

defines when J holds for a particular pair of such environments ϕ and η; J must be closed for satisfaction to be defined.

The time precedence judgment holds when the value of the first time expression is less than or equal to the value of the second:

$$\phi, \eta \models^{\mathcal{J}} t_1 \leq t_2 \quad \text{iff} \quad \mathcal{V}_\eta(t_1) \leq \mathcal{V}_\eta(t_2)$$

The locality judgment holds when the value of the local parameter can be changed at any time other than the current time without affecting the truth of the proposition:[4]

$$\begin{array}{lll} \phi, \eta \models^{\mathcal{J}} p : \mathsf{lo}\,(a) & \text{iff} & \phi, \eta[b \mapsto j] \models^{\mathcal{J}} p \,@\, b \text{ implies } \phi[a \mapsto \pi], \eta[b \mapsto j] \models^{\mathcal{J}} p \,@\, b \\ & & \text{for some } b \text{ and all } \pi, j \text{ such that } \pi(j) = \phi(a)(j) \end{array}$$

[4]Note that locality might alternatively be introduced as a derived concept based on a syntactic encoding such as

$$p : \mathsf{lo}\,(a) \stackrel{\mathsf{def}}{\equiv} \Box(\forall x : \mathsf{fl}.\ x = a \supset p \supset [x/a]\,p)$$

A rigidity judgment holds when a rigid ascription implies that the value of an expression (or the truth of a proposition) is invariant for all times:

$$\phi, \eta \vDash^{\mathcal{J}} e : \rho\,(\alpha) \quad \text{iff} \quad \rho = \mathsf{ri} \text{ implies } \mathcal{V}^{\mathcal{J}}_{\phi}(e)(j_1) = \mathcal{V}^{\mathcal{J}}_{\phi_{(j_1-j_2)..|\alpha}}(e)(j_2) \text{ for all } j_1, j_2$$

$$\begin{array}{lll} \phi, \eta \vDash^{\mathcal{J}} p : \rho\,(\alpha) & \text{iff} & \rho = \mathsf{ri} \\ & \text{implies} & \phi, \eta[b \mapsto j_1] \vDash^{\mathcal{J}} p \mathbin{@} b \\ & & \text{implies } \phi_{(j_1-j_2)..|\alpha}, \eta[b \mapsto j_2] \vDash^{\mathcal{J}} p \mathbin{@} b \\ & & \text{for some } b \text{ and all } j_1, j_2 \end{array}$$

Because the rigidity of a bound variable does not influence the rigidity of the proposition in which it appears,[5] it is necessary to maintain a list of parameters α that have instantiated such variables when the rigidity judgment is defined semantically. At a syntactic level, such parameters are "assumed to be rigid." At a semantic level, rigidity is defined by comparing the value of an expression or proposition at any given time with its value at any other time. The semantic interpretation of the statement "a is assumed to be rigid" is to *shift* the sequence π that a is mapped to such that a appears to have the same value at the current pair of reference times j_1 and j_2.

$\pi_{k..}$ is the notation for the sequence π shifted by k time steps:

$$\pi_{k..} = j \mapsto \begin{cases} \pi(0) & \text{if } j + k < 0 \\ \pi(j+k) & \text{otherwise} \end{cases}$$

$\phi_{k..|\alpha}$ is the environment ϕ such that the sequence for each parameter in α is shifted by k time steps:

$$\phi_{k..|\alpha} = a \mapsto \begin{cases} (\phi(a))_{k..} & \text{if } a \in \alpha \\ \phi(a) & \text{otherwise} \end{cases}$$

Thus, ϕ will agree with $\phi' = \phi_{(j-j')..|\alpha}$ for all parameters in α when ϕ is interpreted at time j and ϕ' is interpreted at time j'. This is the informal justification for the treatment of α in the definition of rigidity—the formal justification is established by the soundness proof.

Note that I often abbreviate $e : \rho\,(\cdot)$ as $e : \rho$ and $p : \rho\,(\cdot)$ as $p : \rho$.

Satisfaction for the remaining judgments is defined in Figure 3.3—this definition is similar to the usual semantic models for temporal logic. Thus, $\phi, \eta \vDash^{\mathcal{J}} p \mathbin{@} t$ ("ϕ and η satisfy p at time t") holds if p is true of ϕ and η at time t. Because term rewriting is inherently syntactic, its semantic manifestation is simple equivalence, and is thus rather uninteresting when viewed in terms of the satisfaction relation. I say that a proposition is *valid* for a given time environment iff it is satisfied by all value environments and that time environment. A proposition is *valid* (in general) iff it is valid for all time environments.

Rigidity, shifting, and satisfaction have the following properties:

Proposition 3.2.9 (Equivalence) $\phi, \eta \vDash^{\mathcal{J}} e : \rho$ *iff* $\mathcal{V}^{\mathcal{J}}_{\phi}(e) : \rho$
if e is closed

[5] For example, $\exists x : \mathsf{fl}.\ x = 5$ is a rigid proposition, even though x is a flexible variable.

$$
\begin{array}{lll}
\phi,\eta \models^{\mathcal{J}} e \Longrightarrow^0 e' & \text{iff} & \mathcal{V}^{\mathcal{J}}_{\phi}(e) = \mathcal{V}^{\mathcal{J}}_{\phi}(e') \\
\phi,\eta \models^{\mathcal{J}} e \Longrightarrow e' & \text{iff} & \mathcal{V}^{\mathcal{J}}_{\phi}(e) = \mathcal{V}^{\mathcal{J}}_{\phi}(e') \\
\phi,\eta \models^{\mathcal{J}} e \Longrightarrow^{**} e' & \text{iff} & \mathcal{V}^{\mathcal{J}}_{\phi}(e) = \mathcal{V}^{\mathcal{J}}_{\phi}(e') \\
\phi,\eta \models^{\mathcal{J}} e \Longrightarrow^{*} e' & \text{iff} & \mathcal{V}^{\mathcal{J}}_{\phi}(e) = \mathcal{V}^{\mathcal{J}}_{\phi}(e') \\
\\
\phi,\eta \models^{\mathcal{J}} p \Longrightarrow^0 p' & \text{iff} & (\phi,\eta[b \mapsto j] \models^{\mathcal{J}} p \,@\, b \text{ iff } \phi,\eta[b \mapsto j] \models^{\mathcal{J}} p' \,@\, b) \\
 & & \quad \text{for some } b \text{ and all } j \\
\phi,\eta \models^{\mathcal{J}} p \Longrightarrow p' & \text{iff} & (\phi,\eta[b \mapsto j] \models^{\mathcal{J}} p \,@\, b \text{ iff } \phi,\eta[b \mapsto j] \models^{\mathcal{J}} p' \,@\, b) \\
 & & \quad \text{for some } b \text{ and all } j \\
\phi,\eta \models^{\mathcal{J}} p \Longrightarrow^{**} p' & \text{iff} & (\phi,\eta[b \mapsto j] \models^{\mathcal{J}} p \,@\, b \text{ iff } \phi,\eta[b \mapsto j] \models^{\mathcal{J}} p' \,@\, b) \\
 & & \quad \text{for some } b \text{ and all } j \\
\phi,\eta \models^{\mathcal{J}} p \Longrightarrow^{*} p' & \text{iff} & (\phi,\eta[b \mapsto j] \models^{\mathcal{J}} p \,@\, b \text{ iff } \phi,\eta[b \mapsto j] \models^{\mathcal{J}} p' \,@\, b) \\
 & & \quad \text{for some } b \text{ and all } j \\
\\
\phi,\eta \models^{\mathcal{J}} R(e_1,\ldots,e_k) \,@\, t & \text{iff} & \langle \mathcal{V}^{\mathcal{J}}_{\phi}(e_1)(\mathcal{V}_{\eta}(t)),\ldots,\mathcal{V}^{\mathcal{J}}_{\phi}(e_k)(\mathcal{V}_{\eta}(t))\rangle \in \mathcal{J}(R) \\
\phi,\eta \models^{\mathcal{J}} (\neg R)(e_1,\ldots,e_k) \,@\, t & \text{iff} & \langle \mathcal{V}^{\mathcal{J}}_{\phi}(e_1)(\mathcal{V}_{\eta}(t)),\ldots,\mathcal{V}^{\mathcal{J}}_{\phi}(e_k)(\mathcal{V}_{\eta}(t))\rangle \notin \mathcal{J}(R) \\
\phi,\eta \models^{\mathcal{J}} p_1 \wedge p_2 \,@\, t & \text{iff} & \phi,\eta \models^{\mathcal{J}} p_1 \,@\, t \text{ and } \phi,\eta \models^{\mathcal{J}} p_2 \,@\, t \\
\phi,\eta \models^{\mathcal{J}} p_1 \vee p_2 \,@\, t & \text{iff} & \phi,\eta \models^{\mathcal{J}} p_1 \,@\, t \text{ or } \phi,\eta \models^{\mathcal{J}} p_2 \,@\, t \\
\phi,\eta \models^{\mathcal{J}} p_1 \supset p_2 \,@\, t & \text{iff} & \phi,\eta \models^{\mathcal{J}} p_1 \,@\, t \text{ implies } \phi,\eta \models^{\mathcal{J}} p_2 \,@\, t \\
\phi,\eta \models^{\mathcal{J}} \forall x^{\tau}\!:\!\rho.\, p \,@\, t & \text{iff} & \phi[a^{\tau} \mapsto \pi^{\tau}],\eta \models^{\mathcal{J}} [a^{\tau}/x^{\tau}]\, p \,@\, t \\
 & & \quad \text{for some } a^{\tau} \notin \mathcal{A}(p) \text{ and all } \pi^{\tau} : \rho \\
\phi,\eta \models^{\mathcal{J}} \exists x^{\tau}\!:\!\rho.\, p \,@\, t & \text{iff} & \phi[a^{\tau} \mapsto \pi^{\tau}],\eta \models^{\mathcal{J}} [a^{\tau}/x^{\tau}]\, p \,@\, t \\
 & & \quad \text{for some } a^{\tau} \notin \mathcal{A}(p) \text{ and some } \pi^{\tau} : \rho \\
\phi,\eta \models^{\mathcal{J}} \bigcirc p_1 \,@\, t & \text{iff} & \phi,\eta[b \mapsto \mathcal{V}_{\eta}(t)+1] \models^{\mathcal{J}} p_1 \,@\, b \text{ for some } b \\
\phi,\eta \models^{\mathcal{J}} \Box p_1 \,@\, t & \text{iff} & \phi,\eta[b \mapsto j] \models^{\mathcal{J}} p_1 \,@\, b \text{ for some } b \text{ and all } j \geq \mathcal{V}_{\eta}(t) \\
\phi,\eta \models^{\mathcal{J}} p_1\, \mathcal{U}^{\Diamond}\, p_2 \,@\, t & \text{iff} & \phi,\eta[b \mapsto j_2] \models^{\mathcal{J}} p_2 \,@\, b \\
 & & \quad \text{for some } b \text{ and some } j_2 \geq \mathcal{V}_{\eta}(t) \\
 & & \quad \text{such that } \phi,\eta[b_1 \mapsto \mathcal{V}_{\eta}(t)][b_2 \mapsto j_2] \models^{\mathcal{J}} p_1 \,@[b_1,b_2) \\
 & & \quad \text{for some } b_1,b_2 \text{ such that } b_1 \neq b_2 \\
\\
\phi,\eta \models^{\mathcal{J}} p \,@[t_1,t_2) & \text{iff} & \phi,\eta[b \mapsto j] \models^{\mathcal{J}} p \,@\, b \text{ for some } b \text{ and all } j \\
 & & \quad \text{such that } \mathcal{V}_{\eta}(t_1) \leq j < \mathcal{V}_{\eta}(t_2) \\
\\
\phi,\eta \models^{\mathcal{J}} p \,\underline{@}\, t & \text{iff} & \phi,\eta \models^{\mathcal{J}} p \,@\, t
\end{array}
$$

Figure 3.3: Satisfaction

PROOF: by the definition of $\models$ $\square$

Proposition 3.2.10 (Rigidity) $\pi_{k..} : \rho$ *if* $\pi : \rho$

PROOF: by the definition of $\pi_{k..}$ $\square$

Proposition 3.2.11 (Cancellation) $(\pi_{k..})_{(-k)..}(j) = \pi(j)$ *if* $j \geq k$

PROOF: by the definition of $\pi_{k..}$ $\square$

Proposition 3.2.12 (Cancellation) $(\phi_{k..|\alpha})_{(-k)..|\alpha}(a)(j) = \phi(a)(j)$ *if* $j \geq k$

PROOF: by Proposition 3.2.11 and the definition of $\phi_{k..|\alpha}$ $\square$

Proposition 3.2.13 (Renaming) $\phi, \eta[b_1 \mapsto j] \models^{\mathcal{J}} p \,@\, b_1$ *iff* $\phi, \eta[b_2 \mapsto j] \models^{\mathcal{J}} p \,@\, b_2$
if p *is closed*

PROOF: by induction on the structure of p $\square$

Proposition 3.2.14 (Renaming) $\phi[a_1 \mapsto \pi], \eta \models^{\mathcal{J}} [a_1/x]\, p \,@\, t$
iff $\phi[a_2 \mapsto \pi], \eta \models^{\mathcal{J}} [a_2/x]\, p \,@\, t$
if a_1 *and* a_2 *do not appear in* p *and* $[a_1/x]\, p$ *is closed*

PROOF: by induction on the structure of p $\square$

Proposition 3.2.15 (Independence) $\phi, \eta[b \mapsto j] \models^{\mathcal{J}} p \,@\, t$ *iff* $\phi, \eta \models^{\mathcal{J}} p \,@\, t$
if b *does not appear in* t *and* p *is closed*

PROOF: by induction on the structure of p $\square$

Proposition 3.2.16 (Independence) $\phi[a \mapsto \pi], \eta \models^{\mathcal{J}} p \,@\, t$ *iff* $\phi, \eta \models^{\mathcal{J}} p \,@\, t$
if a *does not appear in* p *and* p *is closed*

PROOF: by induction on the structure of p $\square$

Proposition 3.2.17 (Past Independence) $\phi[a \mapsto \pi], \eta \models^{\mathcal{J}} p \,@\, t$ *iff* $\phi, \eta \models^{\mathcal{J}} p \,@\, t$
if $\pi(j) = \phi(a)(j)$ *for all* $j \geq \mathcal{V}_\eta(t)$ *and* p *is closed*

PROOF: by induction on the structure of p $\square$

Proposition 3.2.18 (Extraction) $\phi, \eta \models^{\mathcal{J}} p \,@\, t$ *iff* $\phi, \eta[b \mapsto \mathcal{V}_\eta(t)] \models^{\mathcal{J}} p \,@\, b$
if p *is closed*

PROOF: by induction on the structure of p $\square$

Proposition 3.2.19 (Extraction) $\phi, \eta \models^{\mathcal{J}} [e/x]\, p \,@\, t$
iff $\phi[a \mapsto \mathcal{V}_\phi^{\mathcal{J}}(e)], \eta \models^{\mathcal{J}} [a/x]\, p \,@\, t$
if a *does not appear in* p *and* $[e/x]\, p$ *is closed*

PROOF: by induction on the structure of p $\square$

3.3 Proof System

I now develop a proof system for temporal logic that is sound with respect to the model-theoretic semantics.

An *affirmation*

$$\Gamma \vdash^{\mathcal{J}} J$$

asserts that there is a proof of the judgment J under assumptions Γ. Affirmation is predicated on the interpretation function $\mathcal{J}$ because the correct instantiation of certain inference rules depends on the interpretation of particular constants. Note that in most cases, I omit the interpretation function when presenting an inference rule. It should be understood in these cases that the interpretation function is the same for all affirmations in the rule. Note also that locality, rigidity, and (normally) term rewriting affirmations are efficiently decidable.

A *context* Γ is a collection of hypothetical judgments that weaken provability. For example,

$$a : \mathsf{ri} \vdash [a/x]\, p \,@\, t$$

asserts that it is provable that $[a/x]\, p$ holds at time t, assuming that a is rigid. Note that I could clarify the (lack of) dependencies between judgment types by stratifying the context accordingly (*e.g.*, $p \,@\, t$ is a useless assumption for proving $t_1 \leq t_2$), but this is not practical here due to the number of distinct judgment types.

A pair of environments satisfy a context ($\phi, \eta \vDash^{\mathcal{J}} \Gamma$) when they satisfy each judgment in the context (the context must be closed for satisfaction to be defined). Thus, context satisfaction can be defined inductively as follows:

$$\begin{array}{ll} \phi, \eta \vDash^{\mathcal{J}} \cdot & \\ \phi, \eta \vDash^{\mathcal{J}} \Gamma, J & \text{iff} \quad \phi, \eta \vDash^{\mathcal{J}} \Gamma \text{ and } \phi, \eta \vDash^{\mathcal{J}} J \end{array}$$

It has the following properties:

Proposition 3.3.1 (Independence) $\phi, \eta[b \mapsto j] \vDash^{\mathcal{J}} \Gamma$ *iff* $\phi, \eta \vDash^{\mathcal{J}} \Gamma$ *if* b *does not appear in* Γ *and* Γ *is closed*

Proof: by induction on the structure of Γ □

Proposition 3.3.2 (Independence) $\phi[a \mapsto \pi], \eta \vDash^{\mathcal{J}} \Gamma$ *iff* $\phi, \eta \vDash^{\mathcal{J}} \Gamma$ *if* a *does not appear in* Γ *and* Γ *is closed*

Proof: by induction on the structure of Γ □

I now present the natural deduction system for my logic. The proof theory of a similar system is examined in another work [BPW02]. Note that because my proof system includes first-order quantifiers and arithmetic over infinite sets of numbers, I cannot hope to make it complete [Göd31]. In fact, it is actually less strong than most classical systems for temporal logic, because there is no rule for "excluded middle."[6]

The hypothesis rule lets one use a hypothesis as a conclusion in a derivation:

[6] The "excluded middle" axiom is $p \vee p \supset \perp$ for an arbitrary proposition p. In practice, I have not yet found a need for an excluded middle axiom, but adding it to the system would result in no technical complications.

$$\frac{}{\Gamma \vdash 0 \leq t} \leq \mathsf{i}_0 \quad \frac{}{\Gamma \vdash t \leq t+1} \leq \mathsf{i}_{+1} \quad \frac{\Gamma \vdash t+1 \leq t}{\Gamma \vdash p' \,@\, t'} \leq \mathsf{e}_{+1}$$

$$\frac{\Gamma \vdash t_0 \leq t' \quad \Gamma \vdash p \,@\, t_0 \quad \Gamma,\, t_0 \leq b,\, p \,@\, b \vdash p \,@\, b+1}{\Gamma \vdash p \,@\, t'} \leq \mathsf{e}^b_{\mathsf{ind1}}$$

$$\frac{}{\Gamma \vdash t \leq t} \leq \mathsf{i}_{\mathsf{ref}} \quad \frac{\Gamma \vdash t_1 \leq t_2 \quad \Gamma \vdash t_2 \leq t_1 \quad \Gamma \vdash p \,@\, t_1}{\Gamma \vdash p \,@\, t_2} \leq \mathsf{i}_{\mathsf{asym}}$$

$$\frac{\Gamma \vdash t_1 \leq t_2 \quad \Gamma \vdash t_2 \leq t_3}{\Gamma \vdash t_1 \leq t_3} \leq \mathsf{i}_{\mathsf{trans}}$$

$$\frac{\Gamma \vdash t_1 \leq t_0 \quad \Gamma \vdash t_2 \leq t_0 \quad \Gamma,\, t_2 \leq t_1 \vdash p \,@\, t \quad \Gamma,\, t_1+1 \leq t_2 \vdash p \,@\, t}{\Gamma \vdash p \,@\, t} \leq \mathsf{e}_{\mathsf{linp}}$$

$$\frac{\Gamma \vdash t_0 \leq t_1 \quad \Gamma \vdash t_0 \leq t_2 \quad \Gamma,\, t_2 \leq t_1 \vdash p \,@\, t \quad \Gamma,\, t_1+1 \leq t_2 \vdash p \,@\, t}{\Gamma \vdash p \,@\, t} \leq \mathsf{e}_{\mathsf{linf}}$$

Figure 3.4: Inference Rules (Time Structure)

$$\frac{}{\Gamma_1,\, J,\, \Gamma_2 \vdash J} \,\mathsf{hyp}$$

In this section, I label introduction rules with i and elimination rules with e. The inference rules in Figure 3.4 enable one to derive judgments on relations between times based on the underlying time structure I have chosen. The first four rules come from Peano arithmetic and reflect the underlying model of times as natural numbers. The induction rule $\leq \mathsf{e}_{\mathsf{ind1}}$ enables one to infer that a judgment holds at an arbitrary future time if it holds now, and if it is preserved at each future time. This rule is a standard feature of discrete-time temporal logics.[7] The induction rule is actually a specialized version of a more general rule that cannot be translated from the LF representation without awkward notation (see Appendix B.1.3.6)—however, it is equivalent to the following rule in which certain LF constructs are internalized as temporal operators:

$$\frac{\Gamma \vdash p_1 \,@\, t_0 \quad \Gamma,\, t_0 \leq b,\, p_1 \,@\, b \vdash p_1 \vee p_2 \,@\, b+1}{\Gamma \vdash p_1 \,\mathcal{U}\, p_2 \,@\, t_0} \leq \mathsf{e}^b_{\mathsf{ind2}}$$

The next three rules reflect standard properties of partial orders (reflexivity, antisymmetry, transitivity). The final two rules are totality properties that reflect the fact that the time structure is linear in the past and linear in the future (respectively). Because the past is rooted at zero, the past linearity rule is unnecessary: one can always instantiate the first two premises of the future linearity rule with $\leq \mathsf{i}_0$. However, I keep these rules in their present form to preserve orthogonality and because the zero-instantiated form of the future linearity rule cannot be treated as an

[7] When a parameter appears as a superscript of an inference-rule label, it should be understood to mean that the parameter is "fresh" (*i.e.*, it does not appear in the conclusion of the rule).

$$\frac{}{\Gamma \vdash R(e_1, \ldots, e_k) : \mathsf{lo}\ (a)}\ \mathsf{loi}_R \qquad \frac{\Gamma \vdash p_1 : \mathsf{lo}\ (a) \quad \Gamma \vdash p_2 : \mathsf{lo}\ (a)}{\Gamma \vdash p_1 \wedge p_2 : \mathsf{lo}\ (a)}\ \mathsf{loi}_\wedge$$

$$\frac{\Gamma \vdash p_1 : \mathsf{lo}\ (a) \quad \Gamma \vdash p_2 : \mathsf{lo}\ (a)}{\Gamma \vdash p_1 \vee p_2 : \mathsf{lo}\ (a)}\ \mathsf{loi}_\vee \qquad \frac{\Gamma \vdash p_1 : \mathsf{lo}\ (a) \quad \Gamma \vdash p_2 : \mathsf{lo}\ (a)}{\Gamma \vdash p_1 \supset p_2 : \mathsf{lo}\ (a)}\ \mathsf{loi}_\supset$$

$$\frac{\Gamma \vdash [a'/x]\, p : \mathsf{lo}\ (a)}{\Gamma \vdash \forall x : \rho.\ p : \mathsf{lo}\ (a)}\ \mathsf{loi}_\forall^{a'} \qquad \frac{\Gamma \vdash [a'/x]\, p : \mathsf{lo}\ (a)}{\Gamma \vdash \exists x : \rho.\ p : \mathsf{lo}\ (a)}\ \mathsf{loi}_\exists^{a'}$$

$$\frac{}{\Gamma \vdash \bigcirc p_1 : \mathsf{lo}\ (a)}\ \mathsf{loi}_\bigcirc^{a \notin \mathcal{A}(p_1)}$$

$$\frac{}{\Gamma \vdash \Box p_1 : \mathsf{lo}\ (a)}\ \mathsf{loi}_\Box^{a \notin \mathcal{A}(p_1)} \qquad \frac{}{\Gamma \vdash p_1\, \mathcal{U}^\diamond\, p_2 : \mathsf{lo}\ (a)}\ \mathsf{loi}_{\mathcal{U}^\diamond}^{a \notin \mathcal{A}(p_1) \cup \mathcal{A}(p_2)}$$

$$\frac{\Gamma \vdash [a/x]\, p : \mathsf{lo}\ (a) \quad \Gamma \vdash e = e' \mathbin{@} t \quad \Gamma \vdash [e/x]\, p \mathbin{@} t}{\Gamma \vdash [e'/x]\, p \mathbin{@} t}\ \mathsf{loe}^a$$

Figure 3.5: Inference Rules (Locality)

elimination rule, and thus has undesirable proof-theoretic properties—see Bernard, *et al.* [BPW02] for a more detailed discussion.

The inference rules in Figure 3.5 enable one to infer locality: any parameter that does not appear in the scope of a temporal operator is local. The rule loe declares that equality at the current time is sufficient to perform substitutions for local variables.[8]

The inference rules in Figure 3.6 enable one to infer rigidity for expressions and propositions. The rule fli declares that all expressions are flexible (*e.g.*, rigid expressions are also flexible). The rule rii_f declares that rigidity is preserved by constant functions. The rule rie lets one infer the truth of a rigid proposition from one time to another.

When interpreted as a logic program, the inference rules in Figure 3.7 and Figure 3.8 implement a standard rewriting algorithm [BN98] on expressions and propositions if the atomic introduction rules are ignored ($\Longrightarrow^0 \mathsf{i}_{\mathsf{exp}}$, $\Longrightarrow^0 \mathsf{i}_{\mathsf{prop}}$, $\Longrightarrow \mathsf{i}_{\mathsf{exp}}$, $\Longrightarrow \mathsf{i}_{\mathsf{prop}}$)[9] and if $\Longrightarrow^{**} \mathsf{i}_{\mathsf{stop\_exp}}$ and $\Longrightarrow^{**} \mathsf{i}_{\mathsf{stop\_prop}}$ are only allowed to succeed when no other rule is applicable.

I optimize this algorithm by providing a special "step" judgment $\Longrightarrow^0$ that is guaranteed to introduce no new redices. In practice, this optimization enables a dramatic increase in performance, because many reductions fall into this category and terms that are already in normal form need not be rebuilt. There are built-in step introduction rules for constants, based on the constant interpretation function $\mathcal{J}$—these rules enable a proof checker to establish ordinary facts about constant ex-

[8] Note that this rule is unsound if the logic is extended to include "next-time expressions" such as are found in Manna and Pnueli [MP91].

[9] In practice, these rules are only used to introduce derived rules.

$$\frac{}{\Gamma \vdash e\!:\!\mathsf{fl}\,(\alpha)}\ \mathsf{fli} \qquad \frac{\Gamma \vdash e\!:\!\rho\,(\alpha)}{\Gamma \vdash e\!:\!\rho\,(\alpha_1, \alpha, \alpha_2)}\ \rho\mathsf{i}_{\mathsf{weak}}$$

$$\frac{}{\Gamma \vdash a\!:\!\mathsf{ri}\,(a)}\ \mathsf{rii}_{\mathsf{par}} \qquad \frac{}{\Gamma \vdash c\!:\!\mathsf{ri}\,(\alpha)}\ \mathsf{rii}_{\mathsf{con}} \qquad \frac{\Gamma \vdash e_1\!:\!\mathsf{ri}\,(\alpha) \quad \ldots \quad \Gamma \vdash e_k\!:\!\mathsf{ri}\,(\alpha)}{\Gamma \vdash f(e_1, \ldots, e_k)\!:\!\mathsf{ri}\,(\alpha)}\ \mathsf{rii}_f$$

$$\frac{\Gamma \vdash e_1\!:\!\rho\,(\alpha) \quad \ldots \quad \Gamma \vdash e_k\!:\!\rho\,(\alpha)}{\Gamma \vdash R(e_1, \ldots, e_k)\!:\!\rho\,(\alpha)}\ \rho\mathsf{i}_R \qquad \frac{\Gamma \vdash p_1\!:\!\rho\,(\alpha) \quad \Gamma \vdash p_2\!:\!\rho\,(\alpha)}{\Gamma \vdash p_1 \wedge p_2\!:\!\rho\,(\alpha)}\ \rho\mathsf{i}_\wedge$$

$$\frac{\Gamma \vdash p_1\!:\!\rho\,(\alpha) \quad \Gamma \vdash p_2\!:\!\rho\,(\alpha)}{\Gamma \vdash p_1 \vee p_2\!:\!\rho\,(\alpha)}\ \rho\mathsf{i}_\vee \qquad \frac{\Gamma \vdash p_1\!:\!\rho\,(\alpha) \quad \Gamma \vdash p_2\!:\!\rho\,(\alpha)}{\Gamma \vdash p_1 \supset p_2\!:\!\rho\,(\alpha)}\ \rho\mathsf{i}_\supset$$

$$\frac{\Gamma \vdash [a/x]\,p\!:\!\rho\,(\alpha, a)}{\Gamma \vdash \forall x\!:\!\rho'.\,p\!:\!\rho\,(\alpha)}\ \rho\mathsf{i}_\forall^a \qquad \frac{\Gamma \vdash [a/x]\,p\!:\!\rho\,(\alpha, a)}{\Gamma \vdash \exists x\!:\!\rho'.\,p\!:\!\rho\,(\alpha)}\ \rho\mathsf{i}_\exists^a$$

$$\frac{\Gamma \vdash p\!:\!\rho\,(\alpha)}{\Gamma \vdash \bigcirc p\!:\!\rho\,(\alpha)}\ \rho\mathsf{i}_\bigcirc \qquad \frac{\Gamma \vdash p\!:\!\rho\,(\alpha)}{\Gamma \vdash \Box p\!:\!\rho\,(\alpha)}\ \rho\mathsf{i}_\Box \qquad \frac{\Gamma \vdash p_1\!:\!\rho\,(\alpha) \quad \Gamma \vdash p_2\!:\!\rho\,(\alpha)}{\Gamma \vdash p_1\,\mathcal{U}^\diamond\,p_2\!:\!\rho\,(\alpha)}\ \rho\mathsf{i}_{\mathcal{U}^\diamond}$$

$$\frac{\Gamma \vdash p\!:\!\mathsf{ri}\,(\cdot) \quad \Gamma \vdash p \,@\, t}{\Gamma \vdash p \,@\, t'}\ \mathsf{rie}$$

Figure 3.6: Inference Rules (Rigidity)

pressions. The other step introduction rules enable the code producer to customize the rewriting strategy according to the certification strategy by introducing derived rules from provable equivalences. The normalization elimination rules enable the results of a rewriting search to be used in an ordinary derivation (usually a derived rule).

The inference rules for connectives (see Figure 3.9) are straightforward adaptations of the standard introduction and elimination rules for natural deduction. Note that the rule $\vee\mathsf{e}$ does not require the time of the first premise to match the time of the conclusion. The generalization of this rule to conclusions on the interval judgment can be derived from within the system. The introduction and elimination rules for these connectives are locally sound and locally complete in the sense of Pfenning [Pfe99] by adapting the standard reductions and expansions. The standard introduction and elimination rules for quantifiers can be adapted to local soundness and local completeness by explicitly considering the rigidity of the appropriate parameter (see Figure 3.9).

The introduction and elimination rules for temporal operators are based on Simpson [Sim94] and Davies [Dav96] (see Figure 3.9). The rule $\Box\mathsf{i}$ enables one to infer that a proposition is true at all future times if it can be proven at any arbitrary future time. The rule $\Box\mathsf{e}$ follows directly from the definition of $\Box$. The rule $\mathcal{U}^\diamond\mathsf{e}$ resembles $\exists\mathsf{e}$: given $p_1\,\mathcal{U}^\diamond\,p_2 \,@\, t$, one can derive $p' \,@\, t'$ if one can derive $p' \,@\, t'$ under the assumption that p_2 holds at some arbitrary point in the future, and p_1 holds

$$\frac{}{\Gamma \vdash^{\mathcal{J}} f(c_1, \ldots, c_k) \Longrightarrow^0 c'} \Longrightarrow^0 \mathsf{i}_f^{\mathcal{J}(f)(\mathcal{J}(c_1),\ldots,\mathcal{J}(c_k))=\mathcal{J}(c')}$$

$$\frac{}{\Gamma \vdash^{\mathcal{J}} R(c_1, \ldots, c_k) \Longrightarrow^0 \top} \Longrightarrow^0 \top\mathsf{i}_R^{\langle\mathcal{J}(c_1),\ldots,\mathcal{J}(c_k)\rangle\in\mathcal{J}(R)}$$

$$\frac{}{\Gamma \vdash^{\mathcal{J}} R(c_1, \ldots, c_k) \Longrightarrow^0 \bot} \Longrightarrow^0 \bot\mathsf{i}_R^{\langle\mathcal{J}(c_1),\ldots,\mathcal{J}(c_k)\rangle\notin\mathcal{J}(R)}$$

$$\frac{\Gamma \vdash e = e' \,@\, b}{\Gamma \vdash e \Longrightarrow^0 e'} \Longrightarrow^0 \mathsf{i}_{\mathsf{exp}}^b \qquad \frac{\Gamma \vdash p \equiv p' \,@\, b}{\Gamma \vdash p \Longrightarrow^0 p'} \Longrightarrow^0 \mathsf{i}_{\mathsf{prop}}^b$$

$$\frac{\Gamma \vdash e = e' \,@\, b}{\Gamma \vdash e \Longrightarrow e'} \Longrightarrow \mathsf{i}_{\mathsf{exp}}^b \qquad \frac{\Gamma \vdash p \equiv p' \,@\, b}{\Gamma \vdash p \Longrightarrow p'} \Longrightarrow \mathsf{i}_{\mathsf{prop}}^b$$

$$\frac{\Gamma \vdash e \Longrightarrow^0 e'}{\Gamma \vdash e \Longrightarrow^{**} e'} \Longrightarrow^{**} \mathsf{i}_{\mathsf{step0\_exp}} \qquad \frac{}{\Gamma \vdash e \Longrightarrow^{**} e} \Longrightarrow^{**} \mathsf{i}_{\mathsf{stop\_exp}}$$

$$\frac{\Gamma \vdash e \Longrightarrow e' \quad \Gamma \vdash e' \Longrightarrow^{*} e''}{\Gamma \vdash e \Longrightarrow^{**} e''} \Longrightarrow^{**} \mathsf{i}_{\mathsf{step\_exp}}$$

$$\frac{\Gamma \vdash p \Longrightarrow^0 p'}{\Gamma \vdash p \Longrightarrow^{**} p'} \Longrightarrow^{**} \mathsf{i}_{\mathsf{step0\_prop}} \qquad \frac{}{\Gamma \vdash p \Longrightarrow^{**} p} \Longrightarrow^{**} \mathsf{i}_{\mathsf{stop\_prop}}$$

$$\frac{\Gamma \vdash p \Longrightarrow p' \quad \Gamma \vdash p' \Longrightarrow^{*} p''}{\Gamma \vdash p \Longrightarrow^{**} p''} \Longrightarrow^{**} \mathsf{i}_{\mathsf{step\_prop}}$$

Figure 3.7: Inference Rules (Term Rewriting)

$$\frac{}{\Gamma \vdash a \Longrightarrow^* a} \Longrightarrow^* \mathsf{i}_{\mathsf{par}} \qquad \frac{}{\Gamma \vdash c \Longrightarrow^* c} \Longrightarrow^* \mathsf{i}_{\mathsf{con}}$$

$$\frac{\Gamma \vdash e_1 \Longrightarrow^* e_1' \quad \ldots \quad \Gamma \vdash e_k \Longrightarrow^* e_k' \quad \Gamma \vdash f(e_1', \ldots, e_k') \Longrightarrow^{**} e''}{\Gamma \vdash f(e_1, \ldots, e_k) \Longrightarrow^* e''} \Longrightarrow^* \mathsf{i}_f$$

$$\frac{\Gamma \vdash e_1 \Longrightarrow^* e_1' \quad \ldots \quad \Gamma \vdash e_k \Longrightarrow^* e_k' \quad \Gamma \vdash R(e_1', \ldots, e_k') \Longrightarrow^{**} p''}{\Gamma \vdash R(e_1, \ldots, e_k) \Longrightarrow^* p''} \Longrightarrow^* \mathsf{i}_R$$

$$\frac{\Gamma \vdash p_1 \Longrightarrow^* p_1' \quad \Gamma \vdash p_2 \Longrightarrow^* p_2' \quad \Gamma \vdash p_1' \wedge p_2' \Longrightarrow^{**} p''}{\Gamma \vdash p_1 \wedge p_2 \Longrightarrow^* p''} \Longrightarrow^* \mathsf{i}_\wedge$$

$$\frac{\Gamma \vdash p_1 \Longrightarrow^* p_1' \quad \Gamma \vdash p_2 \Longrightarrow^* p_2' \quad \Gamma \vdash p_1' \vee p_2' \Longrightarrow^{**} p''}{\Gamma \vdash p_1 \vee p_2 \Longrightarrow^* p''} \Longrightarrow^* \mathsf{i}_\vee$$

$$\frac{\Gamma \vdash p_1 \Longrightarrow^* p_1' \quad \Gamma \vdash p_2 \Longrightarrow^* p_2' \quad \Gamma \vdash p_1' \supset p_2' \Longrightarrow^{**} p''}{\Gamma \vdash p_1 \supset p_2 \Longrightarrow^* p''} \Longrightarrow^* \mathsf{i}_\supset$$

$$\frac{\Gamma \vdash [a/x]\, p \Longrightarrow^* [a/x]\, p' \quad \Gamma \vdash \forall x.\, p' \Longrightarrow^{**} p''}{\Gamma \vdash \forall x.\, p \Longrightarrow^* p''} \Longrightarrow^* \mathsf{i}_\forall^a$$

$$\frac{\Gamma \vdash [a/x]\, p \Longrightarrow^* [a/x]\, p' \quad \Gamma \vdash \exists x.\, p' \Longrightarrow^{**} p''}{\Gamma \vdash \exists x.\, p \Longrightarrow^* p''} \Longrightarrow^* \mathsf{i}_\exists^a$$

$$\frac{\Gamma \vdash p \Longrightarrow^* p' \quad \Gamma \vdash \bigcirc p' \Longrightarrow^{**} p''}{\Gamma \vdash \bigcirc p \Longrightarrow^* p''} \Longrightarrow^* \mathsf{i}_\bigcirc$$

$$\frac{\Gamma \vdash p \Longrightarrow^* p' \quad \Gamma \vdash \Box p' \Longrightarrow^{**} p''}{\Gamma \vdash \Box p \Longrightarrow^* p''} \Longrightarrow^* \mathsf{i}_\Box$$

$$\frac{\Gamma \vdash p_1 \Longrightarrow^* p_1' \quad \Gamma \vdash p_2 \Longrightarrow^* p_2' \quad \Gamma \vdash p_1'\, \mathcal{U}^\diamond\, p_2' \Longrightarrow^{**} p''}{\Gamma \vdash p_1\, \mathcal{U}^\diamond\, p_2 \Longrightarrow^* p''} \Longrightarrow^* \mathsf{i}_{\mathcal{U}^\diamond}$$

$$\frac{\Gamma \vdash e \Longrightarrow^* e'}{\Gamma \vdash e = e' \,@\, t} \Longrightarrow^* \mathsf{e}_{\mathsf{exp}} \qquad \frac{\Gamma \vdash p \Longrightarrow^* p'}{\Gamma \vdash p \equiv p' \,@\, t} \Longrightarrow^* \mathsf{e}_{\mathsf{prop}}$$

Figure 3.8: Inference Rules (Normalization)

$$\frac{\Gamma \vdash p_1 \mathbin{@} t \quad \Gamma \vdash p_2 \mathbin{@} t}{\Gamma \vdash p_1 \wedge p_2 \mathbin{@} t} \wedge\mathsf{i} \quad \frac{\Gamma \vdash p_1 \wedge p_2 \mathbin{@} t}{\Gamma \vdash p_1 \mathbin{@} t} \wedge\mathsf{el} \quad \frac{\Gamma \vdash p_1 \wedge p_2 \mathbin{@} t}{\Gamma \vdash p_2 \mathbin{@} t} \wedge\mathsf{er}$$

$$\frac{\Gamma \vdash p_1 \mathbin{@} t}{\Gamma \vdash p_1 \vee p_2 \mathbin{@} t} \vee\mathsf{il} \quad \frac{\Gamma \vdash p_2 \mathbin{@} t}{\Gamma \vdash p_1 \vee p_2 \mathbin{@} t} \vee\mathsf{ir}$$

$$\frac{\Gamma \vdash p_1 \vee p_2 \mathbin{@} t \quad \Gamma,\, p_1 \mathbin{@} t \vdash p' \mathbin{@} t' \quad \Gamma,\, p_2 \mathbin{@} t \vdash p' \mathbin{@} t'}{\Gamma \vdash p' \mathbin{@} t'} \vee\mathsf{e}$$

$$\frac{\Gamma,\, p_1 \mathbin{@} t \vdash p_2 \mathbin{@} t}{\Gamma \vdash p_1 \supset p_2 \mathbin{@} t} \supset\mathsf{i} \quad \frac{\Gamma \vdash p_1 \supset p_2 \mathbin{@} t \quad \Gamma \vdash p_1 \mathbin{@} t}{\Gamma \vdash p_2 \mathbin{@} t} \supset\mathsf{e}$$

$$\frac{\Gamma,\, a{:}\rho \vdash [a/x]\, p \mathbin{@} t}{\Gamma \vdash \forall x{:}\rho.\, p \mathbin{@} t} \forall\mathsf{i}^a \quad \frac{\Gamma \vdash \forall x{:}\rho.\, p \mathbin{@} t \quad \Gamma \vdash e{:}\rho}{\Gamma \vdash [e/x]\, p \mathbin{@} t} \forall\mathsf{e}$$

$$\frac{\Gamma \vdash e{:}\rho \quad \Gamma \vdash [e/x]\, p \mathbin{@} t}{\Gamma \vdash \exists x{:}\rho.\, p \mathbin{@} t} \exists\mathsf{i} \quad \frac{\Gamma \vdash \exists x{:}\rho.\, p \mathbin{@} t \quad \Gamma,\, a{:}\rho,\, [a/x]\, p \mathbin{@} t \vdash p' \mathbin{@} t'}{\Gamma \vdash p' \mathbin{@} t'} \exists\mathsf{e}^a$$

$$\frac{\Gamma \vdash p \mathbin{@} t+1}{\Gamma \vdash \bigcirc p \mathbin{@} t} \bigcirc\mathsf{i} \quad \frac{\Gamma \vdash \bigcirc p \mathbin{@} t}{\Gamma \vdash p \mathbin{@} t+1} \bigcirc\mathsf{e}$$

$$\frac{\Gamma,\, t \leq b_1 \vdash p_1 \mathbin{@} b_1}{\Gamma \vdash \Box p_1 \mathbin{@} t} \Box\mathsf{i}^{b_1} \quad \frac{\Gamma \vdash \Box p_1 \mathbin{@} t \quad \Gamma \vdash t \leq t_1}{\Gamma \vdash p_1 \mathbin{@} t_1} \Box\mathsf{e}$$

$$\frac{\Gamma \vdash t \leq t_2 \quad \Gamma \vdash p_1 \mathbin{@} [t, t_2) \quad \Gamma \vdash p_2 \mathbin{@} t_2}{\Gamma \vdash p_1\, \mathcal{U}^{\Diamond}\, p_2 \mathbin{@} t} \mathcal{U}^{\Diamond}\mathsf{i}$$

$$\frac{\Gamma \vdash p_1\, \mathcal{U}^{\Diamond}\, p_2 \mathbin{@} t \quad \Gamma,\, t \leq b_2,\, p_1 \mathbin{@} [t, b_2),\, p_2 \mathbin{@} b_2 \vdash p' \mathbin{@} t'}{\Gamma \vdash p' \mathbin{@} t'} \mathcal{U}^{\Diamond}\mathsf{e}^{b_2}$$

Figure 3.9: Inference Rules (Instants)

until then. The introduction and elimination rules for the temporal operators are also locally sound and locally complete [BPW02].

The inference rules for the interval judgment internalize the semantic interpretation of the judgment:

$$\frac{\Gamma,\, t_1 \le b,\, b+1 \le t_2 \vdash p \mathbin{@} b}{\Gamma \vdash p \mathbin{@}[t_1, t_2)}\ \mathsf{ovi}^b \qquad \frac{\Gamma \vdash p \mathbin{@}[t_1, t_2) \quad \Gamma \vdash t_1 \le t \quad \Gamma \vdash t+1 \le t_2}{\Gamma \vdash p \mathbin{@} t}\ \mathsf{ove}$$

$\top$ always holds:

$$\frac{}{\Gamma \vdash \top \mathbin{@} t}\ \top\mathsf{i}$$

Additionally, the following generic inference rules apply to all constant relations:

$$\frac{\Gamma,\, R(e_1, \ldots, e_k) \mathbin{@} t \vdash p' \mathbin{@} t' \quad \Gamma,\, (\neg R)(e_1, \ldots, e_k) \mathbin{@} t \vdash p' \mathbin{@} t'}{\Gamma \vdash p' \mathbin{@} t'}\ R\mathsf{i}_{\mathsf{case}}$$

$$\frac{\Gamma \vdash R(e_1, \ldots, e_k) \mathbin{@} t \quad \Gamma \vdash (\neg R)(e_1, \ldots, e_k) \mathbin{@} t}{\Gamma \vdash p' \mathbin{@} t'}\ R\mathsf{e}_{\mathsf{contr}}$$

Any relation is either true or false (but never both) at a given time for any given set of argument expressions. Note that because this is a natural-deduction system, I do not generalize this principle to all propositions in the form of an "excluded middle" rule.

Finally, the following inference rules take propositions in and out of the restricted formal system, governed by $\underline{@}$:

$$\frac{\Gamma \vdash p \mathbin{@} t}{\Gamma \vdash p \mathbin{\underline{@}} t}\ \underline{@}\mathsf{i} \qquad \frac{\Gamma \vdash p \mathbin{\underline{@}} t}{\Gamma \vdash p \mathbin{@} t}\ \underline{@}\mathsf{e}$$

See Section 8.2 for a discussion of how this judgment is used by the code producer to define a restricted formal system.

3.4 Theories

In this section, I develop theories for equality, natural numbers, pairs, and lists. First, however, I introduce notation for general algebraic properties of functions and relations.

3.4.1 Algebraic Properties

In this section, I catalog a set of derived indexed propositions that encode general algebraic properties of functions and relations. The definitions of these propositions are based on generally accepted mathematical practice [DP90, Cla84], so I do not elaborate them here, but I refer the interested reader to Appendix B.1.1.10 and Appendix B.1.1.11, where they can be examined in their entirety.

In Table 3.2, I enumerate the relevant properties on function constants.

In Table 3.3, I enumerate the relevant properties on relation constants.

$\texttt{inj}(f)$	$\underline{f}$ is an injective function
$\texttt{assoc}(f)$	$\underline{f}$ is an associative binary function
$\texttt{idl}(f, c_1)$	$\underline{c_1}$ is a left identity for binary function $\underline{f}$
$\texttt{idr}(f, c_2)$	$\underline{c_2}$ is a right identity for binary function $\underline{f}$
$\texttt{id}(f, c)$	$\underline{c}$ is an identity for binary function $\underline{f}$
$\texttt{invl}(f, c, f_1)$	$\underline{f_1}$ computes left inverses for binary function $\underline{f}$ and identity element c
$\texttt{invr}(f, c, f_2)$	$\underline{f_2}$ computes right inverses for binary function $\underline{f}$ and identity element c
$\texttt{inv}(f, c, f')$	$\underline{f'}$ computes inverses for binary function $\underline{f}$ and identity element c
$\texttt{comm}(f)$	$\underline{f}$ is a commutative binary function
$\texttt{distl}(f_1, f_2)$	$\underline{f_2}$ distributes to the left over $\underline{f_1}$
$\texttt{distr}(f_1, f_2)$	$\underline{f_2}$ distributes to the right over $\underline{f_1}$
$\texttt{dist}(f_1, f_2)$	$\underline{f_2}$ distributes over $\underline{f_1}$
$\texttt{monoid}(f, c)$	$\underline{f}$ is a monoid with identity element $\underline{c}$
$\texttt{group}(f, c, f')$	$\underline{f}$ is a group product with identity element $\underline{c}$; $\underline{f'}$ computes inverses for $\underline{f}$ and $\underline{c}$
$\texttt{comm\_group}(f, c, f')$	$\underline{f}$ is a commutative group product with identity element $\underline{c}$; $\underline{f'}$ computes inverses for $\underline{f}$ and $\underline{c}$
$\texttt{ring}(f_1, c_1, f_1', f_2)$	$\underline{f_2}$ is a ring product for ring sum $\underline{f_1}$ with identity element $\underline{c_1}$; $\underline{f_1'}$ computes inverses for $\underline{f_1}$ and $\underline{c_1}$
$\texttt{comm\_ring}(f_1, c_1, f_1', f_2, c_2)$	$\underline{f_2}$ is a commutative ring product for ring sum $\underline{f_1}$ with identity element $\underline{c_1}$; $\underline{f_1'}$ computes inverses for $\underline{f_1}$ and $\underline{c_1}$; $\underline{c_2}$ is the identity for f_2
$\texttt{idem}(f)$	$\underline{f}$ is idempotent
$\texttt{absorb}(f_1, f_2)$	$\underline{f_1}$ absorbs $\underline{f_2}$

Table 3.2: Properties of Functions

`fun`(R)	$\underline{R}$ is a partial function
`ref`(R)	$\underline{R}$ is reflexive
`irref`(R)	$\underline{R}$ is irreflexive
`sym`(R)	$\underline{R}$ is symmetric
`antisym`(R)	$\underline{R}$ is antisymmetric
`trans`(R)	$\underline{R}$ is transitive
`total`(R)	$\underline{R}$ is total
`bot`(R, c)	$\underline{c}$ is a bottom element for $\underline{R}$
`top`(R, c)	$\underline{c}$ is a top element for $\underline{R}$
`preorder`(R)	$\underline{R}$ is a preorder
`equiv`(R)	$\underline{R}$ is an equivalence relation
`order`(R)	$\underline{R}$ is an order
`tot_order`(R)	$\underline{R}$ is a total order
`str_order`(R)	$\underline{R}$ is a strict order
`str_tot_order`(R)	$\underline{R}$ is a strict total order
`meet`(R, f)	$\underline{f}$ is the meet operation for order $\underline{R}$
`join`(R, f)	$\underline{f}$ is the join operation for order $\underline{R}$
`latt`(R, f_1, f_2)	Val^τ is a lattice under order $\underline{R^{\tau\times\tau\to o}}$ with meet $\underline{f_1}$ and join $\underline{f_2}$
`dist_latt`(R, f_1, f_2)	Val^τ is a distributive lattice under order $\underline{R^{\tau\times\tau\to o}}$ with meet $\underline{f_1}$ and join $\underline{f_2}$
`comp`(f_1, f_2, c_0, c_1, f')	$\underline{f'}$ computes complements for meet $\underline{f_1}$, join $\underline{f_2}$, zero element $\underline{c_0}$, and one element $\underline{c_1}$
`bool_latt`$(R, f_1, f_2, c_0, c_1, f')$	Val^τ is a boolean lattice under order $\underline{R^{\tau\times\tau\to o}}$ with meet $\underline{f_1}$ and join $\underline{f_2}$; $\underline{f'}$ computes complements for zero element $\underline{c_0}$ and one element $\underline{c_1}$

Table 3.3: Properties of Relations

3.4.2 Equality

3.4.2.1 Semantics

The semantics of equality is based on mathematical identity:

$$\mathcal{J}(=^{\tau}) = \{\langle v, v\rangle \mid v \in \mathit{Val}^{\tau}\}$$

3.4.2.2 Inference Rules

The rules for equality take Necula [Nec98] as a point of departure:

$$\frac{}{\Gamma \vdash e = e \,@\, t} = \mathsf{i}_{\mathsf{ref}} \qquad \frac{\Gamma,\, a\!:\!\rho,\, a = e \,@\, t \vdash p' \,@\, t'}{\Gamma \vdash p' \,@\, t'} = \mathsf{i}^{a}_{\mathsf{some}}$$

$$\frac{\Gamma \vdash e = e' \,@\, b \quad \Gamma \vdash [e/x]\, p \,@\, t}{\Gamma \vdash [e'/x]\, p \,@\, t} = \mathsf{e}^{b}_{\mathsf{cong}}$$

Note that the congruence rule $= \mathsf{e}_{\mathsf{cong}}$ must be weakened to account for the case in which p contains temporal operators: one must show that e and e' are equal at all times (this rule complements loe). The rule $= \mathsf{i}_{\mathsf{some}}$ enables one to introduce a parameter (typically rigid) that is equal to the current value of some expression (typically flexible).

3.4.3 Pairs

$\mathtt{pair}\langle\tau_1\rangle\langle\tau_2\rangle$ is the type of pairs of elements of types τ_1 and τ_2. $\mathtt{mkp}^{\tau_1\times\tau_2\to\mathtt{pair}\langle\tau_1\rangle\langle\tau_2\rangle}$ is a function that constructs a new pair, whereas the function $\mathtt{left}^{\mathtt{pair}\langle\tau_1\rangle\langle\tau_2\rangle\to\tau_1}$ and the function $\mathtt{right}^{\mathtt{pair}\langle\tau_1\rangle\langle\tau_2\rangle\to\tau_2}$ project particular elements out of a given pair.

3.4.3.1 Semantics

Pairs are represented semantically as mathematical pairs:

$$\mathit{Val}^{\mathtt{pair}\langle\tau_1\rangle\langle\tau_2\rangle} = \mathit{Val}^{\tau_1} \times \mathit{Val}^{\tau_2}$$

The interpretation of the pair functions is similarly straightforward:

$$\begin{aligned} \mathcal{J}(\mathtt{mkp}) &= v_1, v_2 \mapsto \langle v_1, v_2\rangle \\ \mathcal{J}(\mathtt{left}) &= \langle v_1, v_2\rangle \mapsto v_1 \\ \mathcal{J}(\mathtt{right}) &= \langle v_1, v_2\rangle \mapsto v_2 \end{aligned}$$

3.4.3.2 Inference Rules

The pair functions are characterized by the following axioms:

$$\frac{}{\Gamma \vdash \mathtt{left}(\mathtt{mkp}(e_1, e_2)) = e_1 \,@\, t}\ \mathsf{left\_mk} \qquad \frac{}{\Gamma \vdash \mathtt{right}(\mathtt{mkp}(e_1, e_2)) = e_2 \,@\, t}\ \mathsf{right\_mk}$$

Additionally, the pair constructor function is injective:

$$\frac{}{\Gamma \vdash \mathtt{inj}(\mathtt{mkp}) \,@\, t}\ \mathsf{mk\_inj}$$

3.4.4 Lists

$\texttt{list}\langle\tau\rangle$ is the type of lists of elements of type τ. $\texttt{empty}^{\texttt{list}\langle\tau\rangle}$ is the empty list of type τ, $\texttt{cons}^{\tau\times\texttt{list}\langle\tau\rangle\rightarrow\texttt{list}\langle\tau\rangle}$ is a function that adds an element to the head of a list, and $\texttt{head}^{\texttt{list}\langle\tau\rangle\rightarrow\tau}$ and $\texttt{tail}^{\texttt{list}\langle\tau\rangle\rightarrow\texttt{list}\langle\tau\rangle}$ are functions that project out the head and tail of a given list.

3.4.4.1 Semantics

The type of a list of type τ is modeled semantically by the set of all finite sequences of values of type τ:

$$Val^{\texttt{list}\langle\tau\rangle} = (Val^{\tau})^*$$

The list functions are interpreted as follows:

$$\begin{aligned}
\mathcal{J}(\texttt{empty}) &= \langle\,\rangle \\
\mathcal{J}(\texttt{cons}) &= v, v' \mapsto \langle v, v'\rangle \\
\mathcal{J}(\texttt{head}^{\texttt{list}\langle\tau\rangle\rightarrow\tau}) &= v \mapsto \begin{cases} v' & \text{if } v = \langle v', v''\rangle \\ v' \in Val^{\tau} & \text{otherwise} \end{cases} \\
\mathcal{J}(\texttt{tail}) &= v \mapsto \begin{cases} v'' & \text{if } v = \langle v', v''\rangle \\ v' \in Val^{\texttt{list}\langle\tau\rangle} & \text{otherwise} \end{cases}
\end{aligned}$$

The value of $\texttt{head}(\texttt{empty}^{\texttt{list}\langle\tau\rangle})$ is an unspecified value of type τ. Similarly, the value of $\texttt{tail}(\texttt{empty}^{\texttt{list}\langle\tau\rangle})$ is an unspecified value of type $\texttt{list}\langle\tau\rangle$.

3.4.4.2 Inference Rules

The list functions are characterized by the following axioms:

$$\frac{}{\Gamma \vdash \texttt{head}(\texttt{cons}(e_1, e_2)) = e_1 \,@\, t}\ \textsf{head\_cons} \qquad \frac{}{\Gamma \vdash \texttt{tail}(\texttt{cons}(e_1, e_2)) = e_2 \,@\, t}\ \textsf{tail\_cons}$$

The list constructor function is also injective:

$$\frac{}{\Gamma \vdash \texttt{inj}(\texttt{cons}) \,@\, t}\ \textsf{cons\_inj}$$

3.5 Soundness

I can now show that my inference rules are sound with respect to the formal model of Section 3.2. In Chapter 4, I show that additional domain-specific inference rules for the machine model are also sound.

Proposition 3.5.1 (Soundness) $\phi, \eta \vDash^{\mathcal{J}} J$ *if* $\phi, \eta \vDash^{\mathcal{J}} \Gamma$ *and* $\Gamma \vdash^{\mathcal{J}} J$

PROOF:

by induction on the derivation of $\Gamma \vdash^{\mathcal{J}} J$

$\phi, \eta \vDash^{\mathcal{J}} \Gamma \qquad \Gamma \vdash^{\mathcal{J}} J$ — Prem.

let $\mathcal{D}$ be the derivation of $\Gamma \vdash^{\mathcal{J}} J$

case: $J = p \,@\, t', \mathcal{D} = \dfrac{\begin{matrix}\mathcal{D}'_1 \\ \Gamma \vdash t_0 \leq t'\end{matrix} \quad \begin{matrix}\mathcal{D}'_2 \\ \Gamma \vdash p \,@\, t_0\end{matrix} \quad \begin{matrix}\mathcal{D}'_3 \\ \Gamma, t_0 \leq b, p \,@\, b \vdash p \,@\, b+1\end{matrix}}{\Gamma \vdash p \,@\, t'} \leq \mathsf{e}^{b}_{\mathsf{ind1}}$

- $\phi, \eta \vDash^{\mathcal{J}} t_0 \leq t' \qquad \phi, \eta \vDash^{\mathcal{J}} p \,@\, t_0$ — I.H.
- let $j_0 = \mathcal{V}_\eta(t_0), j' = \mathcal{V}_\eta(t')$
- $j_0 \leq j'$ — Def. $\vDash$
- let $b \notin \mathcal{B}(t_0) \cup \mathcal{B}(t') \cup \mathcal{B}(\Gamma)$
- $\phi, \eta[b \mapsto j_0] \vDash^{\mathcal{J}} p \,@\, b$ — Prop. 3.2.18
- for all $j \geq j_0$
  - $\phi, \eta[b \mapsto j] \vDash^{\mathcal{J}} p \,@\, b$ — Hyp.
  - let $\eta' = \eta[b \mapsto j]$
  - $\mathcal{V}_{\eta'}(b) = j$ — Def. $\mathcal{V}_\eta$
  - $\phi, \eta' \vDash^{\mathcal{J}} \Gamma$ — Prop. 3.3.1
  - $\phi, \eta' \vDash^{\mathcal{J}} t_0 \leq b$ — Def. $\vDash$
  - $\phi, \eta' \vDash^{\mathcal{J}} \Gamma, t_0 \leq b, p \,@\, b$ — Def. $\vDash$
  - $\phi, \eta' \vDash^{\mathcal{J}} p \,@\, b+1$ — I.H.
  - $\mathcal{V}_{\eta'}(b+1) = j+1$ — Def. $\mathcal{V}_\eta$
  - let $b' \neq b$
  - $\phi, \eta'[b' \mapsto j+1] \vDash^{\mathcal{J}} p \,@\, b'$ — Prop. 3.2.18
  - $\phi, \eta[b \mapsto j+1] \vDash^{\mathcal{J}} p \,@\, b$ — Prop. 3.2.13
- $\phi, \eta[b \mapsto j'] \vDash^{\mathcal{J}} p \,@\, b$ — induction
- $\phi, \eta \vDash^{\mathcal{J}} p \,@\, t'$ — Prop. 3.2.18

other cases are similar

$\square$

Chapter 4

Machine Model

In this section, I develop an abstract model for an idealized subset of the Intel IA-32 processor architecture [Int01], and then prove that a formal encoding of this semantics in temporal logic is sound with respect to the original model. This abstract model was chosen to correspond directly to my PCC implementation and is not the simplest way to model a processor (refer to Bernard and Lee [BL02b], for example, for a simpler machine model). However, the exercise does demonstrate that my approach can accommodate an irregular instruction set with complex addressing modes and few registers without causing an unacceptable amount of complication.

This processor operates on "words," each of which is 32 bits in size. There are a small number of general-purpose registers that each contain a single word, a word-sized program counter, a word-sized collection of status flags, and two mappings from words to words that model the memory and stack.[1] The processor executes a program that is a partial function from addresses to instructions. I assume that the program is effectively in a separate memory and thereby protected from modification.

The stack is in a separate register from the memory so that the code producer can treat it as an extension of the register file (as in Necula [Nec98]) and thereby not have to provide an explicit proof that aliasing cannot occur between the stack and heap. Although this design slightly constrains the code generation options available to the code consumer, I believe that it is a favorable trade-off, given the importance of having an efficient stack representation for processors with a small number of registers. In effect, I treat the stack as though it were in a different *segment* from the heap. In an implementation of this model, one can actually set up an isolated address space for the stack using the IA-32 segment registers. Alternatively, one can use a "flat" address space and design a security policy to ensure that stack and heap addresses cannot alias. In Section 10.2.2, I suggest how this latter approach might be realized.

This chapter is organized as follows: in Section 4.1, I define the instruction set of the abstract machine. Section 4.2 contains the new syntactic elements that

[1] Accesses to the memory and stack are restricted to aligned addresses by the memory-safety policy (see Chapter 5).

are need to model the abstract machine formally in temporal logic. Section 4.3 contains the complete operational semantics of the machine, specified in informal mathematical notation. In Section 4.4, I sample the logical inference rules that formalize the operational semantics in temporal logic (see Appendix B.2 for the complete formalization). Finally, in Section 4.5, I show that the inference rules are sound with respect to the operational semantics.

I will define the following new types to represent abstract machine states and machine-language programs:

`wd`	32-bit machine words
`op1`	Unary operator identifiers
`op2`	Binary operator identifiers
`op3`	Ternary operator identifiers
`cop`	Conditional operator identifiers
`greg`	General-purpose register identifiers/register tokens
`mapg`	Mappings from register tokens to machine words
`mapw`	Mappings from machine words to machine words
`state`	Machine-state tuples
`ma`	Memory addresses
`ea`	Effective addresses
`inst`	Instructions
`prog`	Programs

This table is intended to suggest an informal reading for each of the type identifiers—precise definitions will be given later in this chapter.

I first develop the instruction-set representation at the level of values, then "reflect" this representation into the language of expressions by providing a constant or constant function for each possible value.

4.1 Instruction Set

Figure 4.1 contains a definition of the value-level representation of instructions.

A *machine word n* is a value of type `wd`. The semantic interpretation of `wd` is the first 2^{32} natural numbers. Words are inherently unsigned, but negative numbers can be simulated by signed operators using the two's complement convention. A *register token r* identifies a general-purpose register; each register token is a value of type `greg`. A small subset of the total functions from words to words are designated as *unary operators op1* (type `op1`). A *binary operator op2* (type `op2`) designates a total function from pairs of words to words. A *ternary operator op3* (type `op3`) designates a total function from word triples to words. A *conditional operator cop* (type `cop`) designates a property of the machine status flags. Each operator corresponds to a distinct arithmetic or logical operation in the IA-32 instruction set. Note that an operator that would ordinarily correspond to partial function (such as division) is assigned unspecified results for the undefined elements of its domain in order to make the operator total.

Unary op.	$\mathit{op1}$	$::= \mathtt{op1\_inc} \mid \mathtt{op1\_dec} \mid \mathtt{op1\_neg} \mid \mathtt{op1\_not}$
Binary op.	$\mathit{op2}$	$::= \mathtt{op2\_add} \mid \mathtt{op2\_sub} \mid \mathtt{op2\_imul}$
		$\mid \mathtt{op2\_and} \mid \mathtt{op2\_or} \mid \mathtt{op2\_xor}$
		$\mid \mathtt{op2\_sf2}$
Ternary op.	$\mathit{op3}$	$::= \mathtt{op3\_idiv} \mid \mathtt{op3\_irem}$
Conditional op.	cop	$::= \mathtt{cop\_z} \mid \mathtt{cop\_s} \mid \mathtt{cop\_o} \mid \mathtt{cop\_c} \mid \mathtt{cop\_na} \mid \mathtt{cop\_l} \mid \mathtt{cop\_ng}$
		$\mid \mathtt{cop\_nz} \mid \mathtt{cop\_ns} \mid \mathtt{cop\_no} \mid \mathtt{cop\_nc} \mid \mathtt{cop\_a} \mid \mathtt{cop\_nl} \mid \mathtt{cop\_g}$
Register tokens	r	$::= \mathtt{eax} \mid \mathtt{ebx} \mid \mathtt{ecx} \mid \mathtt{edx} \mid \mathtt{esi} \mid \mathtt{edi} \mid \mathtt{ebp} \mid \mathtt{esp}$
Memory addr.	ma	$::= \mathtt{ma\_d}\langle n\rangle \mid \mathtt{ma\_r}\langle r\rangle\langle n\rangle\langle ma_1\rangle$
Effective addr.	ea	$::= \mathtt{ea\_i}\langle n\rangle \mid \mathtt{ea\_r}\langle r\rangle \mid \mathtt{ea\_s}\langle ma\rangle \mid \mathtt{ea\_m}\langle ma\rangle$
Instructions	I	$::= \mathtt{mov}\langle n_{\mathsf{i}}\rangle\langle ea_1\rangle\langle ea_2\rangle \mid \mathtt{xchg}\langle n_{\mathsf{i}}\rangle\langle ea\rangle\langle r\rangle \mid \mathtt{lea}\langle n_{\mathsf{i}}\rangle\langle ea\rangle\langle r\rangle$
		$\mid \mathtt{push}\langle n_{\mathsf{i}}\rangle\langle ea\rangle \mid \mathtt{pop}\langle n_{\mathsf{i}}\rangle\langle ea\rangle$
		$\mid \mathtt{op1}\langle n_{\mathsf{i}}\rangle\langle \mathit{op1}\rangle\langle ea\rangle$
		$\mid \mathtt{op2}\langle n_{\mathsf{i}}\rangle\langle \mathit{op2}\rangle\langle ea_1\rangle\langle ea_2\rangle \mid \mathtt{op2n}\langle n_{\mathsf{i}}\rangle\langle \mathit{op2}\rangle\langle ea_1\rangle\langle ea_2\rangle$
		$\mid \mathtt{op3}\langle n_{\mathsf{i}}\rangle\langle \mathit{op3}_1\rangle\langle \mathit{op3}_2\rangle\langle ea\rangle\langle r_1\rangle\langle r_2\rangle$
		$\mid \mathtt{jmp}\langle n_{\mathsf{i}}\rangle\langle ea\rangle \mid \mathtt{j}\langle n_{\mathsf{i}}\rangle\langle \mathit{cop}\rangle\langle n\rangle$
		$\mid \mathtt{call}\langle n_{\mathsf{i}}\rangle\langle ea\rangle \mid \mathtt{ret}\langle n_{\mathsf{i}}\rangle$

Figure 4.1: Instruction Set

A *memory address ma* specifies how to compute the run-time address of a particular memory location. An *effective address ea* specifies the location of an operand of an instruction, which can be located in a register, in the instruction stream itself (*i.e.*, *immediate*), on the stack, or in memory. Each memory or stack addressing mode is composed of a *displacement* n_{d}, plus zero or more *index registers* r_{i_j}, each of which is multiplied by a corresponding *scale factor* n_{s_j}. The actual address n of a stack or memory effective address is computed from a register file $v_{\mathtt{mapg}}$ as follows:

$$n = v_{\mathtt{mapg}}(r_{\mathsf{i}_1}) \dot{\times} n_{\mathsf{s}_1} \ldots \dot{+} v_{\mathtt{mapg}}(r_{\mathsf{i}_k}) \dot{\times} n_{\mathsf{s}_k} \dot{+} n_{\mathsf{d}}$$

where $\dot{+}$ is addition modulo 2^{32} and $\dot{\times}$ is multiplication modulo 2^{32}.

An instruction I is a value of type `inst`, a program Φ is a value of type `prog`. Actual IA-32 instructions vary in size. Because the original opcodes of the instruction are lost by my instruction abstraction, each instruction includes a word n_{i} that specifies the size of the original instruction in bytes. Other components of an instruction specify the effective addresses of operands, as well as various operators and immediate operands. Operators are abstracted in order to treat a group of similar instructions (*e.g.*, `add`, `sub`) as essentially the same instruction in the operational semantics.

The effect of each instruction can be summarized informally as follows:

$\texttt{mov}\langle n_{\mathsf{i}}\rangle\langle ea_1\rangle\langle ea_2\rangle$	Move word from ea_1 to ea_2
$\texttt{xchg}\langle n_{\mathsf{i}}\rangle\langle ea\rangle\langle r\rangle$	Exchange word in ea with word in r
$\texttt{lea}\langle n_{\mathsf{i}}\rangle\langle ea\rangle\langle r\rangle$	Load address of ea into r
$\texttt{push}\langle n_{\mathsf{i}}\rangle\langle ea\rangle$	Push word from ea on stack
$\texttt{pop}\langle n_{\mathsf{i}}\rangle\langle ea\rangle$	Pop word from stack into ea
$\texttt{op1}\langle n_{\mathsf{i}}\rangle\langle \mathit{op1}\rangle\langle ea\rangle$	Apply $\mathit{op1}$ to ea
$\texttt{op2}\langle n_{\mathsf{i}}\rangle\langle \mathit{op2}\rangle\langle ea_1\rangle\langle ea_2\rangle$	Apply $\mathit{op2}$ to contents of ea_1 and ea_2
$\texttt{op2n}\langle n_{\mathsf{i}}\rangle\langle \mathit{op2}\rangle\langle ea_1\rangle\langle ea_2\rangle$	Set flags from $\texttt{op2}\langle n_{\mathsf{i}}\rangle\langle \mathit{op2}\rangle\langle ea_1\rangle\langle ea_2\rangle$
$\texttt{op3}\langle n_{\mathsf{i}}\rangle\langle \mathit{op3}_1\rangle\langle \mathit{op3}_2\rangle\langle ea\rangle\langle r_1\rangle\langle r_2\rangle$	Apply $\mathit{op3}_1$ and $\mathit{op3}_2$ in parallel
$\texttt{jmp}\langle n_{\mathsf{i}}\rangle\langle ea\rangle$	Jump to word from ea
$\texttt{j}\langle n_{\mathsf{i}}\rangle\langle \mathit{cop}\rangle\langle n\rangle$	Jump to offset n if cop holds
$\texttt{call}\langle n_{\mathsf{i}}\rangle\langle ea\rangle$	Call procedure at word from ea
$\texttt{ret}\langle n_{\mathsf{i}}\rangle$	Return from current procedure

Several additional instructions can be defined by abbreviation:

$$\begin{aligned}
\texttt{cmp}\langle n_{\mathsf{i}}\rangle\langle ea_1\rangle\langle ea_2\rangle &\stackrel{\mathsf{def}}{=} \texttt{op2n}\langle n_{\mathsf{i}}\rangle\langle \texttt{op2_sub}\rangle\langle ea_1\rangle\langle ea_2\rangle \\
\texttt{nop}\langle n_{\mathsf{i}}\rangle &\stackrel{\mathsf{def}}{=} \texttt{mov}\langle n_{\mathsf{i}}\rangle\langle \texttt{ea_r}\langle \texttt{edi}\rangle\rangle\langle \texttt{ea_r}\langle \texttt{edi}\rangle\rangle \\
\texttt{test}\langle n_{\mathsf{i}}\rangle\langle ea_1\rangle\langle ea_2\rangle &\stackrel{\mathsf{def}}{=} \texttt{op2n}\langle n_{\mathsf{i}}\rangle\langle \texttt{op2_and}\rangle\langle ea_1\rangle\langle ea_2\rangle
\end{aligned}$$

When checking an actual IA-32 program for safety, the object code is mapped onto the abstract machine model. Note that in order to simplify the instruction representation and operational semantics, the machine model includes many instructions that are not available on an actual IA-32 processor (for example, memory-to-memory moves). In practice, this simply means that although these instructions are defined correctly by the semantics, none will ever appear in an actual program. Also note that because a real IA-32 effective address does not distinguish stack from memory accesses, the object-code reader infers whether the stack or memory is referred to based on which segment register is selected in the object code.

As an illustrative example, the following procedure returns the factorial of the argument at stack offset 4 in register `eax`:

4	$\texttt{mov}\langle 5\rangle\langle \texttt{ea\_i}\langle 1\rangle\rangle\langle \texttt{ea\_r}\langle \texttt{eax}\rangle\rangle$	$\texttt{eax} \leftarrow 1$
9	$\texttt{mov}\langle 5\rangle\langle \texttt{ea\_i}\langle 1\rangle\rangle\langle \texttt{ea\_r}\langle \texttt{ebx}\rangle\rangle$	$\texttt{ebx} \leftarrow 1$
14	$\texttt{jmp}\langle 2\rangle\langle \texttt{ea\_i}\langle 20\rangle\rangle$	goto 20
16	$\texttt{op2}\langle 3\rangle\langle \texttt{op2\_imul}\rangle\langle \texttt{ea\_r}\langle \texttt{ebx}\rangle\rangle\langle \texttt{ea\_r}\langle \texttt{eax}\rangle\rangle$	$\texttt{eax} \leftarrow \texttt{ebx} \mathbin{\dot\times} \texttt{eax}$
19	$\texttt{op1}\langle 1\rangle\langle \texttt{op1\_inc}\rangle\langle \texttt{ea\_r}\langle \texttt{ebx}\rangle\rangle$	$\texttt{ebx} \leftarrow \texttt{ebx} \mathbin{\dot+} 1$
20	$\texttt{cmp}\langle 4\rangle\langle \texttt{ea\_s}\langle \texttt{ma\_r}\langle \texttt{esp}\rangle\langle 1\rangle\langle \texttt{ma\_d}\langle 4\rangle\rangle\rangle\rangle\langle \texttt{ea\_r}\langle \texttt{ebx}\rangle\rangle$	$\texttt{ebx} \mathbin{\dot-} \texttt{s}(\texttt{esp} \mathbin{\dot+} 4)$
24	$\texttt{j}\langle 2\rangle\langle \texttt{cop\_ng}\rangle\langle 2^{32}-10\rangle$	goto 16 if $\texttt{ebx} \leq \texttt{s}(\texttt{esp} \mathbin{\dot+} 4)$
26	$\texttt{ret}\langle 1\rangle$	return to caller

4.2 Syntax

I introduce the functions and relations in Table 4.1 and Table 4.2 to model various aspects of the operational semantics in temporal logic, including functions to construct and deconstruct machine-state tuples. State tuples are chiefly an abbreviation that simplifies notation. In addition, each value from Section 4.1 is assigned a constant or a constant function of the appropriate type in Table 4.3. Finally, Table 4.4 contains functions that support the definition of the operational semantics in Section 4.3.

I model a general-purpose register file as a single value of type `mapg`, mapping register tokens to words. The stack and memory are modeled by total functions from words to words (type `mapw`).

The constants $c_0^{\mathtt{wd}}$, $c_1^{\mathtt{wd}}$, ... denote words, but I usually write 0, 1, ... when the type `wd` can be inferred from context. `selw` (select from map) and `updw` (update map) are function constants. For example, $\mathtt{updw}(\mathtt{m}, 3, 4)$ denotes the same map as `m`, except that address 3 is mapped to 4. The constants `selg` (select register) and `updg` (update register) are similar except that they operate on register files. There are no operations yielding register tokens, just designated constants (c_r).

I associate a constant $c_{op1}^{\mathtt{op1}}$ with each unary operator *op1*, and likewise with each binary, ternary, and conditional operator. `app1` is a function constant that applies a unary operator, `app2` is a function constant that applies a binary operator, and `app3` is a function constant that applies a ternary operator. `not` is a function constant that complements a conditional operator (*e.g.*, $\mathtt{not}(\overline{\mathtt{cop\_z}}) = \overline{\mathtt{cop\_nz}}$).

My logic encompasses instructions and programs by means of constant functions and relations. For example, $f_{\mathtt{mov}}$ constructs a move instruction from two effective addresses, and `fetch` assigns a particular instruction to a particular program address. The logic is coupled to a particular untrusted program by means of the constant `pm`: $\mathcal{J}(\mathtt{pm})$ is the program in the "program memory."[2] Instruction expressions enable me to model the operational semantics of the abstract IA-32 machine directly in temporal logic and are also useful for specifying security policies.

Other functions from the figures are used less frequently—their roles will be clarified in the operational semantics in Section 4.3.

Identifiers for the special-purpose registers are chosen from parameters—the interpretation of these parameters is constrained by the machine model. *Reg* is the set of all register parameters (note that these are *not* register tokens). The following are register parameters:

$\mathtt{pc}^{\mathtt{wd}}$	The program counter
$\mathtt{f}^{\mathtt{wd}}$	The status flags
$\mathtt{g}^{\mathtt{mapg}}$	The register file
$\mathtt{s}^{\mathtt{mapw}}$	The stack
$\mathtt{m}^{\mathtt{mapw}}$	The contents of memory

Propositions can express properties of machine states. For example, $\mathtt{selg}(\mathtt{g}, \overline{\mathtt{eax}}) \neq 0$ asserts that general-purpose register `eax` is not zero at the current time.

[2]Because the program code is presumably ready to be run by the code consumer, `pm` is treated as a "stand in" to avoid replicating the program inside the proof. Alternatively, the program code could be stored in the proof and extracted by the code consumer after proof checking (*i.e.*, "code-carrying proof").

$\texttt{zfw}^{\texttt{wd}\to\texttt{wd}}$	Zero flag from a word result
$\texttt{sfw}^{\texttt{wd}\to\texttt{wd}}$	Sign flag from a word result
$\texttt{negw}^{\texttt{wd}\to\texttt{wd}}$	Word negation
$\texttt{addw}^{\texttt{wd}\times\texttt{wd}\to\texttt{wd}}$	Word addition
$\texttt{subw}^{\texttt{wd}\times\texttt{wd}\to\texttt{wd}}$	Word subtraction
$\texttt{mulw}^{\texttt{wd}\times\texttt{wd}\to\texttt{wd}}$	Word multiplication
$\texttt{divw}^{\texttt{wd}\times\texttt{wd}\times\texttt{wd}\to\texttt{wd}}$	Word division
$\texttt{remw}^{\texttt{wd}\times\texttt{wd}\times\texttt{wd}\to\texttt{wd}}$	Word remainder
$\texttt{andw}^{\texttt{wd}\times\texttt{wd}\to\texttt{wd}}$	Word bitwise "and"
$\texttt{orw}^{\texttt{wd}\times\texttt{wd}\to\texttt{wd}}$	Word bitwise "or"
$\texttt{xorw}^{\texttt{wd}\times\texttt{wd}\to\texttt{wd}}$	Word bitwise "exclusive or"
$\texttt{notw}^{\texttt{wd}\to\texttt{wd}}$	Word bitwise complement
$\texttt{ltw}^{\texttt{wd}\times\texttt{wd}\to o}$	Word signed "less than"
$\texttt{ltuw}^{\texttt{wd}\times\texttt{wd}\to o}$	Word unsigned "less than"
$\texttt{incw}^{\texttt{wd}\times\texttt{wd}\to o}$	Word bitwise inclusion
$\texttt{geqw}^{\texttt{wd}\times\texttt{wd}\to o}$	$= \neg\texttt{ltw}$
$\texttt{gequw}^{\texttt{wd}\times\texttt{wd}\to o}$	$= \neg\texttt{ltuw}$
$\texttt{nincw}^{\texttt{wd}\times\texttt{wd}\to o}$	$= \neg\texttt{incw}$

Table 4.1: Functions and Relations on Machine Words

$\texttt{selg}^{\texttt{mapg}\times\texttt{greg}\to\texttt{wd}}$	Select value of general-purpose register
$\texttt{updg}^{\texttt{mapg}\times\texttt{greg}\times\texttt{wd}\to\texttt{mapg}}$	Update value of general-purpose register
$\texttt{selw}^{\texttt{mapw}\times\texttt{wd}\to\texttt{wd}}$	Select value at address
$\texttt{updw}^{\texttt{mapw}\times\texttt{wd}\times\texttt{wd}\to\texttt{mapw}}$	Update value at address
$\texttt{joinw}^{\texttt{mapw}\times\texttt{wd}\times\texttt{mapw}\to\texttt{mapw}}$	Superimpose address spaces
$\texttt{s\_mk}^{\texttt{wd}\times\texttt{wd}\times\texttt{mapg}\times\texttt{mapw}\times\texttt{mapw}\to\texttt{state}}$	Construct machine state
$\texttt{s\_pc}^{\texttt{state}\to\texttt{wd}}$	Select program counter from machine state
$\texttt{s\_f}^{\texttt{state}\to\texttt{wd}}$	Select status flags from machine state
$\texttt{s\_g}^{\texttt{state}\to\texttt{mapg}}$	Select register file from machine state
$\texttt{s\_s}^{\texttt{state}\to\texttt{mapw}}$	Select stack from machine state
$\texttt{s\_m}^{\texttt{state}\to\texttt{mapw}}$	Select memory from machine state

Table 4.2: Functions and Relations for the Machine Model

$c_n^{\texttt{wd}}$	Machine word with value n
$c_{op1}^{\texttt{op1}}$	Unary operator with value *op1*
$c_{op2}^{\texttt{op2}}$	Binary operator with value *op2*
$c_{op3}^{\texttt{op3}}$	Ternary operator with value *op3*
$c_{cop}^{\texttt{cop}}$	Conditional operator with value *cop*
$c_r^{\texttt{greg}}$	Register token with value r
$f_{\texttt{ma\_d}}^{\texttt{wd}\to\texttt{ma}}$ $f_{\texttt{ma\_r}}^{\texttt{greg}\times\texttt{wd}\times\texttt{ma}\to\texttt{ma}}$	Memory-address constructors
$f_{\texttt{ea\_i}}^{\texttt{wd}\to\texttt{ea}}$ $f_{\texttt{ea\_r}}^{\texttt{greg}\to\texttt{ea}}$ $f_{\texttt{ea\_s}}^{\texttt{ma}\to\texttt{ea}}$ $f_{\texttt{ea\_m}}^{\texttt{ma}\to\texttt{ea}}$	Effective-address constructors
$f_{\texttt{mov}}^{\texttt{wd}\times\texttt{ea}\times\texttt{ea}\to\texttt{inst}}$ $f_{\texttt{xchg}}^{\texttt{wd}\times\texttt{ea}\times\texttt{greg}\to\texttt{inst}}$ $f_{\texttt{lea}}^{\texttt{wd}\times\texttt{ea}\times\texttt{greg}\to\texttt{inst}}$ $f_{\texttt{push}}^{\texttt{wd}\times\texttt{ea}\to\texttt{inst}}$ $f_{\texttt{pop}}^{\texttt{wd}\times\texttt{ea}\to\texttt{inst}}$ $f_{\texttt{op1}}^{\texttt{wd}\times\texttt{op1}\times\texttt{ea}\to\texttt{inst}}$ $f_{\texttt{op2}}^{\texttt{wd}\times\texttt{op2}\times\texttt{ea}\times\texttt{ea}\to\texttt{inst}}$ $f_{\texttt{op2n}}^{\texttt{wd}\times\texttt{op2}\times\texttt{ea}\times\texttt{ea}\to\texttt{inst}}$ $f_{\texttt{op3}}^{\texttt{wd}\times\texttt{op3}\times\texttt{op3}\times\texttt{ea}\times\texttt{greg}\times\texttt{greg}\to\texttt{inst}}$ $f_{\texttt{jmp}}^{\texttt{wd}\times\texttt{ea}\to\texttt{inst}}$ $f_{\texttt{j}}^{\texttt{wd}\times\texttt{cop}\times\texttt{wd}\to\texttt{inst}}$ $f_{\texttt{call}}^{\texttt{wd}\times\texttt{ea}\to\texttt{inst}}$ $f_{\texttt{ret}}^{\texttt{wd}\to\texttt{inst}}$	Instruction constructors

Table 4.3: Constants and Functions for the Instruction Set

$\mathtt{app1}^{\mathtt{op1} \times \mathtt{wd} \to \mathtt{wd}}$	Apply unary operator
$\mathtt{app2}^{\mathtt{op2} \times \mathtt{wd} \times \mathtt{wd} \to \mathtt{wd}}$	Apply binary operator
$\mathtt{app3}^{\mathtt{op3} \times \mathtt{wd} \times \mathtt{wd} \times \mathtt{wd} \to \mathtt{wd}}$	Apply ternary operator
$\mathtt{of1}^{\mathtt{op1} \times \mathtt{wd} \to \mathtt{wd}}$	Overflow flag from a unary operator
$\mathtt{of2}^{\mathtt{op2} \times \mathtt{wd} \times \mathtt{wd} \to \mathtt{wd}}$	Overflow flag from a binary operator
$\mathtt{of3}^{\mathtt{op3} \times \mathtt{wd} \times \mathtt{wd} \times \mathtt{wd} \to \mathtt{wd}}$	Overflow flag from a ternary operator
$\mathtt{cf1}^{\mathtt{op1} \times \mathtt{wd} \to \mathtt{wd}}$	Carry flag from a unary operator
$\mathtt{cf2}^{\mathtt{op2} \times \mathtt{wd} \times \mathtt{wd} \to \mathtt{wd}}$	Carry flag from a binary operator
$\mathtt{cf3}^{\mathtt{op3} \times \mathtt{wd} \times \mathtt{wd} \times \mathtt{wd} \to \mathtt{wd}}$	Carry flag from a ternary operator
$\mathtt{selzf}^{\mathtt{wd} \to \mathtt{wd}}$	Select zero flag
$\mathtt{selsf}^{\mathtt{wd} \to \mathtt{wd}}$	Select sign flag
$\mathtt{selof}^{\mathtt{wd} \to \mathtt{wd}}$	Select overflow flag
$\mathtt{selcf}^{\mathtt{wd} \to \mathtt{wd}}$	Select carry flag
$\mathtt{updf1}^{\mathtt{op1} \times \mathtt{wd} \times \mathtt{wd} \to \mathtt{wd}}$	Update status flags for a unary operator
$\mathtt{updf2}^{\mathtt{op2} \times \mathtt{wd} \times \mathtt{wd} \times \mathtt{wd} \to \mathtt{wd}}$	Update status flags for a binary operator
$\mathtt{updf3}^{\mathtt{op3} \times \mathtt{wd} \times \mathtt{wd} \times \mathtt{wd} \times \mathtt{wd} \to \mathtt{wd}}$	Update status flags for a ternary operator
$\mathtt{self}^{\mathtt{cop} \times \mathtt{wd} \to \mathtt{wd}}$	Select flag according to a conditional operator
$\mathtt{not}^{\mathtt{cop} \to \mathtt{cop}}$	Complement conditional operator
$\mathtt{ma\_addr}^{\mathtt{mapg} \times \mathtt{ma} \to \mathtt{wd}}$	Evaluate memory address for register file
$\mathtt{ea\_addr}^{\mathtt{state} \times \mathtt{ea} \to \mathtt{wd}}$	Evaluate effective address for state
$\mathtt{ea\_sel}^{\mathtt{state} \times \mathtt{ea} \to \mathtt{wd}}$	Select value at effective address in state
$\mathtt{ea\_updg}^{\mathtt{state} \times \mathtt{ea} \times \mathtt{wd} \to \mathtt{mapg}}$	Update register file for effective address and state
$\mathtt{ea\_upds}^{\mathtt{state} \times \mathtt{ea} \times \mathtt{wd} \to \mathtt{mapw}}$	Update stack for effective address and state
$\mathtt{ea\_updm}^{\mathtt{state} \times \mathtt{ea} \times \mathtt{wd} \to \mathtt{mapw}}$	Update memory for effective address and state
$\mathtt{nextpc}^{\mathtt{state} \times \mathtt{inst} \to \mathtt{wd}}$	Next-state program counter for instruction
$\mathtt{nextf}^{\mathtt{state} \times \mathtt{inst} \to \mathtt{wd}}$	Next-state status flags for instruction
$\mathtt{nextg}^{\mathtt{state} \times \mathtt{inst} \to \mathtt{mapg}}$	Next-state register file for instruction
$\mathtt{nexts}^{\mathtt{state} \times \mathtt{inst} \to \mathtt{mapw}}$	Next-state stack for instruction
$\mathtt{nextm}^{\mathtt{state} \times \mathtt{inst} \to \mathtt{mapw}}$	Next-state memory for instruction
$\mathtt{fetch}^{\mathtt{prog} \times \mathtt{wd} \times \mathtt{inst} \to o}$	Assign instruction to program address

Table 4.4: Functions for Evaluation

4.3 Operational Semantics

I now specify the operational semantics of the abstract IA-32 machine. This semantics is based directly on the *IA-32 Intel Architecture Software Developer's Manual* [Int01].

4.3.1 Execution Sets

My operational semantics defines a set of executions for each program.

A *state* s maps each register parameter to a value of its type.[3] A state is simply a snapshot of the machine at a particular time. An *execution* σ is an infinite sequence of states representing the trace of a computation. Finite executions are represented by repeating the final state infinitely.

An environment can be transformed into an execution (see Section 3.2) by sampling each register at each time. $\phi|_{Reg}$ is the execution for environment ϕ:

$$\phi|_{Reg} = \sigma \text{ such that } \sigma_j = a \mapsto \phi(a)(j) \text{ for all times } j \text{ and } a \in Reg$$

$\phi|_{Reg}$ is called the *erasure* of ϕ (*i.e.*, non-register parameters are "erased"). An execution σ *satisfies* a proposition p at a time j ($\sigma, j \vDash^{\mathcal{J}} p$) if all environments that erase to σ satisfy p at j:

$$\sigma, j \vDash^{\mathcal{J}} p \text{ iff } \phi, \eta[b \mapsto j] \vDash^{\mathcal{J}} p \,@\, b \text{ for all } \phi \text{ such that } \phi|_{Reg} = \sigma \text{ and some } \eta, b$$

This definition effectively treats the non-register parameters in the proposition as "history registers," or uninterpreted components of the state of a security automaton [Sch99]. When $\mathcal{J}$ is understood from context, the *execution set* Σ_p of a proposition p is the set of executions that satisfy it at time zero:

$$\Sigma_p = \{\sigma \mid \sigma, 0 \vDash^{\mathcal{J}} p\}$$

Temporal logic can now be treated as a formal security-policy language in the sense of Bernard and Lee [BL01]. Given a security policy p, an execution σ does not violate security if and only if $\sigma \in \Sigma_p$. I discuss security policies further in Chapter 5.

The operational semantics is based on a transition relation between states for any given program. $\Phi \triangleright s \to s'$ asserts that there is a valid transition from state s

[3] Note that a state s is distinct from a state tuple (a value of type `state`). A state tuple simply packages together five relevant components of a state.

to state s' when executing program Φ:

$$\begin{array}{l}
\Phi \rhd s \to s[\mathtt{pc} \mapsto n'_{\mathtt{pc}}][\mathtt{f} \mapsto n'_{\mathtt{f}}][\mathtt{g} \mapsto v'_{\mathtt{g}}][\mathtt{s} \mapsto v'_{\mathtt{s}}][\mathtt{m} \mapsto v'_{\mathtt{m}}] \\
\quad \text{if } s(\mathtt{pc}) \in \operatorname{dom}\Phi \\
\quad \text{where } \begin{array}[t]{ll}
n'_{\mathtt{pc}} & = \underline{\mathtt{nextpc}}(v, I) \\
n'_{\mathtt{f}} & = \underline{\mathtt{nextf}}(v, I) \\
v'_{\mathtt{g}} & = \underline{\mathtt{nextg}}(v, I) \\
v'_{\mathtt{s}} & = \underline{\mathtt{nexts}}(v, I) \\
v'_{\mathtt{m}} & = \underline{\mathtt{nextm}}(v, I) \\
v & = \langle s(\mathtt{pc}), s(\mathtt{f}), s(\mathtt{g}), s(\mathtt{s}), s(\mathtt{m})\rangle \\
I & = \Phi(s(\mathtt{pc}))
\end{array}
\end{array}$$

$$\begin{array}{l}
\Phi \rhd s \to s \\
\quad \text{if } s(\mathtt{pc}) \notin \operatorname{dom}\Phi
\end{array}$$

Note that certain instructions can generate run-time exceptions (*e.g.*, divide by zero) that are handled transparently by the run-time system—in the interest of simplicity, I do not attempt to model such exceptions in the operational semantics. The execution set of a program (*i.e.*, its possible behavior) comprises all executions with valid transitions:

$$\Sigma_\Phi = \{\sigma \mid \Phi \rhd \sigma_j \to \sigma_{j+1} \text{ for all } j \geq 0\}$$

4.3.2 Types

The semantic model for each of the types introduced in this section can be summarized as follows:

$$\begin{array}{ll}
Val^{\mathtt{wd}} & = \{n \in \mathbb{N} \mid n < 2^{32}\} \\
Val^{\mathtt{op1}} & = \{op1\}_{op1} \\
Val^{\mathtt{op2}} & = \{op2\}_{op2} \\
Val^{\mathtt{op3}} & = \{op3\}_{op3} \\
Val^{\mathtt{cop}} & = \{cop\}_{cop} \\
Val^{\mathtt{greg}} & = \{r\}_{r} \\
Val^{\mathtt{mapg}} & = Val^{\mathtt{greg}} \to Val^{\mathtt{wd}} \\
Val^{\mathtt{mapw}} & = Val^{\mathtt{wd}} \to Val^{\mathtt{wd}} \\
Val^{\mathtt{state}} & = Val^{\mathtt{wd}} \times Val^{\mathtt{wd}} \times Val^{\mathtt{mapg}} \times Val^{\mathtt{mapw}} \times Val^{\mathtt{mapw}} \\
Val^{\mathtt{ma}} & = \{ma\}_{ma} \\
Val^{\mathtt{ea}} & = \{ea\}_{ea} \\
Val^{\mathtt{inst}} & = \{I\}_{I} \\
Val^{\mathtt{prog}} & = Val^{\mathtt{wd}} \rightharpoonup Val^{\mathtt{inst}}
\end{array}$$

I will use various arithmetic and logical operations on machine words that are defined mathematically as follows:

$$
\begin{aligned}
\ulcorner n \urcorner &= \begin{cases} n - 2^{32} & \text{if } n \geq 2^{31} \\ n & \text{otherwise} \end{cases} \\
\ulcorner\!\ulcorner n_1, n_2 \urcorner\!\urcorner &= \begin{cases} n' - 2^{64} & \text{if } n_1 \geq 2^{31} \\ n' & \text{otherwise} \end{cases} \\
& \quad \text{where } n' = n_1 \times 2^{32} + n_2 \\
n_1 \dot{+} n_2 &= (n_1 + n_2) \bmod 2^{32} \\
n_1 \dot{-} n_2 &= (n_1 - n_2) \bmod 2^{32} \\
n_1 \dot{\times} n_2 &= (n_1 \times n_2) \bmod 2^{32} \\
n_1 \dot{\wedge} n_2 &= \textstyle\sum_{j=0}^{31} 2^j \times (\lfloor n_1/2^j \rfloor \bmod 2) \times (\lfloor n_2/2^j \rfloor \bmod 2) \\
n_1 \dot{\vee} n_2 &= \dot{\neg}(\dot{\neg} n_1 \dot{\wedge} \dot{\neg} n_2) \\
n_1 \dot{\underline{\vee}} n_2 &= (n_1 \dot{\vee} n_2) \dot{\wedge} (\dot{\neg} n_1 \dot{\vee} \dot{\neg} n_2) \\
\dot{\neg} n &= 2^{32} - 1 - n
\end{aligned}
$$

$\ulcorner \cdot \urcorner$ converts a word to an integer (*i.e.* a member of $\mathbb{Z}$) according to the two's complement convention. $\dot{\wedge}$, *etc.* are bitwise operations on words.

4.3.3 IA-32 Functions

I provide interpretations for the functions (*e.g.*, $\underline{\texttt{nextpc}}$) used to define the transition relation in this subsection. Recall that I use underlining to express the interpretation of a constant, so, for example, $\underline{\texttt{nextpc}}$ denotes the mathematical function that is the interpretation of the constant `nextpc` according to the interpretation function $\mathcal{J}$. Specifying the mathematical interpretation of the various function constants listed in Section 4.2 amounts to constructing a mathematical model for the operation of the abstract processor. These constants will appear later in the formal specification of the abstract processor as provided by the code consumer.

I proceed through the functions that comprise the semantic model in "top-down" order, so it may be advisable to skim the rest of this section prior to a detailed reading.

First, I define functions that compute the individual components of the next-time state tuple according to the current-time state tuple. I use the following abbreviation in these definitions:

$$(v)_{ea} \stackrel{\mathsf{def}}{=} \underline{\texttt{ea\_sel}}(v, ea)$$

$\underline{\texttt{nextpc}}(v, I)$ determines the next-state program counter according to the current state tuple v and current instruction I:

I	$\underline{\mathtt{nextpc}}(v, I)$, where $v = \langle n_{\mathtt{pc}}, n_{\mathtt{f}}, v_{\mathtt{g}}, v_{\mathtt{s}}, v_{\mathtt{m}} \rangle$
$\mathtt{mov}\langle n_{\mathsf{i}} \rangle \langle ea_1 \rangle \langle ea_2 \rangle$	$n_{\mathtt{pc}} \dot{+} n_{\mathsf{i}}$
$\mathtt{xchg}\langle n_{\mathsf{i}} \rangle \langle ea \rangle \langle r \rangle$	$n_{\mathtt{pc}} \dot{+} n_{\mathsf{i}}$
$\mathtt{lea}\langle n_{\mathsf{i}} \rangle \langle ea \rangle \langle r \rangle$	$n_{\mathtt{pc}} \dot{+} n_{\mathsf{i}}$
$\mathtt{push}\langle n_{\mathsf{i}} \rangle \langle ea \rangle$	$n_{\mathtt{pc}} \dot{+} n_{\mathsf{i}}$
$\mathtt{pop}\langle n_{\mathsf{i}} \rangle \langle ea \rangle$	$n_{\mathtt{pc}} \dot{+} n_{\mathsf{i}}$
$\mathtt{op1}\langle n_{\mathsf{i}} \rangle \langle op1 \rangle \langle ea \rangle$	$n_{\mathtt{pc}} \dot{+} n_{\mathsf{i}}$
$\mathtt{op2}\langle n_{\mathsf{i}} \rangle \langle op2 \rangle \langle ea_1 \rangle \langle ea_2 \rangle$	$n_{\mathtt{pc}} \dot{+} n_{\mathsf{i}}$
$\mathtt{op2n}\langle n_{\mathsf{i}} \rangle \langle op2 \rangle \langle ea_1 \rangle \langle ea_2 \rangle$	$n_{\mathtt{pc}} \dot{+} n_{\mathsf{i}}$
$\mathtt{op3}\langle n_{\mathsf{i}} \rangle \langle op3_1 \rangle \langle op3_2 \rangle \langle ea \rangle \langle r_1 \rangle \langle r_2 \rangle$	$n_{\mathtt{pc}} \dot{+} n_{\mathsf{i}}$
$\mathtt{jmp}\langle n_{\mathsf{i}} \rangle \langle ea \rangle$	$(v)_{ea}$
$\mathtt{j}\langle n_{\mathsf{i}} \rangle \langle cop \rangle \langle n \rangle$	$n_{\mathtt{pc}} \dot{+} n_{\mathsf{i}} \dot{+} n \dot{\times} \underline{\mathtt{self}}(cop, n_{\mathtt{f}})$
$\mathtt{call}\langle n_{\mathsf{i}} \rangle \langle ea \rangle$	$(v)_{ea}$
$\mathtt{ret}\langle n_{\mathsf{i}} \rangle$	$v_{\mathtt{s}}(v_{\mathtt{g}}(\mathtt{esp}))$

$\underline{\mathtt{nextf}}(v, I)$ determines the next-state status flags according to the current state tuple v and current instruction I:

I	$\underline{\mathtt{nextf}}(v, I)$, where $v = \langle n_{\mathtt{pc}}, n_{\mathtt{f}}, v_{\mathtt{g}}, v_{\mathtt{s}}, v_{\mathtt{m}} \rangle$
$\mathtt{mov}\langle n_{\mathsf{i}} \rangle \langle ea_1 \rangle \langle ea_2 \rangle$	$n_{\mathtt{f}}$
$\mathtt{xchg}\langle n_{\mathsf{i}} \rangle \langle ea \rangle \langle r \rangle$	$n_{\mathtt{f}}$
$\mathtt{lea}\langle n_{\mathsf{i}} \rangle \langle ea \rangle \langle r \rangle$	$n_{\mathtt{f}}$
$\mathtt{push}\langle n_{\mathsf{i}} \rangle \langle ea \rangle$	$n_{\mathtt{f}}$
$\mathtt{pop}\langle n_{\mathsf{i}} \rangle \langle ea \rangle$	$n_{\mathtt{f}}$
$\mathtt{op1}\langle n_{\mathsf{i}} \rangle \langle op1 \rangle \langle ea \rangle$	$\underline{\mathtt{updf1}}(op1, n_{\mathtt{f}}, (v)_{ea})$
$\mathtt{op2}\langle n_{\mathsf{i}} \rangle \langle op2 \rangle \langle ea_1 \rangle \langle ea_2 \rangle$	$\underline{\mathtt{updf2}}(op2, n_{\mathtt{f}}, (v)_{ea_2}, (v)_{ea_1})$
$\mathtt{op2n}\langle n_{\mathsf{i}} \rangle \langle op2 \rangle \langle ea_1 \rangle \langle ea_2 \rangle$	$\underline{\mathtt{updf2}}(op2, n_{\mathtt{f}}, (v)_{ea_2}, (v)_{ea_1})$
$\mathtt{op3}\langle n_{\mathsf{i}} \rangle \langle op3_1 \rangle \langle op3_2 \rangle \langle ea \rangle \langle r_1 \rangle \langle r_2 \rangle$	$\underline{\mathtt{updf3}}(op3_1, n_{\mathtt{f}}, v_{\mathtt{g}}(r_1), v_{\mathtt{g}}(r_2), (v)_{ea})$
$\mathtt{jmp}\langle n_{\mathsf{i}} \rangle \langle ea \rangle$	$n_{\mathtt{f}}$
$\mathtt{j}\langle n_{\mathsf{i}} \rangle \langle cop \rangle \langle n \rangle$	$n_{\mathtt{f}}$
$\mathtt{call}\langle n_{\mathsf{i}} \rangle \langle ea \rangle$	$n_{\mathtt{f}}$
$\mathtt{ret}\langle n_{\mathsf{i}} \rangle$	$n_{\mathtt{f}}$

$\underline{\mathtt{nextg}}(v, I)$ determines the next-state register file according to the current state tuple v and current instruction I:

I	$\underline{\mathtt{nextg}}(v, I)$, where $v = \langle n_{\mathtt{pc}}, n_{\mathtt{f}}, v_{\mathtt{g}}, v_{\mathtt{s}}, v_{\mathtt{m}} \rangle$
$\mathtt{mov}\langle n_{\mathsf{i}} \rangle \langle ea_1 \rangle \langle ea_2 \rangle$	$\underline{\mathtt{ea\_updg}}(v, ea_2, (v)_{ea_1})$
$\mathtt{xchg}\langle n_{\mathsf{i}} \rangle \langle ea \rangle \langle r \rangle$	$\underline{\mathtt{ea\_updg}}(v, ea, v_{\mathtt{g}}(r))[r \mapsto (v)_{ea}]$
$\mathtt{lea}\langle n_{\mathsf{i}} \rangle \langle ea \rangle \langle r \rangle$	$v_{\mathtt{g}}[r \mapsto \underline{\mathtt{ea\_addr}}(v, ea)]$
$\mathtt{push}\langle n_{\mathsf{i}} \rangle \langle ea \rangle$	$v_{\mathtt{g}}[\mathtt{esp} \mapsto v_{\mathtt{g}}(\mathtt{esp}) \dot{-} 4]$
$\mathtt{pop}\langle n_{\mathsf{i}} \rangle \langle ea \rangle$	$\underline{\mathtt{ea\_updg}}(v, ea, v_{\mathtt{s}}(v_{\mathtt{g}}(\mathtt{esp})))[\mathtt{esp} \mapsto v_{\mathtt{g}}(\mathtt{esp}) \dot{+} 4]$
$\mathtt{op1}\langle n_{\mathsf{i}} \rangle \langle op1 \rangle \langle ea \rangle$	$\underline{\mathtt{ea\_updg}}(v, ea, \underline{\mathtt{app1}}(op1, (v)_{ea}))$
$\mathtt{op2}\langle n_{\mathsf{i}} \rangle \langle op2 \rangle \langle ea_1 \rangle \langle ea_2 \rangle$	$\underline{\mathtt{ea\_updg}}(v, ea_2, \underline{\mathtt{app2}}(op2, (v)_{ea_2}, (v)_{ea_1}))$
$\mathtt{op2n}\langle n_{\mathsf{i}} \rangle \langle op2 \rangle \langle ea_1 \rangle \langle ea_2 \rangle$	$v_{\mathtt{g}}$
$\mathtt{op3}\langle n_{\mathsf{i}} \rangle \langle op3_1 \rangle \langle op3_2 \rangle \langle ea \rangle \langle r_1 \rangle \langle r_2 \rangle$	$v_{\mathtt{g}}[r_1 \mapsto \underline{\mathtt{app3}}(op3_1, v_{\mathtt{g}}(r_1), v_{\mathtt{g}}(r_2), (v)_{ea})]$ $[r_2 \mapsto \underline{\mathtt{app3}}(op3_2, v_{\mathtt{g}}(r_1), v_{\mathtt{g}}(r_2), (v)_{ea})]$
$\mathtt{jmp}\langle n_{\mathsf{i}} \rangle \langle ea \rangle$	$v_{\mathtt{g}}$
$\mathtt{j}\langle n_{\mathsf{i}} \rangle \langle cop \rangle \langle n \rangle$	$v_{\mathtt{g}}$
$\mathtt{call}\langle n_{\mathsf{i}} \rangle \langle ea \rangle$	$v_{\mathtt{g}}[\mathtt{esp} \mapsto v_{\mathtt{g}}(\mathtt{esp}) \dot{-} 4]$
$\mathtt{ret}\langle n_{\mathsf{i}} \rangle$	$v_{\mathtt{g}}[\mathtt{esp} \mapsto v_{\mathtt{g}}(\mathtt{esp}) \dot{+} 4]$

$\underline{\mathtt{nexts}}(v, I)$ determines the next-state stack according to the current state tuple v and current instruction I:

I	$\underline{\mathtt{nexts}}(v, I)$, where $v = \langle n_{\mathtt{pc}}, n_{\mathtt{f}}, v_{\mathtt{g}}, v_{\mathtt{s}}, v_{\mathtt{m}} \rangle$
$\mathtt{mov}\langle n_{\mathsf{i}} \rangle \langle ea_1 \rangle \langle ea_2 \rangle$	$\underline{\mathtt{ea\_upds}}(v, ea_2, (v)_{ea_1})$
$\mathtt{xchg}\langle n_{\mathsf{i}} \rangle \langle ea \rangle \langle r \rangle$	$\underline{\mathtt{ea\_upds}}(v, ea, v_{\mathtt{g}}(r))$
$\mathtt{lea}\langle n_{\mathsf{i}} \rangle \langle ea \rangle \langle r \rangle$	$v_{\mathtt{s}}$
$\mathtt{push}\langle n_{\mathsf{i}} \rangle \langle ea \rangle$	$v_{\mathtt{s}}[(v_{\mathtt{g}}(\mathtt{esp}) \dot{-} 4) \mapsto (v)_{ea}]$
$\mathtt{pop}\langle n_{\mathsf{i}} \rangle \langle ea \rangle$	$\underline{\mathtt{ea\_upds}}(v, ea, v_{\mathtt{s}}(v_{\mathtt{g}}(\mathtt{esp})))$
$\mathtt{op1}\langle n_{\mathsf{i}} \rangle \langle op1 \rangle \langle ea \rangle$	$\underline{\mathtt{ea\_upds}}(v, ea, \underline{\mathtt{app1}}(op1, (v)_{ea}))$
$\mathtt{op2}\langle n_{\mathsf{i}} \rangle \langle op2 \rangle \langle ea_1 \rangle \langle ea_2 \rangle$	$\underline{\mathtt{ea\_upds}}(v, ea_2, \underline{\mathtt{app2}}(op2, (v)_{ea_2}, (v)_{ea_1}))$
$\mathtt{op2n}\langle n_{\mathsf{i}} \rangle \langle op2 \rangle \langle ea_1 \rangle \langle ea_2 \rangle$	$v_{\mathtt{s}}$
$\mathtt{op3}\langle n_{\mathsf{i}} \rangle \langle op3_1 \rangle \langle op3_2 \rangle \langle ea \rangle \langle r_1 \rangle \langle r_2 \rangle$	$v_{\mathtt{s}}$
$\mathtt{jmp}\langle n_{\mathsf{i}} \rangle \langle ea \rangle$	$v_{\mathtt{s}}$
$\mathtt{j}\langle n_{\mathsf{i}} \rangle \langle cop \rangle \langle n \rangle$	$v_{\mathtt{s}}$
$\mathtt{call}\langle n_{\mathsf{i}} \rangle \langle ea \rangle$	$v_{\mathtt{s}}[(v_{\mathtt{g}}(\mathtt{esp}) \dot{-} 4) \mapsto n_{\mathtt{pc}} \dot{+} n_{\mathsf{i}}]$
$\mathtt{ret}\langle n_{\mathsf{i}} \rangle$	$v_{\mathtt{s}}$

$\underline{\mathtt{nextm}}(v, I)$ determines the next-state memory according to the current state tuple v and current instruction I:

I	$\underline{\texttt{nextm}}(v, I)$, where $v = \langle n_{\texttt{pc}}, n_{\texttt{f}}, v_{\texttt{g}}, v_{\texttt{s}}, v_{\texttt{m}}\rangle$
$\texttt{mov}\langle n_{\mathsf{i}}\rangle\langle ea_1\rangle\langle ea_2\rangle$	$\underline{\texttt{ea\_updm}}(v, ea_2, (v)_{ea_1})$
$\texttt{xchg}\langle n_{\mathsf{i}}\rangle\langle ea\rangle\langle r\rangle$	$\underline{\texttt{ea\_updm}}(v, ea, v_{\texttt{g}}(r))$
$\texttt{lea}\langle n_{\mathsf{i}}\rangle\langle ea\rangle\langle r\rangle$	$v_{\texttt{m}}$
$\texttt{push}\langle n_{\mathsf{i}}\rangle\langle ea\rangle$	$v_{\texttt{m}}$
$\texttt{pop}\langle n_{\mathsf{i}}\rangle\langle ea\rangle$	$\underline{\texttt{ea\_updm}}(v, ea, v_{\texttt{s}}(v_{\texttt{g}}(\texttt{esp})))$
$\texttt{op1}\langle n_{\mathsf{i}}\rangle\langle \mathit{op1}\rangle\langle ea\rangle$	$\underline{\texttt{ea\_updm}}(v, ea, \underline{\texttt{app1}}(\mathit{op1}, (v)_{ea}))$
$\texttt{op2}\langle n_{\mathsf{i}}\rangle\langle \mathit{op2}\rangle\langle ea_1\rangle\langle ea_2\rangle$	$\underline{\texttt{ea\_updm}}(v, ea_2, \underline{\texttt{app2}}(\mathit{op2}, (v)_{ea_2}, (v)_{ea_1}))$
$\texttt{op2n}\langle n_{\mathsf{i}}\rangle\langle \mathit{op2}\rangle\langle ea_1\rangle\langle ea_2\rangle$	$v_{\texttt{m}}$
$\texttt{op3}\langle n_{\mathsf{i}}\rangle\langle \mathit{op3}_1\rangle\langle \mathit{op3}_2\rangle\langle ea\rangle\langle r_1\rangle\langle r_2\rangle$	$v_{\texttt{m}}$
$\texttt{jmp}\langle n_{\mathsf{i}}\rangle\langle ea\rangle$	$v_{\texttt{m}}$
$\texttt{j}\langle n_{\mathsf{i}}\rangle\langle cop\rangle\langle n\rangle$	$v_{\texttt{m}}$
$\texttt{call}\langle n_{\mathsf{i}}\rangle\langle ea\rangle$	$v_{\texttt{m}}$
$\texttt{ret}\langle n_{\mathsf{i}}\rangle$	$v_{\texttt{m}}$

Next, I define functions that evaluate memory addresses and effective addresses into actual addresses, and load and store values based on these addresses. The function $\underline{\texttt{ma\_addr}}(v_{\texttt{g}}, ma)$ evaluates a given memory address ma to an actual address with respect to a given register file $v_{\texttt{g}}$:

$$\underline{\texttt{ma\_addr}}(v_{\texttt{g}}, ma) = \begin{cases} n_{\mathsf{d}} & \text{if } ma = \texttt{ma\_d}\langle n_{\mathsf{d}}\rangle \\ v_{\texttt{g}}(r_{\mathsf{i}}) \mathbin{\dot\times} n_{\mathsf{s}} \mathbin{\dot+} \underline{\texttt{ma\_addr}}(v_{\texttt{g}}, ma') & \text{if } ma = \texttt{ma\_r}\langle r_{\mathsf{i}}\rangle\langle n_{\mathsf{s}}\rangle\langle ma'\rangle \end{cases}$$

$\underline{\texttt{ea\_addr}}(v, ea)$ evaluates a given effective address ea to an actual address with respect to a given state tuple v:

$$\underline{\texttt{ea\_addr}}(\langle n_{\texttt{pc}}, n_{\texttt{f}}, v_{\texttt{g}}, v_{\texttt{s}}, v_{\texttt{m}}\rangle, ea) = \begin{cases} n' \in \mathit{Val}^{\texttt{wd}} & \text{if } ea = \texttt{ea\_i}\langle n\rangle \\ n' \in \mathit{Val}^{\texttt{wd}} & \text{if } ea = \texttt{ea\_r}\langle r\rangle \\ \underline{\texttt{ma\_addr}}(v_{\texttt{g}}, ma) & \text{if } ea = \texttt{ea\_s}\langle ma\rangle \\ \underline{\texttt{ma\_addr}}(v_{\texttt{g}}, ma) & \text{if } ea = \texttt{ea\_m}\langle ma\rangle \end{cases}$$

$\underline{\texttt{ea\_sel}}(v, ea)$ (often abbreviated $(v)_{ea}$) is the value stored at a given effective address ea according to a given state tuple v:

$$\underline{\texttt{ea\_sel}}(\langle n_{\texttt{pc}}, n_{\texttt{f}}, v_{\texttt{g}}, v_{\texttt{s}}, v_{\texttt{m}}\rangle, ea) = \begin{cases} n & \text{if } ea = \texttt{ea\_i}\langle n\rangle \\ v_{\texttt{g}}(r) & \text{if } ea = \texttt{ea\_r}\langle r\rangle \\ v_{\texttt{s}}(\underline{\texttt{ma\_addr}}(v_{\texttt{g}}, ma)) & \text{if } ea = \texttt{ea\_s}\langle ma\rangle \\ v_{\texttt{m}}(\underline{\texttt{ma\_addr}}(v_{\texttt{g}}, ma)) & \text{if } ea = \texttt{ea\_m}\langle ma\rangle \end{cases}$$

$\underline{\texttt{ea\_updg}}(v, ea, n)$ is the new register file that results from storing a given value

n at a given effective address ea with respect to a given state tuple v:

$$\underline{\texttt{ea\_updg}}(\langle n_{\texttt{pc}}, n_{\texttt{f}}, v_{\texttt{g}}, v_{\texttt{s}}, v_{\texttt{m}}\rangle, ea, n) = \begin{cases} n' \in \mathit{Val}^{\texttt{wd}} & \text{if } ea = \texttt{ea\_i}\langle n_1\rangle \\ v_{\texttt{g}}[r \mapsto n] & \text{if } ea = \texttt{ea\_r}\langle r\rangle \\ v_{\texttt{g}} & \text{if } ea = \texttt{ea\_s}\langle ma\rangle \\ v_{\texttt{g}} & \text{if } ea = \texttt{ea\_m}\langle ma\rangle \end{cases}$$

$\underline{\texttt{ea\_upds}}(v, ea, n)$ is the new stack that results from storing a given value n at a given effective address ea with respect to a given state tuple v:

$$\underline{\texttt{ea\_upds}}(\langle n_{\texttt{pc}}, n_{\texttt{f}}, v_{\texttt{g}}, v_{\texttt{s}}, v_{\texttt{m}}\rangle, ea, n) = \begin{cases} n' \in \mathit{Val}^{\texttt{wd}} & \text{if } ea = \texttt{ea\_i}\langle n_1\rangle \\ v_{\texttt{s}} & \text{if } ea = \texttt{ea\_r}\langle r\rangle \\ v_{\texttt{s}}[\underline{\texttt{ma\_addr}}(v_{\texttt{g}}, ma) \mapsto n] & \text{if } ea = \texttt{ea\_s}\langle ma\rangle \\ v_{\texttt{s}} & \text{if } ea = \texttt{ea\_m}\langle ma\rangle \end{cases}$$

$\underline{\texttt{ea\_updm}}(v, ea, n)$ is the new memory that results from storing a given value n at a given effective address ea with respect to a given state tuple v:

$$\underline{\texttt{ea\_updm}}(\langle n_{\texttt{pc}}, n_{\texttt{f}}, v_{\texttt{g}}, v_{\texttt{s}}, v_{\texttt{m}}\rangle, ea, n) = \begin{cases} n' \in \mathit{Val}^{\texttt{wd}} & \text{if } ea = \texttt{ea\_i}\langle n_1\rangle \\ v_{\texttt{m}} & \text{if } ea = \texttt{ea\_r}\langle r\rangle \\ v_{\texttt{m}} & \text{if } ea = \texttt{ea\_s}\langle ma\rangle \\ v_{\texttt{m}}[\underline{\texttt{ma\_addr}}(v_{\texttt{g}}, ma) \mapsto n] & \text{if } ea = \texttt{ea\_m}\langle ma\rangle \end{cases}$$

$\underline{\texttt{app1}}(\mathit{op1}, n)$ applies unary operator $\mathit{op1}$ to argument n:

$$\begin{aligned} \underline{\texttt{app1}}(\texttt{op1\_inc}, n) &= n \mathbin{\dot{+}} 1 \\ \underline{\texttt{app1}}(\texttt{op1\_dec}, n) &= n \mathbin{\dot{-}} 1 \\ \underline{\texttt{app1}}(\texttt{op1\_neg}, n) &= 0 \mathbin{\dot{-}} n \\ \underline{\texttt{app1}}(\texttt{op1\_not}, n) &= \dot{\neg} n \end{aligned}$$

$\underline{\texttt{app2}}(\mathit{op2}, n_1, n_2)$ applies binary operator $\mathit{op2}$ to arguments n_1 and n_2:

$$\begin{aligned} \underline{\texttt{app2}}(\texttt{op2\_add}, n_1, n_2) &= n_1 \mathbin{\dot{+}} n_2 \\ \underline{\texttt{app2}}(\texttt{op2\_sub}, n_1, n_2) &= n_1 \mathbin{\dot{-}} n_2 \\ \underline{\texttt{app2}}(\texttt{op2\_imul}, n_1, n_2) &= n_1 \mathbin{\dot{\times}} n_2 \\ \underline{\texttt{app2}}(\texttt{op2\_and}, n_1, n_2) &= n_1 \mathbin{\dot{\wedge}} n_2 \\ \underline{\texttt{app2}}(\texttt{op2\_or}, n_1, n_2) &= n_1 \mathbin{\dot{\vee}} n_2 \\ \underline{\texttt{app2}}(\texttt{op2\_xor}, n_1, n_2) &= n_1 \mathbin{\dot{\veebar}} n_2 \\ \underline{\texttt{app2}}(\texttt{op2\_sf2}, n_1, n_2) &= (2^{32} - 1) \times \underline{\texttt{sfw}}(n_2) \end{aligned}$$

$\underline{\texttt{app3}}(\mathit{op3}, n_1, n_2, n_3)$ applies ternary operator $\mathit{op3}$ to arguments n_1, n_2, and n_3:

$$\begin{aligned} \underline{\texttt{app3}}(\texttt{op3\_idiv}, n_1, n_2, n_3) &= \underline{\texttt{divw}}(n_1, n_2, n_3) \\ \underline{\texttt{app3}}(\texttt{op3\_irem}, n_1, n_2, n_3) &= \underline{\texttt{remw}}(n_1, n_2, n_3) \end{aligned}$$

$$
\begin{aligned}
&\underline{\texttt{of1}}(\texttt{op1\_inc}, n) &&= \underline{\texttt{of2}}(\texttt{op2\_add}, n, 1) \\
&\underline{\texttt{of1}}(\texttt{op1\_dec}, n) &&= \underline{\texttt{of2}}(\texttt{op2\_sub}, n, 1) \\
&\underline{\texttt{of1}}(\texttt{op1\_neg}, n) &&= \underline{\texttt{of2}}(\texttt{op2\_sub}, 0, n) \\
&\underline{\texttt{of1}}(\texttt{op1\_not}, n) &&= 0 \\[1ex]
&\underline{\texttt{of2}}(\texttt{op2\_add}, n_1, n_2) &&= (\underline{\texttt{sfw}}(n_1 \mathbin{\dot{\veebar}} n_2) \mathbin{\dot{\veebar}} 1) \mathbin{\dot{\wedge}} \underline{\texttt{sfw}}((n_1 \mathbin{\dot{+}} n_2) \mathbin{\dot{\veebar}} n_1) \\
&\underline{\texttt{of2}}(\texttt{op2\_sub}, n_1, n_2) &&= \underline{\texttt{sfw}}(n_1 \mathbin{\dot{\veebar}} n_2) \mathbin{\dot{\wedge}} \underline{\texttt{sfw}}((n_1 \mathbin{\dot{-}} n_2) \mathbin{\dot{\veebar}} n_1) \\[1ex]
&\underline{\texttt{of2}}(\texttt{op2\_imul}, n_1, n_2) &&= \begin{cases} 0 & \text{if } -(2^{31}) \leq \ulcorner n_1 \urcorner \times \ulcorner n_2 \urcorner \leq 2^{31} - 1 \\ 1 & \text{otherwise} \end{cases} \\[1ex]
&\underline{\texttt{of2}}(\texttt{op2\_and}, n_1, n_2) &&= 0 \\
&\underline{\texttt{of2}}(\texttt{op2\_or}, n_1, n_2) &&= 0 \\
&\underline{\texttt{of2}}(\texttt{op2\_xor}, n_1, n_2) &&= 0 \\
&\underline{\texttt{of2}}(\texttt{op2\_sf2}, n_1, n_2) &&= 0 \\[1ex]
&\underline{\texttt{of3}}(\mathit{op3}, n_1, n_2, n_3) &&= n \in \{0, 1\}
\end{aligned}
$$

Figure 4.2: Computing the Overflow Flag

$$
\begin{aligned}
&\underline{\texttt{cf1}}(\texttt{op1\_inc}, n) &&= \underline{\texttt{cf2}}(\texttt{op2\_add}, n, 1) \\
&\underline{\texttt{cf1}}(\texttt{op1\_dec}, n) &&= \underline{\texttt{cf2}}(\texttt{op2\_sub}, n, 1) \\
&\underline{\texttt{cf1}}(\texttt{op1\_neg}, n) &&= \underline{\texttt{cf2}}(\texttt{op2\_sub}, 0, n) \\
&\underline{\texttt{cf1}}(\texttt{op1\_not}, n) &&= 0 \\[1ex]
&\underline{\texttt{cf2}}(\texttt{op2\_add}, n_1, n_2) &&= \begin{cases} 1 & \text{if } n_1 \mathbin{\dot{+}} n_2 < n_1 \\ 0 & \text{otherwise} \end{cases} \\[1ex]
&\underline{\texttt{cf2}}(\texttt{op2\_sub}, n_1, n_2) &&= \begin{cases} 1 & \text{if } n_1 < n_2 \\ 0 & \text{otherwise} \end{cases} \\[1ex]
&\underline{\texttt{cf2}}(\texttt{op2\_imul}, n_1, n_2) &&= \underline{\texttt{of2}}(\texttt{op2\_imul}, n_1, n_2) \\
&\underline{\texttt{cf2}}(\texttt{op2\_and}, n_1, n_2) &&= 0 \\
&\underline{\texttt{cf2}}(\texttt{op2\_or}, n_1, n_2) &&= 0 \\
&\underline{\texttt{cf2}}(\texttt{op2\_xor}, n_1, n_2) &&= 0 \\
&\underline{\texttt{cf2}}(\texttt{op2\_sf2}, n_1, n_2) &&= 0 \\[1ex]
&\underline{\texttt{cf3}}(\mathit{op3}, n_1, n_2, n_3) &&= n \in \{0, 1\}
\end{aligned}
$$

Figure 4.3: Computing the Carry Flag

$$
\begin{aligned}
\underline{\texttt{selzf}}(n) &= (n \mathbin{\dot\wedge} 2^6)/2^6 \\
\underline{\texttt{selsf}}(n) &= (n \mathbin{\dot\wedge} 2^7)/2^7 \\
\underline{\texttt{selof}}(n) &= (n \mathbin{\dot\wedge} 2^{11})/2^{11} \\
\underline{\texttt{selcf}}(n) &= (n \mathbin{\dot\wedge} 2^0)/2^0
\end{aligned}
$$

$$
\underline{\texttt{updf1}}(\mathit{op1}, n, n_1) = \begin{cases}
\begin{array}{l}
\quad n \mathbin{\dot\wedge} \dot\neg(2^6 \mathbin{\dot\vee} 2^7 \mathbin{\dot\vee} 2^{11}) \\
\mathbin{\dot\vee} \underline{\texttt{zfw}}(\underline{\texttt{app1}}(\mathit{op1}, n_1)) \times 2^6 \\
\mathbin{\dot\vee} \underline{\texttt{sfw}}(\underline{\texttt{app1}}(\mathit{op1}, n_1)) \times 2^7 \\
\mathbin{\dot\vee} \underline{\texttt{of1}}(\mathit{op1}, n_1) \times 2^{11}
\end{array} & \text{if } \mathit{op1} \in \{\texttt{op1\_inc}, \texttt{op1\_dec}\} \\[2ex]
\begin{array}{l}
\quad n \mathbin{\dot\wedge} \dot\neg(2^6 \mathbin{\dot\vee} 2^7 \mathbin{\dot\vee} 2^{11} \mathbin{\dot\vee} 2^0) \\
\mathbin{\dot\vee} \underline{\texttt{zfw}}(\underline{\texttt{app1}}(\mathit{op1}, n_1)) \times 2^6 \\
\mathbin{\dot\vee} \underline{\texttt{sfw}}(\underline{\texttt{app1}}(\mathit{op1}, n_1)) \times 2^7 \\
\mathbin{\dot\vee} \underline{\texttt{of1}}(\mathit{op1}, n_1) \times 2^{11} \\
\mathbin{\dot\vee} \underline{\texttt{cf1}}(\mathit{op1}, n_1) \times 2^0
\end{array} & \text{if } \mathit{op1} = \texttt{op1\_neg} \\[2ex]
n & \text{if } \mathit{op1} = \texttt{op1\_not}
\end{cases}
$$

$$
\underline{\texttt{updf2}}(\mathit{op2}, n, n_1, n_2) = \begin{cases}
\begin{array}{l}
\quad n \mathbin{\dot\wedge} \dot\neg(2^{11} \mathbin{\dot\vee} 2^0) \\
\mathbin{\dot\vee} \underline{\texttt{of2}}(\mathit{op2}, n_1, n_2) \times 2^{11} \\
\mathbin{\dot\vee} \underline{\texttt{cf2}}(\mathit{op2}, n_1, n_2) \times 2^0
\end{array} & \text{if } \mathit{op2} = \texttt{op2\_imul} \\[2ex]
\begin{array}{l}
\quad n \mathbin{\dot\wedge} \dot\neg(2^6 \mathbin{\dot\vee} 2^7 \mathbin{\dot\vee} 2^{11} \mathbin{\dot\vee} 2^0) \\
\mathbin{\dot\vee} \underline{\texttt{zfw}}(\underline{\texttt{app2}}(\mathit{op2}, n_1, n_2)) \times 2^6 \\
\mathbin{\dot\vee} \underline{\texttt{sfw}}(\underline{\texttt{app2}}(\mathit{op2}, n_1, n_2)) \times 2^7 \\
\mathbin{\dot\vee} \underline{\texttt{of2}}(\mathit{op2}, n_1, n_2) \times 2^{11} \\
\mathbin{\dot\vee} \underline{\texttt{cf2}}(\mathit{op2}, n_1, n_2) \times 2^0
\end{array} & \text{otherwise}
\end{cases}
$$

$$
\underline{\texttt{updf3}}(\mathit{op3}, n, n_1, n_2, n_3) = n' \in \mathit{Val}^{\texttt{wd}}
$$

Figure 4.4: Encoding the Status Flags

The functions in Figure 4.2 compute the overflow flag for the result of applying a given operator to a given set of arguments. The functions in Figure 4.3 compute the carry flag for the result of applying a given operator to a given set of arguments. Note that the effect of the two ternary operators (`op3_idiv`, `op3_irem`) on the status flags is not defined according to the IA-32 specification [Int01]. Finally, the functions in Figure 4.4 encode and decode individual status flags that are stored collectively in the status flags register.

$\underline{\texttt{self}}(cop, n)$ determines if a given conditional operator cop holds according to the encoded status flags n:

$$\begin{aligned}
\underline{\texttt{self}}(\texttt{cop_z}, n) &= \underline{\texttt{selzf}}(n) \\
\underline{\texttt{self}}(\texttt{cop_s}, n) &= \underline{\texttt{selsf}}(n) \\
\underline{\texttt{self}}(\texttt{cop_o}, n) &= \underline{\texttt{selof}}(n) \\
\underline{\texttt{self}}(\texttt{cop_c}, n) &= \underline{\texttt{selcf}}(n) \\
\underline{\texttt{self}}(\texttt{cop_na}, n) &= \underline{\texttt{selzf}}(n) \mathbin{\dot{\vee}} \underline{\texttt{selcf}}(n) \\
\underline{\texttt{self}}(\texttt{cop_l}, n) &= \underline{\texttt{selsf}}(n) \mathbin{\dot{\veebar}} \underline{\texttt{selof}}(n) \\
\underline{\texttt{self}}(\texttt{cop_ng}, n) &= \underline{\texttt{selzf}}(n) \mathbin{\dot{\vee}} \underline{\texttt{self}}(\texttt{cop_l}, n)
\end{aligned}$$

$$\begin{aligned}
\underline{\texttt{self}}(\texttt{cop_nz}, n) &= \underline{\texttt{selzf}}(n) \mathbin{\dot{\veebar}} 1 \\
\underline{\texttt{self}}(\texttt{cop_ns}, n) &= \underline{\texttt{selsf}}(n) \mathbin{\dot{\veebar}} 1 \\
\underline{\texttt{self}}(\texttt{cop_no}, n) &= \underline{\texttt{selof}}(n) \mathbin{\dot{\veebar}} 1 \\
\underline{\texttt{self}}(\texttt{cop_nc}, n) &= \underline{\texttt{selcf}}(n) \mathbin{\dot{\veebar}} 1 \\
\underline{\texttt{self}}(\texttt{cop_a}, n) &= \underline{\texttt{self}}(\texttt{cop_na}, n) \mathbin{\dot{\veebar}} 1 \\
\underline{\texttt{self}}(\texttt{cop_nl}, n) &= \underline{\texttt{self}}(\texttt{cop_l}, n) \mathbin{\dot{\veebar}} 1 \\
\underline{\texttt{self}}(\texttt{cop_g}, n) &= \underline{\texttt{self}}(\texttt{cop_ng}, n) \mathbin{\dot{\veebar}} 1
\end{aligned}$$

$\underline{\texttt{not}}(cop)$ is the complement of the conditional operator cop:

$$\begin{aligned}
\underline{\texttt{not}}(\texttt{cop_z}) &= \texttt{cop_nz} & \underline{\texttt{not}}(\texttt{cop_nz}) &= \texttt{cop_z} \\
\underline{\texttt{not}}(\texttt{cop_s}) &= \texttt{cop_ns} & \underline{\texttt{not}}(\texttt{cop_ns}) &= \texttt{cop_s} \\
\underline{\texttt{not}}(\texttt{cop_o}) &= \texttt{cop_no} & \underline{\texttt{not}}(\texttt{cop_no}) &= \texttt{cop_o} \\
\underline{\texttt{not}}(\texttt{cop_c}) &= \texttt{cop_nc} & \underline{\texttt{not}}(\texttt{cop_nc}) &= \texttt{cop_c} \\
\underline{\texttt{not}}(\texttt{cop_na}) &= \texttt{cop_a} & \underline{\texttt{not}}(\texttt{cop_a}) &= \texttt{cop_na} \\
\underline{\texttt{not}}(\texttt{cop_l}) &= \texttt{cop_nl} & \underline{\texttt{not}}(\texttt{cop_nl}) &= \texttt{cop_l} \\
\underline{\texttt{not}}(\texttt{cop_ng}) &= \texttt{cop_g} & \underline{\texttt{not}}(\texttt{cop_g}) &= \texttt{cop_ng}
\end{aligned}$$

The complement of cop holds exactly when cop does not hold itself.

Finally, the interpretation of an instruction-set constant or function is governed by its subscript ($e.g.$, $\mathcal{J}(c_{op2}) = op2$, and $\mathcal{J}(f_{\texttt{ea\_r}})(r) = \texttt{ea\_r}(r)$).

4.3.4 Generic Functions

The remaining functions, defined in this subsection, are generic operations that are not specifically tailored to the IA-32 architecture.

Figure 4.5 contains basic arithmetic and logical operations on machine words. Note that divide and remainder return an unspecified value when the divisor is zero.

$$\underline{\texttt{zfw}}(n) = \begin{cases} 1 & \text{if } n = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\underline{\texttt{sfw}}(n) = \begin{cases} 1 & \text{if } n \geq 2^{31} \\ 0 & \text{otherwise} \end{cases}$$

$$\underline{\texttt{negw}}(n) = 0 \mathbin{\dot{-}} n$$

$$\begin{aligned} \underline{\texttt{addw}}(n_1, n_2) &= n_1 \mathbin{\dot{+}} n_2 \\ \underline{\texttt{subw}}(n_1, n_2) &= n_1 \mathbin{\dot{-}} n_2 \\ \underline{\texttt{mulw}}(n_1, n_2) &= n_1 \mathbin{\dot{\times}} n_2 \end{aligned}$$

$$\underline{\texttt{divw}}(n_1, n_2, n_3) = \begin{cases} \lfloor n' \rfloor & \text{if } n_3 \neq 0 \text{ and } 0 \leq n' < 2^{31} \\ \lceil n' \rceil + 2^{32} & \text{if } n_3 \neq 0 \text{ and } -2^{31} \leq n' < 0 \\ n \in \mathit{Val}^{\texttt{wd}} & \text{otherwise} \end{cases}$$
$$\text{where } n' = \ulcorner\!\ulcorner n_1, n_2 \urcorner\!\urcorner / \ulcorner n_3 \urcorner$$

$$\underline{\texttt{remw}}(n_1, n_2, n_3) = \begin{cases} \ulcorner\!\ulcorner n_1, n_2 \urcorner\!\urcorner - \lfloor n' \rfloor \times n_3 & \text{if } n_3 \neq 0 \text{ and } 0 \leq n' < 2^{31} \\ \ulcorner\!\ulcorner n_1, n_2 \urcorner\!\urcorner - (\lceil n' \rceil + 2^{32}) \times n_3 & \text{if } n_3 \neq 0 \text{ and } -2^{31} \leq n' < 0 \\ n \in \mathit{Val}^{\texttt{wd}} & \text{otherwise} \end{cases}$$
$$\text{where } n' = \ulcorner\!\ulcorner n_1, n_2 \urcorner\!\urcorner / \ulcorner n_3 \urcorner$$

$$\begin{aligned} \underline{\texttt{andw}}(n_1, n_2) &= n_1 \mathbin{\dot{\wedge}} n_2 \\ \underline{\texttt{orw}}(n_1, n_2) &= n_1 \mathbin{\dot{\vee}} n_2 \\ \underline{\texttt{xorw}}(n_1, n_2) &= n_1 \mathbin{\dot{\veebar}} n_2 \end{aligned}$$

$$\underline{\texttt{notw}}(n) = n \mathbin{\dot{\veebar}} 2^{32} - 1$$

$$\begin{aligned} \underline{\texttt{ltw}}(n_1, n_2) \quad &\text{iff} \quad \ulcorner n_1 \urcorner < \ulcorner n_2 \urcorner \\ \underline{\texttt{ltuw}}(n_1, n_2) \quad &\text{iff} \quad n_1 < n_2 \\ \underline{\texttt{incw}}(n_1, n_2) \quad &\text{iff} \quad n_1 \mathbin{\dot{\vee}} n_2 = n_2 \end{aligned}$$

Figure 4.5: Arithmetic and Logical Operations

The following functions construct and project elements out of state tuples:

$$
\begin{aligned}
\underline{\texttt{s\_mk}}(n_{\texttt{pc}}, n_{\texttt{f}}, v_{\texttt{g}}, v_{\texttt{s}}, v_{\texttt{m}}) &= \langle n_{\texttt{pc}}, n_{\texttt{f}}, v_{\texttt{g}}, v_{\texttt{s}}, v_{\texttt{m}} \rangle \\[4pt]
\underline{\texttt{s\_pc}}(\langle n_{\texttt{pc}}, n_{\texttt{f}}, v_{\texttt{g}}, v_{\texttt{s}}, v_{\texttt{m}} \rangle) &= n_{\texttt{pc}} \\
\underline{\texttt{s\_f}}(\langle n_{\texttt{pc}}, n_{\texttt{f}}, v_{\texttt{g}}, v_{\texttt{s}}, v_{\texttt{m}} \rangle) &= n_{\texttt{f}} \\
\underline{\texttt{s\_g}}(\langle n_{\texttt{pc}}, n_{\texttt{f}}, v_{\texttt{g}}, v_{\texttt{s}}, v_{\texttt{m}} \rangle) &= v_{\texttt{g}} \\
\underline{\texttt{s\_s}}(\langle n_{\texttt{pc}}, n_{\texttt{f}}, v_{\texttt{g}}, v_{\texttt{s}}, v_{\texttt{m}} \rangle) &= v_{\texttt{s}} \\
\underline{\texttt{s\_m}}(\langle n_{\texttt{pc}}, n_{\texttt{f}}, v_{\texttt{g}}, v_{\texttt{s}}, v_{\texttt{m}} \rangle) &= v_{\texttt{m}}
\end{aligned}
$$

The following functions are the fundamental operations on register files and word mappings:

$$
\begin{aligned}
\underline{\texttt{selg}}(v, r) &= v(r) \\
\underline{\texttt{updg}}(v, r, n) &= v[r \mapsto n] \\[4pt]
\underline{\texttt{selw}}(v, n) &= v(n) \\
\underline{\texttt{updw}}(v, n_1, n_2) &= v[n_1 \mapsto n_2] \\
\underline{\texttt{joinw}}(v_1, n, v_2) &= n' \mapsto \begin{cases} v_1(n') & \text{if } n' < n \\ v_2(n') & \text{otherwise} \end{cases}
\end{aligned}
$$

The `joinw` operation is used to "superimpose" two distinct address spaces. This function is used in the memory safety policy (see Chapter 5) to enforce that the stack is not changed above a particular address.

$\underline{\texttt{fetch}}(\Phi, n, I)$ assigns the instruction I to address n of program Φ:

$$\mathcal{J}(\texttt{fetch})(\Phi, n, I) \quad \text{iff} \quad n \in \operatorname{dom} \Phi \text{ and } \Phi(n) = I$$

4.4 Inference Rules

At this point, the semantics of the abstract machine is fully specified. However, in order to construct a safety proof based on this semantics, the code producer must be provided with a set of inference rules that formalizes this semantics in logical notation. The presentation in this chapter follows the same pattern as in Chapter 3. First, a formal semantics is defined, then a set of inference rules are introduced that are sound with respect to this semantics.

The inference rules in this section are specific to the IA-32 machine model. Specifically, I introduce rules that encode properties of the machine-model functions and relations from Table 4.1, Table 4.2, and Table 4.3. In Section 4.4.7 I introduce rules that encode properties of the transition relation. These rules only hold for executions that respect the transition relation $\Phi \triangleright s \to s'$ (see Section 4.3).

Many inference rules are direct encodings of semantic definitions of particular machine-model functions. For example,

$$\frac{}{\Gamma \vdash \mathsf{app2}(\overline{\texttt{op2\_add}}, e_1, e_2) = \texttt{addw}(e_1, e_2) \mathbin{@} t}\ \mathsf{app2\_add}$$

formally encodes the defining equation of the addition operator:

$$\underline{\texttt{app2}}(\texttt{op2\_add}, n_1, n_2) = n_1 \dot{+} n_2$$

Because most of the formal encodings are so similar to the machine semantics, I do not reproduce them here—instead, I refer the interested reader to Appendix B.2, in which I give a complete LF encoding of the machine model. In this section, I only explicitly mention theories that do not simply embody equations from the machine semantics.

4.4.1 Machine Words

The induction rule is a variation on the standard rule from Peano arithmetic that can demonstrate universal properties of the natural numbers:

$$\frac{\Gamma \vdash e : \mathsf{ri} \quad \Gamma,\, a : \mathsf{ri},\, \forall x : \mathsf{ri}.\, \texttt{ltuw}(x, a) \supset p \mathbin{@} t \vdash [a/x]\, p \mathbin{@} t}{\Gamma \vdash [e/x]\, p \mathbin{@} t}\ \mathsf{wd\_ind}^a$$

The second premise requires the property p to be established for an arbitrary machine word a, under the assumption that p holds for all machine words that are strictly less than a. Note that the hypothetical proposition here is an internalized version of an LF construct that is awkward to express in standard deductive notation—see Appendix B.2.2.1 for the true LF representation.

The following rules formalize the mathematical definition of the "zero flag" and "sign flag" functions:

$$\frac{}{\Gamma \vdash \texttt{zfw}(0) = 1 \mathbin{@} t}\ \texttt{zfwi1} \qquad \frac{\Gamma \vdash e \neq 0 \mathbin{@} t}{\Gamma \vdash \texttt{zfw}(e) = 0 \mathbin{@} t}\ \texttt{zfwi0}$$

$$\frac{\Gamma \vdash \texttt{zfw}(e) = 1 \mathbin{@} t}{\Gamma \vdash e = 0 \mathbin{@} t}\ \texttt{zfwe1} \qquad \frac{\Gamma \vdash \texttt{zfw}(e) = 0 \mathbin{@} t}{\Gamma \vdash e \neq 0 \mathbin{@} t}\ \texttt{zfwe0}$$

$$\frac{\Gamma \vdash \texttt{ltw}(e, 0) \mathbin{@} t}{\Gamma \vdash \texttt{sfw}(e) = 1 \mathbin{@} t}\ \texttt{sfwi1} \qquad \frac{\Gamma \vdash \texttt{geqw}(e, 0) \mathbin{@} t}{\Gamma \vdash \texttt{sfw}(e) = 0 \mathbin{@} t}\ \texttt{sfwi0}$$

$$\frac{\Gamma \vdash \texttt{sfw}(e) = 1 \mathbin{@} t}{\Gamma \vdash \texttt{ltw}(e, 0) \mathbin{@} t}\ \texttt{sfwe1} \qquad \frac{\Gamma \vdash \texttt{sfw}(e) = 0 \mathbin{@} t}{\Gamma \vdash \texttt{geqw}(e, 0) \mathbin{@} t}\ \texttt{sfwe0}$$

The following rules establish the algebraic properties of the machine-word operations:

$$\frac{}{\Gamma \vdash \texttt{comm\_ring}(\texttt{addw}, 0, \texttt{negw}, \texttt{mulw}, 1) \mathbin{@} t}\ \mathsf{wd\_comm\_ring}$$

$$\frac{}{\Gamma \vdash \texttt{mulw}(e, 0) = 0 \mathbin{@} t}\ \mathsf{mulw\_id\_0}$$

$$\frac{}{\Gamma \vdash \texttt{bool\_latt}(\texttt{incw}, \texttt{andw}, \texttt{orw}, 0, 2^{32} - 1, \texttt{notw}) \mathbin{@} t}\ \mathsf{wd\_bool\_latt}$$

$$\frac{}{\Gamma \vdash \texttt{subw}(e_1, e_2) = \texttt{addw}(e_1, \texttt{negw}(e_2)) \mathbin{@} t}\ \mathsf{wd\_sub}$$

$$\frac{}{\Gamma \vdash \mathtt{xorw}(e_1, e_2) = \mathtt{andw}(\mathtt{orw}(e_1, e_2), \mathtt{orw}(\mathtt{notw}(e_1), \mathtt{notw}(e_2))) \mathbin{@} t}\ \textsf{wd\_xor}$$

$$\frac{}{\Gamma \vdash \mathtt{str\_tot\_order}(\mathtt{ltw}) \mathbin{@} t}\ \textsf{ltw\_str\_tot\_order}$$

$$\frac{}{\Gamma \vdash \mathtt{bot}(\mathtt{ltw}, 2^{31}) \mathbin{@} t}\ \textsf{ltw\_bot} \qquad \frac{}{\Gamma \vdash \mathtt{top}(\mathtt{ltw}, 2^{31} - 1) \mathbin{@} t}\ \textsf{ltw\_top}$$

$$\frac{}{\Gamma \vdash \mathtt{str\_tot\_order}(\mathtt{ltuw}) \mathbin{@} t}\ \textsf{ltuw\_str\_tot\_order}$$

$$\frac{}{\Gamma \vdash \mathtt{bot}(\mathtt{ltuw}, 0) \mathbin{@} t}\ \textsf{ltuw\_bot} \qquad \frac{}{\Gamma \vdash \mathtt{top}(\mathtt{ltuw}, 2^{32} - 1) \mathbin{@} t}\ \textsf{ltuw\_top}$$

$$\frac{}{\Gamma \vdash \mathtt{bot}(\mathtt{incw}, 0) \mathbin{@} t}\ \textsf{incw\_bot} \qquad \frac{}{\Gamma \vdash \mathtt{top}(\mathtt{incw}, 2^{32} - 1) \mathbin{@} t}\ \textsf{incw\_top}$$

4.4.2 Machine Operators

In the interest of brevity, the rules in this section are presented in tabular form. The header of each table contains a prototype inference rule. Each column of the table identifies a particular meta variable of the prototype that has a restriction on how it can be instantiated. Each row of the table contains a single set of valid instantiations for the meta variables that are identified by the column header. Thus, each row of each table effectively introduces a distinct inference rule.

The following rules encode the behavior of a given flag-selection function when applied to a corresponding unary flag-update function:

$$\frac{}{\Gamma \vdash f(\mathtt{updf1}(c_{op1}, e_{\mathtt{f}}, e)) = e' \mathbin{@} t}\ \textsf{updf1}_{f, c_{op1}}$$

f	$op1$	e'
selzf	op1_inc	$\mathtt{zfw}(\mathtt{app1}(c_{\mathtt{op1\_inc}}, e))$
selzf	op1_dec	$\mathtt{zfw}(\mathtt{app1}(c_{\mathtt{op1\_dec}}, e))$
selzf	op1_neg	$\mathtt{zfw}(\mathtt{app1}(c_{\mathtt{op1\_neg}}, e))$
selzf	op1_not	$\mathtt{selzf}(e_{\mathtt{f}})$
selsf	op1_inc	$\mathtt{sfw}(\mathtt{app1}(c_{\mathtt{op1\_inc}}, e))$
selsf	op1_dec	$\mathtt{sfw}(\mathtt{app1}(c_{\mathtt{op1\_dec}}, e))$
selsf	op1_neg	$\mathtt{sfw}(\mathtt{app1}(c_{\mathtt{op1\_neg}}, e))$
selsf	op1_not	$\mathtt{selsf}(e_{\mathtt{f}})$
selof	op1_inc	$\mathtt{of1}(c_{\mathtt{op1\_inc}}, e)$
selof	op1_dec	$\mathtt{of1}(c_{\mathtt{op1\_dec}}, e)$
selof	op1_neg	$\mathtt{of1}(c_{\mathtt{op1\_neg}}, e)$
selof	op1_not	$\mathtt{selof}(e_{\mathtt{f}})$
selcf	op1_inc	$\mathtt{selcf}(e_{\mathtt{f}})$
selcf	op1_dec	$\mathtt{selcf}(e_{\mathtt{f}})$
selcf	op1_neg	$\mathtt{cf1}(c_{\mathtt{op1\_neg}}, e)$
selcf	op1_not	$\mathtt{selcf}(e_{\mathtt{f}})$

The following rules encode the behavior of a given flag-selection function when applied to a corresponding binary flag-update function:

$$\frac{}{\Gamma \vdash f(\mathtt{updf2}(c_{op2}, e_{\mathtt{f}}, e_1, e_2)) = e' \,@\, t} \; \mathsf{updf2}_f$$

f	e'
`selzf`	$\mathtt{zfw}(\mathtt{app2}(c_{op2}, e_1, e_2))$
`selsf`	$\mathtt{sfw}(\mathtt{app2}(c_{op2}, e_1, e_2))$
`selof`	$\mathtt{of2}(c_{op2}, e_1, e_2)$
`selcf`	$\mathtt{cf2}(c_{op2}, e_1, e_2)$

There are no rules for `updf3`, because its behavior should be undefined according to the IA-32 specification [Int01].

Finally, the following rules restrict the value of an overflow flag or a carry flag to zero or one:

$$\frac{\Gamma \vdash e \neq 0 \,@\, t}{\Gamma \vdash e = 1 \,@\, t} \; \mathsf{neq0}_f$$

e
$\mathtt{of1}(e_{\mathtt{op1}}, e_1)$
$\mathtt{of2}(e_{\mathtt{op2}}, e_1, e_2)$
$\mathtt{of3}(e_{\mathtt{op3}}, e_1, e_2, e_3)$
$\mathtt{cf1}(e_{\mathtt{op1}}, e_1)$
$\mathtt{cf2}(e_{\mathtt{op2}}, e_1, e_2)$
$\mathtt{cf3}(e_{\mathtt{op3}}, e_1, e_2, e_3)$

4.4.3 Conditional Operators

The rule $\mathtt{self}_{\mathsf{neq0}}$ constrains the value of a conditional operator to zero or one:

$$\frac{\Gamma \vdash \mathtt{self}(e_{\mathtt{cop}}, e_{\mathtt{f}}) \neq 0 \,@\, t}{\Gamma \vdash \mathtt{self}(e_{\mathtt{cop}}, e_{\mathtt{f}}) = 1 \,@\, t} \; \mathtt{self}_{\mathsf{neq0}}$$

$\mathtt{self}_{\mathsf{not}}$ specifies that the effect of the `not` function is to invert the state of a given conditional operator:

$$\frac{}{\Gamma \vdash \mathtt{self}(\mathtt{not}(e_{\mathtt{cop}}), e_{\mathtt{f}}) = \mathtt{xorw}(\mathtt{self}(e_{\mathtt{cop}}, e_{\mathtt{f}}), 1) \,@\, t} \; \mathtt{self}_{\mathsf{not}}$$

$\mathtt{not}_{\mathsf{not}}$ specifies that the complement of a complement is the identity:

$$\frac{}{\Gamma \vdash \mathtt{not}(\mathtt{not}(e_{\mathtt{cop}})) = e_{\mathtt{cop}} \,@\, t} \; \mathtt{not}_{\mathsf{not}}$$

4.4.4 Register Files

The following theory encodes the semantics of register files:

$$\frac{}{\Gamma \vdash \mathtt{selg}(\mathtt{updg}(e_{\mathtt{m}}, e_{\mathtt{r}}, e_{\mathsf{n}}), e_{\mathtt{r}}) = e_{\mathsf{n}} \,@\, t} \; \mathtt{selg}_{\mathsf{mc0}}$$

$$\frac{\Gamma \vdash e'_{\mathsf{r}} \neq e_{\mathsf{r}} \,@\, t}{\Gamma \vdash \mathtt{selg}(\mathtt{updg}(e_{\mathtt{m}}, e_{\mathsf{r}}, e_{\mathsf{n}}), e'_{\mathsf{r}}) = \mathtt{selg}(e_{\mathtt{m}}, e'_{\mathsf{r}}) \,@\, t}\ \mathtt{selg}_{\mathsf{mc1}}$$

$$\frac{\Gamma \vdash \mathtt{selg}(e_{\mathtt{m}}, a_{\mathsf{r}}) = \mathtt{selg}(e'_{\mathtt{m}}, a_{\mathsf{r}}) \,@\, t}{\Gamma \vdash e_{\mathtt{m}} = e'_{\mathtt{m}} \,@\, t}\ \mathtt{selg}^{a_{\mathsf{r}}}_{\mathsf{ext}}$$

$\mathtt{selg}_{\mathsf{mc0}}$ and $\mathtt{selg}_{\mathsf{mc1}}$ are the standard McCarthy rules [MP67] for arrays here adapted to register files. $\mathtt{selg}_{\mathsf{ext}}$ expresses an extensionality principle for register files.

4.4.5 Machine-word Mappings

The following theory encodes the semantics of word mappings:

$$\frac{}{\Gamma \vdash \mathtt{selw}(\mathtt{updw}(e_{\mathtt{m}}, e_{\mathsf{n}_1}, e_{\mathsf{n}_2}), e_{\mathsf{n}_1}) = e_{\mathsf{n}_2} \,@\, t}\ \mathtt{selw}_{\mathsf{mc0}}$$

$$\frac{\Gamma \vdash e'_{\mathsf{n}_1} \neq e_{\mathsf{n}_1} \,@\, t}{\Gamma \vdash \mathtt{selw}(\mathtt{updw}(e_{\mathtt{m}}, e_{\mathsf{n}_1}, e_{\mathsf{n}_2}), e'_{\mathsf{n}_1}) = \mathtt{selw}(e_{\mathtt{m}}, e'_{\mathsf{n}_1}) \,@\, t}\ \mathtt{selw}_{\mathsf{mc1}}$$

$$\frac{\Gamma \vdash \mathtt{ltuw}(e'_{\mathsf{n}}, e_{\mathsf{n}}) \,@\, t}{\Gamma \vdash \mathtt{selw}(\mathtt{joinw}(e_{\mathtt{m}_1}, e_{\mathsf{n}}, e_{\mathtt{m}_2}), e'_{\mathsf{n}}) = \mathtt{selw}(e_{\mathtt{m}_1}, e'_{\mathsf{n}}) \,@\, t}\ \mathtt{selw}_{\mathsf{ltu}}$$

$$\frac{\Gamma \vdash \mathtt{gequw}(e'_{\mathsf{n}}, e_{\mathsf{n}}) \,@\, t}{\Gamma \vdash \mathtt{selw}(\mathtt{joinw}(e_{\mathtt{m}_1}, e_{\mathsf{n}}, e_{\mathtt{m}_2}), e'_{\mathsf{n}}) = \mathtt{selw}(e_{\mathtt{m}_2}, e'_{\mathsf{n}}) \,@\, t}\ \mathtt{selw}_{\mathsf{gequ}}$$

$$\frac{\Gamma \vdash \mathtt{selw}(e_{\mathtt{m}}, a_{\mathsf{n}}) = \mathtt{selw}(e'_{\mathtt{m}}, a_{\mathsf{n}}) \,@\, t}{\Gamma \vdash e_{\mathtt{m}} = e'_{\mathtt{m}} \,@\, t}\ \mathtt{selw}^{a_{\mathsf{n}}}_{\mathsf{ext}}$$

The McCarthy rules and extensionality are similar to the rules for register files. $\mathtt{selw}_{\mathsf{ltu}}$ and $\mathtt{selw}_{\mathsf{gequ}}$ encode the semantics of the "join" operation.

4.4.6 Instruction-Set Constructors

The following rules establish the injectivity of the various constructors that encode the IA-32 instruction set:

$$\frac{}{\Gamma \vdash \mathtt{inj}(f) \,@\, t}\ \mathsf{ma\_inj}^{f \in \{f_{\mathtt{ma\_d}}, f_{\mathtt{ma\_r}}\}}$$

$$\frac{}{\Gamma \vdash \mathtt{inj}(f) \,@\, t}\ \mathsf{ea\_inj}^{f \in \{f_{\mathtt{ea\_i}}, f_{\mathtt{ea\_r}}, f_{\mathtt{ea\_s}}, f_{\mathtt{ea\_m}}\}}$$

$$\frac{}{\Gamma \vdash \mathtt{inj}(f) \,@\, t}\ \mathsf{inst\_inj}^{f \in \{f_{\mathtt{mov}}, f_{\mathtt{xchg}}, f_{\mathtt{lea}}, f_{\mathtt{push}}, f_{\mathtt{pop}}, f_{\mathtt{op1}}, f_{\mathtt{op2}}, f_{\mathtt{op2n}}, f_{\mathtt{op3}}, f_{\mathtt{jmp}}, f_{\mathtt{j}}, f_{\mathtt{call}}, f_{\mathtt{ret}}\}}$$

4.4.7 Transition Rules

The following transition rules specify how the special-purpose registers change from state to state according to the transition relation:

$$\frac{}{\Gamma \vdash \forall x : \mathsf{ri}.\, \forall x_{\mathsf{l}} : \mathsf{ri}.\, x = \mathtt{ss} \supset \mathtt{fetch}(\mathtt{pm}, \mathtt{s\_pc}(x), x_{\mathsf{l}}) \supset \bigcirc(\mathtt{pc} = \mathtt{nextpc}(x, x_{\mathsf{l}})) \mathbin{@} t}\ \mathsf{nextpc\_fetch}$$

$$\frac{}{\Gamma \vdash \forall x : \mathsf{ri}.\, \forall x_{\mathsf{l}} : \mathsf{ri}.\, x = \mathtt{ss} \supset \mathtt{fetch}(\mathtt{pm}, \mathtt{s\_pc}(x), x_{\mathsf{l}}) \supset \bigcirc(\mathtt{f} = \mathtt{nextf}(x, x_{\mathsf{l}})) \mathbin{@} t}\ \mathsf{nextf\_fetch}$$

$$\frac{}{\Gamma \vdash \forall x : \mathsf{ri}.\, \forall x_{\mathsf{l}} : \mathsf{ri}.\, x = \mathtt{ss} \supset \mathtt{fetch}(\mathtt{pm}, \mathtt{s\_pc}(x), x_{\mathsf{l}}) \supset \bigcirc(\mathtt{g} = \mathtt{nextg}(x, x_{\mathsf{l}})) \mathbin{@} t}\ \mathsf{nextg\_fetch}$$

$$\frac{}{\Gamma \vdash \forall x : \mathsf{ri}.\, \forall x_{\mathsf{l}} : \mathsf{ri}.\, x = \mathtt{ss} \supset \mathtt{fetch}(\mathtt{pm}, \mathtt{s\_pc}(x), x_{\mathsf{l}}) \supset \bigcirc(\mathtt{s} = \mathtt{nexts}(x, x_{\mathsf{l}})) \mathbin{@} t}\ \mathsf{nexts\_fetch}$$

$$\frac{}{\Gamma \vdash \forall x : \mathsf{ri}.\, \forall x_{\mathsf{l}} : \mathsf{ri}.\, x = \mathtt{ss} \supset \mathtt{fetch}(\mathtt{pm}, \mathtt{s\_pc}(x), x_{\mathsf{l}}) \supset \bigcirc(\mathtt{m} = \mathtt{nextm}(x, x_{\mathsf{l}})) \mathbin{@} t}\ \mathsf{nextm\_fetch}$$

where $\mathtt{ss}$ is an abbreviation:

$$\mathtt{ss} \stackrel{\mathsf{def}}{=} \mathtt{s\_mk}(\mathtt{pc}, \mathtt{f}, \mathtt{g}, \mathtt{s}, \mathtt{m})$$

(*i.e.*, all the special-purpose registers packaged together as a single state tuple).

In each rule, the current-time values of the registers are identified with the rigid variable x. Then, new values of the registers are provided at the next time instant according to the instruction of the program in the program memory at the current program counter. Rigid variables name the previous-time values of the registers inside the $\bigcirc$ operator. The new value of the register in question is determined by a function (*e.g.*, $\mathtt{nextpc}$) of the current-time state (x) and the current-time instruction ($\mathtt{fetch}(\mathtt{pm}, \mathtt{s\_pc}(x))$).

Note that the transition rules do not check that the program has proper control flow, unlike other implementations of PCC. Any control flow that has a valid security proof is permitted, but the security policy will generally require that the program counter stay within the program.

It is occasionally convenient to refer to a complete next-time state, so I use the following abbreviation:

$$\begin{aligned} &\mathtt{s\_next}(e, e_{\mathsf{l}}) \\ \stackrel{\mathsf{def}}{=}\ &\mathtt{s\_mk}(\mathtt{nextpc}(e, e_{\mathsf{l}}), \mathtt{nextf}(e, e_{\mathsf{l}}), \mathtt{nextg}(e, e_{\mathsf{l}}), \mathtt{nexts}(e, e_{\mathsf{l}}), \mathtt{nextm}(e, e_{\mathsf{l}})) \end{aligned}$$

Additionally, $\mathtt{gsp}$ is the current stack pointer:

$$\mathtt{gsp} \stackrel{\mathsf{def}}{=} \mathtt{selg}(\mathtt{g}, \overline{\mathtt{esp}})$$

Finally, I provide an inference rule that associates each address of the program memory with the corresponding instruction:

$$\frac{}{\Gamma \vdash^{\mathcal{J}} \mathtt{fetch}(\mathtt{pm}, \overline{n}, e) \mathbin{@} t}\ \mathsf{fetch\_pm}^{n \in \mathrm{dom}\, \mathcal{J}(\mathtt{pm}), \mathcal{J}(\mathtt{pm})(n) = \mathcal{V}^{\mathcal{J}}_{\phi}(e)}$$

e is the instruction at address n, and is composed solely of constants and constant functions for this rule, so ϕ is arbitrary. Note that in my implementation, there is no single $\mathsf{fetch\_pm}$ rule. It is simpler to extend the LF signature with individual rules for each address according to the contents of the program.

Additionally, the fetch relation is a (partial) function:

$$\frac{}{\Gamma \vdash \mathtt{fun}(\mathtt{fetch}) \mathbin{@} t}\ \mathsf{fetch\_fun}$$

4.5 Soundness

In this section, I show that the formal encoding of the machine model is sound with respect to the operational semantics. Informally, I am simply extending Proposition 3.5.1 to encompass more inference rules, but in order to make my argument more convincing to a skeptic, I will be more thorough and assign a subscript to the affirmation judgment according to which inference rules were used to derive the affirmation:

- $\Gamma \vdash^{\mathcal{J}} J$ asserts that J is derivable from Γ using only inference rules from Chapter 3.
- $\Gamma \vdash^{\mathcal{J}}_{\mathsf{x86}} J$ asserts that J is derivable from Γ using only inference rules from Chapter 3 and Section 4.4.1 through Section 4.4.5.
- $\Gamma \vdash^{\mathcal{J}}_{\mathsf{x86+}} J$ asserts that J is derivable from Γ using only inference rules from Chapter 3 and Section 4.4.

Proposition 3.5.1 shows that $\phi, \eta \vDash^{\mathcal{J}} J$ follows from $\phi, \eta \vDash^{\mathcal{J}} \Gamma$ and $\Gamma \vdash^{\mathcal{J}} J$. I now extend this proposition to the functions that define the machine-model:

Proposition 4.5.1 (Soundness) *$\phi, \eta \vDash^{\mathcal{J}} J$ if $\phi, \eta \vDash^{\mathcal{J}} \Gamma$ and $\Gamma \vdash^{\mathcal{J}}_{\mathsf{x86}} J$*

Proof:

by induction on the derivation of $\Gamma \vdash^{\mathcal{J}}_{\mathsf{x86}} J$, using the proof of Proposition 3.5.1

$\phi, \eta \vDash^{\mathcal{J}} \Gamma \qquad \Gamma \vdash^{\mathcal{J}}_{\mathsf{x86}} J$ Prem.

let $\mathcal{D}$ be the derivation of $\Gamma \vdash^{\mathcal{J}}_{\mathsf{x86}} J$

case: $J = [e/x]\, p \mathbin{@} t$,

$$\mathcal{D} = \frac{\overset{\mathcal{D}_1'}{\Gamma \vdash e : \mathsf{ri}} \quad \overset{\mathcal{D}_2'}{\Gamma, a : \mathsf{ri}, \forall x : \mathsf{ri}.\, \mathtt{ltuw}(x, a) \supset p \mathbin{@} t \vdash [a/x]\, p \mathbin{@} t}}{\Gamma \vdash [e/x]\, p \mathbin{@} t}\ \mathsf{wd\_ind}^{a}$$

let $j = \mathcal{V}_\eta(t), n' = \mathcal{V}^{\mathcal{J}}_\phi(e)(j), \pi' = (j \mapsto n')$

$\phi, \eta \vDash^{\mathcal{J}} e : \mathsf{ri}$ I.H.

$\mathcal{V}^{\mathcal{J}}_\phi(e) : \mathsf{ri}$ Prop. 3.2.9

for all n

$\phi[a \mapsto (j \mapsto n_1)], \eta \vDash^{\mathcal{J}} [a/x]\, p \mathbin{@} t$ for all $n_1 < n$ Hyp.

let $\pi = (j \mapsto n)$

$\pi : \mathsf{ri}$ — Def. ri
$\mathcal{V}^{\mathcal{J}}_{\phi[a\mapsto\pi]}(a) : \mathsf{ri}$ — Def. $\mathcal{V}^{\mathcal{J}}_{\phi}$
$\phi[a \mapsto \pi], \eta \vDash^{\mathcal{J}} a\!:\!\mathsf{ri}$ — Prop. 3.2.9
let $a_1 \notin \mathcal{A}(p) \cup \{a\}$
for all $\pi_1 : \mathsf{ri}$
  $\phi[a \mapsto \pi][a_1 \mapsto \pi_1], \eta \vDash^{\mathcal{J}} \mathtt{ltuw}(a_1, a) \,@\, t$ — Hyp.
  $\mathcal{V}^{\mathcal{J}}_{\phi[a\mapsto\pi][a_1\mapsto\pi_1]}(a_1) = \pi_1$ — Def. $\mathcal{V}^{\mathcal{J}}_{\phi}$
  $\mathcal{V}^{\mathcal{J}}_{\phi[a\mapsto\pi][a_1\mapsto\pi_1]}(a) = \pi$ — Def. $\mathcal{V}^{\mathcal{J}}_{\phi}$
  $\pi_1(j) < \pi(j)$ — Def. $\vDash$
  let $n_1 = \pi_1(j)$
  $n_1 < n$
  $\phi[a \mapsto \pi_1], \eta \vDash^{\mathcal{J}} [a/x]\, p \,@\, t$ — Def. ri
  $\phi[a_1 \mapsto \pi_1], \eta \vDash^{\mathcal{J}} [a_1/x]\, p \,@\, t$ — Prop. 3.2.14
  $\phi[a \mapsto \pi][a_1 \mapsto \pi_1], \eta \vDash^{\mathcal{J}} [a_1/x]\, p \,@\, t$ — Prop. 3.2.16
$\phi[a \mapsto \pi], \eta \vDash^{\mathcal{J}} \forall x\!:\!\mathsf{ri}.\, \mathtt{ltuw}(x, a) \supset p \,@\, t$ — Def. $\vDash$
$\phi[a \mapsto \pi], \eta \vDash^{\mathcal{J}} \Gamma,\ a\!:\!\mathsf{ri},\ \forall x\!:\!\mathsf{ri}.\, \mathtt{ltuw}(x, a) \supset p \,@\, t$ — Def. $\vDash$
$\phi[a \mapsto \pi], \eta \vDash^{\mathcal{J}} [a/x]\, p \,@\, t$ — I.H.
$\phi[a \mapsto \pi'], \eta \vDash^{\mathcal{J}} [a/x]\, p \,@\, t$ — induction
$\mathcal{V}^{\mathcal{J}}_{\phi}(e) = \pi'$ — Def. ri
$\phi, \eta \vDash^{\mathcal{J}} [e/x]\, p \,@\, t$ — Prop. 3.2.19
other cases are similar

$\square$

Finally, I show that the proof system including the transition rules is sound for any environment that erases to a valid execution:

Proposition 4.5.2 (Soundness) $\phi, \eta \vDash^{\mathcal{J}} J$
if $\phi, \eta \vDash^{\mathcal{J}} \Gamma$ *and* $\Gamma \vdash^{\mathcal{J}}_{\mathsf{x86+}} J$ *and* $\phi|_{Reg} \in \Sigma_{\mathcal{J}(\mathtt{pm})}$

PROOF:
by induction on the derivation of $\Gamma \vdash^{\mathcal{J}}_{\mathsf{x86+}} J$, using the proof of Proposition 4.5.1
$\phi, \eta \vDash^{\mathcal{J}} \Gamma \qquad \Gamma \vdash^{\mathcal{J}}_{\mathsf{x86+}} J \qquad \phi|_{Reg} \in \Sigma_{\mathcal{J}(\mathtt{pm})}$ — Prem.
let $\mathcal{D}$ be the derivation of $\Gamma \vdash^{\mathcal{J}}_{\mathsf{x86+}} J$
case: $J = \forall x\!:\!\mathsf{ri}.\, \forall x_{\mathsf{l}}\!:\!\mathsf{ri}.\, x = \mathtt{ss} \supset \mathtt{fetch}(\mathtt{pm}, \mathtt{s\_pc}(x), x_{\mathsf{l}}) \supset \bigcirc(\mathtt{pc} = \mathtt{nextpc}(x, x_{\mathsf{l}})) \,@\, t$,

$$\mathcal{D} = \frac{}{\Gamma \vdash J}\ \mathsf{nextpc\_fetch}$$

 let $j = \mathcal{V}_{\eta}(t), \sigma = \phi|_{Reg}, \Phi = \mathcal{J}(\mathtt{pm}), s = \sigma_j, s' = \sigma_{j+1}$
 $\Phi \rhd s \to s'$ — Def. Σ_{Φ}
 let $a, a_{\mathsf{l}} \notin Reg$ such that $a \neq a_{\mathsf{l}}$
 for all $\pi : \mathsf{ri}, \pi_{\mathsf{l}} : \mathsf{ri}$
  let $\phi_{\mathsf{l}} = \phi[a \mapsto \pi][a_{\mathsf{l}} \mapsto \pi_{\mathsf{l}}]$
  $\phi_{\mathsf{l}}, \eta \vDash^{\mathcal{J}} a = \mathtt{ss} \,@\, t \qquad \phi_{\mathsf{l}}, \eta \vDash^{\mathcal{J}} \mathtt{fetch}(\mathtt{pm}, \mathtt{s\_pc}(a), a_{\mathsf{l}}) \,@\, t$ — Hyp.
  $\mathcal{V}^{\mathcal{J}}_{\phi_{\mathsf{l}}}(a)(j) = \mathcal{V}^{\mathcal{J}}_{\phi_{\mathsf{l}}}(\mathtt{ss})(j)$ — Def. $\vDash$
  $= \mathcal{V}^{\mathcal{J}}_{\phi}(\mathtt{ss})(j)$ — Prop. 3.2.5
  $\pi(j) = \langle \phi(\mathtt{pc})(j), \phi(\mathtt{f})(j), \phi(\mathtt{g})(j), \phi(\mathtt{s})(j), \phi(\mathtt{m})(j) \rangle$ — Def. $\mathcal{V}^{\mathcal{J}}_{\phi}$
  $= \langle s(\mathtt{pc}), s(\mathtt{f}), s(\mathtt{g}), s(\mathtt{s}), s(\mathtt{m}) \rangle$ — Def. $\phi|_{Reg}$

$$
\begin{array}{ll}
\quad \underline{\texttt{fetch}}(\mathcal{V}^{\mathcal{J}}_{\phi_\mathsf{l}}(\texttt{pm})(j), \mathcal{V}^{\mathcal{J}}_{\phi_\mathsf{l}}(\texttt{s\_pc}(a))(j), \mathcal{V}^{\mathcal{J}}_{\phi_\mathsf{l}}(a_\mathsf{l})(j)) & \text{Def. } \models \\
\quad \underline{\texttt{fetch}}(\Phi, s(\texttt{pc}), \pi_\mathsf{l}(j)) & \text{Def. } \mathcal{V}^{\mathcal{J}}_{\phi} \\
\quad s(\texttt{pc}) \in \operatorname{dom}\Phi \qquad \Phi(s(\texttt{pc})) = \pi_\mathsf{l}(j) & \text{Def. } \texttt{fetch} \\
\quad \underline{\texttt{nextpc}}(\pi(j), \Phi(s(\texttt{pc}))) = s'(\texttt{pc}) & \text{Def. } \Phi \rhd s \to s' \\
\quad = \phi_\mathsf{l}(\texttt{pc})(j+1) & \text{Def. } \phi|_{Reg} \\
\quad \phi_\mathsf{l}(\texttt{pc})(j+1) = \underline{\texttt{nextpc}}(\pi(j), \pi_\mathsf{l}(j)) & \\
\quad = \underline{\texttt{nextpc}}(\pi(j+1), \pi_\mathsf{l}(j+1)) & \text{Def. } \mathsf{ri} \\
\quad = \underline{\texttt{nextpc}}(\phi_\mathsf{l}(a)(j+1), \phi_\mathsf{l}(a_\mathsf{l})(j+1)) & \\
\quad \mathcal{V}^{\mathcal{J}}_{\phi_\mathsf{l}}(\texttt{pc})(j+1) = \underline{\texttt{nextpc}}(\mathcal{V}^{\mathcal{J}}_{\phi_\mathsf{l}}(a)(j+1), \mathcal{V}^{\mathcal{J}}_{\phi_\mathsf{l}}(a_\mathsf{l})(j+1)) & \text{Def. } \mathcal{V}^{\mathcal{J}}_{\phi} \\
\quad \mathcal{V}_{\eta[b \mapsto j+1]}(b) = j+1 & \text{Def. } \mathcal{V}_{\eta} \\
\quad \phi_\mathsf{l}, \eta[b \mapsto j+1] \models^{\mathcal{J}} \texttt{pc} = \texttt{nextpc}(a, a_\mathsf{l}) \mathbin{@} b & \text{Def. } \models \\
\quad \phi_\mathsf{l}, \eta \models^{\mathcal{J}} \bigcirc(\texttt{pc} = \texttt{nextpc}(a, a_\mathsf{l})) \mathbin{@} t & \text{Def. } \models \\
\phi, \eta \models^{\mathcal{J}} J & \text{Def. } \models
\end{array}
$$

other cases are similar

$\square$

Chapter 5

Security Policies

I now address the principal concern of the code consumer: "how do I tell if my system is secure when I execute an untrusted program?"

In this chapter, I will focus principally on ensuring that each access to a memory location is properly aligned, and that it respects an access mode assigned to the location by the run-time system. This property is known as *memory safety*.

I choose to focus on memory safety for the following reasons:

- Memory safety is an important safety property in and of itself for ensuring that run-time data structures are not compromised or exposed.
- A formal encoding of memory safety provides a concrete evidence that the temporal-logic security-policy notation introduced in Chapter 3 is adequate for specifying at least one nontrivial safety property.
- Current certifying compilers are developed with memory safety as the principal target safety property—in order to take advantage of such automatic certification tools, the code consumer must be able to enforce memory safety.

Thus, this chapter supports my thesis statement by demonstrating that temporal logic is in fact a practical notation for specifying at least on nontrivial security policy that is used in current certified-code systems.

5.1 Overview

The purpose of this chapter to is fully develop a formal security policy for memory safety that ensures that access to memory is restricted to a prescribed set of safe operations. The formal memory-safety property is designed to ensure that three informal properties hold:

Control-flow safety The program counter is restricted to memory locations with defined instructions (except when the program makes a call to a specific run-time procedure provided by the code consumer). This property ensures that the untrusted program will not try to read invalid addresses or execute undefined instructions.

precond.: $\texttt{js\_mem(m)} \wedge \texttt{selg(g,}\overline{\texttt{eax}}\texttt{)} : \texttt{array(int)}$	$\Rightarrow$ `eax` is aligned
$\vdots$	$\Downarrow$
$\texttt{ebx} \leftarrow 24(\texttt{eax})$	$\texttt{eax} \dotplus 24$ is aligned
$\vdots$	
`ret`	
postcond.: $\texttt{js\_mem(m)}$	heap is still o.k.

Figure 5.1: Type Safety Implies Memory Safety

Stack continuity The stack is only accessed based on small positive offsets from the current stack pointer, and the stack grows contiguously downwards in small increments. This property ensures that accesses to the stack do not corrupt other parts of memory, and that stack overflow can be detected by a run-time mechanism.

Heap integrity The heap is only accessed through properly aligned addresses, and each such access respects a read/write access mode assigned to the address by the code consumer. Additionally, when data structures provided by the code consumer are modified, the program preserves certain structural invariants that are defined by a type system. This property ensures that the untrusted program cannot read sensitive parts of the heap, and that the data structures of the code consumer in the heap will not be corrupted when the untrusted program is run.

Thus, the memory safety policy is designed to ensure system integrity when the untrusted program shares an address space with other software used by the code consumer. Recall that the machine model is based on a "segmented" memory in which the program, stack, and heap are mapped to distinct address spaces. However, the memory-safety policy can also be applied to systems that have a "flat" address space—I outline how such an arrangement might be realized in Section 10.2.2.

The formal memory-safety property is comprised of alignment constraints and access modes for memory addresses that ensure that running the untrusted program will not result in a processor exception. This property is satisfied indirectly, however, by first satisfying a higher-level type-safety property that additionally guarantees that certain internal structural invariants hold for the current heap. This strategy is illustrated in Figure 5.1 for a procedure that loads a single element from offset 24 of an array. Not only does this ensure that future memory accesses will continue to respect the low-level memory-safety property (*i.e.*, because the heap is still internally consistent), it also ensures that data structures that are visible to both the code consumer and the untrusted program are mutually consistent.[1] To a certain

[1] This use of type safety to share complex data structures is in contrast to the viewpoint taken by foundational PCC (F-PCC) [App01, AF00], in which the type system is effectively invented by the code producer, and is thus unavailable to the code consumer for the purpose of developing procedure specifications.

extent, ensuring heap consistency is actually more important than preventing processor exceptions. Many instruction-set architectures provide a facility for handling run-time exceptions, but heap corruption is much harder to prevent with a run-time check.

In order to formalize a memory-safety policy, I first develop an abstraction of memory-management hardware. In particular, the code consumer is able to stipulate that a given address is "readable" or "writable" when an instruction is executed. Instead of including the abstract memory manager in the abstract machine model, I prefer to include it in the security-policy specification. This approach has the notable advantage of making it possible to specify several distinct memory-safety policies for a given machine model. However, including the memory manager in the abstract machine model is also a defensible choice, but I will not explore it further in this dissertation.

I need to be able to track the abstract state of the memory manager as it evolves over time. Security automata [Sch99] (see Section 2.1.3.4) provide a systematic framework for the representation of such states, in addition to being an attractive notation for specifying "expressive" safety policies. Security automata will thus play a key role in specifying the memory-safety policy by providing "history registers" [MP95] to track the abstract memory manager, among other things. Essentially, access constraints on the heap are modeled as a map from heap addresses to access modes that is updated as new blocks are allocated from the heap. Additionally, as part of the specification of type safety, the interpretation of each type is explicitly modeled as a representation function (assigning sets of values to types) that is refined as the program executes.[2] Finally, security automata provide an expedient mechanism for ensuring stack continuity by tracking the base of the current stack frame.

Because security automata are so important to specifying memory safety and type safety policies, I develop a systematic formal representation of automaton states in Section 5.4. This representation is additionally suitable for encoding expressive safety policies such as resource bounds, access control, and confidentiality (see Bernard and Lee [BL01] for concrete examples).

In order to formalize the memory-safety policy in temporal logic, certain additional low-level data types are needed. These data types are introduced in Section 5.5 as a simple lattice of access modes and maps from addresses to access modes. Given this theory, the memory-safety policy itself can be encoded—this is the subject of Section 5.6, in which the essential memory-safety relations are defined, and in which I prove the key theorems that establish control-flow safety and heap integrity.

As I stated earlier in this section, the heap-integrity policy is not established directly—rather, a stronger property of *Java type safety* [GJS96, LY99, CLN$^+$00] is demonstrated, and the representation invariants of this type system are used to show that the heap-integrity policy is indirectly satisfied. This approach also gives the code consumer the opportunity to pass typed data structures to the untrusted

[2]This approach to type safety was originally studied by Necula [Nec98], but the representation function is notably *not* assigned to a named register in conventional PCC implementations.

program as procedure parameters or global variables. The formal representation of the Java type system is developed in Section 5.7. This representation follows an approach developed by Necula [Nec98] for encoding a programming-language type system in logical notation. However, I deconstruct this encoding further by defining the type environment and memory validity constraints syntactically, thus exposing the precise low-level invariants that must be preserved by the heap allocation procedures. The informal operational semantics for the type system is therefore much simpler than in previous work, and could itself be derived in a more expressive framework such as higher-order logic—perhaps paving the way to a more foundational treatment of type safety in SpecialJ.

Based on the material in Section 5.7, I can *derive* the typing rules that are actually used in safety proofs constructed by the SpecialJ compiler. This is the subject of Section 5.8, in which the typing rules are presented in terms of the low-level encoding of the Java type system. Finally, the derivability and soundness theorems for the inference rules introduced in this chapter are presented in Section 5.9.

I now proceed with a study of enforcement in my PCC system.

5.2 Enforcement

Conventional PCC enforcement mechanisms are implemented in the C programming language and generate a *verification condition* [Kin71] (VC) that is true only if the program does not violate the security policy. An LF type checker establishes that the security proof is a correct proof of the VC. I argue in an earlier report [BL01] that the VC generator should interpret a security policy specification instead of "hard coding" a security policy. However, temporal logic is expressive enough to encode security properties directly—therefore, no special interpreter is needed.

For temporal-logic PCC, the code producer provides a proof of

$$\vdash p_{\mathsf{sp}} \mathbin{@} 0$$

instead of a proof of a VC. p_{sp} is a *security property* that must hold for the system to be secure.[3] p_{sp} is specified by the code consumer directly. The definition of satisfaction can be used to verify that it has the intended meaning.

Contrast this approach with a conventional PCC system, in which the code producer proves a VC derived from the security property by a trusted analysis. In my system, the code producer proves the security property directly from a formal encoding of the abstract machine model (compare Figure 5.2 with Figure 5.3). To show that my enforcement mechanism is sound, I need only show that the machine-model encoding is valid (see Section 4.5).

Note that the standard connectives of temporal logic allow one to combine security properties in a modular way. For example, $\Sigma_{p_1 \wedge p_2}$ is equal to $\Sigma_{p_1} \cap \Sigma_{p_2}$ (*i.e.*, the program must simultaneously satisfy both p_1 and p_2). Disjunction can similarly be interpreted as the union of execution sets (*i.e.*, the code producer can choose which of two possible security properties to satisfy). Additionally, one can

[3]Recall that a security property is a security policy that corresponds to an execution set.

$$\boxed{\begin{array}{l}\text{code consumer computes } VC_{\mathsf{sp}}(\Phi) = \forall x_1.\ldots \supset x_1 : \mathtt{array(int)} \wedge \ldots \\ VC_{\mathsf{sp}}(\Phi) \text{ implies } p_{\mathsf{sp}} \text{ (by an informal argument)}\end{array}}$$

$$\frac{\begin{array}{c}\text{typing rules}\\ \vdots\end{array}}{\vdash \forall x_1.\ldots \supset x_1 : \mathtt{array(int)} \wedge \ldots}$$

Figure 5.2: Safety Proof (Conventional PCC)

universally quantify a *flexible* history parameter (such as `n` in the instruction bound example from Section 2.2.1) to specify that the parameter is local to a given security property. This quantification ensures that the parameter will not be interpreted inconsistently when the security property is combined with other security properties. These properties, among others, make temporal logic an excellent foundation for a security-property language.

5.3 Soundness of Enforcement

Let p_{sp} be a security property. The following proposition establishes that the system is secure with respect to any program that has a security proof:

Proposition 5.3.1 (Soundness) $\Sigma_{\mathcal{J}(\mathtt{pm})} \subseteq \Sigma_{p_{\mathsf{sp}}}$ *if* $\vdash^{\mathcal{J}} p_{\mathsf{sp}} @ 0$

PROOF:
for all $\sigma \in \Sigma_{\mathcal{J}(\mathtt{pm})}$
  for all ϕ such that $\phi|_{Reg} = \sigma$
    $\phi, \eta \models^{\mathcal{J}} p_{\mathsf{sp}} @ 0$ — Proposition 4.5.2
  $\sigma, 0 \models^{\mathcal{J}} p_{\mathsf{sp}}$ — Def. $\sigma, j \models^{\mathcal{J}} p$
  $\sigma \in \Sigma_{p_{\mathsf{sp}}}$ — Def. Σ_p
□

i.e., if there is a proof of the security property, then the execution set of the program is contained within the execution set of the security property.

Let $\Phi = \mathcal{J}(\mathtt{pm})$. The code producer provides a derivation of

$$\vdash p_{\mathsf{sp}} @ 0$$

$$\frac{\begin{array}{ccc}\text{machine semantics} & & \text{typing rules}\\ \vdots & & \vdots \\ & \vdots & \end{array}}{\vdash p_{\mathsf{sp}} @ 0}$$

Figure 5.3: Safety Proof (Temporal-Logic PCC)

along with Φ. The code consumer uses a trusted proof checker to verify the correctness of the derivation. From Proposition 5.3.1, the code consumer can conclude $\Sigma_\Phi \subseteq \Sigma_{p_{\mathsf{sp}}}$: no execution of Φ violates p_{sp}.

5.4 Security Automata

In Section 2.1.3, I introduced security automata as a universal representation for safety properties. In this section, I define a formal representation for security automata that facilitates the specification of new safety properties.

Following Schneider [Sch99], a *security automaton* is a set of states (including a distinguished "bad" state), plus a transition function that specifies how the automaton changes from state to state. The current machine state is also a parameter of the transition function, and a given execution is safe if and only if the security automaton never reaches the bad state. In my formal encoding, I depart from the standard security-automaton model slightly by allowing any state to be designated a bad state. Essentially, I provide a generic representation for history registers [MP95], then explicitly specify which states of the generic representation are safe.

Let `sa` be the type of security-automaton states. These states are inherently composite values, mapping each element of an unspecified finite set of *security registers* to a value. $\mathtt{sreg}\langle\tau\rangle$ is the type of security registers that have values of type τ. By convention, the parameter $\mathtt{q}^{\mathsf{sa}}$ holds the current state of the security automaton.[4]

It is important to note that because `q` is *not* part of the machine model, it need not be implemented with any concrete representation. In fact, `q` can be viewed as a fictional or "ghost" state that only exists for the purpose of specifying security properties. The state of `q` might coincidentally correspond to some particular portion of the concrete machine state, but it not necessary to provide it with any actual run-time representation. This is an important departure from other implementations of security automata such as SASI [ES99, ES00] that necessarily associate the security automaton with a concrete run-time representation. As I point out in Section 10.2, however, instrumentation tools such as SASI are still potentially useful to the code *producer* for constructing certified code.

Giving the security registers first-class status for procedure specifications enables a relatively high degree of expressiveness when specifying safety properties. For example, one can require that a given resource bound be unchanged by a procedure, or that the bound stay within some specified limit relative to its value at the start of the procedure. Note that a liveness or termination property is also needed in conjunction with this kind of specification to ensure that the postcondition is actually reached at some point.

[4] Note that `q` is not a register parameter, and is thus not assigned a value by program states (*i.e.*, `q` is "erased," along with all other non-register parameters). However, the state of `q` can affect which executions are allowed by a given security property according to the definition of satisfaction for executions (see Section 4.3).

$$
\begin{array}{ll}
\mathtt{q\_mk}(e_{\mathtt{pc}}, e_{\mathtt{f}}, e_{\mathtt{g}}, e_{\mathtt{s}}, e_{\mathtt{m}}, e_{\mathtt{q}}) & \stackrel{\mathsf{def}}{=} \mathtt{mkp}(\mathtt{s\_mk}(e_{\mathtt{pc}}, e_{\mathtt{f}}, e_{\mathtt{g}}, e_{\mathtt{s}}, e_{\mathtt{m}}), e_{\mathtt{q}}) \\
\mathtt{q\_pc}(e) & \stackrel{\mathsf{def}}{=} \mathtt{s\_pc}(\mathtt{left}(e)) \\
\mathtt{q\_f}(e) & \stackrel{\mathsf{def}}{=} \mathtt{s\_f}(\mathtt{left}(e)) \\
\mathtt{q\_g}(e) & \stackrel{\mathsf{def}}{=} \mathtt{s\_g}(\mathtt{left}(e)) \\
\mathtt{q\_s}(e) & \stackrel{\mathsf{def}}{=} \mathtt{s\_s}(\mathtt{left}(e)) \\
\mathtt{q\_m}(e) & \stackrel{\mathsf{def}}{=} \mathtt{s\_m}(\mathtt{left}(e)) \\
\mathtt{q\_q}(e) & \stackrel{\mathsf{def}}{=} \mathtt{right}(e) \\
\mathtt{q\_esp}(e) & \stackrel{\mathsf{def}}{=} \mathtt{selg}(\mathtt{q\_g}(e), \overline{\mathtt{esp}}) \\
\mathtt{q\_fps}(e) & \stackrel{\mathsf{def}}{=} \mathtt{sels}(\mathtt{q\_q}(e), \overline{\mathtt{fps}}) \\
\mathtt{q\_accm}(e) & \stackrel{\mathsf{def}}{=} \mathtt{sels}(\mathtt{q\_q}(e), \overline{\mathtt{accm}}) \\
\mathtt{q\_ta}(e) & \stackrel{\mathsf{def}}{=} \mathtt{sels}(\mathtt{q\_q}(e), \overline{\mathtt{ta}}) \\
\mathtt{q\_ts}(e) & \stackrel{\mathsf{def}}{=} \mathtt{mkp}(\mathtt{q\_ta}(e), \mathtt{q\_accm}(e)) \\
\mathtt{q\_next}(e, e_{\mathsf{l}}) & \stackrel{\mathsf{def}}{=} \mathtt{mkp}(\mathtt{s\_next}(\mathtt{left}(e), e_{\mathsf{l}}), \mathtt{nextq}(e, e_{\mathsf{l}}))
\end{array}
$$

Figure 5.4: Operations on Paired States

Two function constants operate on security-automaton states as though they were register files. The function constant $\mathtt{sels}^{\mathtt{sa}\times\mathtt{sreg}\langle\tau\rangle\to\tau}$ selects the value of a given security register from a given security-automaton state. Likewise, the function constant $\mathtt{upds}^{\mathtt{sa}\times\mathtt{sreg}\langle\tau\rangle\times\tau\to\mathtt{sa}}$ redefines the value of a given security register in a given security-automaton state.

In practice, the code consumer chooses a suitable set of security registers to record properties that are of interest to his or her safety property. For example, one security register might count the number of instructions executed, while another might become nonzero once any confidential information is read from the heap. A suitable set of history registers is sufficient to rewrite any safety property into the form of an invariance property [Sch99]. The invariance properties [MP90] are safety properties that have a certain syntactic form in temporal logic notation:

$$\Box p$$

where p contains no temporal operators (*i.e.*, p is a predicate on individual states). The security-automaton framework thus provides the code consumer with a convenient theory that is expressive enough to encompass all safety properties.

It is often useful to pair a state tuple with an explicit security-automaton state. Thus, the type qstate is an abbreviation for the type of pairs consisting of a machine state and a security-automaton state:

$$\mathtt{qstate} \stackrel{\mathsf{def}}{=} \mathtt{pair}\langle\mathtt{state}\rangle\langle\mathtt{sa}\rangle$$

$\texttt{sq}$ is also an abbreviation:

$$\texttt{sq} \stackrel{\mathsf{def}}{=} \texttt{mkp}(\texttt{ss}, \texttt{q})$$

i.e., $\texttt{sq}$ is the pair consisting of $\texttt{ss}$ and $\texttt{q}$. Figure 5.4 contains additional abbreviations for operations on paired states. In this figure, $\texttt{fps}$, $\texttt{accm}$, and $\texttt{ta}$ are security registers that I introduce in Section 5.6 and Section 5.8.

5.4.1 Operational Semantics

The operational semantics of security automata is straightforward.

SReg^τ is an unspecified finite set containing all the security registers of type τ. Security automata are modeled as total functions from security registers to values of the same type:

$$\begin{aligned} \mathit{Val}^{\texttt{sreg}\langle\tau\rangle} &= \mathit{SReg}^\tau \\ \mathit{Val}^{\mathsf{sa}} &= \mathit{SReg} \rightarrow \mathit{Val} \end{aligned}$$

The select and update operations are modeled as function application and pointwise extension, as for register files:

$$\begin{aligned} \underline{\texttt{sels}}(q, v) &= q(v) \\ \underline{\texttt{upds}}(q, v, v') &= q[v \mapsto v'] \end{aligned}$$

5.4.2 Inference Rules

The McCarthy and extensionality rules for security-automaton states are similar to the rules for register files:

$$\frac{}{\Gamma \vdash \texttt{sels}(\texttt{upds}(e_{\mathsf{q}}, e_{\mathsf{r}}, e), e_{\mathsf{r}}) = e \mathbin{@} t}\ \texttt{sels}_{\mathsf{mc0}}$$

$$\frac{\Gamma \vdash e'_{\mathsf{r}} \neq e_{\mathsf{r}} \mathbin{@} t}{\Gamma \vdash \texttt{sels}(\texttt{upds}(e_{\mathsf{q}}, e_{\mathsf{r}}, e), e'_{\mathsf{r}}) = \texttt{sels}(e_{\mathsf{q}}, e'_{\mathsf{r}}) \mathbin{@} t}\ \texttt{sels}_{\mathsf{mc1}}$$

$$\frac{}{\Gamma \vdash \texttt{sels}(\texttt{upds}(e_{\mathsf{q}}, e_{\mathsf{r}}^{\texttt{sreg}\langle\tau\rangle}, e), e_{\mathsf{r}}^{\texttt{sreg}\langle\tau'\rangle}) = \texttt{sels}(e_{\mathsf{q}}, e_{\mathsf{r}}^{\texttt{sreg}\langle\tau'\rangle}) \mathbin{@} t}\ \texttt{sels}_{\mathsf{mc1'}}^{\tau\neq\tau'}$$

$$\frac{\Gamma \vdash \texttt{sels}(e_{\mathsf{q}}, a_{\mathsf{r}}) = \texttt{sels}(e'_{\mathsf{q}}, a_{\mathsf{r}}) \mathbin{@} t}{\Gamma \vdash e_{\mathsf{q}} = e'_{\mathsf{q}} \mathbin{@} t}\ \texttt{sels}_{\mathsf{ext}}^{a_{\mathsf{r}}}$$

The rule $\texttt{sels}_{\mathsf{mc1'}}$ is needed to distinguish security registers of different types (inequality is only defined for expressions of the same type). The meta-theoretic side condition requires the types at which the rule is instantiated to be distinct. In practice, the enforcement mechanism simply instantiates the rule at each distinct pair of types that has an associated security register.

5.5 Memory Access

In this section, I introduce data types that enable me to model the constraints on memory accesses that might be imposed on the code producer by an operating system or run-time system.

`acc` is the type of memory access modes. These modes (`nacc`, `rd`, `wr`, `rw`) are partially ordered according to whether read or write access is permitted by the mode:

`nacc`	No access
`rd`	Read-only access
`wr`	Write-only access
`rw`	Read-write access

`mapa` is the type of access-mode maps. A value of this type maps each machine-word address to an access mode according to whether or not that address is readable or writable.

The following functions and relations operate on access modes:

$\mathtt{leqa}^{\mathtt{acc}\times\mathtt{acc}\to o}$	Partial order relation for access modes
$\mathtt{sela}^{\mathtt{mapa}\times\mathtt{wd}\times\mathtt{wd}\to\mathtt{acc}}$	Select access mode for address and length

The result of $\underline{\mathtt{sela}}(v, n_1, n_2)$ is the greatest access mode (see Section 5.5.1) that is less than or equal to the access mode of each addresses in the range $n_1, \ldots, n_1+n_2-1$. Note that my encoding of the memory access policy does not require an "update" operation on access-mode maps.

$\mathtt{leqam}^{\mathtt{mapa}\times\mathtt{mapa}\to o}$ is a defined relation that is the point-wise partial order on access-mode maps:

$$\mathtt{leqam}(e_{\mathsf{m}_1}, e_{\mathsf{m}_2}) \stackrel{\mathsf{def}}{\equiv} \forall x_{\mathsf{n}_1}\!:\!\mathsf{fl}.\, \forall x_{\mathsf{n}_2}\!:\!\mathsf{fl}.\, \mathtt{leqa}(\mathtt{sela}(e_{\mathsf{m}_1}, x_{\mathsf{n}_1}, x_{\mathsf{n}_2}), \mathtt{sela}(e_{\mathsf{m}_2}, x_{\mathsf{n}_1}, x_{\mathsf{n}_2}))$$

5.5.1 Operational Semantics

The type `acc` is modeled as a finite set, whereas the type `mapa` is modeled as a set of total functions:

$$\begin{aligned} Val^{\mathtt{acc}} &= \{\mathtt{nacc}, \mathtt{rd}, \mathtt{wr}, \mathtt{rw}\} \\ Val^{\mathtt{mapa}} &= Val^{\mathtt{wd}} \to Val^{\mathtt{acc}} \end{aligned}$$

The formal definitions of `leqa` and `sela` follow from the informal understanding:

$$\underline{\mathtt{leqa}}(v_1, v_2) \quad \text{iff} \quad v_1 = v_2 \text{ or } v_1 = \mathtt{nacc} \text{ or } v_2 = \mathtt{rw}$$

$$\begin{aligned} \underline{\mathtt{sela}}(v, n_1, n_2) = \quad & v_{\mathsf{acc}} \text{ such that } v_{\mathsf{acc}} \leq v_{n_1,n_2} \\ & \text{and } \underline{\mathtt{leqa}}(v'_{\mathsf{acc}}, v_{\mathsf{acc}}) \text{ for all } v'_{\mathsf{acc}} \leq v_{n_1,n_2} \end{aligned}$$

where I write $v_{\mathsf{acc}} \leq v_{n_1,n_2}$ when $\underline{\mathtt{leqa}}(v_{\mathsf{acc}}, v(n))$ for all n such that $n_1 \leq n < n_1 \dot{+} n_2$.

5.5.2 Inference Rules

The following inference rules encode the algebraic properties of `leqa`:

$$\frac{}{\Gamma \vdash \texttt{order}(\texttt{leqa}) \mathbin{@} t}\ \textsf{leqaorder}$$

$$\frac{}{\Gamma \vdash \texttt{bot}(\texttt{leqa}, \overline{\texttt{nacc}}) \mathbin{@} t}\ \textsf{leqabot} \qquad \frac{}{\Gamma \vdash \texttt{top}(\texttt{leqa}, \overline{\texttt{rw}}) \mathbin{@} t}\ \textsf{leqatop}$$

There are no inference rules for `sela`.

5.6 Memory Safety

I can now formalize the memory safety policy.

I first introduce two security registers. The security register $\texttt{fps} \in \mathit{SReg}^{\texttt{list}\langle\texttt{wd}\rangle}$ contains the current stack of frame pointers. This stack consists of all the frame pointers of procedures that have been called, but have not yet returned. The security register $\texttt{accm} \in \mathit{SReg}^{\texttt{mapa}}$ is the current access-mode map. It assigns an access mode to each possible memory address. Note that the access-mode granularity of a typical operating system or run-time system is likely to be much coarser than individual addresses (*e.g.*, whole pages), but this need not be reflected in the formalization of the memory-safety policy.

I use the following functions and relations to encode the memory safety policy:

$\texttt{nextq}^{\texttt{qstate}\times\texttt{inst}\to\texttt{sa}}$	Next security-automaton state for instruction
$\texttt{safe\_sp}^{\texttt{qstate}\times\texttt{wd}\to o}$	Stack-pointer address is safe for state
$\texttt{safe\_rds}^{\texttt{qstate}\times\texttt{wd}\to o}$	Stack address is safe to read
$\texttt{safe\_wrs}^{\texttt{qstate}\times\texttt{wd}\times\texttt{wd}\to o}$	Stack address is safe to write with value
$\texttt{safe\_rdm}^{\texttt{qstate}\times\texttt{wd}\to o}$	Memory address is safe to read
$\texttt{safe\_wrm}^{\texttt{qstate}\times\texttt{wd}\times\texttt{wd}\to o}$	Memory address is safe to write with value
$\texttt{safe\_rdea}^{\texttt{qstate}\times\texttt{ea}\to o}$	Effective address is safe to read
$\texttt{safe\_wrea}^{\texttt{qstate}\times\texttt{ea}\times\texttt{wd}\to o}$	Effective address is safe to write with value
$\texttt{safe\_inst}^{\texttt{qstate}\times\texttt{inst}\to o}$	Instruction is safe to execute in state

Recall that $\tau_1 \times \cdots \times \tau_k \to o$ is the type annotation for a relation on parameters of types $\tau_1, \ldots, \tau_k$.

The memory-safety policy for a procedure is specified in Figure 5.5. This safety policy is imposed on the initial entry point, which is called directly by the code consumer to execute the untrusted program. This safety policy imposes a particular calling convention on the interface between the code consumer and code producer, but this calling convention need not be used by the code producer for any other procedures.

e_{pc_0} is the address of the first instruction of the procedure. e_{esp_0} is the stack pointer at the time the procedure is called. p_{pre_0} is the specific precondition of the procedure, ordinarily asserting that particular registers and stack slots have particular types. p_{post_0} is similarly the specific postcondition of the procedure. p_{cs_0}

$$
\begin{array}{lll}
\multicolumn{3}{l}{\Box(\forall x_{\mathsf{sq}_0}\!:\!\mathsf{ri}.\ \mathtt{sq} = x_{\mathsf{sq}_0} \supset p_{\mathsf{pre}}(x_{\mathsf{sq}_0}) \supset \mathtt{safe}\,\mathcal{U}\,p_{\mathsf{post}}(x_{\mathsf{sq}_0}))} \\
\text{where } \mathtt{safe} & \stackrel{\mathsf{def}}{\equiv} & \exists x_{\mathsf{I}}\!:\!\mathsf{fl}.\ \mathtt{fetch}(\mathtt{pm}, \mathtt{pc}, x_{\mathsf{I}}) \wedge \mathtt{safe\_inst}(\mathtt{sq}, x_{\mathsf{I}}) \\
p_{\mathsf{pre}}(x_{\mathsf{sq}_0}) & \stackrel{\mathsf{def}}{\equiv} & \quad \mathtt{pc} = e_{\mathtt{pc}_0} \\
& & \wedge\ \mathtt{gsp} = e_{\mathtt{esp}_0} \\
& & \wedge\ (p_{\mathsf{pre}_0} \\
& & \wedge\ (\mathtt{sels}(\mathtt{q}, \overline{\mathtt{fps}}) = \mathtt{cons}(\mathtt{q\_esp}(x_{\mathsf{sq}_0}), \mathtt{tail}(\mathtt{q\_fps}(x_{\mathsf{sq}_0}))) \\
& & \wedge\ \top)) \\
p_{\mathsf{post}}(x_{\mathsf{sq}_0}) & \stackrel{\mathsf{def}}{\equiv} & \quad \mathtt{pc} = \mathtt{selw}(\mathtt{q\_s}(x_{\mathsf{sq}_0}), \mathtt{q\_esp}(x_{\mathsf{sq}_0})) \\
& & \wedge\ (p_{\mathsf{post}_0} \\
& & \wedge\ (\mathtt{gsp} = \mathtt{addw}(\mathtt{q\_esp}(x_{\mathsf{sq}_0}), 4) \\
& & \wedge\ (\mathtt{s} = \mathtt{joinw}(\mathtt{s}, \mathtt{addw}(\mathtt{q\_esp}(x_{\mathsf{sq}_0}), e_{\mathsf{ncs}_0}), \mathtt{q\_s}(x_{\mathsf{sq}_0})) \\
& & \wedge\ (\mathtt{sels}(\mathtt{q}, \overline{\mathtt{fps}}) = \mathtt{tail}(\mathtt{q\_fps}(x_{\mathsf{sq}_0})) \\
& & \wedge\ p_{\mathsf{cs}_0}))))
\end{array}
$$

Figure 5.5: Memory Safety for a Procedure

is the callee-save register set of the procedure, asserting that particular registers and stack slots are unchanged by the procedure. This proposition is a list of equalities that relate registers in $\mathtt{sq}$ at the postcondition to corresponding registers in $x_{\mathtt{sq}}$. e_{ncs_0} is the least offset from the stack pointer above which the entire stack is unchanged by the procedure.

The safety specification for a procedure requires that once the precondition holds, the instruction-level memory-safety property $\mathtt{safe}$ must hold until the postcondition holds. $\mathtt{safe}$ requires that there be some instruction I at the current program counter, and that I be safe according to the definition of safety for an individual instruction. In addition to the specific precondition for the procedure, the general precondition p_{pre} requires that the current stack pointer be on the top of the frame-pointer stack $\mathtt{fps}$. The general postcondition p_{post} ensures that the program counter is set to the address on the top of the stack at the time the procedure was called. Additionally, the stack pointer will be one word higher than when the procedure was called, the stack contents will be preserved above this address, and the frame-pointer stack will be popped to reflect the loss of the current frame pointer.[5]

To elaborate on the stack-preservation requirement, the proposition

$$\mathtt{q\_s}(\mathtt{sq}) = \mathtt{joinw}(\mathtt{q\_s}(\mathtt{sq}), \mathtt{addw}(\mathtt{q\_esp}(x_{\mathsf{sq}_0}), e_{\mathsf{ncs}_0}), \mathtt{q\_s}(x_{\mathsf{sq}_0}))$$

stipulates that the current contents of the stack ($\mathtt{q\_s}(\mathtt{sq})$) be equal to the contents of the stack at the time the procedure was called ($\mathtt{q\_s}(x_{\mathsf{sq}_0})$) above the stack pointer in effect at the time of the call ($\mathtt{q\_esp}(x_{\mathsf{sq}_0})$), biased by the callee-save offset (e_{ncs_0}).

[5]Note that because the frame-pointer stack is held in a security register, it need not given a concrete representation at run time.

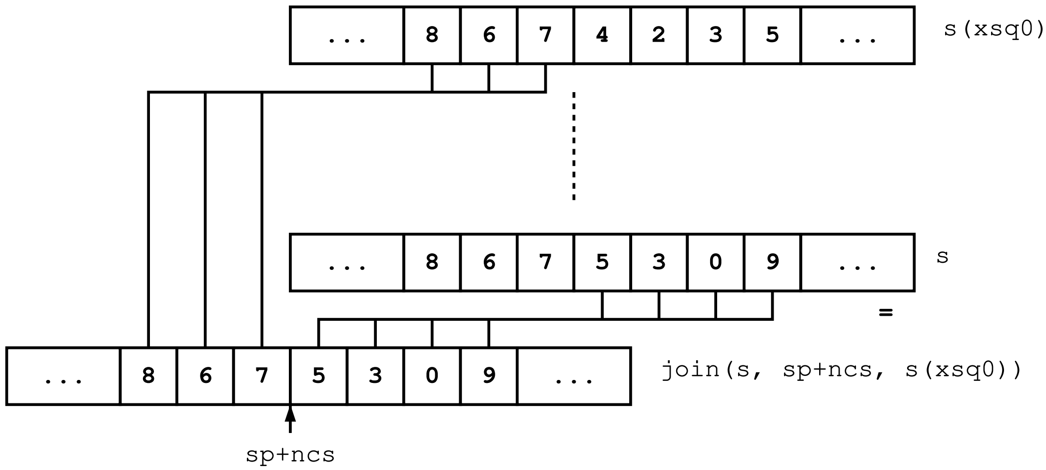

Figure 5.6: Stack Preservation

The current contents of the stack need only be equal to *itself* below the stack pointer (plus the callee-save offset), and thus can be modified arbitrarily by callee throughout this region (see Figure 5.6).

5.6.1 Operational Semantics

In this section, I give a formal semantics for the memory-safety relations that embodies a full formalization of the memory-safety policy.

$\underline{\texttt{nextq}}(v, I)$ determines the next security-automaton state according to the current state tuple v and current instruction I:

I	$\underline{\texttt{nextq}}(v, I)$ where $v = \langle\langle n_{\texttt{pc}}, n_{\texttt{f}}, v_{\texttt{g}}, v_{\texttt{s}}, v_{\texttt{m}}\rangle, v_{\texttt{q}}\rangle$
$\texttt{mov}\langle n_{\mathsf{i}}\rangle\langle ea_1\rangle\langle ea_2\rangle$	$v_{\texttt{q}}$
$\texttt{xchg}\langle n_{\mathsf{i}}\rangle\langle ea\rangle\langle r\rangle$	$v_{\texttt{q}}$
$\texttt{lea}\langle n_{\mathsf{i}}\rangle\langle ea\rangle\langle r\rangle$	$v_{\texttt{q}}$
$\texttt{push}\langle n_{\mathsf{i}}\rangle\langle ea\rangle$	$v_{\texttt{q}}$
$\texttt{pop}\langle n_{\mathsf{i}}\rangle\langle ea\rangle$	$v_{\texttt{q}}$
$\texttt{op1}\langle n_{\mathsf{i}}\rangle\langle \mathit{op1}\rangle\langle ea\rangle$	$v_{\texttt{q}}$
$\texttt{op2}\langle n_{\mathsf{i}}\rangle\langle \mathit{op2}\rangle\langle ea_1\rangle\langle ea_2\rangle$	$v_{\texttt{q}}$
$\texttt{op2n}\langle n_{\mathsf{i}}\rangle\langle \mathit{op2}\rangle\langle ea_1\rangle\langle ea_2\rangle$	$v_{\texttt{q}}$
$\texttt{op3}\langle n_{\mathsf{i}}\rangle\langle \mathit{op3}_1\rangle\langle \mathit{op3}_2\rangle\langle ea\rangle\langle r_1\rangle\langle r_2\rangle$	$v_{\texttt{q}}$
$\texttt{jmp}\langle n_{\mathsf{i}}\rangle\langle ea\rangle$	$v_{\texttt{q}}$
$\texttt{j}\langle n_{\mathsf{i}}\rangle\langle cop\rangle\langle n\rangle$	$v_{\texttt{q}}$
$\texttt{call}\langle n_{\mathsf{i}}\rangle\langle ea\rangle$	$v_{\texttt{q}}[\texttt{fps} \mapsto \underline{\texttt{cons}}(v_{\texttt{g}}(\texttt{esp}) \dot{-} 4, v_{\texttt{q}}(\texttt{fps}))]$
$\texttt{ret}\langle n_{\mathsf{i}}\rangle$	$v_{\texttt{q}}[\texttt{fps} \mapsto \underline{\texttt{tail}}(v_{\texttt{q}}(\texttt{fps}))]$

Essentially, the frame pointer is pushed on each procedure call, and popped for each procedure return. Note that trusted procedures of the run-time system may change other aspects of the security-automaton state, but these changes are outside the scope of the `nextq` function.

The relation $\underline{\texttt{safe\_sp}}(v, n)$ holds exactly when n is a safe next-state stack pointer with respect to current state v:

$$\underline{\texttt{safe\_sp}}(\langle v, v_{\texttt{q}}\rangle, n) \text{ iff } v_{\texttt{q}}(\texttt{fps}) = \langle n_{\mathsf{fp}}, v'\rangle \text{ and } n_{\mathsf{fp}} \dot{-} n < \underline{\texttt{page}} \text{ and } (n_{\mathsf{fp}} \dot{-} n) \,\dot{\wedge}\, 3 = 0$$

The stack pointer is allowed to be explicitly decremented to an aligned address within a page of the current frame pointer. This guarantees that each page of memory will be touched at least once as stack frames are allocated from the stack, and thus that the stack grows contiguously downwards. This constraint ensures that the runtime system can detect when the stack and heap are about to collide,[6] and thus prevent the heap from being corrupted by the stack (see Section 5.6.4 for further discussion of stack overflow).

$\underline{\texttt{safe\_rds}}(v, n)$ holds when n is a readable stack address with respect to current state v:

$$\underline{\texttt{safe\_rds}}(\langle\langle n_{\texttt{pc}}, n_{\texttt{f}}, v_{\texttt{g}}, v_{\texttt{s}}, v_{\texttt{m}}\rangle, v_{\texttt{q}}\rangle, n) \text{ iff } \begin{array}{l} n \dot{-} v_{\texttt{g}}(\texttt{esp}) < \underline{\texttt{page}} \\ \text{and } (n \dot{-} v_{\texttt{g}}(\texttt{esp})) \,\dot{\wedge}\, 3 = 0 \end{array}$$

Similarly, $\underline{\texttt{safe\_wrs}}(v, n, n')$ holds when n is a writable stack address with respect to current state v:

$$\underline{\texttt{safe\_wrs}}(\langle\langle n_{\texttt{pc}}, n_{\texttt{f}}, v_{\texttt{g}}, v_{\texttt{s}}, v_{\texttt{m}}\rangle, v_{\texttt{q}}\rangle, n, n') \text{ iff } \begin{array}{l} n \dot{-} v_{\texttt{g}}(\texttt{esp}) < \underline{\texttt{page}} \\ \text{and } (n \dot{-} v_{\texttt{g}}(\texttt{esp})) \,\dot{\wedge}\, 3 = 0 \end{array}$$

Thus, all stack accesses must be within the page above the current stack pointer. Note that the interpretations of `safe_rds` and `safe_wrs` coincide in my implementation, but because it is reasonable to imagine systems where they do not, I treat them as distinct functions in anticipation of such a possibility.

$\underline{\texttt{safe\_rdm}}(v, n)$ holds when n is a readable memory address with respect to current state v:

$$\underline{\texttt{safe\_rdm}}(\langle v', v_{\texttt{q}}\rangle, n) \text{ iff } \underline{\texttt{leqa}}(\texttt{rd}, \underline{\texttt{sela}}(v_{\texttt{q}}(\texttt{accm}), n, 4)) \text{ and } n \,\dot{\wedge}\, 3 = 0$$

Similarly, $\underline{\texttt{safe\_wrm}}(v, n, n')$ holds when n is a writable memory address with respect to current state v:

$$\underline{\texttt{safe\_wrm}}(\langle v', v_{\texttt{q}}\rangle, n, n') \text{ iff } \underline{\texttt{leqa}}(\texttt{wr}, \underline{\texttt{sela}}(v_{\texttt{q}}(\texttt{accm}), n, 4)) \text{ and } n \,\dot{\wedge}\, 3 = 0$$

Thus, any memory access must be aligned and explicitly "approved" by the current access-mode map `accm`.

[6] For example, by mapping an "inaccessible" page between the stack and heap.

$\underline{\texttt{safe\_rdea}}(v, ea)$ holds when ea is a readable effective address with respect to current state v:

$$\underline{\texttt{safe\_rdea}}(v, ea) \text{ iff } \begin{cases} \text{true} & \text{if } ea = \texttt{ea\_i}\langle n \rangle \\ \text{true} & \text{if } ea = \texttt{ea\_r}\langle r \rangle \\ \underline{\texttt{safe\_rds}}(v, \underline{\texttt{ma\_addr}}(v_{\texttt{g}}, ma)) & \text{if } ea = \texttt{ea\_s}\langle ma \rangle \\ \underline{\texttt{safe\_rdm}}(v, \underline{\texttt{ma\_addr}}(v_{\texttt{g}}, ma)) & \text{if } ea = \texttt{ea\_m}\langle ma \rangle \end{cases}$$

where $v = \langle\langle n_{\texttt{pc}}, n_{\texttt{f}}, v_{\texttt{g}}, v_{\texttt{s}}, v_{\texttt{m}}\rangle, v_{\texttt{q}}\rangle$

Similarly, $\underline{\texttt{safe\_wrea}}(v, ea, n)$ holds when ea is a writable effective address with respect to current state v:

$$\underline{\texttt{safe\_wrea}}(v, ea, n) \text{ iff } \begin{cases} \text{false} & \text{if } ea = \texttt{ea\_i}\langle n' \rangle \\ \text{true} & \text{if } \quad ea = \texttt{ea\_r}\langle r \rangle \\ & \quad \text{and } r \neq \texttt{esp} \\ \underline{\texttt{safe\_sp}}(v, n) & \text{if } ea = \texttt{ea\_r}\langle \texttt{esp} \rangle \\ \underline{\texttt{safe\_wrs}}(v, \underline{\texttt{ma\_addr}}(v_{\texttt{g}}, ma), n) & \text{if } ea = \texttt{ea\_s}\langle ma \rangle \\ \underline{\texttt{safe\_wrm}}(v, \underline{\texttt{ma\_addr}}(v_{\texttt{g}}, ma), n) & \text{if } ea = \texttt{ea\_m}\langle ma \rangle \end{cases}$$

where $v = \langle\langle n_{\texttt{pc}}, n_{\texttt{f}}, v_{\texttt{g}}, v_{\texttt{s}}, v_{\texttt{m}}\rangle, v_{\texttt{q}}\rangle$

Thus, an access mode is safe to use in a state when the address to be accessed is safe according to the appropriate stack or memory access relation.

$\underline{\texttt{safe\_inst}}(v, I)$, defined in Table 5.1, holds when I is a safe instruction with respect to current state v. Essentially, instruction safety holds when all the effective addresses of the instruction are safe to use in the current state.

5.6.2 Inference Rules

In this section, I provide a deductive system that encodes the formal semantics of the memory-safety policy.

The following transition rule specifies how the security-automaton state changes from state to state:

$$\frac{}{\Gamma \vdash \forall x\!:\!\mathsf{ri}.\, \forall x_{\mathsf{I}}\!:\!\mathsf{ri}.\, x = \texttt{sq} \supset \texttt{fetch}(\texttt{pm}, \texttt{q\_pc}(x), x_{\mathsf{I}}) \supset \bigcirc(\texttt{q} = \texttt{nextq}(x, x_{\mathsf{I}})) \mathbin{@} t} \ \mathsf{nextq\_fetch}$$

It states that the next-time value of $\texttt{q}$ is the function $\texttt{nextq}$ applied to the current-time value of of $\texttt{q}$ and the current-time state and instruction.

The semantics of the function $\texttt{nextq}$ is encoded by a set inference rules. I show only a few representative cases here:

$$\frac{}{\Gamma \vdash \texttt{nextq}(\texttt{mkp}(e, e_{\texttt{q}}), f_{\texttt{mov}}(e_{\mathsf{i}}, e_{\texttt{ea}_1}, e_{\texttt{ea}_2})) = e_{\texttt{q}} \mathbin{@} t} \ \mathsf{nextq\_mov}$$

$$\frac{}{\Gamma \vdash \texttt{nextq}(\texttt{q\_mk}(e_{\texttt{pc}}, e_{\texttt{f}}, e_{\texttt{g}}, e_{\texttt{s}}, e_{\texttt{m}}, e_{\texttt{q}}), f_{\texttt{call}}(e_{\mathsf{i}}, e_{\texttt{ea}})) = e'_{\texttt{q}} \mathbin{@} t} \ \mathsf{nextq\_call}$$

where $e'_{\texttt{q}} \stackrel{\mathsf{def}}{=} \texttt{upds}(e_{\texttt{q}}, \overline{\texttt{fps}}, \texttt{cons}(\texttt{addw}(\texttt{selg}(e_{\texttt{g}}, \overline{\texttt{esp}}), -4), \texttt{sels}(e_{\texttt{q}}, \overline{\texttt{fps}})))$

I	$\underline{\texttt{safe\_inst}}(v, I)$, where $v = \langle v', v_{\mathtt{q}}\rangle = \langle\langle n_{\mathtt{pc}}, n_{\mathtt{f}}, v_{\mathtt{g}}, v_{\mathtt{s}}, v_{\mathtt{m}}\rangle, v_{\mathtt{q}}\rangle$
$\mathtt{mov}\langle n_{\mathsf{i}}\rangle\langle ea_1\rangle\langle ea_2\rangle$	$\underline{\texttt{safe\_rdea}}(v, ea_1)$ and $\underline{\texttt{safe\_wrea}}(v, ea_2, (v')_{ea_1})$
$\mathtt{xchg}\langle n_{\mathsf{i}}\rangle\langle ea\rangle\langle r\rangle$	$\underline{\texttt{safe\_rdea}}(v, ea)$ and $\underline{\texttt{safe\_wrea}}(v, \mathtt{ea\_r}\langle r\rangle, (v')_{ea})$ and $\underline{\texttt{safe\_wrea}}(v, ea, v_{\mathtt{g}}(r))$
$\mathtt{lea}\langle n_{\mathsf{i}}\rangle\langle ea\rangle\langle r\rangle$	$\underline{\texttt{safe\_wrea}}(v, \mathtt{ea\_r}\langle r\rangle, \underline{\texttt{ea\_addr}}(v', ea))$
$\mathtt{push}\langle n_{\mathsf{i}}\rangle\langle ea\rangle$	$\underline{\texttt{safe\_rdea}}(v, ea)$
$\mathtt{pop}\langle n_{\mathsf{i}}\rangle\langle ea\rangle$	$\underline{\texttt{safe\_wrea}}(v, ea, v_{\mathtt{s}}(v_{\mathtt{g}}(\mathtt{esp})))$
$\mathtt{op1}\langle n_{\mathsf{i}}\rangle\langle op1\rangle\langle ea\rangle$	$\underline{\texttt{safe\_rdea}}(v, ea)$ and $\underline{\texttt{safe\_wrea}}(v, ea, \underline{\texttt{app1}}(op1, (v')_{ea}))$
$\mathtt{op2}\langle n_{\mathsf{i}}\rangle\langle op2\rangle\langle ea_1\rangle\langle ea_2\rangle$	$\underline{\texttt{safe\_rdea}}(v, ea_1)$ and $\underline{\texttt{safe\_rdea}}(v, ea_2)$ and $\underline{\texttt{safe\_wrea}}(v, ea_2, n')$ where $n' = \underline{\texttt{app2}}(op2, (v')_{ea_2}, (v')_{ea_1})$
$\mathtt{op2n}\langle n_{\mathsf{i}}\rangle\langle op2\rangle\langle ea_1\rangle\langle ea_2\rangle$	$\underline{\texttt{safe\_rdea}}(v, ea_1)$ and $\underline{\texttt{safe\_rdea}}(v, ea_2)$
$\mathtt{op3}\langle n_{\mathsf{i}}\rangle\langle op3_1\rangle\langle op3_2\rangle\langle ea\rangle\langle r_1\rangle\langle r_2\rangle$	$\underline{\texttt{safe\_rdea}}(v, ea)$ and $\underline{\texttt{safe\_wrea}}(v, \mathtt{ea\_r}\langle r_1\rangle, n'_1)$ and $\underline{\texttt{safe\_wrea}}(v, \mathtt{ea\_r}\langle r_2\rangle, n'_2)$ where $n'_1 = \underline{\texttt{app3}}(op3_1, v_{\mathtt{g}}(r_1), v_{\mathtt{g}}(r_2), (v')_{ea})$ and $n'_2 = \underline{\texttt{app3}}(op3_2, v_{\mathtt{g}}(r_1), v_{\mathtt{g}}(r_2), (v')_{ea})$
$\mathtt{jmp}\langle n_{\mathsf{i}}\rangle\langle ea\rangle$	$\underline{\texttt{safe\_rdea}}(v, ea)$
$\mathtt{j}\langle n_{\mathsf{i}}\rangle\langle cop\rangle\langle n\rangle$	true
$\mathtt{call}\langle n_{\mathsf{i}}\rangle\langle ea\rangle$	$\underline{\texttt{safe\_rdea}}(v, ea)$
$\mathtt{ret}\langle n_{\mathsf{i}}\rangle$	true

Table 5.1: Instruction Safety

$$\frac{\Gamma \vdash \mathtt{subw}(\mathtt{head}(\mathtt{sels}(\mathtt{q\_q}(e), \overline{\mathtt{fps}})), e_{\mathsf{n}}) = e'_{\mathsf{n}} \mathbin{@} t}{\Gamma \vdash \mathtt{safe\_sp}(e, e_{\mathsf{n}}) \equiv \mathtt{ltuw}(e'_{\mathsf{n}}, \mathtt{page}) \wedge \mathtt{andw}(e'_{\mathsf{n}}, 3) = 0 \mathbin{@} t} \ \mathsf{safe\_sp}$$

$$\frac{\Gamma \vdash \mathtt{subw}(e_{\mathsf{n}}, \mathtt{selg}(\mathtt{q\_g}(e), \overline{\mathtt{esp}})) = e'_{\mathsf{n}} \mathbin{@} t}{\Gamma \vdash \mathtt{safe\_rds}(e, e_{\mathsf{n}}) \equiv \mathtt{ltuw}(e'_{\mathsf{n}}, \mathtt{page}) \wedge \mathtt{andw}(e'_{\mathsf{n}}, 3) = 0 \mathbin{@} t} \ \mathsf{safe\_rds}$$

$$\frac{\Gamma \vdash \mathtt{subw}(e_{\mathsf{n}}, \mathtt{selg}(\mathtt{q\_g}(e), \overline{\mathtt{esp}})) = e''_{\mathsf{n}} \mathbin{@} t}{\Gamma \vdash \mathtt{safe\_wrs}(e, e_{\mathsf{n}}, e'_{\mathsf{n}}) \equiv \mathtt{ltuw}(e''_{\mathsf{n}}, \mathtt{page}) \wedge \mathtt{andw}(e''_{\mathsf{n}}, 3) = 0 \mathbin{@} t} \ \mathsf{safe\_wrs}$$

$$\frac{}{\Gamma \vdash \mathtt{safe\_rdm}(e, e_{\mathsf{n}}) \equiv \mathtt{leqa}(\overline{\mathtt{rd}}, e_{\mathsf{acc}}) \wedge \mathtt{andw}(e_{\mathsf{n}}, 3) = 0 \mathbin{@} t} \ \mathsf{safe\_rdm}$$

$$\text{where } e_{\mathsf{acc}} \stackrel{\mathsf{def}}{=} \mathtt{sela}(\mathtt{q\_accm}(e), e_{\mathsf{n}}, 4)$$

$$\frac{}{\Gamma \vdash \mathtt{safe\_wrm}(e, e_{\mathsf{n}}, e'_{\mathsf{n}}) \equiv \mathtt{leqa}(\overline{\mathtt{wr}}, e_{\mathsf{acc}}) \wedge \mathtt{andw}(e_{\mathsf{n}}, 3) = 0 \mathbin{@} t} \ \mathsf{safe\_wrm}$$

$$\text{where } e_{\mathsf{acc}} \stackrel{\mathsf{def}}{=} \mathtt{sela}(\mathtt{q\_accm}(e), e_{\mathsf{n}}, 4)$$

$$\frac{}{\Gamma \vdash \mathtt{safe\_rdea}(e, f_{\mathsf{ea\_i}}(e_{\mathsf{n}})) \mathbin{@} t} \ \mathsf{safe\_rdea\_i} \qquad \frac{}{\Gamma \vdash \mathtt{safe\_rdea}(e, f_{\mathsf{ea\_r}}(e_{\mathsf{r}})) \mathbin{@} t} \ \mathsf{safe\_rdea\_r}$$

$$\frac{}{\Gamma \vdash \mathtt{safe\_rdea}(e, f_{\mathsf{ea\_s}}(e_{\mathtt{ma}})) \equiv \mathtt{safe\_rds}(e, \mathtt{ma\_addr}(\mathtt{q\_g}(e), e_{\mathtt{ma}})) \mathbin{@} t} \ \mathsf{safe\_rdea\_s}$$

$$\frac{}{\Gamma \vdash \mathtt{safe\_rdea}(e, f_{\mathsf{ea\_m}}(e_{\mathtt{ma}})) \equiv \mathtt{safe\_rdm}(e, \mathtt{ma\_addr}(\mathtt{q\_g}(e), e_{\mathtt{ma}})) \mathbin{@} t} \ \mathsf{safe\_rdea\_m}$$

$$\frac{\Gamma \vdash e_{\mathsf{r}} \neq \overline{\mathtt{esp}} \mathbin{@} t}{\Gamma \vdash \mathtt{safe\_wrea}(e, f_{\mathsf{ea\_r}}(e_{\mathsf{r}}), e_{\mathsf{n}}) \mathbin{@} t} \ \mathsf{safe\_wrea\_r}$$

$$\frac{}{\Gamma \vdash \mathtt{safe\_wrea}(e, f_{\mathsf{ea\_r}}(\overline{\mathtt{esp}}), e_{\mathsf{n}}) \equiv \mathtt{safe\_sp}(e, e_{\mathsf{n}}) \mathbin{@} t} \ \mathsf{safe\_wrea\_sp}$$

$$\frac{}{\Gamma \vdash \mathtt{safe\_wrea}(e, f_{\mathsf{ea\_s}}(e_{\mathtt{ma}}), e_{\mathsf{n}}) \equiv \mathtt{safe\_wrs}(e, \mathtt{ma\_addr}(\mathtt{q\_g}(e), e_{\mathtt{ma}}), e_{\mathsf{n}}) \mathbin{@} t} \ \mathsf{safe\_wrea\_s}$$

$$\frac{}{\Gamma \vdash \mathtt{safe\_wrea}(e, f_{\mathsf{ea\_m}}(e_{\mathtt{ma}}), e_{\mathsf{n}}) \equiv \mathtt{safe\_wrm}(e, \mathtt{ma\_addr}(\mathtt{q\_g}(e), e_{\mathtt{ma}}), e_{\mathsf{n}}) \mathbin{@} t} \ \mathsf{safe\_wrea\_m}$$

Figure 5.7: Inference Rules for Memory Safety

$$\frac{}{\begin{array}{ll}\Gamma \vdash & \mathtt{safe\_inst}(e, f_{\mathtt{mov}}(e_{\mathsf{i}}, e_{\mathsf{ea}_1}, e_{\mathsf{ea}_2})) \\ & \equiv \ \mathtt{safe\_rdea}(e, e_{\mathsf{ea}_1}) \\ & \quad \wedge\ \mathtt{safe\_wrea}(e, e_{\mathsf{ea}_2}, \mathtt{ea\_sel}(\mathtt{left}(e), e_{\mathsf{ea}_1}))\end{array} \ @\, t} \ \textsf{safe\_inst\_mov}$$

$$\frac{}{\begin{array}{ll}\Gamma \vdash & \mathtt{safe\_inst}(e, f_{\mathtt{xchg}}(e_{\mathsf{i}}, e_{\mathsf{ea}}, e_{\mathsf{r}})) \\ & \equiv \ \mathtt{safe\_rdea}(e, e_{\mathsf{ea}}) \\ & \quad \wedge\ \mathtt{safe\_wrea}(e, f_{\mathsf{ea\_r}}(e_{\mathsf{r}}), \mathtt{ea\_sel}(\mathtt{left}(e), e_{\mathsf{ea}})) \\ & \quad \wedge\ \mathtt{safe\_wrea}(e, e_{\mathsf{ea}}, \mathtt{selg}(\mathtt{q\_g}(e), e_{\mathsf{r}}))\end{array} \ @\, t} \ \textsf{safe\_inst\_xchg}$$

$$\frac{}{\begin{array}{ll}\Gamma \vdash & \mathtt{safe\_inst}(e, f_{\mathtt{lea}}(e_{\mathsf{i}}, e_{\mathsf{ea}}, e_{\mathsf{r}})) \\ & \equiv \mathtt{safe\_wrea}(e, f_{\mathsf{ea\_r}}(e_{\mathsf{r}}), \mathtt{ea\_addr}(\mathtt{left}(e), e_{\mathsf{ea}}))\end{array} \ @\, t} \ \textsf{safe\_inst\_lea}$$

$$\frac{}{\Gamma \vdash \mathtt{safe\_inst}(e, f_{\mathtt{push}}(e_{\mathsf{i}}, e_{\mathsf{ea}})) \equiv \mathtt{safe\_rdea}(e, e_{\mathsf{ea}}) \ @\, t} \ \textsf{safe\_inst\_push}$$

$$\frac{}{\begin{array}{ll}\Gamma \vdash & \mathtt{safe\_inst}(e, f_{\mathtt{pop}}(e_{\mathsf{i}}, e_{\mathsf{ea}})) \\ & \equiv \mathtt{safe\_wrea}(e, e_{\mathsf{ea}}, \mathtt{selw}(\mathtt{q\_s}(e), \mathtt{selg}(\mathtt{q\_g}(e), \overline{\mathtt{esp}})))\end{array} \ @\, t} \ \textsf{safe\_inst\_pop}$$

Figure 5.8: Inference Rules for Instruction Safety (1)

$$\frac{}{\Gamma \vdash \mathtt{nextq}(\mathtt{mkp}(e, e_{\mathsf{q}}), f_{\mathtt{ret}}(e_{\mathsf{i}})) = \mathtt{upds}(e_{\mathsf{q}}, \overline{\mathtt{fps}}, \mathtt{tail}(\mathtt{sels}(e_{\mathsf{q}}, \overline{\mathtt{fps}}))) \ @\, t} \ \textsf{nextq\_ret}$$

The inference rules in Figure 5.7 enable the safety of particular addresses and addressing modes to be inferred. A stack pointer is safe as long as it is properly aligned and within the first page below the current frame pointer. A stack access is safe as long as it is properly aligned and within the first page *above* the current stack pointer. The safety of an access through an effective address mode is determined according to the value of the effective address in the current state. Rules for inferring safety for individual memory accesses (`safe_wrm`, `safe_rdm`) are based on the type system of Section 5.8.

The inference rules in Figure 5.8 and Figure 5.9 enable the safety of particular instructions to be inferred according to the memory safety policy. Essentially, any access to the stack must be in range, and any access to memory must have an explicit proof of safety. When an instruction accesses the stack or memory is governed by the semantics of the particular instruction and its embedded effective addresses.

5.6.3 Soundness

I can now establish meta-theoretic properties of the formal memory-safety policy. First, the memory-safety policy ensures that control flow stays within the untrusted program:

Proposition 5.6.1 (Control-Flow Safety) $\sigma_j(\mathtt{pc}) \in \mathrm{dom}\,\mathcal{J}(\mathtt{pm})$ *if* $\sigma, j \vDash^{\mathcal{J}} \mathtt{safe}$

$$\frac{}{\begin{array}{l}\Gamma \vdash \quad \texttt{safe\_inst}(e, f_{\texttt{op1}}(e_{\mathsf{i}}, e_{\texttt{op1}}, e_{\texttt{ea}})) \\ \quad \equiv \quad \texttt{safe\_rdea}(e, e_{\texttt{ea}}) \\ \qquad \wedge\ \texttt{safe\_wrea}(e, e_{\texttt{ea}}, \texttt{app1}(e_{\texttt{op1}}, \texttt{ea\_sel}(\texttt{left}(e), e_{\texttt{ea}})))\end{array} \ @\, t}\ \textsf{safe\_inst\_op1}$$

$$\frac{}{\begin{array}{l}\Gamma \vdash \quad \texttt{safe\_inst}(e, f_{\texttt{op2}}(e_{\mathsf{i}}, e_{\texttt{op2}}, e_{\texttt{ea}_1}, e_{\texttt{ea}_2})) \\ \quad \equiv \quad \texttt{safe\_rdea}(e, e_{\texttt{ea}_1}) \wedge \texttt{safe\_rdea}(e, e_{\texttt{ea}_2}) \\ \qquad \wedge\ \texttt{safe\_wrea}(e, e_{\texttt{ea}_2}, e')\end{array} \ @\, t}\ \textsf{safe\_inst\_op2}$$

$$\text{where } e' \stackrel{\mathsf{def}}{=} \texttt{app2}(e_{\texttt{op2}}, \texttt{ea\_sel}(\texttt{left}(e), e_{\texttt{ea}_2}), \texttt{ea\_sel}(\texttt{left}(e), e_{\texttt{ea}_1}))$$

$$\frac{}{\begin{array}{l}\Gamma \vdash \quad \texttt{safe\_inst}(e, f_{\texttt{op2n}}(e_{\mathsf{i}}, e_{\texttt{op2}}, e_{\texttt{ea}_1}, e_{\texttt{ea}_2})) \\ \quad \equiv \texttt{safe\_rdea}(e, e_{\texttt{ea}_1}) \wedge \texttt{safe\_rdea}(e, e_{\texttt{ea}_2})\end{array} \ @\, t}\ \textsf{safe\_inst\_op2n}$$

$$\frac{\begin{array}{l}\Gamma \vdash \texttt{ea\_sel}(\texttt{left}(e), e_{\texttt{ea}}) = e' \ @\, t \\ \Gamma \vdash \texttt{selg}(\texttt{q\_g}(e), e_{\mathsf{r}_1}) = e'_1 \ @\, t \quad \Gamma \vdash \texttt{selg}(\texttt{q\_g}(e), e_{\mathsf{r}_2}) = e'_2 \ @\, t\end{array}}{\begin{array}{l}\Gamma \vdash \quad \texttt{safe\_inst}(e, f_{\texttt{op3}}(e_{\mathsf{i}}, e_{\texttt{op3}_1}, e_{\texttt{op3}_2}, e_{\texttt{ea}}, e_{\mathsf{r}_1}, e_{\mathsf{r}_2})) \\ \quad \equiv \quad \texttt{safe\_rdea}(e, e_{\texttt{ea}}) \\ \qquad \wedge\ \texttt{safe\_wrea}(e, f_{\texttt{ea\_r}}(e_{\mathsf{r}_1}), \texttt{app3}(e_{\texttt{op3}_1}, e'_1, e'_2, e')) \\ \qquad \wedge\ \texttt{safe\_wrea}(e, f_{\texttt{ea\_r}}(e_{\mathsf{r}_2}), \texttt{app3}(e_{\texttt{op3}_2}, e'_1, e'_2, e'))\end{array} \ @\, t}\ \textsf{safe\_inst\_op3}$$

$$\frac{}{\Gamma \vdash \texttt{safe\_inst}(e, f_{\texttt{jmp}}(e_{\mathsf{i}}, e_{\texttt{ea}})) \equiv \texttt{safe\_rdea}(e, e_{\texttt{ea}}) \ @\, t}\ \textsf{safe\_inst\_jmp}$$

$$\frac{}{\Gamma \vdash \texttt{safe\_inst}(e, f_{\mathsf{j}}(e_{\mathsf{i}}, e_{\texttt{cop}}, e_{\mathsf{n}})) \ @\, t}\ \textsf{safe\_inst\_j}$$

$$\frac{}{\Gamma \vdash \texttt{safe\_inst}(e, f_{\texttt{call}}(e_{\mathsf{i}}, e_{\texttt{ea}})) \equiv \texttt{safe\_rdea}(e, e_{\texttt{ea}}) \ @\, t}\ \textsf{safe\_inst\_call}$$

$$\frac{}{\Gamma \vdash \texttt{safe\_inst}(e, f_{\texttt{ret}}(e_{\mathsf{i}})) \ @\, t}\ \textsf{safe\_inst\_ret}$$

Figure 5.9: Inference Rules for Instruction Safety (2)

PROOF:

$\sigma, j \models^{\mathcal{J}} \mathtt{safe}$	Prem.
$\phi, \eta[b \mapsto j] \models^{\mathcal{J}} \mathtt{safe}$ for some ϕ, η, b such that $\phi\vert_{Reg} = \sigma$	Def. $\models$
$\phi, \eta[b \mapsto j] \models^{\mathcal{J}} \exists x_{\mathsf{I}}\!:\!\mathsf{fl}.\, \mathtt{fetch}(\mathtt{pm}, \mathtt{pc}, x_{\mathsf{I}}) \wedge \mathtt{safe\_inst}(\mathtt{sq}, x_{\mathsf{I}})$	Def. $\mathtt{safe}$
$\phi[a_{\mathsf{I}} \mapsto \pi_{\mathsf{I}}], \eta[b \mapsto j] \models^{\mathcal{J}} \mathtt{fetch}(\mathtt{pm}, \mathtt{pc}, a_{\mathsf{I}})$ for some $a_{\mathsf{I}}, \pi_{\mathsf{I}}$	Def. $\models$
$a_{\mathsf{I}} \notin Reg$	Prop. 3.2.14
let $\phi_{\mathsf{I}} = \phi[a_{\mathsf{I}} \mapsto \pi_{\mathsf{I}}]$	
$\underline{\mathtt{fetch}}(\mathcal{V}^{\mathcal{J}}_{\phi_{\mathsf{I}}}(\mathtt{pm})(j), \mathcal{V}^{\mathcal{J}}_{\phi_{\mathsf{I}}}(\mathtt{pc})(j), \mathcal{V}^{\mathcal{J}}_{\phi_{\mathsf{I}}}(a_{\mathsf{I}})(j))$	Def. $\models$
$\underline{\mathtt{fetch}}(\mathcal{J}(\mathtt{pm}), \phi(\mathtt{pc})(j), \pi_{\mathsf{I}}(j))$	Def. $\mathcal{V}^{\mathcal{J}}_{\phi}$
$\phi(\mathtt{pc})(j) \in \mathrm{dom}\, \mathcal{J}(\mathtt{pm})$	Def. $\mathtt{fetch}$
$\sigma_j(\mathtt{pc}) \in \mathrm{dom}\, \mathcal{J}(\mathtt{pm})$	Def. $\phi\vert_{Reg}$

□

Next, I will demonstrate that the formal memory-safety policy ensures that all memory accesses will respect the access-mode map.

First, let $MemRd(s)$ be the set of memory addresses that will be read by executing one instruction from state s. $MemRd(s)$ can be formalized as follows:

$\mathcal{J}(\mathtt{pm})(s(\mathtt{pc}))$	$MemRd(s)$
$\mathtt{mov}\langle n_{\mathsf{i}}\rangle\langle ea_1\rangle\langle ea_2\rangle$	$MemAcc(s, ea_1)$
$\mathtt{xchg}\langle n_{\mathsf{i}}\rangle\langle ea\rangle\langle r\rangle$	$MemAcc(s, ea)$
$\mathtt{lea}\langle n_{\mathsf{i}}\rangle\langle ea\rangle\langle r\rangle$	$\emptyset$
$\mathtt{push}\langle n_{\mathsf{i}}\rangle\langle ea\rangle$	$MemAcc(s, ea)$
$\mathtt{pop}\langle n_{\mathsf{i}}\rangle\langle ea\rangle$	$\emptyset$
$\mathtt{op1}\langle n_{\mathsf{i}}\rangle\langle op1\rangle\langle ea\rangle$	$MemAcc(s, ea)$
$\mathtt{op2}\langle n_{\mathsf{i}}\rangle\langle op2\rangle\langle ea_1\rangle\langle ea_2\rangle$	$MemAcc(s, ea_1) \cup MemAcc(s, ea_2)$
$\mathtt{op2n}\langle n_{\mathsf{i}}\rangle\langle op2\rangle\langle ea_1\rangle\langle ea_2\rangle$	$MemAcc(s, ea_1) \cup MemAcc(s, ea_2)$
$\mathtt{op3}\langle n_{\mathsf{i}}\rangle\langle op3_1\rangle\langle op3_2\rangle\langle ea\rangle\langle r_1\rangle\langle r_2\rangle$	$MemAcc(s, ea)$
$\mathtt{jmp}\langle n_{\mathsf{i}}\rangle\langle ea\rangle$	$MemAcc(s, ea)$
$\mathtt{j}\langle n_{\mathsf{i}}\rangle\langle cop\rangle\langle n\rangle$	$\emptyset$
$\mathtt{call}\langle n_{\mathsf{i}}\rangle\langle ea\rangle$	$MemAcc(s, ea)$
$\mathtt{ret}\langle n_{\mathsf{i}}\rangle$	$\emptyset$

And let $MemRd(s) = \emptyset$ if $s(\mathtt{pc}) \notin \mathcal{J}(\mathtt{pm})$. $MemAcc(s, ea)$ is the set of memory addresses (zero or one) that are accessed by the effective address ea for in state s:

$$MemAcc(s, ea) = \begin{cases} \{\underline{\mathtt{ma\_addr}}(s(\mathtt{g}), ma)\} & \text{if if } ea = \mathtt{ea\_m}\langle ma\rangle \\ \emptyset & \text{otherwise} \end{cases}$$

Similarly, let $MemWr(s)$ be the set of memory addresses that will be written by executing one instruction from state s:

$\mathcal{J}(\mathtt{pm})(s(\mathtt{pc}))$	$MemWr(s)$
$\mathtt{mov}\langle n_{\mathsf{i}}\rangle\langle ea_1\rangle\langle ea_2\rangle$	$MemAcc(s, ea_2)$
$\mathtt{xchg}\langle n_{\mathsf{i}}\rangle\langle ea\rangle\langle r\rangle$	$MemAcc(s, ea)$
$\mathtt{lea}\langle n_{\mathsf{i}}\rangle\langle ea\rangle\langle r\rangle$	$\emptyset$
$\mathtt{push}\langle n_{\mathsf{i}}\rangle\langle ea\rangle$	$\emptyset$
$\mathtt{pop}\langle n_{\mathsf{i}}\rangle\langle ea\rangle$	$MemAcc(s, ea)$
$\mathtt{op1}\langle n_{\mathsf{i}}\rangle\langle op1\rangle\langle ea\rangle$	$MemAcc(s, ea)$
$\mathtt{op2}\langle n_{\mathsf{i}}\rangle\langle op2\rangle\langle ea_1\rangle\langle ea_2\rangle$	$MemAcc(s, ea_2)$
$\mathtt{op2n}\langle n_{\mathsf{i}}\rangle\langle op2\rangle\langle ea_1\rangle\langle ea_2\rangle$	$\emptyset$
$\mathtt{op3}\langle n_{\mathsf{i}}\rangle\langle op3_1\rangle\langle op3_2\rangle\langle ea\rangle\langle r_1\rangle\langle r_2\rangle$	$\emptyset$
$\mathtt{jmp}\langle n_{\mathsf{i}}\rangle\langle ea\rangle$	$\emptyset$
$\mathtt{j}\langle n_{\mathsf{i}}\rangle\langle cop\rangle\langle n\rangle$	$\emptyset$
$\mathtt{call}\langle n_{\mathsf{i}}\rangle\langle ea\rangle$	$\emptyset$
$\mathtt{ret}\langle n_{\mathsf{i}}\rangle$	$\emptyset$

Memory safety now follows essentially by definition:

Proposition 5.6.2 (Memory Safety)

1. $\mathtt{rd} \leq (s(\mathtt{q})(\mathtt{accm}))_{n,4}$ *for all* $n \in MemRd(\sigma_j)$*, and*
2. $\mathtt{wr} \leq (s(\mathtt{q})(\mathtt{accm}))_{n,4}$ *for all* $n \in MemWr(\sigma_j)$*, and*
3. $n \dot{\wedge} 3 = 0$ *for all* $n \in MemRd(\sigma_j) \cup MemWr(\sigma_j)$

if $\sigma, j \models^{\mathcal{J}} \mathtt{safe}$

PROOF: by the definition of $\models$, $\mathcal{V}_{\phi}^{\mathcal{J}}$, and $\mathtt{safe}$ □

These are concrete examples of how it is possible to reason about a formal security property in order to reach a particular informal conclusion.

5.6.4 Stack Overflow

The code consumer needs to ensure that the stack will not grow too large and collide with the heap, thereby corrupting the data structures of the trusted run-time system. There are a number of approaches that can ensure this. For example, the code consumer can stipulate a procedure calling convention and check that the untrusted code uses this convention (Necula [Nec98]) takes this approach), or the code consumer can impose a stack resource bound and require the code producer to provide a proof that the bound is satisfied. Implementing the latter approach is likely to be quite difficult, so I will use a simpler technique for this dissertation that combines elements of both approaches.

In order to prevent stack overflow, the code consumer places an "inaccessible" boundary page above the heap (if the address space is flat) and (well) below the initial stack pointer.[7] The formal memory-safety policy ensures that the untrusted program will allocate only small, contiguous frames from the stack, and that at least one location in each frame (the return address slot) will be accessed before the next frame is allocated. The stack-allocation policy is tied to the IA-32 `call` instruction. The code producer must use this instruction for procedure calls if he or she wishes to use the stack to allocate activation records, but the rest of the standard calling convention need not be used. Note that if the code producer uses heap-allocated activation records [AS94, App92] no restrictions are placed on the procedure calling convention (other than for the initial entry point). Because the stack grows contiguously downwards in small increments, it is guaranteed to access the boundary page before the heap is corrupted. Accessing the boundary page will result in a run-time exception, thus preventing the untrusted program from corrupting the heap.

Note that because several approaches are possible for limiting stack growth, I do not attempt to model the boundary page formally in Chapter 4. Additionally, the code producer is provided with the following inference rule, which is justified by the stack overflow check:

$$\frac{\Gamma \vdash \mathtt{fetch}(\mathsf{pm}, \mathsf{q\_pc}(\mathsf{sq}), f_{\mathtt{call}}(e_{\mathsf{i}}, e_{\mathsf{ea}})) \mathbin{@} t \quad \Gamma \vdash \mathsf{gequw}(\mathsf{gsp}, \mathsf{page}) \mathbin{@} t}{\Gamma \vdash \mathsf{gequw}(\mathsf{gsp}, \mathsf{addw}(\mathsf{page}, \mathsf{page})) \mathbin{@} t + 1} \ \mathsf{sp\_under}$$

It asserts that whenever the stack pointer is above the first page and a call instruction is executed, the stack pointer will be above the second page in the next time step.

5.7 Java Types

In this section, I encode a machine-level fragment of the Java [GJS96] type system that is strong enough to establish memory safety for a useful set of programs. Essentially, a machine-language Java type is inherited by a register or memory location when a source-language variable or expression is compiled to that location. The machine model obviously does not know what the source-level program code is, so the code producer provides an explicit typing derivation for each memory location that is accessed by the program. This approach was originated by Necula [Nec98] for a safe variant of the C programming language, and has later been extended to encompass the Java programming language [CLN+00]. My type system is based on this latter implementation.

I only address a subset of the Java type system in this dissertation—features for long integers, floating point, subtyping, or interfaces are not included. Note, however, that the current SpecialJ [CLN+00] implementation does include these features, so I do not anticipate any serious difficulty in extending my type system to accommodate these additional types.

`jty` is the type of Java types, `jta` is the type of Java *type assignments*, and `jts` is the type of Java *type environments*. A type assignment is a representation function (assigning sets of values to types), paired with a memory assignment function

[7] If the address space is segmented, the boundary page is simply the page at address zero.

$c^{\texttt{jty}}_{\texttt{j\_bool}}$ $c^{\texttt{jty}}_{\texttt{j\_char}}$ $c^{\texttt{jty}}_{\texttt{j\_byte}}$ $c^{\texttt{jty}}_{\texttt{j\_short}}$ $c^{\texttt{jty}}_{\texttt{j\_int}}$ $c^{\texttt{jty}}_{\texttt{j\_Class}}$	Java type
$f^{\texttt{jty}\to\texttt{jty}}_{\texttt{j\_array}}$	Java array type constructor
$f^{\texttt{wd}\to\texttt{jty}}_{\texttt{j\_inst}}$	Java instance type constructor
$\texttt{j\_size}^{\texttt{jty}\to\texttt{wd}}$	Size of type
$\texttt{ja\_leq}^{\texttt{jta}\times\texttt{jta}\to o}$	Containment for type assignments
$\texttt{ja\_of}^{\texttt{jta}\times\texttt{wd}\times\texttt{jty}\to o}$	Type assignment for values
$\texttt{ja\_ptr}^{\texttt{jta}\times\texttt{wd}\times\texttt{jty}\to o}$	Type assignment for memory locations
$\texttt{ja\_field}^{\texttt{jta}\times\texttt{wd}\times\texttt{wd}\times\texttt{jty}\to o}$	Type assignment for field offsets

Table 5.2: Operations on Java Types

(assigning types to addresses), and a field assignment function (assigning types to field offsets of classes). The type jta is only inhabited by type assignments that are internally consistent; I define this condition formally in Section 5.7.1. A type environment is a type assignment paired with a memory access mode map:

$$\texttt{jts} \stackrel{\mathsf{def}}{=} \texttt{pair}\langle\texttt{jta}\rangle\langle\texttt{mapa}\rangle$$

I introduce the constants, functions, and relations in Table 5.2 to model Java types formally. Essentially, mathematical values are "reflected" into the syntax of the logic in much the same way as in Section 4. The underlying mathematical values are introduced in Section 5.7.1.

I define the following abbreviations to compute the address of the length field of an array, as well as the address of an arbitrary array element:

$$\begin{aligned} \texttt{j\_len}(e_{\mathsf{na}}) &\stackrel{\mathsf{def}}{=} \texttt{addw}(e_{\mathsf{na}}, 16) \\ \texttt{j\_elem}(e_{\mathsf{na}}, e_{\mathsf{ni}}, e_{\texttt{jty}}) &\stackrel{\mathsf{def}}{=} \texttt{addw}(e_{\mathsf{na}}, \texttt{addw}(\texttt{mulw}(e_{\mathsf{ni}}, \texttt{j\_size}(e_{\texttt{jty}})), 20)) \end{aligned}$$

The offsets in these definitions (16, 20) are determined by the run-time system. The following abbreviations assert that a given type is of machine-word size, and that a given machine-word pointer is aligned with respect to a given type, respectively:

$$\begin{aligned} \texttt{j\_wd}(e_{\texttt{jty}}) &\stackrel{\mathsf{def}}{\equiv} \texttt{j\_size}(e_{\texttt{jty}}) = 4 \\ \texttt{j\_align}(e_{\mathsf{np}}, e_{\texttt{jty}}) &\stackrel{\mathsf{def}}{\equiv} \texttt{andw}(e_{\mathsf{np}}, \texttt{subw}(\texttt{j\_size}(e_{\texttt{jty}}), 1)) = 0 \end{aligned}$$

The following defined relations hold when a given machine word is a pointer to an array length, array element, or object field, respectively:

$$\begin{aligned}
\mathtt{ja_ptr_len}(e_{\mathtt{jta}}, e_{\mathsf{np}}) \quad &\stackrel{\mathsf{def}}{\equiv} \exists x_{\mathsf{na}}\!:\!\mathsf{fl}.\ \exists x_{\mathtt{jty}}\!:\!\mathsf{fl}.\\
&\qquad \mathtt{ja_of}(e_{\mathtt{jta}}, x_{\mathsf{na}}, f_{\mathtt{j_array}}(x_{\mathtt{jty}}))\\
&\quad \wedge\ x_{\mathsf{na}} \neq 0\\
&\quad \wedge\ e_{\mathsf{np}} = \mathtt{j_len}(x_{\mathsf{na}})
\end{aligned}$$

$$\begin{aligned}
\mathtt{ja_ptr_elem}(e_{\mathtt{jta}}, e_{\mathtt{m}}, e_{\mathsf{np}}, e_{\mathtt{jty}}) &\stackrel{\mathsf{def}}{\equiv} \exists x_{\mathsf{na}}\!:\!\mathsf{fl}.\ \exists x_{\mathsf{ni}}\!:\!\mathsf{fl}.\\
&\qquad \mathtt{ja_of}(e_{\mathtt{jta}}, x_{\mathsf{na}}, f_{\mathtt{j_array}}(e_{\mathtt{jty}}))\\
&\quad \wedge\ x_{\mathsf{na}} \neq 0\\
&\quad \wedge\ \mathtt{ltuw}(x_{\mathsf{ni}}, \mathtt{selw}(e_{\mathtt{m}}, \mathtt{j_len}(x_{\mathsf{na}})))\\
&\quad \wedge\ e_{\mathsf{np}} = \mathtt{j_elem}(x_{\mathsf{na}}, x_{\mathsf{ni}}, e_{\mathtt{jty}})
\end{aligned}$$

$$\begin{aligned}
\mathtt{ja_ptr_field}(e_{\mathtt{jta}}, e_{\mathsf{np}}, e_{\mathtt{jty}}) \quad &\stackrel{\mathsf{def}}{\equiv} \exists x_{\mathsf{no}}\!:\!\mathsf{fl}.\ \exists x_{\mathsf{nc}}\!:\!\mathsf{fl}.\ \exists x_{\mathsf{nf}}\!:\!\mathsf{fl}.\\
&\qquad \mathtt{ja_of}(e_{\mathtt{jta}}, x_{\mathsf{no}}, f_{\mathtt{j_inst}}(x_{\mathsf{nc}}))\\
&\quad \wedge\ x_{\mathsf{no}} \neq 0\\
&\quad \wedge\ \mathtt{ja_field}(e_{\mathtt{jta}}, x_{\mathsf{nc}}, x_{\mathsf{nf}}, e_{\mathtt{jty}})\\
&\quad \wedge\ e_{\mathsf{np}} = \mathtt{addw}(x_{\mathsf{no}}, x_{\mathsf{nf}})
\end{aligned}$$

$\mathtt{ja\_valid}^{\mathtt{jta}\to o}$ is a defined relation that holds when a given a given type assignment is internally consistent:

$$\begin{aligned}
\mathtt{ja_valid}(e_{\mathtt{jta}}) \stackrel{\mathsf{def}}{\equiv} &\ \forall x_{\mathsf{np}}\!:\!\mathsf{fl}.\ \forall x_{\mathtt{jty}}\!:\!\mathsf{fl}.\\
&\qquad (\mathtt{ja_ptr_len}(e_{\mathtt{jta}}, x_{\mathsf{np}}) \supset \mathtt{ja_ptr}(e_{\mathtt{jta}}, x_{\mathsf{np}}, \overline{\mathtt{j_int}}))\\
&\quad \wedge (\mathtt{ja_ptr_field}(e_{\mathtt{jta}}, x_{\mathsf{np}}, x_{\mathtt{jty}}) \supset \mathtt{ja_ptr}(e_{\mathtt{jta}}, x_{\mathsf{np}}, x_{\mathtt{jty}}))\\
&\quad \wedge (\mathtt{ja_ptr}(e_{\mathtt{jta}}, x_{\mathsf{np}}, x_{\mathtt{jty}}) \supset \mathtt{j_align}(x_{\mathsf{np}}, x_{\mathtt{jty}}))
\end{aligned}$$

This relation requires that the length location of an array be assigned the type `j_int`, and that each field location of an object be assigned the corresponding type of the field. Note that the type of an array element cannot be constrained at this level (*e.g.*, as in Necula [Nec98]), because the length of the array must be retrieved from memory in order to bound the number of elements (refer to the definition of `ja_mem`, appearing next). Additionally, pointers must be aligned according to the size of the value whose address is taken.

$\mathtt{ja\_mem}^{\mathtt{jta}\times\mathtt{mapw}\to o}$ is a defined relation that holds when a given memory value is consistent with a given type assignment:

$$\begin{aligned}
&\mathtt{ja_mem}(e_{\mathtt{jta}}, e_{\mathtt{m}})\\
&\ \stackrel{\mathsf{def}}{\equiv} \forall x_{\mathsf{np}}\!:\!\mathsf{fl}.\ \forall x_{\mathtt{jty}}\!:\!\mathsf{fl}.\\
&\qquad (\mathtt{ja_ptr_elem}(e_{\mathtt{jta}}, e_{\mathtt{m}}, x_{\mathsf{np}}, x_{\mathtt{jty}}) \supset \mathtt{ja_ptr}(e_{\mathtt{jta}}, x_{\mathsf{np}}, x_{\mathtt{jty}}))\\
&\quad \wedge (\mathtt{ja_ptr}(e_{\mathtt{jta}}, x_{\mathsf{np}}, x_{\mathtt{jty}}) \supset \mathtt{j_wd}(x_{\mathtt{jty}}) \supset \mathtt{ja_of}(e_{\mathtt{jta}}, \mathtt{selw}(e_{\mathtt{m}}, x_{\mathsf{np}}), x_{\mathtt{jty}}))
\end{aligned}$$

This relation asserts that each element of an array be assigned the corresponding type of the element and that the value in each memory location agrees with the type assigned to that memory location.

Internal consistency is extended from type assignments to type environments by requiring that array length and object fields be accessible according to the access map in the environment:

$$\begin{aligned}\texttt{js\_valid}(e_{\texttt{jts}}) \stackrel{\mathsf{def}}{\equiv} \forall x_{\mathsf{np}}\!:\!\mathsf{fl}.\,\forall x_{\texttt{jty}}\!:\!\mathsf{fl}.\ & \texttt{ja\_valid}(\texttt{left}(e_{\texttt{jts}})) \\ & \wedge \left(\begin{array}{l}\ \ \texttt{ja\_ptr\_len}(\texttt{left}(e_{\texttt{jts}}), x_{\mathsf{np}}) \\ \supset \texttt{sela}(\texttt{right}(e_{\texttt{jts}}), x_{\mathsf{np}}, \texttt{j\_size}(\overline{\texttt{j\_int}})) = \overline{\texttt{rd}}\end{array}\right) \\ & \wedge \left(\begin{array}{l}\ \ \texttt{ja\_ptr\_field}(\texttt{left}(e_{\texttt{jts}}), x_{\mathsf{np}}, x_{\texttt{jty}}) \\ \supset \texttt{sela}(\texttt{right}(e_{\texttt{jts}}), x_{\mathsf{np}}, \texttt{j\_size}(x_{\texttt{jty}})) = \overline{\texttt{rw}}\end{array}\right)\end{aligned}$$

Memory consistency is extended from type assignments to type environments by requiring that array elements be accessible according to the access map in the environment:

$$\begin{aligned}\texttt{js\_mem}(e_{\texttt{jts}}, e_{\texttt{m}}) \stackrel{\mathsf{def}}{\equiv} \forall x_{\mathsf{np}}\!:\!\mathsf{fl}.\,\forall x_{\texttt{jty}}\!:\!\mathsf{fl}.\ & \texttt{ja\_valid}(e_{\texttt{jts}}) \\ & \wedge\ \texttt{ja\_mem}(\texttt{left}(e_{\texttt{jts}}), e_{\texttt{m}}) \\ & \wedge \left(\begin{array}{l}\ \ \texttt{ja\_ptr\_elem}(\texttt{left}(e_{\texttt{jts}}), e_{\texttt{m}}, x_{\mathsf{np}}, x_{\texttt{jty}}) \\ \supset \texttt{sela}(\texttt{right}(e_{\texttt{jts}}), x_{\mathsf{np}}, \texttt{j\_size}(x_{\texttt{jty}})) = \overline{\texttt{rw}}\end{array}\right)\end{aligned}$$

I now extend the standard typing relations from type assignments to type environments:

$$\begin{aligned}\texttt{js\_leq}(e_{\texttt{jts}_1}, e_{\texttt{jts}_2}) &\stackrel{\mathsf{def}}{\equiv} \ \ \texttt{ja\_leq}(\texttt{left}(e_{\texttt{jts}_1}), \texttt{left}(e_{\texttt{jts}_2})) \\ &\qquad \wedge\ \texttt{leqam}(\texttt{right}(e_{\texttt{jts}_1}), \texttt{right}(e_{\texttt{jts}_2})) \\ \texttt{js\_of}(e_{\texttt{jts}}, e_{\mathsf{np}}, e_{\texttt{jty}}) &\stackrel{\mathsf{def}}{\equiv} \ \ \texttt{js\_valid}(e_{\texttt{jts}}) \\ &\qquad \wedge\ \texttt{ja\_of}(\texttt{left}(e_{\texttt{jts}}), e_{\mathsf{np}}, e_{\texttt{jty}}) \\ \texttt{js\_ptr}(e_{\texttt{jts}}, e_{\mathsf{np}}, e_{\texttt{jty}}, e_{\texttt{acc}}) &\stackrel{\mathsf{def}}{\equiv} \ \ \texttt{js\_valid}(e_{\texttt{jts}}) \\ &\qquad \wedge\ \texttt{ja\_ptr}(\texttt{left}(e_{\texttt{jts}}), e_{\mathsf{np}}, e_{\texttt{jty}}) \\ &\qquad \wedge\ \texttt{sela}(\texttt{right}(e_{\texttt{jts}}), e_{\mathsf{np}}, \texttt{j\_size}(e_{\texttt{jty}})) = e_{\texttt{acc}} \\ \texttt{js\_field}(e_{\texttt{jts}}, e_{\mathsf{nc}}, e_{\mathsf{nf}}, e_{\texttt{jty}}) &\stackrel{\mathsf{def}}{\equiv} \texttt{ja\_field}(\texttt{left}(e_{\texttt{jts}}), e_{\mathsf{nc}}, e_{\mathsf{nf}}, e_{\texttt{jty}})\end{aligned}$$

Containment for type environments, $\texttt{js\_leq}^{\texttt{jts}\times\texttt{jts}\to o}$, holds when the component type assignments and access maps are contained. The type assignment relation for memory locations, $\texttt{js\_ptr}^{\texttt{jts}\times\texttt{wd}\times\texttt{jty}\times\texttt{acc}\to o}$ is extended to specify an access mode. It holds when the specified address has the specified type and has at least the specified access mode. Both type assignment relations for type environments imply that the type environment is internally consistent. This approach is taken primarily to reduce the number of inferences required to reconstruct a typing derivation.

5.7.1 Operational Semantics

I begin by providing a semantic model for types and type assignments:

$$\begin{array}{lrl}\text{Java types} & \mathit{jty} ::= & \texttt{j\_bool} \mid \texttt{j\_char} \mid \texttt{j\_byte} \mid \texttt{j\_short} \mid \texttt{j\_int} \\ & & \mid \texttt{j\_array}\langle \mathit{jty}\rangle \mid \texttt{j\_inst}\langle n\rangle \mid \texttt{j\_Class}\end{array}$$

Classes are identified by distinct addresses. An instance of a class n is assigned the type $\mathtt{j\_inst}\langle n\rangle$. The class n is itself assigned the type `j_Class`.

A type assignment is modeled by a triple consisting of a total representation function that assigns a set of types to each machine word, a partial memory assignment function that assigns a possible type to each memory address, and a set of field assignments that associates a possible type with a field offset and class:

$$\begin{aligned} \mathit{Val}^{\mathtt{jty}} &= \{\mathit{jty}\}_{\mathit{jty}} \\ \mathit{Val}^{\mathtt{jta}} &= (\mathit{Val}^{\mathtt{wd}} \to \mathbf{2}^{\mathit{Val}^{\mathtt{jty}}}) \times (\mathit{Val}^{\mathtt{wd}} \rightharpoonup \mathit{Val}^{\mathtt{jty}}) \times \mathbf{2}^{\mathit{Val}^{\mathtt{wd}} \times \mathit{Val}^{\mathtt{wd}} \times \mathit{Val}^{\mathtt{jty}}} \end{aligned}$$

The size function assigns the size 4 to each Java type:

$$\underline{\mathtt{j\_size}}(\mathit{jty}) = 4$$

This definition is an artifact of the SpecialJ implementation that I use: each "small" type occupies 4 bytes. Note that distinct arithmetic types (`j_byte`, `j_short`, `j_int`) are still useful for constraining procedure parameters even when the size of each type is always 4 bytes.

The operation of the type assignment functions is defined as follows:

$$\begin{array}{lll}
\underline{\mathtt{ja_leq}}(\langle v_1, v_2, v_3\rangle, \langle v_1', v_2', v_3'\rangle) & \text{iff} & \quad v_1(n) \subseteq v_1'(n) \text{ for all } n \\
 & & \text{and } \operatorname{dom} v_2 \subseteq \operatorname{dom} v_2' \\
 & & \text{and } v_2(n) = v_2'(n) \text{ for all } n \in \operatorname{dom} v_2 \\
 & & \text{and } v_3 \subseteq v_3' \\
\underline{\mathtt{ja_of}}(\langle v_1, v_2, v_3\rangle, n, \mathit{jty}) & \text{iff} & \quad \mathit{jty} \in v_1(n) \\
 & & \text{or } n < 2 \text{ and } \mathit{jty} = \mathtt{j_bool} \\
 & & \text{or } n < 2^{16} \text{ and } \mathit{jty} = \mathtt{j_char} \\
 & & \text{or } -2^7 \leq \ulcorner n \urcorner < 2^7 \text{ and } \mathit{jty} = \mathtt{j_byte} \\
 & & \text{or } -2^{15} \leq \ulcorner n \urcorner < 2^{15} \text{ and } \mathit{jty} = \mathtt{j_short} \\
 & & \text{or } \mathit{jty} = \mathtt{j_int} \\
 & & \text{or } n = 0 \text{ and } \mathit{jty} = \mathtt{j_array}\langle \mathit{jty}'\rangle \\
 & & \text{or } n = 0 \text{ and } \mathit{jty} = \mathtt{j_inst}\langle n'\rangle \\
\underline{\mathtt{ja_ptr}}(\langle v_1, v_2, v_3\rangle, n, \mathit{jty}) & \text{iff} & n \in \operatorname{dom} v_2 \text{ and } v_2(n) = \mathit{jty} \\
\underline{\mathtt{ja_field}}(\langle v_1, v_2, v_3\rangle, n_1, n_2, \mathit{jty}) & \text{iff} & \langle n_1, n_2, \mathit{jty}\rangle \in v_3
\end{array}$$

Containment for type assignments (`ja_leq`) holds when the representation functions are point-wise contained, the memory assignment function v_2 assigns a subset of the types assigned by v_2', and the field offset functions are compatible. The type assignment relation (`ja_of`) holds when the type is assigned by the representation function, when the type is an atomic type and the value is in range, or when the value is null and the type is a pointer type. Note that I only rely on the representation function to assign types to non-null pointers. The pointer relation holds when the memory assignment function assigns the value the appropriate type. Finally, the field relation holds when field offset function assigns the class and offset the appropriate type.

$$\frac{\Gamma \vdash \texttt{ltuw}(e_{\mathsf{n}}, 2) \mathbin{@} t}{\Gamma \vdash \texttt{ja\_of}(e_{\texttt{jta}}, e_{\mathsf{n}}, \overline{\texttt{j\_bool}}) \mathbin{@} t}\ \textsf{ja\_ofi}_{\texttt{j\_bool}}$$

$$\frac{\Gamma \vdash \texttt{ltuw}(e_{\mathsf{n}}, 65536) \mathbin{@} t}{\Gamma \vdash \texttt{ja\_of}(e_{\texttt{jta}}, e_{\mathsf{n}}, \overline{\texttt{j\_char}}) \mathbin{@} t}\ \textsf{ja\_ofi}_{\texttt{j\_char}}$$

$$\frac{\Gamma \vdash \texttt{geqw}(e_{\mathsf{n}}, 4294967168) \mathbin{@} t \quad \Gamma \vdash \texttt{ltw}(e_{\mathsf{n}}, 128) \mathbin{@} t}{\Gamma \vdash e_{\texttt{jta}}, \texttt{ja\_of}(e_{\mathsf{n}}, \overline{\texttt{j\_byte}}) \mathbin{@} t}\ \textsf{ja\_ofi}_{\texttt{j\_byte}}$$

$$\frac{\Gamma \vdash \texttt{geqw}(e_{\mathsf{n}}, 4294934528) \mathbin{@} t \quad \Gamma \vdash \texttt{ltw}(e_{\mathsf{n}}, 32768) \mathbin{@} t}{\Gamma \vdash \texttt{ja\_of}(e_{\texttt{jta}}, e_{\mathsf{n}}, \overline{\texttt{j\_short}}) \mathbin{@} t}\ \textsf{ja\_ofi}_{\texttt{j\_short}}$$

$$\frac{}{\Gamma \vdash \texttt{ja\_of}(e_{\texttt{jta}}, e_{\mathsf{n}}, \overline{\texttt{j\_int}}) \mathbin{@} t}\ \textsf{ja\_ofi}_{\texttt{j\_int}}$$

$$\frac{}{\Gamma \vdash \texttt{ja\_of}(e_{\texttt{jta}}, 0, f_{\texttt{j\_array}}(e_{\texttt{jty}})) \mathbin{@} t}\ \textsf{ja\_ofi}_{\texttt{j\_array}}0$$

$$\frac{}{\Gamma \vdash \texttt{ja\_of}(e_{\texttt{jta}}, 0, f_{\texttt{j\_inst}}(e_{\mathsf{nc}})) \mathbin{@} t}\ \textsf{ja\_ofi}_{\texttt{j\_inst}}0$$

Figure 5.10: Java Typing Rules for Constants

5.7.2 Inference Rules

The following inference rules encode the semantics of the `j_size` function:

$$\frac{}{\Gamma \vdash \texttt{j\_size}(f_{\texttt{j\_array}}(e_{\texttt{jty}})) = 4 \mathbin{@} t}\ \textsf{j\_sizei}_{\texttt{j\_array}}$$

$$\frac{}{\Gamma \vdash \texttt{j\_size}(f_{\texttt{j\_inst}}(e_{\mathsf{n}})) = 4 \mathbin{@} t}\ \textsf{j\_sizei}_{\texttt{j\_inst}}$$

Note that the size of an atomic type can be inferred by using term rewriting rules for constants.

The following inference rules assert that the array and instance type constructors are injective:

$$\frac{}{\Gamma \vdash \texttt{inj}(f_{\texttt{j\_array}}) \mathbin{@} t}\ \textsf{array\_inj} \qquad \frac{}{\Gamma \vdash \texttt{inj}(f_{\texttt{j\_inst}}) \mathbin{@} t}\ \textsf{inst\_inj}$$

$$\frac{}{\Gamma \vdash \texttt{order}(\texttt{ja\_leq}) \mathbin{@} t}\ \textsf{ja\_leqorder}$$

The inference rules in Figure 5.10 enable types to be inferred for various machine-word constants.

Finally, the following inference rules stipulate that the pointer and field assignment relations behave as partial functions:

$$\frac{}{\Gamma \vdash \texttt{fun}(\texttt{ja\_ptr}) \mathbin{@} t}\ \textsf{ja\_ptrfun} \qquad \frac{}{\Gamma \vdash \texttt{fun}(\texttt{ja\_field}) \mathbin{@} t}\ \textsf{ja\_fieldfun}$$

5.8 Java Type Safety

At this point, the encoding of the Java type system is sufficient to satisfy the memory-safety policy when all accesses to memory are performed through pointer types. Thus, I see Java type safety as a means to establish memory safety. To make this process more systematic, I introduce a set of derived rules in Section 5.8.1 and Section 5.8.2 that resemble the typing rules used by the SpecialJ PCC implementation. The ability to use Java types to satisfy memory safety depends critically on the code consumer providing run-time procedures to allocate well-typed blocks of memory, and providing appropriate types to arguments that are passed to the entry point of the program. Currently, the soundness of these type assignments can only be established informally.

A Java typing derivation is sufficient evidence to conclude `safe_rdm` or `safe_wrm` for a specific address expression. However, the formal system does not stipulate that Java type safety is a necessary condition for memory safety, so the code consumer is free to provide additional disjoint type systems to the code producer for this purpose. For example, the code consumer could simultaneously support Java, ML, and "Safe C" type systems with no particular complications.

The Java type system here is an essential part of the trusted computing base, unlike other approaches to foundational PCC [MA00, HST+02]. Note that Proposition 5.9.1 only establishes the soundness of the encoding used for type safety (*i.e.*, that it corresponds to the mathematical model). It is the soundness of the allocation procedure that provides the real evidence that type safety is an adequate justification for memory safety.

The security register $\mathtt{ta} \in \mathit{SReg}^{\mathtt{jta}}$ contains the current Java type assignment, assigning a set of word-size values to each Java type. Types are effectively assigned by a trusted allocation procedure of the run-time system for all non-null pointer values.

5.8.1 Pointer Rules

$\boxed{\mathtt{ja\_ptr}(e_{\mathtt{jta}}, e_{\mathsf{np}}, e_{\mathtt{jty}})}$

The following rule asserts that the length field of an array is an integer:

$$\frac{\Gamma \vdash \mathtt{ja\_valid}(e_{\mathtt{jta}}) \mathbin{@} t \quad \Gamma \vdash \mathtt{ja\_of}(e_{\mathtt{jta}}, e_{\mathsf{na}}, f_{\mathtt{j\_array}}(e_{\mathtt{jty}})) \mathbin{@} t \quad \Gamma \vdash e_{\mathsf{na}} \neq 0 \mathbin{@} t}{\Gamma \vdash \mathtt{ja\_ptr}(e_{\mathtt{jta}}, \mathtt{j\_len}(e_{\mathsf{na}}), \overline{\mathtt{j\_int}}) \mathbin{@} t}\ \mathsf{ja\_ptri_{len}}$$

The following rule asserts that each element of an array has the appropriate type:

$$\frac{\Gamma \vdash \mathtt{ja\_mem}(e_{\mathtt{jta}}, e_{\mathtt{m}}) \mathbin{@} t \quad \Gamma \vdash \mathtt{ja\_of}(e_{\mathtt{jta}}, e_{\mathsf{na}}, f_{\mathtt{j\_array}}(e_{\mathtt{jty}})) \mathbin{@} t \quad \Gamma \vdash e_{\mathsf{na}} \neq 0 \mathbin{@} t \quad \Gamma \vdash \mathtt{ltuw}(e_{\mathsf{ni}}, \mathtt{selw}(e_{\mathtt{m}}, \mathtt{j\_len}(e_{\mathsf{na}}))) \mathbin{@} t}{\Gamma \vdash \mathtt{ja\_ptr}(e_{\mathtt{jta}}, \mathtt{j\_elem}(e_{\mathsf{na}}, e_{\mathsf{ni}}, e_{\mathtt{jty}}), e_{\mathtt{jty}}) \mathbin{@} t}\ \mathsf{ja\_ptri_{elem}}$$

The following rule asserts that each field of an instance has its assigned type:

$$\frac{\Gamma \vdash \texttt{ja\_valid}(e_{\texttt{jta}}) \mathbin{@} t \quad \Gamma \vdash \texttt{ja\_of}(e_{\texttt{jta}}, e_{\mathsf{no}}, f_{\texttt{j\_inst}}(e_{\mathsf{nc}})) \mathbin{@} t \\ \Gamma \vdash e_{\mathsf{no}} \neq 0 \mathbin{@} t \quad \Gamma \vdash \texttt{ja\_field}(e_{\texttt{jta}}, e_{\mathsf{nc}}, e_{\mathsf{nf}}, e_{\texttt{jty}}) \mathbin{@} t}{\Gamma \vdash \texttt{ja\_ptr}(e_{\texttt{jta}}, \texttt{addw}(e_{\mathsf{no}}, e_{\mathsf{nf}}), e_{\texttt{jty}}) \mathbin{@} t} \ \mathsf{ja\_ptri_{field}}$$

The following rule asserts that each pointer is aligned to its element size:

$$\frac{\Gamma \vdash \texttt{ja\_valid}(e_{\texttt{jta}}) \mathbin{@} t \quad \Gamma \vdash \texttt{ja\_ptr}(e_{\texttt{jta}}, e_{\mathsf{np}}, e_{\texttt{jty}}) \mathbin{@} t}{\Gamma \vdash \texttt{j\_align}(e_{\mathsf{np}}, e_{\texttt{jty}}) \mathbin{@} t} \ \mathsf{ja\_ptre_{align}}$$

The following rule asserts that dereferencing a pointer results in a value of the appropriate type:

$$\frac{\Gamma \vdash \texttt{ja\_mem}(e_{\texttt{jta}}, e_{\mathtt{m}}) \mathbin{@} t \quad \Gamma \vdash \texttt{ja\_ptr}(e_{\texttt{jta}}, e_{\mathsf{np}}, e_{\texttt{jty}}) \mathbin{@} t \\ \Gamma \vdash \texttt{j\_wd}(e_{\texttt{jty}}) \mathbin{@} t}{\Gamma \vdash \texttt{ja\_of}(e_{\texttt{jta}}, \texttt{selw}(e_{\mathtt{m}}, e_{\mathsf{np}}), e_{\texttt{jty}}) \mathbin{@} t} \ \mathsf{ja\_ptre_{sel}}$$

5.8.2 Memory Rule

$\boxed{\texttt{ja\_mem}(e_{\texttt{jta}}, e_{\mathtt{m}})}$

The following rule asserts that writing to a pointer results preserves the validity of memory:

$$\frac{\Gamma \vdash \texttt{ja\_mem}(e_{\texttt{jta}}, e_{\mathtt{m}}) \mathbin{@} t \quad \Gamma \vdash \texttt{ja\_ptr}(e_{\texttt{jta}}, e_{\mathsf{np}}, e_{\texttt{jty}}) \mathbin{@} t \\ \Gamma \vdash \texttt{ja\_of}(e_{\texttt{jta}}, e_{\mathsf{n}}, e_{\texttt{jty}}) \mathbin{@} t \\ \Gamma \vdash \forall x_{\mathsf{np}}{:}\mathsf{fl}.\ \texttt{ja\_ptr\_len}(e_{\texttt{jta}}, x_{\mathsf{np}}) \supset x_{\mathsf{np}} \neq e_{\mathsf{np}} \mathbin{@} t}{\Gamma \vdash \texttt{ja\_mem}(e_{\texttt{jta}}, \texttt{updw}(e_{\mathtt{m}}, e_{\mathsf{np}}, e_{\mathsf{n}})) \mathbin{@} t} \ \mathsf{ja\_memupd}$$

5.9 Soundness

I now show that the inference rules in this chapter are sound (some rules are additionally derivable).

Let

$$\Gamma \vdash^{\mathcal{J}}_{\mathsf{sp}} J$$

assert that J is derivable from Γ using only inference rules from Chapter 3, Chapter 4, Section 5.4, Section 5.5, Section 5.6, and Section 5.7. Additionally, let

$$\Gamma \vdash^{\mathcal{J}}_{\texttt{jty}} J$$

assert that J is derivable from Γ using only inference rules from Chapter 3, Chapter 4, Section 5.4, Section 5.5, Section 5.6, Section 5.7, and Section 5.8.

Proposition 5.9.1 (Soundness) $\phi, \eta \vDash^{\mathcal{J}} J$
if $\phi, \eta \vDash^{\mathcal{J}} \Gamma$ *and* $\Gamma \vdash^{\mathcal{J}}_{\mathsf{sp}} J$ *and* $\phi|_{Reg} \in \Sigma_{\mathcal{J}(\texttt{pm})}$

Proof: by induction on the derivation of $\Gamma \vdash^{\mathcal{J}}_{\mathsf{sp}} J$, using the proof of Proposition 4.5.2 □

Proposition 5.9.2 (Derivability) $\Gamma \vdash^{\mathcal{J}}_{\mathsf{sp}} J$ *if* $\Gamma \vdash^{\mathcal{J}}_{\texttt{jty}} J$

Proof: refer to the LF implementation □

Chapter 6

Program Logic

In this chapter, I develop a *logic of programs* [Flo67, Hoa69, LS82] for proving invariance properties of machine-code programs. In general, a logic of programs (or *program logic*) consists of a set of formal inference rules that can be used, together with ordinary mathematical reasoning, to demonstrate the correctness (or safety) of a given program. A program logic embodies the abstract properties of the machine on which a program is be executed. For example, in Floyd-Hoare logic, a rule for a conditional statement allows one to conclude that if both arms of the conditional satisfy a certain property, then the conditional statement itself satisfies the property:

$$\frac{\{p \wedge e \neq 0\}\ c_1\ \{p'\} \quad \{p \wedge e = 0\}\ c_2\ \{p'\}}{\{p\}\ \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \{p'\}}\ \mathsf{if}$$

Program logics rely on mathematical reasoning to prove ordinary facts about mathematical objects that have no essential relation to programs. For example, when applying a loop rule to the correctness of a factorial procedure, one might need to show $n! = n \times (n-1)!$. Such premises are ordinarily discharged by informal algebraic reasoning. See Dijkstra [Dij76] and Reynolds [Rey81, Rey98] for a more in-depth treatment of standard program logics.

The program logic in this dissertation provides an important conceptual foundation for my approach to proof construction, and for my approach to proof engineering. The program logic additionally establishes the formal basis for the logic program for proof reconstruction that I present in Chapter 8.

In one valid approach to PCC, the code consumer might provide the code producer with a set of program-logic rules, and expect a proof of safety for each untrusted program that is made up of a tree of these rules. Such rules are ordinarily proven sound with respect to an operational semantics, but the proofs are rarely formalized to the same degree of rigor as a PCC safety proof. The code consumer would have more confidence in the program logic if the soundness argument were formalized to such a degree as to be verifiable by machine. This reduces the size of the trusted computing base (TCB), and therefore reduces the vulnerability of the code consumer, because the program logic need not be a trusted component.[1]

[1] A component must be *trusted* when it can only be verified informally. In this dissertation,

$$\frac{\vdash [e_{\mathtt{sq}}/x_{\mathtt{sq}}]\, p_{\mathsf{gl}} \mathbin{@} 0}{\vdash e_{\mathtt{sq}} \overset{p_{\mathtt{safe}}}{\rightsquigarrow} p_{\mathsf{gl}}} \rightsquigarrow\mathsf{i}_0$$

$$\downarrow$$

$$\frac{\Gamma \vdash [e_{\mathtt{sq}}/x_{\mathtt{sq}}]\, p_{\mathsf{gl}} \mathbin{@} t \quad \Gamma \vdash [a_{\mathtt{sq}}/x_{\mathtt{sq}}]\, p_{\mathsf{gl}} : \mathsf{lo}\,(a_{\mathtt{sq}})}{\Gamma \vdash \mathtt{sq} = e_{\mathtt{sq}} \supset [\mathtt{sq}/x_{\mathtt{sq}}]\,(p_{\mathtt{safe}}\,\mathcal{U}\,p_{\mathsf{gl}}) \mathbin{@} t} \; \mathtt{pl\_evali}_0^{a_{\mathtt{sq}}}$$

($\mathtt{pl\_evali}_0$ is derived formally)

Figure 6.1: Deriving an Inference Rule

In fact, the formal framework of Chapter 3 and Chapter 4 already provides the code consumer with the tools needed to accomplish exactly this task—the soundness of a program logic with respect to an operational semantics can be established entirely within the framework of temporal-logic reasoning. However, this approach can be taken a step further. Based on a suitable encoding, the rules of the program logic are not only sound, but admissible, and actually *derivable*.[2] Now, because the program logic can be synthesized from within the formalism, it need not even be a component of the code consumer—an explicit derivation of the program logic can be attached to the safety proof provided by the code producer (see Figure 6.1). In fact, the code producer is free to "invent" any suitable program logic that can be synthesized from within the formal system. This is the approach I take in this dissertation: the program logic is a derived component that is used internally in the proof reconstruction program supplied by the code producer.

6.1 Overview

The program logic is important enough conceptually that I will present it first in Section 6.3 using a higher-level syntax that is outside my standard logical formalism. I do not use this embodiment of the program logic directly—it is effectively an organizational tool that clarifies how derived rules should be developed. Thus, I will not attempt to show that the high-level view of the program logic is sound.

Note that most program logics are intended to demonstrate partial or total correctness properties. My program logic departs from this practice by demonstrating *safety* properties,[3] although it can also be applied to partial correctness. Intuitively,

the temporal-logic inference rules, the formal machine model, and the proof checker are trusted components. A key feature of PCC is that it requires only a small TCB, and thus enables a trustworthy system to be deployed.

[2] An *admissible* inference rule does not enlarge the set of theorems that are provable in a given formal system. For a *derived* inference rule, there is a single tree of inferences that establishes the conclusion from the premises for any instantiation of the rule. Because derived rules have explicit formal proofs, they need not be trusted.

[3] To be more precise, my program logic demonstrates *invariance* properties (see Section 5.4).

correctness tells the code consumer *what* the program computes, but for PCC he or she is really more interested in *how* it is computed. A novel feature of my program logic is that the inference rules are explicitly instantiated with a formal representation of the desired safety property during construction of a derivation.

After presenting the high-level view of the program logic, I show in Section 6.4 how an equivalent program logic can be *derived* entirely in terms of temporal-logic syntax and inference rules. Refer to Gordon [Gor88] for a similar approach based on higher-order logic. Thus, the program logic is an *untrusted* component of the PCC infrastructure that enables the code producer to develop a coherent theory for proving invariance properties.

In Section 6.5, I survey additional derived rules that are useful for establishing memory-safety properties. Finally, in Section 6.6, I introduce the proof outline representation that can be used to distill only the essential content out of a safety proof based on the derived program logic.

Later, in Chapter 8, I outline how the core program logic is further specialized into a derived deterministic logic program for reconstructing safety proofs. By specializing the program logic into a deterministic logic program, the SpecialJ symbolic evaluator is effectively converted from a trusted component to an untrusted component. Because the logic-program rules are more complex and idiosyncratic (due to efficiency and decidability considerations), the program logic in this chapter also serves as a useful starting point for understanding the operation of the proof reconstruction logic program.

6.2 Program Logics in PCC

To date, little direct attention has been paid to program logics in PCC research. Typically, the focus has been on developing a verification-condition (VC) generator as a self-contained artifact without much regard to the implicit program logic upon which it is based (see Necula [Nec98], for example).

A *VC generator* is a program that implicitly constructs a proof, using program-logic rules, of some property of a program. For its a result, the VC generator produces a list of premises that cannot be satisfied by applying some rule of the program logic—these are the "VCs." Each VC is typically a mathematical assertion that must be verified by hand, or by a search that can be costly, in general. Current theorem-proving technology is only capable of discovering relatively simple and/or tightly-constrained VC proofs. Typical VC generators must be provided with an explicit invariant for each loop in the program—loop invariants are also notoriously difficult to discover automatically.

The PCC focus on VC generators is in keeping with some of the VC generator literature. For example, in King's [Kin71] original paper, the program logic is not developed explicitly, though it can be inferred to be based on Floyd's [Flo67] earlier work. Significantly, Appel and Felty [AF00] originally argued in favor of a derived Floyd-Hoare logic in their seminal paper on foundational PCC, though in later work they have advocated a derived typed assembly language (TAL) instead [App01].

$$\begin{array}{c} \mathcal{D}_{\texttt{gl}} \\ \vdash [e_{\texttt{sq}}/x_{\texttt{sq}}]\, p_{\texttt{gl}} \mathbin{@} 0 \\ \\ + \\ \\ \vdash [e_{\texttt{sq}}/x_{\texttt{sq}}]\, p_{\texttt{gl}} \mathbin{@} 0 \\ \mathcal{D}_{\texttt{safe}} \\ \vdash e_{\texttt{sq}} \overset{p_{\texttt{safe}}}{\rightsquigarrow} p_{\texttt{gl}} \\ \\ \Big\downarrow \\ \\ \mathcal{D}_{\texttt{gl}} \\ \mathcal{D}_{\texttt{safe}} \\ \vdash e_{\texttt{sq}} \overset{p_{\texttt{safe}}}{\rightsquigarrow} p_{\texttt{gl}} \end{array}$$

Figure 6.2: Completing a Program-Logic Derivation

The lack of attention to program logics in the PCC literature is perhaps unfortunate, because I believe that an explicit program logic does much to clarify the reasoning principles upon which the VC generator is based.[4] In the context of my work, a program logic provides an additional benefit, because it can be *derived* entirely from within the framework of temporal logic, and thus becomes a useful artifact in and of itself.

The program logic is a natural complement to the VC generator. Essentially, any steps taken implicitly by the VC generator algorithm are assigned some explicit program-logic rule. Conversely, proofs of VCs are relegated to less formal "mathematical reasoning" by most program logics, but note that these are exactly the proof terms that come out when one uses a certifying compiler based on a VC generator. In a conventional PCC system, the VC proof is made up of explicit inferences that must be be discovered by search (typically, by using an automatic theorem prover). If one inserts the correct VC proof terms into a tree of program-logic inference rules, then one obtains a "fully explicit" proof that accounts for both mathematical and operational reasoning (see Figure 6.2). Furthermore, if the program logic itself can be foundationally established within the same framework that accounts for general mathematical reasoning, then one can reap the benefits of the program logic without increasing the complexity of the trusted framework. Note that one can also view the VC proofs as the "residue" that remains after program-logic rules are deleted from a fully-explicit safety proof.

The program logic I use is based on *symbolic evaluation* [Nec98] as opposed to a more standard Floyd-Hoare logic [Hoa69, Dij76]. In symbolic evaluation, substitutions into propositions are postponed by maintaining an explicit representation of the machine state. This is also known as *deferred substitution.* The deferred substi-

[4] For a potential practical benefit, one might simplify informal soundness proofs for VC generators by starting from a sound program logic.

tution strategy has important practical consequences because it enables a compact proof representation. The evolution of a specification over time can be described as a sequence of individual state changes, as opposed to a sequence of instantiated propositions. This makes "predicate factoring" as advocated by Appel and Felty [AF00] unnecessary, because only one specification is ever stored in the proof for a given loop or procedure.

6.3 A Logic of Programs for Invariance Properties

In the interest of a simple presentation, the program logic is formalized in natural deduction style *without* internalized hypotheses. A judgment J is affirmed without an accompanying context Γ:

$$\vdash J$$

A free hypothesis in a derivation is indicated by an italic label u. The hypothesis is *discharged* (*i.e.*, removed from the context) by the rule with the matching label. For example, the implication-introduction rule

$$\frac{\Gamma, p_1 \mathbin{@} t \vdash p_2 \mathbin{@} t}{\Gamma \vdash p_1 \supset p_2 \mathbin{@} t} \supset \mathsf{i}$$

is presented as follows without internalized hypotheses:

$$\cfrac{\cfrac{}{\vdash p_1 \mathbin{@} t}\, u \\ \vdots \\ \vdash p_2 \mathbin{@} t}{\vdash p_1 \supset p_2 \mathbin{@} t} \supset \mathsf{i}^u$$

Strictly speaking, because of this change of notation, the program-logic rules are not part of (or compatible with) the formal system of Chapter 3. However, because the role of the high-level view of the program logic is solely illustrative, this is not an important technical complication. I will include closed affirmations based on the Chapter 3 system as premises in some program-logic inference rules. These premises perform the same function as do "mathematical reasoning" premises in a standard Floyd-Hoare system. For example, to require a proof that $3 + 4 = 7$ in a program-logic rule, I would use the premise

$$\vdash 3 + 4 = 7 \mathbin{@} 0$$

Essentially, I am treating temporal logic in this section as a first-order nonmodal system to avoid the need for a distinct restricted formal system.

I refer to proof of a premise that is not based on one or more program-logic rules as a *residual proof* later in this chapter. Residual proofs correspond to "mathematical reasoning" in an ordinary program logic, and to proofs of VCs when a VC generator is employed. This distinction is crucial in Chapter 7, where I show how a complete safety proof can be constructed from the right combination of program-logic rules and residual proofs.

This is a program logic for invariance properties. I must stipulate that loop invariants, procedure specifications, and the target invariance property $p_{\texttt{safe}}$ do not contain temporal operators (*i.e.*, the security-property $p_{\texttt{sp}}$ is roughly $\Box p_{\texttt{safe}}$). Because many inference rules implicitly employ local reasoning, these rules are not sound if the properties to be established are not themselves local. This stipulation is developed into a formal locality requirement when the program logic is reconstructed as a derived system in Section 6.4.

Note that in principle, the program logic can be used to establish arbitrary invariance properties, but in this dissertation I will only apply it to memory safety and Java type safety. However, I expect that much of the existing implementation can be applied directly to certify more general safety properties, such as resource bounds and access control. Adapting a decidable type system would be particularly straightforward, because no residual proofs would be needed, and practical techniques have been developed [Nec98] for automatically synthesizing type-based loop invariants.

In an attempt to be systematic, I characterize many of the program-logic rules as either introduction or elimination rules. However, the program logic does not seem to yield a coherent type theory, at least as I present it here. For example, local soundness and local completeness (in the sense of Pfenning [Pfe99]) do not hold.

Symbolic states are represented as expressions of type `qstate`, normally in the form $\texttt{q\_mk}(c_{\texttt{pc}}, e_{\texttt{f}}, e_{\texttt{g}}, e_{\texttt{s}}, e_{\texttt{m}}, e_{\texttt{q}})$, where the subexpressions are the usual components of an extended state tuple. The program counter is ordinarily a constant to ensure that the current instruction can be identified. Each of the other components of the state are zero or more "upd" functions applied to a parameter. A rigid, discharged parameter (*e.g.*, $a_{\texttt{sq}}$) represents the unpredictable/underdetermined value of the state at the start of executing a procedure or loop body. The series of "upd" functions reflects modifications made to the machine state by previously interpreted code. For example, a state might be represented as follows at program counter 12 inside a loop:

$$\texttt{q\_mk}(12, \texttt{q\_f}(a_{\texttt{sq}}), \texttt{updg}(\texttt{q\_g}(a_{\texttt{sq}}), \overline{\texttt{eax}}, 5), \texttt{q\_s}(a_{\texttt{sq}}), \texttt{q\_m}(a_{\texttt{sq}}), \texttt{q\_q}(a_{\texttt{sq}}))$$

In this example state, the register `eax` has been explicitly loaded by a machine-language instruction inside the loop body with the value 5, but the remainder of the machine state is undetermined. The undetermined state at the head of the loop is named by the parameter $a_{\texttt{sq}}$.

By convention, loop invariants and procedure specifications have two free variables that are instantiated with either a symbolic state representation (*e.g.*, $e_{\texttt{sq}}$) or the machine-state parameter `sq`, depending on the context. The variable $x_{\texttt{sq}}$ refers to the current-time value of the machine state, whereas the variable $x_{\texttt{sq}_0}$ refers to the value of the machine state at the start of the loop or procedure. For example, the loop invariant

$$\texttt{q\_m}(x_{\texttt{sq}}) = \texttt{q\_m}(x_{\texttt{sq}_0})$$

stipulates that the memory does not change inside the body of the loop. Note that these variables are always bound and/or instantiated when a specification appears in a judgment. In Figure 6.3, I illustrate how specification variables are instantiated.

Loop Invariant

$$\begin{array}{l} \vdots \\ \texttt{sq} = e_{\mathsf{sq}} \\ \vdash [e_{\mathsf{sq}}/x_{\mathsf{sq}_0}]\,[e_{\mathsf{sq}}/x_{\mathsf{sq}}]\,p_{\mathsf{inv}_n} \mathbin{@} t \\ \texttt{sq} = a_{\mathsf{sq}} \\ \vdash [e_{\mathsf{sq}}/x_{\mathsf{sq}_0}]\,[a_{\mathsf{sq}}/x_{\mathsf{sq}}]\,p_{\mathsf{inv}_n} \mathbin{@} b \\ \vdots \\ \texttt{sq} = e'_{\mathsf{sq}} \\ \vdash [e_{\mathsf{sq}}/x_{\mathsf{sq}_0}]\,[e'_{\mathsf{sq}}/x_{\mathsf{sq}}]\,p_{\mathsf{inv}_n} \mathbin{@} t' \end{array}$$

Procedure Body

$$\begin{array}{l} \texttt{sq} = a_{\mathsf{sq}} \\ \vdash [a_{\mathsf{sq}}/x_{\mathsf{sq}_0}]\,[a_{\mathsf{sq}}/x_{\mathsf{sq}}]\,p_{\mathsf{pre}_n} \mathbin{@} b \\ \vdots \\ \texttt{ret}\langle n_{\mathsf{i}}\rangle \\ \texttt{sq} = e'_{\mathsf{sq}} \\ \vdash [a_{\mathsf{sq}}/x_{\mathsf{sq}_0}]\,[e'_{\mathsf{sq}}/x_{\mathsf{sq}}]\,p_{\mathsf{post}_n} \mathbin{@} t' \end{array}$$

Procedure Call

$$\begin{array}{l} \vdots \\ \texttt{sq} = e_{\mathsf{sq}} \\ \vdash [e_{\mathsf{sq}}/x_{\mathsf{sq}_0}]\,[e_{\mathsf{sq}}/x_{\mathsf{sq}}]\,p_{\mathsf{pre}_n} \mathbin{@} t \\ \texttt{call}\langle n_{\mathsf{i}}\rangle\langle \texttt{ea\_i}\langle n\rangle\rangle \\ \texttt{sq} = a'_{\mathsf{sq}} \\ \vdash [e_{\mathsf{sq}}/x_{\mathsf{sq}_0}]\,[a'_{\mathsf{sq}}/x_{\mathsf{sq}}]\,p_{\mathsf{post}_n} \mathbin{@} b' \\ \vdots \end{array}$$

Figure 6.3: Instantiating Specification Variables

A loop invariant (procedure preconditions and postconditions are similar) is normally in the form

$$\mathsf{pc} = e_{\mathsf{pc}} \wedge (p_{\mathsf{u}} \wedge p_{\mathsf{z}})$$

where e_{pc} is the program counter at which the loop invariant holds. p_{u} is the "unconstrained" part of the loop invariant—this proposition is a right-associative conjunction of *specification elements*: $p_1 \wedge (\ldots \wedge (p_k \wedge \top))$, each of which is a basic relation on expressions. Typically, the unconstrained part of a loop invariant is made up of type ascriptions and nonzero assertions. During verification, each specification element in the unconstrained part will be discharged individually by a specific fragment of a residual proof. p_{z} is the "automatic" part of the loop invariant—this proposition is also a right-associative conjunction of specification elements, but the specification elements are restricted to certain specific relations that can be established automatically by term rewriting. The automatic part of the loop invariant is normally made up of *preservation elements* (see Section 6.5.2) that assert that some particular aspect of the machine state is unchanged throughout the loop body.

Each specification is used symmetrically as either a hypothesis (premise) or an obligation (conclusion) in a proof, depending on the context. A loop invariant is an obligation when a loop is entered for the first time, and when the loop head is reached for a second time—a loop invariant is used hypothetically within the body of the loop. Similarly, a procedure precondition is an obligation when a procedure is called, but it is used hypothetically within the body of a procedure being verified. Finally, a procedure postcondition is used hypothetically in a continuation after a procedure is called, but it is an obligation when it is reached from the body of a procedure being verified. The following table summarizes how specifications are used:

	Hypothetical	Obligation
Loop Invariant	in loop body	at loop head
Precondition	in procedure body	at procedure call
Postcondition	after procedure call	at procedure return

Next, I explain each of the program-logic judgments in detail.

6.3.1 Transitions $\boxed{e_{\mathsf{sq}} \to e'_{\mathsf{sq}}}$

The transition judgment,

$$e_{\mathsf{sq}} \to e'_{\mathsf{sq}}$$

asserts that a transition is possible between symbolic state e_{sq} and symbolic state e'_{sq} over a single time step.

Note that given $e_{\mathsf{sq}} \to e'_{\mathsf{sq}}$,

$$\mathsf{sq} = e'_{\mathsf{sq}}$$

is the *strongest possible postcondition* relative to

$$\mathsf{sq} = e_{\mathsf{sq}}$$

in the sense of Floyd [Flo67]. Thus, the symbolic evaluator preserves maximal information about the symbolic state until a loop invariant or postcondition is reached. This is in contrast to type-based systems such as TAL [MWCG98] that *immediately* replace a state with its approximation in terms of types—however, TAL-style systems are potentially more efficient, because less state information needs to be retained as the program is checked.

The rule $\to \mathsf{i}$ introduces the transition judgment. The premises require that the transition be derivable from the formal encoding of the machine semantics.

$$\frac{\vdash \mathtt{fetch}(\mathtt{pm}, \mathtt{q\_pc}(e_{\mathtt{sq}}), e_{\mathsf{l}}) \mathbin{@} 0 \quad \vdash \mathtt{q\_next}(e_{\mathtt{sq}}, e_{\mathsf{l}}) = e'_{\mathtt{sq}} \mathbin{@} 0}{\vdash e_{\mathtt{sq}} \to e'_{\mathtt{sq}}} \to \mathsf{i}$$

6.3.2 Strict Evaluation

$\boxed{e_{\mathtt{sq}} \overset{p_{\mathtt{safe}}}{\rightsquigarrow}{}^{+} p_{\mathsf{gl}}}$

The strict evaluation judgment,

$$e_{\mathtt{sq}} \overset{p_{\mathtt{safe}}}{\rightsquigarrow}{}^{+} p_{\mathsf{gl}}$$

asserts that the invariance property $p_{\mathtt{safe}}$ holds from symbolic state $e_{\mathtt{sq}}$ until the goal property p_{gl} holds, and that at least one transition is possible before p_{gl} holds. This is a partial correctness judgment, so p_{gl} may never hold during an actual program run, but in such a case $p_{\mathtt{safe}}$ will hold indefinitely.

Strict evaluation is needed for inductive arguments where standard evaluation does not suffice. Standard evaluation is like strict evaluation, except that empty steps are allowed (see Section 6.3.3). Strict evaluation ensures that some "progress" is made during the induction step of an argument.

Strict evaluation is introduced by one of two rules ($\rightsquigarrow^{+}\mathsf{i}_1$ or $\rightsquigarrow^{+}\mathsf{i}_2$), according to whether or not a branch is possible for the current instruction. Each rule requires that the invariance property be provable for the current symbolic state, and that a transition is possible to a new symbolic state. From the new symbolic state, the invariance property must hold until the goal goal property holds, but this may be an empty step (see Section 6.3.3). The branch rule $\rightsquigarrow^{+}\mathsf{i}_2$ additionally performs a case split on whether some arbitrary expression is nonzero. In practice, the expression is instantiated to the condition of the branch instruction, and this enables one to infer the appropriate program counter from the machine semantics.

$$\frac{\vdash [e_{\mathtt{sq}}/x_{\mathtt{sq}}]\, p_{\mathtt{safe}} \mathbin{@} 0 \quad \vdash e_{\mathtt{sq}} \to e'_{\mathtt{sq}} \quad \vdash e'_{\mathtt{sq}} \overset{p_{\mathtt{safe}}}{\rightsquigarrow} p_{\mathsf{gl}}}{\vdash e_{\mathtt{sq}} \overset{p_{\mathtt{safe}}}{\rightsquigarrow}{}^{+} p_{\mathsf{gl}}} \rightsquigarrow^{+}\mathsf{i}_1$$

$$\frac{\vdash [e_{\mathtt{sq}}/x_{\mathtt{sq}}]\, p_{\mathtt{safe}} \mathbin{@} 0 \quad \begin{array}{c}\overline{\vdash J_1}\ u_1 \\ \vdots \\ \vdash e_{\mathtt{sq}} \to e'_{\mathtt{sq}_1}\end{array} \quad \begin{array}{c}\overline{\vdash J_2}\ u_2 \\ \vdots \\ \vdash e_{\mathtt{sq}} \to e'_{\mathtt{sq}_2}\end{array} \quad \begin{array}{c}\overline{\vdash J_1}\ u_1 \\ \vdots \\ \vdash e'_{\mathtt{sq}_1} \overset{p_{\mathtt{safe}}}{\rightsquigarrow} p_{\mathsf{gl}}\end{array} \quad \begin{array}{c}\overline{\vdash J_2}\ u_2 \\ \vdots \\ \vdash e'_{\mathtt{sq}_2} \overset{p_{\mathtt{safe}}}{\rightsquigarrow} p_{\mathsf{gl}}\end{array}}{\vdash e_{\mathtt{sq}} \overset{p_{\mathtt{safe}}}{\rightsquigarrow}{}^{+} p_{\mathsf{gl}}} \rightsquigarrow^{+}\mathsf{i}_2^{u_1,u_2}$$

where $J_1 \overset{\mathsf{def}}{\equiv} e_{\mathsf{n}} \neq 0 \mathbin{@} 0$
and $J_2 \overset{\mathsf{def}}{\equiv} e_{\mathsf{n}} = 0 \mathbin{@} 0$

Strict evaluation is eliminated by the rule $\rightsquigarrow^{+}\mathsf{e}$, which simply converts it to standard evaluation.

$$\frac{\vdash e_{\mathsf{sq}} \overset{p_{\mathtt{safe}}}{\rightsquigarrow}{}^{+} p_{\mathsf{gl}}}{\vdash e_{\mathsf{sq}} \overset{p_{\mathtt{safe}}}{\rightsquigarrow} p_{\mathsf{gl}}} \; \rightsquigarrow^{+}\mathsf{e}$$

6.3.3 Evaluation

$\boxed{e_{\mathsf{sq}} \overset{p_{\mathtt{safe}}}{\rightsquigarrow} p_{\mathsf{gl}}}$

The standard evaluation judgment,

$$e_{\mathsf{sq}} \overset{p_{\mathtt{safe}}}{\rightsquigarrow} p_{\mathsf{gl}}$$

asserts that the invariance property $p_{\mathtt{safe}}$ holds from symbolic state e_{sq} until the goal property p_{gl} holds. This judgment is satisfied if p_{gl} simply holds for e_{sq}. As for strict evaluation, $p_{\mathtt{safe}}$ will hold indefinitely if p_{gl} never holds.

Standard evaluation is introduced by $\rightsquigarrow\mathsf{i}_0$ when the goal property is reached.

$$\frac{\vdash [e_{\mathsf{sq}}/x_{\mathsf{sq}}]\, p_{\mathsf{gl}} \,@\, 0}{\vdash e_{\mathsf{sq}} \overset{p_{\mathtt{safe}}}{\rightsquigarrow} p_{\mathsf{gl}}} \; \rightsquigarrow\mathsf{i}_0$$

The loop rule $\rightsquigarrow\mathsf{loop}$ enables the invariance property to be inferred throughout a closed loop, at least until the goal property is satisfied. The first premise requires that the loop invariant eventually hold from the current state, after traveling along a safe path (unless the goal property is reached first). The second premise requires that whenever the loop invariant holds for a state, it will eventually hold from that state, after traveling along a safe path (again, unless the goal property is reached first). The path inferred for the second premise must be nonempty to rule out the case where the hypothesis is used to satisfy the premise with $\rightsquigarrow\mathsf{i}_0$. Thus, the loop rule lets one travel from invariant to invariant along any safe path, even an infinite one, but each interval between invariants must be finite.

$$\frac{\vdash e_{\mathsf{sq}} \overset{p_{\mathtt{safe}}}{\rightsquigarrow} [e_{\mathsf{sq}}/x_{\mathsf{sq}_0}]\, p_{\mathsf{i}} \vee p_{\mathsf{gl}} \qquad \begin{array}{c} \dfrac{}{\vdash [e_{\mathsf{sq}}/x_{\mathsf{sq}_0}]\,[a_{\mathsf{sq}}/x_{\mathsf{sq}}]\, p_{\mathsf{i}} \,@\, 0}\; u \\ \vdots \\ \vdash a_{\mathsf{sq}} \overset{p_{\mathtt{safe}}}{\rightsquigarrow}{}^{+} [e_{\mathsf{sq}}/x_{\mathsf{sq}_0}]\, p_{\mathsf{i}} \vee p_{\mathsf{gl}} \end{array}}{\vdash e_{\mathsf{sq}} \overset{p_{\mathtt{safe}}}{\rightsquigarrow} p_{\mathsf{gl}}} \; \rightsquigarrow\mathsf{loop}^{a_{\mathsf{sq}},u}$$

6.3.4 Procedures

$\boxed{p_{\mathsf{p}} \overset{p_{\mathtt{safe}}}{\rightsquigarrow}{}^{\star} p_{\mathsf{q}}}$

The procedure-call judgment,

$$p_{\mathsf{p}} \overset{p_{\mathtt{safe}}}{\rightsquigarrow}{}^{\star} p_{\mathsf{q}}$$

asserts that the invariance property $p_{\mathtt{safe}}$ holds from any symbolic state satisfying the precondition p_{p} until the postcondition p_{q} is satisfied. The precondition typically identifies the address at which the code of the procedure is located. Thus, a precondition typically holds immediately after a call instruction is executed, whereas a postcondition typically holds immediately after a return instruction is executed. The procedure-call judgment only ensures partial correctness, so the postcondition need never hold if the invariance property holds indefinitely.

The procedure-call judgment is introduced by the rule $\leadsto^{\star}\mathsf{i}$, which requires that from any symbolic state satisfying the precondition, the postcondition must eventually be satisfied after tracing out a safe path.

$$\dfrac{\begin{array}{c}\overline{\vdash [a_{\mathsf{sq}}/x_{\mathsf{sq}_0}]\,[a_{\mathsf{sq}}/x_{\mathsf{sq}}]\,p_{\mathsf{p}} \mathbin{@} 0}\;u\\ \vdots \\ \vdash a_{\mathsf{sq}} \overset{p_{\mathsf{safe}}}{\leadsto} [a_{\mathsf{sq}}/x_{\mathsf{sq}_0}]\,p_{\mathsf{q}}\end{array}}{\vdash p_{\mathsf{p}} \overset{p_{\mathsf{safe}}}{\leadsto}{}^{\star}\, p_{\mathsf{q}}}\;\leadsto^{\star}\mathsf{i}^{a_{\mathsf{sq}},u}$$

The procedure-call judgment is eliminated by the rule $\leadsto^{\star}\mathsf{e}$, which resembles the loop rule. The first premise requires that the current state satisfy the precondition of the procedure. The second premise requires that whenever the postcondition holds for a state, the goal property will eventually hold from that state, after traveling along a safe path.

$$\dfrac{\vdash p_{\mathsf{p}} \overset{p_{\mathsf{safe}}}{\leadsto}{}^{\star}\, p_{\mathsf{q}} \qquad \vdash [e_{\mathsf{sq}}/x_{\mathsf{sq}_0}]\,[e_{\mathsf{sq}}/x_{\mathsf{sq}}]\,p_{\mathsf{p}} \mathbin{@} 0 \qquad \begin{array}{c}\overline{\vdash [e_{\mathsf{sq}}/x_{\mathsf{sq}_0}]\,[a_{\mathsf{sq}}/x_{\mathsf{sq}}]\,p_{\mathsf{q}} \mathbin{@} 0}\;u\\ \vdots \\ \vdash a_{\mathsf{sq}} \overset{p_{\mathsf{safe}}}{\leadsto} p_{\mathsf{gl}}\end{array}}{\vdash e_{\mathsf{sq}} \overset{p_{\mathsf{safe}}}{\leadsto} p_{\mathsf{gl}}}\;\leadsto^{\star}\mathsf{e}^{a_{\mathsf{sq}},u}$$

The recursion rule $\leadsto^{\star}\mathsf{rec}$ enables a procedure to call itself recursively (recursion would not be provably safe without this rule). The hypothetical inference rule u in this rule enables the procedure to use its own specification during its safety proof, provided that the recursive call is made with a smaller index expression (*e.g.*, the stack pointer). This latter condition enables an inductive argument that shows the soundness of this rule. The notation $u^{(e'_{\mathsf{n}})}$ in this rule indicates that e'_{n} is *schematic* (as opposed to *parametric*) with respect to u. Thus, the inference rule u may be instantiated at various different values of e'_{n}, as long as each is smaller than the parameter a_{n}. Note that this rule supports only simple forms of recursion where a single procedure calls itself. The rule can be extended to support mutual recursion by forming the conjunction of all procedures in each connected component of the call graph, but I will not formalize such a rule here in order to keep the presentation simple.

$$\dfrac{\begin{array}{c}\dfrac{\vdash \mathtt{ltuw}(e'_{\mathsf{n}}, a_{\mathsf{n}}) \mathbin{@} 0}{\vdash [e'_{\mathsf{n}}/x_{\mathsf{n}}]\,p_{\mathsf{p}} \overset{p_{\mathsf{safe}}}{\leadsto}{}^{\star}\, p_{\mathsf{q}}}\;u^{(e'_{\mathsf{n}})}\\ \vdots \\ \vdash [a_{\mathsf{n}}/x_{\mathsf{n}}]\,p_{\mathsf{p}} \overset{p_{\mathsf{safe}}}{\leadsto}{}^{\star}\, p_{\mathsf{q}}\end{array}}{\vdash [e_{\mathsf{n}}/x_{\mathsf{n}}]\,p_{\mathsf{p}} \overset{p_{\mathsf{safe}}}{\leadsto}{}^{\star}\, p_{\mathsf{q}}}\;\leadsto^{\star}\mathsf{rec}^{a_{\mathsf{n}},u}$$

6.3.5 Demonstration

At this point, I will demonstrate how a safety property of the factorial program from Section 4.1 is derived using the program-logic. I will also build on this example in subsequent chapters to illustrate the techniques for proof generation and proof engineering. In the interest of brevity, I will restrict my attention to the body of the factorial loop:

$$
\begin{array}{ll}
16 & \texttt{op2}\langle 3\rangle\langle\texttt{op2\_imul}\rangle\langle\texttt{ea\_r}\langle\texttt{ebx}\rangle\rangle\langle\texttt{ea\_r}\langle\texttt{eax}\rangle\rangle \\
19 & \texttt{op1}\langle 1\rangle\langle\texttt{op1\_inc}\rangle\langle\texttt{ea\_r}\langle\texttt{ebx}\rangle\rangle \\
20 & \texttt{cmp}\langle 4\rangle\langle\texttt{ea\_s}\langle\texttt{ma\_r}\langle\texttt{esp}\rangle\langle 1\rangle\langle\texttt{ma\_d}\langle 4\rangle\rangle\rangle\rangle\langle\texttt{ea\_r}\langle\texttt{ebx}\rangle\rangle \\
24 & \texttt{j}\langle 2\rangle\langle\texttt{cop\_ng}\rangle\langle 2^{32}-10\rangle \\
26 & \texttt{ret}\langle 1\rangle
\end{array}
$$

and I will only show that the memory is valid throughout the loop, and that the type environment is unchanged. Let Φ_{16} be the above program.

The specification elements $p_{\mathsf{u}_{16}}$ for the loop invariant are thus

$$p_{\mathsf{u}_{16}} \stackrel{\mathsf{def}}{\equiv} \texttt{js\_mem}(\texttt{q\_ts}(x_{\mathsf{sq}}), \texttt{q\_m}(x_{\mathsf{sq}})) \wedge \top$$

and the preservation elements $p_{\mathsf{z}_{16}}$ are

$$p_{\mathsf{z}_{16}} \stackrel{\mathsf{def}}{\equiv} \texttt{q\_ts}(x_{\mathsf{sq}}) = \texttt{q\_ts}(x_{\mathsf{sq}_0}) \wedge \top$$

and the complete loop invariant is

$$p_{\mathsf{i}_{16}} \stackrel{\mathsf{def}}{\equiv} \texttt{q\_pc}(x_{\mathsf{sq}}) = 16 \wedge (p_{\mathsf{u}_{16}} \wedge p_{\mathsf{z}_{16}})$$

which is associated with program address 16.

The invariance property will be based on the standard instruction-level memory-safety property $\texttt{safe}$ from Section 5.6:

$$p_{\texttt{safe}} \stackrel{\mathsf{def}}{\equiv} \exists x_{\mathsf{l}}\!:\!\mathsf{fl}.\, \texttt{fetch}(\mathsf{pm}, \texttt{q\_pc}(x_{\mathsf{sq}}), x_{\mathsf{l}}) \wedge \texttt{safe\_inst}(x_{\mathsf{sq}}, x_{\mathsf{l}})$$

The safety property will hold for an arbitrary initial state, so let a_{sq_0} be a parameter that stands for this state. I intend to show that the memory remains valid and that the type environment is unchanged when program address 26 is reached,[5] so the goal property $p_{\mathsf{gl}_{26}}$ is

$$p_{\mathsf{gl}_{26}} \stackrel{\mathsf{def}}{\equiv} \texttt{q\_pc}(x_{\mathsf{sq}}) = 26 \wedge [a_{\mathsf{sq}_0}/x_{\mathsf{sq}_0}]\,(p_{\mathsf{u}_{16}} \wedge p_{\mathsf{z}_{16}})$$

I presume that the memory is valid before entering the loop. The ultimate objective is to apply the loop rule at address 16, so the safety derivation proceeds as follows:

[5] If ever—the program logic only ensures partial correctness.

Proposition 6.3.1 (Safety of Factorial Loop) $\vdash a_{\mathtt{sq}_0} \overset{p_{\mathtt{safe}}}{\rightsquigarrow} p_{\mathsf{gl}_{26}}$
if $\vdash \mathtt{js\_mem}(\mathtt{q\_ts}(a_{\mathtt{sq}_0}), \mathtt{q\_m}(a_{\mathtt{sq}_0})) \mathbin{@} 0$ *and* $\mathcal{J}(\mathtt{pm}) = \Phi_{16}$

PROOF:

$\vdash \mathtt{js\_mem}(\mathtt{q\_ts}(a_{\mathtt{sq}_0}), \mathtt{q\_m}(a_{\mathtt{sq}_0})) \mathbin{@} 0$	Prem.
$\vdash [a_{\mathtt{sq}_0}/x_{\mathtt{sq}}]\,([a_{\mathtt{sq}_0}/x_{\mathtt{sq}_0}]\,p_{\mathsf{i}_{16}} \vee p_{\mathsf{gl}_{26}}) \mathbin{@} 0$	App. $\vee\mathsf{il}, \wedge\mathsf{i}$, etc.
$\vdash a_{\mathtt{sq}_0} \overset{p_{\mathtt{safe}}}{\rightsquigarrow} [a_{\mathtt{sq}_0}/x_{\mathtt{sq}_0}]\,p_{\mathsf{i}_{16}} \vee p_{\mathsf{gl}_{26}}$	App. $\rightsquigarrow\mathsf{i}_0$
for all $a_{\mathtt{sq}}$	
$\quad\vdash [a_{\mathtt{sq}_0}/x_{\mathtt{sq}_0}]\,[a_{\mathtt{sq}}/x_{\mathtt{sq}}]\,p_{\mathsf{i}_{16}} \mathbin{@} 0$	Hyp.
$\quad$let $e_{\mathtt{sq}_{16}} = \mathtt{q\_mk}(16, \mathtt{q\_f}(a_{\mathtt{sq}}), \mathtt{q\_g}(a_{\mathtt{sq}}), \mathtt{q\_s}(a_{\mathtt{sq}}), \mathtt{q\_m}(a_{\mathtt{sq}}), \mathtt{q\_q}(a_{\mathtt{sq}}))$	
$\quad\vdash e_{\mathtt{sq}_{16}} = a_{\mathtt{sq}} \mathbin{@} 0$	Def. $\mathtt{q\_mk}$, etc.
$\quad\vdash [e_{\mathtt{sq}_{16}}/x_{\mathtt{sq}}]\,p_{\mathtt{safe}} \mathbin{@} 0$	App. $\mathsf{safe\_inst\_op2}$, etc.
$\quad$let $e_{\mathtt{eax}_{19}} = \mathtt{selg}(\mathtt{q\_g}(a_{\mathtt{sq}}), \overline{\mathtt{eax}})$	
$\quad$let $e_{\mathtt{ebx}_{19}} = \mathtt{selg}(\mathtt{q\_g}(a_{\mathtt{sq}}), \overline{\mathtt{ebx}})$	
$\quad$let $e_{\mathtt{f}_{19}} = \mathtt{updf2}(\overline{\mathtt{op2\_imul}}, \mathtt{q\_f}(a_{\mathtt{sq}}), e_{\mathtt{eax}_{19}}, e_{\mathtt{ebx}_{19}})$	
$\quad$let $e_{\mathtt{g}_{19}} = \mathtt{updg}(\mathtt{q\_g}(a_{\mathtt{sq}}), \overline{\mathtt{eax}}, \mathtt{mulw}(e_{\mathtt{eax}_{19}}, e_{\mathtt{ebx}_{19}}))$	
$\quad$let $e_{\mathtt{sq}_{19}} = \mathtt{q\_mk}(19, e_{\mathtt{f}_{19}}, e_{\mathtt{g}_{19}}, \mathtt{q\_s}(a_{\mathtt{sq}}), \mathtt{q\_m}(a_{\mathtt{sq}}), \mathtt{q\_q}(a_{\mathtt{sq}}))$	
$\quad\vdash e_{\mathtt{sq}_{16}} \to e_{\mathtt{sq}_{19}}$	App. $\to\mathsf{i}$, etc.
$\quad\vdash [e_{\mathtt{sq}_{19}}/x_{\mathtt{sq}}]\,p_{\mathtt{safe}} \mathbin{@} 0$	App. $\mathsf{safe\_inst\_op1}$, etc.
$\quad$let $e_{\mathtt{f}_{20}} = \mathtt{updf1}(\overline{\mathtt{op1\_inc}}, e_{\mathtt{f}_{19}}, e_{\mathtt{ebx}_{19}})$	
$\quad$let $e_{\mathtt{g}_{20}} = \mathtt{updg}(e_{\mathtt{g}_{19}}, \overline{\mathtt{ebx}}, \mathtt{addw}(e_{\mathtt{ebx}_{19}}, 1))$	
$\quad$let $e_{\mathtt{sq}_{20}} = \mathtt{q\_mk}(20, e_{\mathtt{f}_{20}}, e_{\mathtt{g}_{20}}, \mathtt{q\_s}(a_{\mathtt{sq}}), \mathtt{q\_m}(a_{\mathtt{sq}}), \mathtt{q\_q}(a_{\mathtt{sq}}))$	
$\quad\vdash e_{\mathtt{sq}_{19}} \to e_{\mathtt{sq}_{20}}$	App. $\to\mathsf{i}$, etc.
$\quad\vdash [e_{\mathtt{sq}_{20}}/x_{\mathtt{sq}}]\,p_{\mathtt{safe}} \mathbin{@} 0$	App. $\mathsf{safe\_inst\_op2n}$, etc.
$\quad$let $e_{\mathtt{f}_{24}} = \mathtt{updf2}(\overline{\mathtt{op2\_sub}}, e_{\mathtt{f}_{20}}, \mathtt{addw}(e_{\mathtt{ebx}_{19}}, 1), \mathtt{ea\_sel}(\mathtt{left}(e_{\mathtt{sq}_{20}}), \ldots))$	
$\quad$let $e_{\mathtt{sq}_{24}} = \mathtt{q\_mk}(24, e_{\mathtt{f}_{24}}, e_{\mathtt{g}_{20}}, \mathtt{q\_s}(a_{\mathtt{sq}}), \mathtt{q\_m}(a_{\mathtt{sq}}), \mathtt{q\_q}(a_{\mathtt{sq}}))$	
$\quad\vdash e_{\mathtt{sq}_{20}} \to e_{\mathtt{sq}_{24}}$	App. $\to\mathsf{i}$, etc.
$\quad\vdash [e_{\mathtt{sq}_{24}}/x_{\mathtt{sq}}]\,p_{\mathtt{safe}} \mathbin{@} 0$	App. $\mathsf{safe\_inst\_j}$, etc.
$\qquad\vdash \mathtt{geqw}(\mathtt{ea\_sel}(\mathtt{left}(e_{\mathtt{sq}_{20}}), \ldots), \mathtt{addw}(e_{\mathtt{ebx}_{19}}, 1)) \mathbin{@} 0$	Hyp.
$\qquad$let $e'_{\mathtt{sq}_{16}} = \mathtt{q\_mk}(16, e_{\mathtt{f}_{24}}, e_{\mathtt{g}_{20}}, \mathtt{q\_s}(a_{\mathtt{sq}}), \mathtt{q\_m}(a_{\mathtt{sq}}), \mathtt{q\_q}(a_{\mathtt{sq}}))$	
$\qquad\vdash e_{\mathtt{sq}_{24}} \to e'_{\mathtt{sq}_{16}}$	App. $\to\mathsf{i}$, etc.
$\qquad\vdash [e'_{\mathtt{sq}_{16}}/x_{\mathtt{sq}}]\,([a_{\mathtt{sq}_0}/x_{\mathtt{sq}_0}]\,p_{\mathsf{i}_{16}} \vee p_{\mathsf{gl}_{26}}) \mathbin{@} 0$	App. $\vee\mathsf{il}, \wedge\mathsf{i}$, etc.
$\qquad\vdash e'_{\mathtt{sq}_{16}} \overset{p_{\mathtt{safe}}}{\rightsquigarrow} [a_{\mathtt{sq}_0}/x_{\mathtt{sq}_0}]\,p_{\mathsf{i}_{16}} \vee p_{\mathsf{gl}_{26}}$	App. $\rightsquigarrow\mathsf{i}_0$
$\qquad\vdash \mathtt{ltw}(\mathtt{ea\_sel}(\mathtt{left}(e_{\mathtt{sq}_{20}}), \ldots), \mathtt{addw}(e_{\mathtt{ebx}_{19}}, 1)) \mathbin{@} 0$	Hyp.
$\qquad$let $e_{\mathtt{sq}_{26}} = \mathtt{q\_mk}(26, e_{\mathtt{f}_{24}}, e_{\mathtt{g}_{20}}, \mathtt{q\_s}(a_{\mathtt{sq}}), \mathtt{q\_m}(a_{\mathtt{sq}}), \mathtt{q\_q}(a_{\mathtt{sq}}))$	
$\qquad\vdash e_{\mathtt{sq}_{24}} \to e_{\mathtt{sq}_{26}}$	App. $\to\mathsf{i}$, etc.
$\qquad\vdash [e_{\mathtt{sq}_{26}}/x_{\mathtt{sq}}]\,([a_{\mathtt{sq}_0}/x_{\mathtt{sq}_0}]\,p_{\mathsf{i}_{16}} \vee p_{\mathsf{gl}_{26}}) \mathbin{@} 0$	App. $\vee\mathsf{ir}, \wedge\mathsf{i}$, etc.
$\qquad\vdash e_{\mathtt{sq}_{26}} \overset{p_{\mathtt{safe}}}{\rightsquigarrow} [a_{\mathtt{sq}_0}/x_{\mathtt{sq}_0}]\,p_{\mathsf{i}_{16}} \vee p_{\mathsf{gl}_{26}}$	App. $\rightsquigarrow\mathsf{i}_0$
$\quad\vdash e_{\mathtt{sq}_{24}} \overset{p_{\mathtt{safe}}}{\rightsquigarrow}{}^{+} [a_{\mathtt{sq}_0}/x_{\mathtt{sq}_0}]\,p_{\mathsf{i}_{16}} \vee p_{\mathsf{gl}_{26}}$	App. $\rightsquigarrow^{+}\mathsf{i}_2$
$\quad\vdash e_{\mathtt{sq}_{24}} \overset{p_{\mathtt{safe}}}{\rightsquigarrow} [a_{\mathtt{sq}_0}/x_{\mathtt{sq}_0}]\,p_{\mathsf{i}_{16}} \vee p_{\mathsf{gl}_{26}}$	App. $\rightsquigarrow^{+}\mathsf{e}$
$\quad\vdash e_{\mathtt{sq}_{20}} \overset{p_{\mathtt{safe}}}{\rightsquigarrow} [a_{\mathtt{sq}_0}/x_{\mathtt{sq}_0}]\,p_{\mathsf{i}_{16}} \vee p_{\mathsf{gl}_{26}}$	App. $\rightsquigarrow^{+}\mathsf{i}_1, \rightsquigarrow^{+}\mathsf{e}$

$$\vdash e_{\mathsf{sq}_{19}} \overset{p_{\mathtt{safe}}}{\leadsto} [a_{\mathsf{sq}_0}/x_{\mathsf{sq}_0}]\, p_{\mathsf{i}_{16}} \vee p_{\mathsf{gl}_{26}} \qquad \text{App. } \leadsto^{+}\mathsf{i}_1, \leadsto^{+}\mathsf{e}$$
$$\vdash a_{\mathsf{sq}} \overset{p_{\mathtt{safe}}}{\leadsto}{}^{+} [a_{\mathsf{sq}_0}/x_{\mathsf{sq}_0}]\, p_{\mathsf{i}_{16}} \vee p_{\mathsf{gl}_{26}} \qquad \text{App. } \leadsto^{+}\mathsf{i}_1$$
$$\vdash a_{\mathsf{sq}_0} \overset{p_{\mathtt{safe}}}{\leadsto} p_{\mathsf{gl}_{26}} \qquad \text{App. } \leadsto\mathsf{loop}$$

$\square$

Because the purpose of this demonstration is to show how the program-logic inference rules are applied, I omit the derivations of residual proofs in the interest of brevity.

6.4 Deriving the Logic of Programs

As I argue in the introduction to this chapter, the enforcement mechanism is more trustworthy if the program logic is derived by the code producer, as opposed to being trusted by the code consumer. In this section, I show how the program logic of Section 6.3 can be reconstructed in terms of derived temporal-logic constructs.

For each judgment I introduce in Section 6.3, I specify a proposition in temporal logic that expresses the content of that judgment. Similarly, for each inference rule I introduce in Section 6.3, I derive an inference rule that mimics the action of the original rule.

There is a natural correspondence between temporal reasoning and program logics. Essentially, any rule that involves reasoning at more than one instant properly belongs to the domain of the program logic. Similarly, any rule that involves reasoning within a single time step can be assigned to a residual proof. This is the discipline I follow in this chapter: temporal operators appear only where program-logic rules are involved. This is also the rationale for segregating the "support" rules into their own section (see Section 6.5), even though these rules are critical to the foundation of the logic program for proof reconstruction.

Because the semantics of the abstract machine is presented in terms of state changes over time, explicit times will also appear in the formal derivation of the program logic. Essentially, whenever a single instruction is executed, the current time step t will be incremented by one $(t + 1)$ in the conclusion of the rule. Additionally, an abstract time parameter b is introduced whenever a property must hold for all possible future times. For example, a loop invariant must hold for any time at which the particular program counter is encountered, so a time parameter is introduced when verifying the body of a loop.

6.4.1 Semantic Rigidity $\boxed{\mathtt{ri}(e)}$

The premises of derived program-logic rules may be assigned different time instants. In order to know that the component expressions of these premises denote the same value at different times, one must know that the expressions are rigid. However, because the usual rigidity judgment of Chapter 3 is a purely syntactic condition that does not admit a substitution principle, I derive an equivalent conception of rigidity that is encoded as an ordinary proposition. This latter encoding allows the standard equality-based substitutions to be carried out on rigidity judgments.

$\mathtt{ri}(e)$ holds when e denotes a rigid value, and is defined as follows:

$$\mathtt{ri}(e) \stackrel{\mathsf{def}}{\equiv} \forall x\!:\!\mathsf{ri}.\ \Diamond(e = x) \supset \Box(e = x)$$

It asserts that whenever e is equal to a rigid variable x, it is always equal to x.

$\mathtt{ri}(e)$ is introduced by the derived rule $\mathtt{ri}\mathsf{i}$:

$$\frac{\Gamma \vdash e\!:\!\mathsf{ri}}{\Gamma \vdash \mathtt{ri}(e) \mathbin{@} t}\ \mathtt{ri}\mathsf{i}$$

The premise requires that e is rigid in the standard, syntactic rigidity system.

$\mathtt{ri}(e)$ is eliminated by the derived rule $\mathtt{rie}$:

$$\frac{\Gamma \vdash \mathtt{ri}(e) \mathbin{@} t_0 \quad \Gamma \vdash e'\!:\!\mathsf{ri} \quad \Gamma \vdash e = e' \mathbin{@} t \quad \Gamma \vdash t_0 \leq t \quad \Gamma \vdash t_0 \leq t'}{\Gamma \vdash e = e' \mathbin{@} t'}\ \mathtt{rie}$$

Given a syntactically rigid expression e', it allows one to conclude that e is equal to e' at an arbitrary future time if e is equal to e' at some time.

Later in this section, I will use a semantic rigidity premise in a derived inference rule whenever an expression is referred to at more than one time instant.

6.4.2 Transitions

$\boxed{\mathtt{pl\_next}(e_{\mathtt{sq}}, e'_{\mathtt{sq}})}$

The transition judgment $e_{\mathtt{sq}} \to e'_{\mathtt{sq}}$ is encoded by the proposition $\mathtt{pl\_next}(e_{\mathtt{sq}}, e'_{\mathtt{sq}})$ as follows:

$$\mathtt{pl\_next}(e_{\mathtt{sq}}, e'_{\mathtt{sq}}) \stackrel{\mathsf{def}}{\equiv} \exists x_{\mathsf{I}}\!:\!\mathsf{ri}.\ \mathtt{fetch}(\mathtt{pm}, \mathtt{q\_pc}(e_{\mathtt{sq}}), x_{\mathsf{I}}) \wedge \mathtt{q\_next}(e_{\mathtt{sq}}, x_{\mathsf{I}}) = e'_{\mathtt{sq}}$$

It asserts that the instruction in the program memory at the program counter of $e_{\mathtt{sq}}$ takes $e_{\mathtt{sq}}$ to $e'_{\mathtt{sq}}$ according to the machine semantics.

$\mathtt{pl\_next}$ is introduced by the derived rule $\mathtt{pl\_next}\mathsf{i}$, which corresponds directly to $\to\mathsf{i}$:

$$\frac{\Gamma \vdash \mathtt{fetch}(\mathtt{pm}, \mathtt{q\_pc}(e_{\mathtt{sq}}), e_{\mathsf{I}}) \mathbin{@} t \quad \Gamma \vdash \mathtt{q\_next}(e_{\mathtt{sq}}, e_{\mathsf{I}}) = e'_{\mathtt{sq}} \mathbin{@} t}{\Gamma \vdash \mathtt{pl\_next}(e_{\mathtt{sq}}, e'_{\mathtt{sq}}) \mathbin{@} t}\ \mathtt{pl\_next}\mathsf{i}$$

6.4.3 Strict Evaluation

$\boxed{\mathtt{pl\_eval+}(p_{\mathtt{safe}}, e_{\mathtt{sq}}, p_{\mathsf{gl}})}$

The proposition $\mathtt{pl\_eval+}(p_{\mathtt{safe}}, e_{\mathtt{sq}}, p_{\mathsf{gl}})$ encodes the strict evaluation judgment $e_{\mathtt{sq}} \overset{p_{\mathtt{safe}}}{\leadsto}{}^{+} p_{\mathsf{gl}}$ as follows:

$$\mathtt{pl\_eval+}(p_{\mathtt{safe}}, e_{\mathtt{sq}}, p_{\mathsf{gl}}) \stackrel{\mathsf{def}}{\equiv} \mathtt{sq} = e_{\mathtt{sq}} \supset [\mathtt{sq}/x_{\mathtt{sq}}]\,(p_{\mathtt{safe}} \wedge \bigcirc(p_{\mathtt{safe}}\, \mathcal{U}\, p_{\mathsf{gl}}))$$

It asserts that whenever $e_{\mathtt{sq}}$ is the current machine state, the invariance property $p_{\mathtt{safe}}$ will hold for the machine state until the goal property p_{gl} holds for the machine state. Evaluation is "strict" because this proposition is not implied immediately by the goal proposition—at least one step must be taken before it is sufficient to establish strict evaluation.

$\mathtt{pl\_eval+}$ can be introduced by the derived rule $\mathtt{pl\_eval+}\mathsf{i}_1$, which corresponds to $\leadsto^{+}\mathsf{i}_1$:

$$\frac{\begin{array}{l}\Gamma \vdash [e_{\mathsf{sq}}/x_{\mathsf{sq}}]\, p_{\mathtt{safe}} \mathbin{@} t \quad \Gamma \vdash [a_{\mathsf{sq}}/x_{\mathsf{sq}}]\, p_{\mathtt{safe}} : \mathsf{lo}\ (a_{\mathsf{sq}}) \\ \Gamma \vdash \mathtt{pl\_next}(e_{\mathsf{sq}}, e'_{\mathsf{sq}}) \mathbin{@} t \quad \Gamma \vdash \mathtt{ri}(e'_{\mathsf{sq}}) \mathbin{@} 0 \\ \Gamma \vdash \mathtt{pl\_eval}(p_{\mathtt{safe}}, e'_{\mathsf{sq}}, p_{\mathsf{gl}}) \mathbin{@} t+1\end{array}}{\Gamma \vdash \mathtt{pl\_eval+}(p_{\mathtt{safe}}, e_{\mathsf{sq}}, p_{\mathsf{gl}}) \mathbin{@} t}\ \mathtt{pl\_eval+}\mathsf{i}_1^{a_{\mathsf{sq}}}$$

$\mathtt{pl\_eval+}\mathsf{i}_1$ has additional premises which ensure that $p_{\mathtt{safe}}$ is an invariance property (*i.e.*, it has no temporal operators), and that e'_{sq} is rigid. The rigidity of e'_{sq} ensures that it has the same value at time t and time $t+1$.

$\mathtt{pl\_eval+}$ can also be introduced by the derived rule $\mathtt{pl\_eval+}\mathsf{i}_2$, which corresponds to $\rightsquigarrow^{+}\mathsf{i}_2$:

$$\frac{\begin{array}{l}\Gamma \vdash [e_{\mathsf{sq}}/x_{\mathsf{sq}}]\, p_{\mathtt{safe}} \mathbin{@} t \quad \Gamma \vdash [a_{\mathsf{sq}}/x_{\mathsf{sq}}]\, p_{\mathtt{safe}} : \mathsf{lo}\ (a_{\mathsf{sq}}) \\ \Gamma,\ e_{\mathsf{n}} \neq 0 \mathbin{@} t \vdash \mathtt{pl\_next}(e_{\mathsf{sq}}, e'_{\mathsf{sq}_1}) \mathbin{@} t \\ \Gamma,\ e_{\mathsf{n}} = 0 \mathbin{@} t \vdash \mathtt{pl\_next}(e_{\mathsf{sq}}, e'_{\mathsf{sq}_2}) \mathbin{@} t \\ \Gamma \vdash \mathtt{ri}(e'_{\mathsf{sq}_1}) \mathbin{@} 0 \quad \Gamma \vdash \mathtt{ri}(e'_{\mathsf{sq}_2}) \mathbin{@} 0 \quad \Gamma \vdash \mathtt{ri}(e_{\mathsf{n}}) \mathbin{@} 0 \\ \Gamma,\ e_{\mathsf{n}} \neq 0 \mathbin{@} t+1 \vdash \mathtt{pl\_eval}(p_{\mathtt{safe}}, e'_{\mathsf{sq}_1}, p_{\mathsf{gl}}) \mathbin{@} t+1 \\ \Gamma,\ e_{\mathsf{n}} = 0 \mathbin{@} t+1 \vdash \mathtt{pl\_eval}(p_{\mathtt{safe}}, e'_{\mathsf{sq}_2}, p_{\mathsf{gl}}) \mathbin{@} t+1\end{array}}{\Gamma \vdash \mathtt{pl\_eval+}(p_{\mathtt{safe}}, e_{\mathsf{sq}}, p_{\mathsf{gl}}) \mathbin{@} t}\ \mathtt{pl\_eval+}\mathsf{i}_2^{a_{\mathsf{sq}}}$$

$\mathtt{pl\_eval+}\mathsf{i}_2$ has additional premises, as for $\mathtt{pl\_eval+}\mathsf{i}_1$.

$\mathtt{pl\_eval+}$ is eliminated by the derived rule $\mathtt{pl\_eval+}\mathsf{e}$, which corresponds directly to $\rightsquigarrow^{+}\mathsf{e}$:

$$\frac{\Gamma \vdash \mathtt{pl\_eval+}(p_{\mathtt{safe}}, e_{\mathsf{sq}}, p_{\mathsf{gl}}) \mathbin{@} t}{\Gamma \vdash \mathtt{pl\_eval}(p_{\mathtt{safe}}, e_{\mathsf{sq}}, p_{\mathsf{gl}}) \mathbin{@} t}\ \mathtt{pl\_eval+}\mathsf{e}$$

6.4.4 Evaluation

$\boxed{\mathtt{pl\_eval}(p_{\mathtt{safe}}, e_{\mathsf{sq}}, p_{\mathsf{gl}})}$

The proposition $\mathtt{pl\_eval}(p_{\mathtt{safe}}, e_{\mathsf{sq}}, p_{\mathsf{gl}})$ encodes the standard evaluation judgment $e_{\mathsf{sq}} \overset{p_{\mathtt{safe}}}{\rightsquigarrow} p_{\mathsf{gl}}$ as follows:

$$\mathtt{pl\_eval}(p_{\mathtt{safe}}, e_{\mathsf{sq}}, p_{\mathsf{gl}}) \overset{\mathsf{def}}{\equiv} \mathsf{sq} = e_{\mathsf{sq}} \supset [\mathsf{sq}/x_{\mathsf{sq}}]\, (p_{\mathtt{safe}}\, \mathcal{U}\, p_{\mathsf{gl}})$$

This encoding resembles $\mathtt{pl\_eval}$, except that it can be satisfied when the goal proposition holds immediately.

$\mathtt{pl\_eval}$ can be introduced by the derived rule $\mathtt{pl\_eval}\mathsf{i}_0$, which corresponds to $\rightsquigarrow\mathsf{i}_0$:

$$\frac{\Gamma \vdash [e_{\mathsf{sq}}/x_{\mathsf{sq}}]\, p_{\mathsf{gl}} \mathbin{@} t \quad \Gamma \vdash [a_{\mathsf{sq}}/x_{\mathsf{sq}}]\, p_{\mathsf{gl}} : \mathsf{lo}\ (a_{\mathsf{sq}})}{\Gamma \vdash \mathtt{pl\_eval}(p_{\mathtt{safe}}, e_{\mathsf{sq}}, p_{\mathsf{gl}}) \mathbin{@} t}\ \mathtt{pl\_eval}\mathsf{i}_0^{a_{\mathsf{sq}}}$$

The additional premise ensures that p_{gl} is an invariance property.

The loop rule $\mathtt{pl\_eval}\mathsf{loop}$ corresponds to the program-logic rule $\rightsquigarrow\mathsf{loop}$:

$$\frac{\begin{array}{l}\Gamma \vdash [a_{\mathsf{sq}}/x_{\mathsf{sq}}]\, p'_{\mathsf{i}} : \mathsf{lo}\ (a_{\mathsf{sq}}) \quad \Gamma \vdash \texttt{pl\_eval}(p_{\texttt{safe}}, e_{\mathsf{sq}}, p'_{\mathsf{i}} \vee p_{\mathsf{gl}}) \mathbin{@} t \\ \Gamma,\ [a_{\mathsf{sq}}/x_{\mathsf{sq}}]\, p'_{\mathsf{i}} \mathbin{@} b',\ a_{\mathsf{sq}} : \mathsf{ri} \vdash \texttt{pl\_eval+}(p_{\texttt{safe}}, a_{\mathsf{sq}}, p'_{\mathsf{i}} \vee p_{\mathsf{gl}}) \mathbin{@} b'\end{array}}{\Gamma \vdash \texttt{pl\_eval}(p_{\texttt{safe}}, e_{\mathsf{sq}}, p_{\mathsf{gl}}) \mathbin{@} t}\ \texttt{pl\_evalloop}^{a_{\mathsf{sq}}, b'}$$

$$\text{where } p'_{\mathsf{i}} \stackrel{\mathsf{def}}{\equiv} [e_{\mathsf{sq}}/x_{\mathsf{sq}_0}]\, p_{\mathsf{i}}$$

The additional premise ensures that the loop invariant is an invariance property. The body of the loop is evaluated for a time parameter b' because the head of the loop may be encountered at an arbitrary future time.

6.4.5 Procedures

$\boxed{\texttt{pl\_proc}(p_{\texttt{safe}}, p_{\mathsf{p}}, p_{\mathsf{q}})}$

The proposition $\texttt{pl\_proc}(p_{\texttt{safe}}, p_{\mathsf{p}}, p_{\mathsf{q}})$ encodes the procedure judgment $p_{\mathsf{p}} \overset{p_{\texttt{safe}}}{\rightsquigarrow}{}^{\star} p_{\mathsf{q}}$ as follows:

$$\texttt{pl\_proc}(p_{\texttt{safe}}, p_{\mathsf{p}}, p_{\mathsf{q}}) \stackrel{\mathsf{def}}{\equiv} \Box(\forall x_{\mathsf{sq}_0} : \mathsf{ri}.\ \mathsf{sq} = x_{\mathsf{sq}_0} \supset [\mathsf{sq}/x_{\mathsf{sq}}]\, (p_{\mathsf{p}} \supset p_{\texttt{safe}}\, \mathcal{U}\, p_{\mathsf{q}}))$$

It asserts that whenever the precondition p_{p} holds, the invariance property $p_{\texttt{safe}}$ will hold for the machine state until the postcondition p_{q} holds for the machine state. The rigid variable x_{sq_0} can be used in the postcondition to refer to the machine state at the time that the precondition held. For example, the postcondition can specify that the program counter after a procedure call is the value that was on the top of the stack when the procedure was called.

`pl_proc` is introduced by the derived rule $\texttt{pl\_proc}\mathsf{i}$, which corresponds directly to $\rightsquigarrow^{\star}\mathsf{i}$:

$$\frac{\begin{array}{l}\Gamma \vdash [a'_{\mathsf{sq}}/x_{\mathsf{sq}}]\, p'_{\mathsf{p}} : \mathsf{lo}\ (a'_{\mathsf{sq}}) \\ \Gamma,\ [a_{\mathsf{sq}}/x_{\mathsf{sq}}]\, p'_{\mathsf{p}} \mathbin{@} b',\ a_{\mathsf{sq}} : \mathsf{ri} \vdash \texttt{pl\_eval}(p_{\texttt{safe}}, a_{\mathsf{sq}}, p'_{\mathsf{q}}) \mathbin{@} b'\end{array}}{\Gamma \vdash \texttt{pl\_proc}(p_{\texttt{safe}}, p_{\mathsf{p}}, p_{\mathsf{q}}) \mathbin{@} 0}\ \texttt{pl\_proc}\mathsf{i}^{a_{\mathsf{sq}}, a'_{\mathsf{sq}}, b'}$$

$$\text{where } p'_{\mathsf{p}} \stackrel{\mathsf{def}}{\equiv} [a_{\mathsf{sq}}/x_{\mathsf{sq}_0}]\, p_{\mathsf{p}}$$
$$\text{and } p'_{\mathsf{q}} \stackrel{\mathsf{def}}{\equiv} [a_{\mathsf{sq}}/x_{\mathsf{sq}_0}]\, p_{\mathsf{q}}$$

`pl_proc` is eliminated by the derived rule $\texttt{pl\_proc}\mathsf{e}$, which corresponds to $\rightsquigarrow^{\star}\mathsf{e}$:

$$\frac{\begin{array}{l}\Gamma \vdash \texttt{pl\_proc}(p_{\texttt{safe}}, p_{\mathsf{p}}, p_{\mathsf{q}}) \mathbin{@} 0 \quad \Gamma \vdash [e_{\mathsf{sq}}/x_{\mathsf{sq}}]\, p'_{\mathsf{p}} \mathbin{@} t \\ \Gamma \vdash [a_{\mathsf{sq}}/x_{\mathsf{sq}}]\, p'_{\mathsf{p}} : \mathsf{lo}\ (a_{\mathsf{sq}}) \quad \Gamma \vdash [a_{\mathsf{sq}}/x_{\mathsf{sq}}]\, p'_{\mathsf{q}} : \mathsf{lo}\ (a_{\mathsf{sq}}) \\ \Gamma \vdash \texttt{ri}(e_{\mathsf{sq}}) \mathbin{@} 0 \\ \Gamma,\ [a_{\mathsf{sq}}/x_{\mathsf{sq}}]\, p'_{\mathsf{q}} \mathbin{@} b',\ a_{\mathsf{sq}} : \mathsf{ri} \vdash \texttt{pl\_eval}(p_{\texttt{safe}}, a_{\mathsf{sq}}, p_{\mathsf{gl}}) \mathbin{@} b'\end{array}}{\Gamma \vdash \texttt{pl\_eval}(p_{\texttt{safe}}, e_{\mathsf{sq}}, p_{\mathsf{gl}}) \mathbin{@} t}\ \texttt{pl\_proc}\mathsf{e}^{a_{\mathsf{sq}}, b'}$$

$$\text{where } p'_{\mathsf{p}} \stackrel{\mathsf{def}}{\equiv} [e_{\mathsf{sq}}/x_{\mathsf{sq}_0}]\, p_{\mathsf{p}}$$
$$\text{and } p'_{\mathsf{q}} \stackrel{\mathsf{def}}{\equiv} [e_{\mathsf{sq}}/x_{\mathsf{sq}_0}]\, p_{\mathsf{q}}$$

The additional premises ensure that the precondition and postcondition are invariance properties, and that the value of the machine-state representation at the time of the procedure call is consistent with its value at the time of the procedure return.

The recursion rule $\mathtt{pl\_procrec}$ corresponds directly to the program-logic rule $\leadsto^{\star}\mathsf{rec}$:

$$\frac{\Gamma,\, p_{\mathsf{proc}} \mathbin{@} 0,\; a_{\mathsf{n}} : \mathsf{ri} \vdash \mathtt{pl\_proc}(p_{\mathtt{safe}}, [a_{\mathsf{n}}/x_{\mathsf{n}}]\, p_{\mathsf{p}}, p_{\mathsf{q}}) \mathbin{@} 0}{\Gamma \vdash \mathtt{pl\_proc}(p_{\mathtt{safe}}, [e_{\mathsf{n}}/x_{\mathsf{n}}]\, p_{\mathsf{p}}, p_{\mathsf{q}}) \mathbin{@} 0}\ \mathtt{pl\_procrec}^{a_{\mathsf{n}}}$$

$$\text{where } p_{\mathsf{proc}} \stackrel{\mathsf{def}}{\equiv} \forall x_{\mathsf{n}} : \mathsf{fl}.\ \mathtt{ltuw}(x_{\mathsf{n}}, a_{\mathsf{n}}) \supset \mathtt{pl\_proc}(p_{\mathtt{safe}}, p_{\mathsf{p}}, p_{\mathsf{q}})$$

6.4.6 Derivability

Each of the inference rules in this section is derivable.

Let

$$\Gamma \vdash^{\mathcal{J}}_{\mathsf{sp}} J$$

assert that J is derivable from Γ using only inference rules from Chapter 3, Chapter 4, and Chapter 5. Let

$$\Gamma \vdash^{\mathcal{J}}_{\mathsf{pl}} J$$

assert that J is derivable from Γ using only inference rules from Chapter 3, Chapter 4, Chapter 5, and Section 6.4.

Proposition 6.4.1 (Derivability) $\Gamma \vdash^{\mathcal{J}}_{\mathsf{sp}} J$ *if* $\Gamma \vdash^{\mathcal{J}}_{\mathsf{pl}} J$

PROOF: refer to the LF implementation □

6.5 Supporting the Logic of Programs

In this section, I introduce derived "support" rules for the program logic that involve only local, nontemporal reasoning. These rules are not part of the program logic *per se*, because, traditionally, such inferences are lumped together under the rubric of "mathematical reasoning." The derived rules shown here correspond to elements of the high-level structure of a residual proof—other, purely internal, support rules are not shown in the interest of brevity.

I classify the rules into three subsections. "Specification elements" are the basis for verifying that a given procedure or loop invariant specification holds. "Preservation elements" are the basis for verifying that a given machine register, memory location, or group of memory locations is preserved unchanged through a given procedure or loop body. Finally, "residual proofs" are tailored to constructing residual proofs of Java type-safety properties.

6.5.1 Specification Elements

These rules essentially walk down a list of specification elements and attach a residual proof to each one. The list is represented as a right-associative conjunction of specification elements $p_1, \ldots, p_k$. The specification elements are normally typing assertions and nonzero assertions.

The rules in this section prove propositions in the following form:

$$p_1 \wedge (\ldots \wedge (p_k \wedge \top))$$

It asserts that one can infer a given composite specification that is instantiated with a rigid representation of the current machine state. See Section 5.6 for sample specifications.

$\mathsf{checku\_z}$ is the base case:

$$\frac{}{\Gamma \vdash \top \mathbin{@} t}\ \mathsf{checku\_z}$$

The remaining rule in this section is an iteration step.

$\mathsf{checku\_u}$ is the iteration step that takes a residual proof as its first premise and conjoins it with a proof of the rest of the specifications (this is where the residual proof is "attached"):

$$\frac{\Gamma \vdash p_{\mathsf{u}} \mathbin{@} t \quad \Gamma \vdash p'_{\mathsf{u}} \mathbin{@} t}{\Gamma \vdash p_{\mathsf{u}} \wedge p'_{\mathsf{u}} \mathbin{@} t}\ \mathsf{checku\_u}$$

The second premise is the "recursive call" that establishes the conjunction for the remaining specification elements.

6.5.2 Preservation Elements

These rules essentially walk down a list of preservation elements and prove each one via a proof of equality. The equalities are normally established by rewriting before the preservation elements are checked, so the logic program that is derived from these rules is fully automatic (*i.e.*, no residual proof is needed). Again, the list is represented as a right-associative conjunction of specification elements $p_1, \ldots, p_k$. The specification elements are normally equalities.

The rules in this section prove propositions in the following form:

$$p_1 \wedge (\ldots \wedge (p_k \wedge \top))$$

It asserts that one can infer a given preservation element instantiated with a rigid representation of the current state.

$\mathsf{checkz\_none}$ is the base case:

$$\frac{}{\Gamma \vdash \top \mathbin{@} t}\ \mathsf{checkz\_none}$$

The remaining two rules in this section are iteration steps.

$\mathsf{checkz\_eq}$ ensures that a given pair of values are equal:

$$\frac{\Gamma \vdash p'_{\mathsf{z}} \mathbin{@} t}{\Gamma \vdash e = e \wedge p'_{\mathsf{z}} \mathbin{@} t}\ \mathsf{checkz\_eq}$$

e is normally the saved value of the register at the head of a loop or the beginning of a procedure. The premise is the "recursive call" that establishes the specification for the remaining specification elements.

checkz_leq_ts ensures that a given type environment is subsumed by the current type environment:

$$\frac{\Gamma \vdash \texttt{js\_leq}(e_{\texttt{jts}_1}, e_{\texttt{jts}_2}) \mathbin{@} t \quad \Gamma \vdash p'_{\mathsf{z}} \mathbin{@} t}{\Gamma \vdash \texttt{js\_leq}(e_{\texttt{jts}_1}, e_{\texttt{jts}_2}) \wedge p'_{\mathsf{z}} \mathbin{@} t}\ \mathsf{checkz\_leq\_ts}$$

The first premise is established by a logic program for transitive closure. The second premise is the "recursive call" that establishes the specification for the remaining specification elements.

6.5.3 Residual Proofs

The derived rules in this section are "utility" rules for inferences that arise frequently in residual proofs. Many of these rules are based on equivalent *trusted* rules in the signature used by the SpecialJ compiler [CLN+00].

arrElem is a slight generalization of $\texttt{ja\_ptri}_{\mathsf{elem}}$ (see Section 5.8):

$$\frac{\begin{array}{l}\Gamma \vdash \texttt{js\_of}(e_{\texttt{jts}}, e_{\mathsf{na}}, f_{\texttt{j\_array}}(e_{\texttt{jty}})) \mathbin{@} t \quad \Gamma \vdash \texttt{js\_mem}(e_{\texttt{jts}}, e_{\texttt{m}}) \mathbin{@} t \\ \Gamma \vdash e_{\mathsf{na}} \neq 0 \mathbin{@} t \quad \Gamma \vdash \texttt{j\_size}(e_{\texttt{jty}}) = e_{\mathsf{ns}} \mathbin{@} b \quad \Gamma \vdash \texttt{j\_len}(e_{\mathsf{na}}) = e'_{\mathsf{np}} \mathbin{@} b \\ \Gamma \vdash \texttt{selw}(e_{\texttt{m}}, e'_{\mathsf{np}}) = e'_{\mathsf{n}} \mathbin{@} b \quad \Gamma \vdash \texttt{ltuw}(e_{\mathsf{ni}}, e'_{\mathsf{n}}) \mathbin{@} t\end{array}}{\Gamma \vdash \texttt{js\_ptr}(e_{\texttt{jts}}, \texttt{addw}(e_{\mathsf{na}}, \texttt{addw}(\texttt{mulw}(e_{\mathsf{ni}}, e_{\mathsf{ns}}), 20)), e_{\texttt{jty}}, \overline{\texttt{rw}}) \mathbin{@} t}\ \mathsf{arrElem}^b$$

The additional premises enable the bounds check to be put into normal form $\texttt{ltuw}(e_{\mathsf{ni}}, e'_{\mathsf{n}})$ by rewriting. This proposition matches the conclusion of the incoming residual proof.

The rules in Figure 6.4 are used in SpecialJ safety proofs to extract properties of numbers from the condition code register, based on standard IA32 idioms. For example, in nullcandne, the state of the condition codes becomes

$$\texttt{updf2}(\overline{\texttt{op2\_and}}, e_{\texttt{f}}, e, e)$$

after executing the "test" instruction. If the zero flag is cleared in this state, then one can infer that the tested value is nonzero. Similarly, executing the "compare" instruction yields state

$$\texttt{updf2}(\overline{\texttt{op2\_sub}}, e_{\texttt{f}}, e_1, e_2)$$

where e_1 and e_2 are the values being compared.

Finally, ptrCong substitutes one pointer for another based on a proof of equality:

$$\frac{\Gamma \vdash \texttt{js\_ptr}(e_{\texttt{jts}}, e_{\mathsf{np}}, e_{\texttt{jty}}, e_{\mathsf{acc}}) \mathbin{@} t \quad \Gamma \vdash e_{\mathsf{np}} = e'_{\mathsf{np}} \mathbin{@} t}{\Gamma \vdash \texttt{js\_ptr}(e_{\texttt{jts}}, e'_{\mathsf{np}}, e_{\texttt{jty}}, e_{\mathsf{acc}}) \mathbin{@} t}\ \mathsf{ptrCong}$$

Perhaps surprisingly, this is the only rule needed by the SpecialJ compiler for equality-based substitutions.

$$\frac{\Gamma \vdash \mathtt{self}(\overline{\mathtt{cop\_nz}}, \mathtt{updf2}(\overline{\mathtt{op2\_and}}, e_{\mathtt{f}}, e, e)) \neq 0 \,@\, t}{\Gamma \vdash e \neq 0 \,@\, t} \ \textsf{nullcandne}$$

$$\frac{\Gamma \vdash \mathtt{self}(\overline{\mathtt{cop\_nz}}, \mathtt{updf2}(\overline{\mathtt{op2\_sub}}, e_{\mathtt{f}}, e, 0)) \neq 0 \,@\, t}{\Gamma \vdash e \neq 0 \,@\, t} \ \textsf{nullcsubne}$$

$$\frac{\Gamma \vdash \mathtt{self}(\overline{\mathtt{cop\_nz}}, \mathtt{updf2}(\overline{\mathtt{op2\_and}}, e_{\mathtt{f}}, 0, 0)) \neq 0 \,@\, t}{\Gamma \vdash \bot \,@\, t} \ \textsf{candneqz}$$

$$\frac{\Gamma \vdash \mathtt{self}(\overline{\mathtt{cop\_c}}, \mathtt{updf2}(\overline{\mathtt{op2\_sub}}, e_{\mathtt{f}}, e_1, e_2)) \neq 0 \,@\, t}{\Gamma \vdash \mathtt{ltuw}(e_1, e_2) \,@\, t} \ \textsf{ultcsubb}$$

$$\frac{\Gamma \vdash \mathtt{self}(\overline{\mathtt{cop\_a}}, \mathtt{updf2}(\overline{\mathtt{op2\_sub}}, e_{\mathtt{f}}, e_2, e_1)) \neq 0 \,@\, t}{\Gamma \vdash \mathtt{ltuw}(e_1, e_2) \,@\, t} \ \textsf{ultcsubnbe}$$

$$\frac{\Gamma \vdash \mathtt{self}(\overline{\mathtt{cop\_a}}, \mathtt{updf2}(\overline{\mathtt{op2\_and}}, e_{\mathtt{f}}, e, e)) \neq 0 \,@\, t}{\Gamma \vdash \mathtt{ltuw}(0, e) \,@\, t} \ \textsf{ultzero}$$

$$\frac{\Gamma \vdash \mathtt{self}(\overline{\mathtt{cop\_nl}}, \mathtt{updf2}(\overline{\mathtt{op2\_and}}, e_{\mathtt{f}}, e, e)) \neq 0 \,@\, t}{\Gamma \vdash \mathtt{geqw}(e, 0) \,@\, t} \ \textsf{gecandnlt}$$

$$\frac{\Gamma \vdash \mathtt{self}(\overline{\mathtt{cop\_nl}}, \mathtt{updf2}(\overline{\mathtt{op2\_sub}}, e_{\mathtt{f}}, e, 0)) \neq 0 \,@\, t}{\Gamma \vdash \mathtt{geqw}(e, 0) \,@\, t} \ \textsf{gecsubnlt}$$

Figure 6.4: Condition-Code Rules

6.5.4 Derivability

Each of the inference rules in this section is derivable.

Let

$$\Gamma \vdash_{\mathsf{ck}}^{\mathcal{J}} J$$

assert that J is derivable from Γ using only inference rules from Chapter 3, Chapter 4, Chapter 5, and Section 6.5.

Proposition 6.5.1 (Derivability) $\Gamma \vdash_{\mathsf{sp}}^{\mathcal{J}} J$ *if* $\Gamma \vdash_{\mathsf{ck}}^{\mathcal{J}} J$

PROOF: refer to the LF implementation □

6.6 Proof Outlines for the Logic of Programs

In this section, I specify a *proof outline* representation for the logic of programs. A proof outline is a data structure that contains just enough information to efficiently construct a proof of safety in the logic of programs. Essentially, a proof outline shows one how to put together derived rules to produce a proof of safety. Instantiations of meta-variables are only included if they are not determined by other rules, or by the conclusion of the inference tree—I assume that the target safety property is known in advance.

The proof outline representation enables a reasonably compact proof representation, as I will demonstrate in Chapter 9. Note that based on my experience [BL02a], most explicit proof representations (including binary LF and LFi [NL98b]) are not dense enough to yield competitive results when the proof obligation is a formal safety property and not an intermediate result like a VC. Rather, some form of implicit proof representation is needed such as in Necula and Rahul [NR01]. Keep in mind that there are only a few bits available for each instruction if the proof representation is to be smaller than the code itself. The IA32 architecture is particularly challenging in this respect, because some common instructions are only one byte long.

Proof outlines play a key role in proof representation and proof reconstruction, which is the subject of Chapter 8. Essentially, the proof outline becomes a search constraint for a deterministic logic program. In Chapter 7, I show how a proof outline is automatically constructed from a VC proof emitted by the SpecialJ compiler. However, I present the proof outline specification here because there is a structural correspondence with the derived program logic that I have presented in this chapter, and because I want it to serve as a common point of departure for the remainder of the dissertation.

Note that in this chapter, I have generally only presented derived rules that have corresponding proof outlines. There are many other derived rules, but they are only used internally by the proof reconstruction logic program that is the subject of Chapter 8.

$$
\begin{array}{lll}
\text{Callee Outl.} & oc ::= & \mathtt{callee}\langle oe\rangle \\[4pt]
\text{Eval. Outl.} & oe ::= & \mathtt{eval\_head}\langle e^{\mathtt{wd}}_{\mathsf{pc}}\rangle\langle p_{\mathsf{iuz}}\rangle\langle e^{\mathtt{wd}}_{\mathsf{nh}}\rangle\langle oe_1\rangle\langle oe_2\rangle \mid \mathtt{eval\_tail}\langle ock\rangle \\
& & \mid \mathtt{eval\_step}\langle oe_1\rangle \\
& & \mid \mathtt{eval\_unsafe}\langle ovc\rangle\langle oe_1\rangle \\
& & \mid \mathtt{eval\_unsafe\_mem}\langle ovc\rangle\langle e^{\mathtt{wd}}_{\mathsf{nm}}\rangle\langle oe_1\rangle \\
& & \mid \mathtt{eval\_split}\langle oe_1\rangle\langle oe_2\rangle \\
& & \mid \mathtt{eval\_call\_and}\langle e^{\mathtt{wd}}_{\mathsf{nh}}\rangle\langle ock\rangle\langle oe\rangle \mid \mathtt{eval\_call\_false}\langle ock\rangle \\[4pt]
\text{Checking Outl.} & ock ::= & \mathtt{ck\_z} \mid \mathtt{ck\_u}\langle ovc\rangle\langle ock_1\rangle \\[4pt]
\text{VC Outlines} & ovc ::= & \mathtt{vc\_hyp\_cont}\langle e^{\mathtt{wd}}_{\mathsf{nh}}\rangle \mid \mathtt{vc\_hyp\_mem}\langle e^{\mathtt{wd}}_{\mathsf{nm}}\rangle \\
& & \mid \mathtt{vc\_hyp\_pre} \mid \mathtt{vc\_hyp\_cop} \mid \mathtt{vc\_hyp\_lt\_ts} \\
& & \mid \mathtt{vc\_and\_el}\langle ovc_1\rangle \mid \mathtt{vc\_and\_er}\langle ovc_1\rangle \mid \mathtt{vc\_normE\_e} \\
& & \mid \mathtt{vc\_auto0} \mid \mathtt{vc\_auto1}\langle ovc_1\rangle \\
& & \mid \mathtt{vc\_auto2}\langle ovc_1\rangle\langle ovc_2\rangle \mid \mathtt{vc\_auto3}\langle ovc_1\rangle\langle ovc_2\rangle\langle ovc_3\rangle \\
& & \mid \mathtt{vc\_arrLen}\langle ovc_1\rangle\langle ovc_2\rangle \\
& & \mid \mathtt{vc\_arrElem}\langle ovc_1\rangle\langle ovc_2\rangle\langle ovc_3\rangle\langle ovc_4\rangle\langle ovc_5\rangle \\
& & \mid \mathtt{vc\_instFld}\langle ovc_1\rangle\langle ovc_2\rangle\langle ovc_3\rangle \\
& & \mid \mathtt{vc\_nullcandne}\langle ovc_1\rangle \mid \mathtt{vc\_nullcsubne}\langle ovc_1\rangle \\
& & \mid \mathtt{vc\_candneqz}\langle ovc_1\rangle \\
& & \mid \mathtt{vc\_ultcsubb}\langle ovc_1\rangle \mid \mathtt{vc\_ultcsubnbe}\langle ovc_1\rangle \\
& & \mid \mathtt{vc\_ultzero}\langle ovc_1\rangle \\
& & \mid \mathtt{vc\_gecandnlt}\langle ovc_1\rangle \mid \mathtt{vc\_gecsubnlt}\langle ovc_1\rangle \\
& & \mid \mathtt{vc\_ptrCong}\langle ovc_1\rangle\langle ovc_2\rangle
\end{array}
$$

Figure 6.5: Abstract Syntax for Proof Outlines

6.6.1 Syntax

The syntax of a proof outline is defined in Figure 6.5. Essentially, each proof-outline constructor specifies that some mixture of program-logic rules from Section 6.3, derived rules from Section 6.5, and standard inference rules should be applied.

A "callee" outline identifies a proof reconstruction strategy for a procedure based on the point-of-view of the callee. Currently, there is only one strategy (symbolic evaluation), based on an "evaluation" outline. Thus, the `callee` case specifies that the program-logic procedure rules $\rightsquigarrow^{\star}\mathsf{i}$ and $\rightsquigarrow^{\star}\mathsf{rec}$ should be applied (it is assumed that all procedures are recursive).

An "evaluation" outline specifies a sequence of (possibly strict) evaluation rules (from Section 6.3) that verify the safety of a procedure body. The `eval_head` case specifies that the loop rule $\rightsquigarrow\mathsf{loop}$ should be applied. e_{pc} is the address of the head of the loop, p_{iuz} is the loop invariant, and e_{nh} is an index that precisely identifies the (hypothetical) loop invariant when it is used in a VC proof.[6] oe_1 is an evaluation outline that brings the current state to the head of the loop and establishes the initial loop invariant, whereas oe_2 is an evaluation outline for the body of the loop. Note that a loop "head" here need not be the first instruction in the loop–in practice, a loop invariant can be assigned to any address in the loop body.

The `eval_tail` case specifies that the $\rightsquigarrow\mathsf{i}_0$ rule should be applied—the goal property (a loop invariant or postcondition) has been reached. *ock* is a "checking" outline that can establish the goal property.

The `eval_step`, `eval_unsafe`, and `eval_unsafe_mem` cases each simulate a single instruction by applying $\rightsquigarrow^{+}\mathsf{i}_1$ in conjunction with $\rightarrow \mathsf{i}$. $\rightsquigarrow^{+}\mathsf{e}$ is also applied if non-strict evaluation is needed. The cases discriminate based on whether the safety of the current instruction is evident (`eval_step`), and thus provable by rewriting, or whether it requires an explicit proof of safety (`eval_unsafe` and `eval_unsafe_mem`). *ovc* is an outline of the explicit safety proof. `eval_unsafe_mem` is needed when the memory is changed by the instruction—it provides an index that precisely identifies the new memory state.[7]

The `eval_split` case simulates a branch instruction by applying $\rightsquigarrow^{+}\mathsf{i}_2$ in conjunction with $\rightarrow \mathsf{i}$. $\rightsquigarrow^{+}\mathsf{e}$ is also applied if non-strict evaluation is needed. Safety is always evident for branch instructions on the IA32—another architecture might also need an unsafe version of `eval_split`.

`eval_call_and` and `eval_call_false` simulate procedure calls by applying the program-logic rule $\rightsquigarrow^{\star}\mathsf{e}$. `eval_call_false` is used only for procedures such as exception generators that never return, and thus have a postcondition of "false." e_{nh} is an index that precisely identifies the (hypothetical) postcondition when it is used in a VC proof. *ock* is a checking outline that can establish the precondition for the current state, whereas *oe* is the evaluation outline for the code left to be executed after the call (*i.e.*, the "continuation"). Note that I assume that procedure

[6] Each active hypothesis is assigned a unique index during proof generation (see Section 7.2.1).

[7] This is an optimization provided by SpecialJ [CLN^{+}00]: when the memory is changed, validity (see Section 5) for the new memory value is proven immediately and this fact is noted under the given index. Later, a proof can simply refer to the validity index to justify that the specific memory state is valid. The logic program for proof reconstruction (see Chapter 8) simulates this mechanism.

specifications are stored separately and that the precondition and postcondition can be recovered by indexing on the current program counter (see Chapter 8 for the mechanism used during proof reconstruction).

A "checking" outline applies one of two supporting derived rules from Section 6.5.1. The `ck_z` case applies check_z, then applies preservation rules from Section 6.5.2 according to the current goal. Because the subgoals can be satisfied by rewriting, `ck_z` requires no internal information. The `ck_u` case applies check_u. *ovc* is an outline of a VC proof that can establish the third premise of this rule. ock_1 is an outline of a proof that can establish the remaining components of the current specification.

Note that there are no "preservation" outlines for the rules in Section 6.5.2, because the premises of these rules can be established by rewriting, and thus do not need outlines: the goal contains sufficient information to reconstruct a proof.

A "VC" outline specifies a tree of inference rules (any of hypothetical, derived, and built-in) that establish the truth of some residual proof. The hypothetical outlines `vc_hyp_cont`, `vc_hyp_mem`, `vc_hyp_pre`, and `vc_hyp_cop` identify judgments in the current context that are sufficient to establish a given VC. Judgments are inserted into the context by the hypothetical premises of the derived rules in Section 6.4. For example, the procedure call rule inserts the postcondition of the procedure into the context when verifying the "continuation" of the call.

`vc_hyp_cont` identifies a continuation hypothesis that is either the postcondition of a procedure (after it is called) or a loop invariant (within the body of the loop). e_{nh} is the index assigned to the hypothesis during proof construction (see Chapter 7). `vc_hyp_mem` identifies a memory-validity hypothesis that is established whenever memory is changed. e_{nm} is an index assigned by the SpecialJ compiler. `vc_hyp_pre` identifies a precondition hypothesis. This hypothesis is the precondition of the procedure whose body is currently being certified. `vc_hyp_cop` identifies a conditional hypothesis that is established after a branch is taken. It asserts that the condition corresponding to the current arm of the branch is true. Note that the rest of the proof contains sufficient contextual information to identify which hypothesis is needed.

`vc_and_el` and `vc_and_er` apply the left and right elimination rules for the $\wedge$ connective, respectively. `vc_normE_e` applies the elimination rule for expression rewriting (*i.e.*, an expression is rewritten to establish equality with its normal form).

`vc_auto0`, `vc_auto1`, `vc_auto2`, `vc_auto3` are place holders for cases where only one derived rule is applicable (*i.e.*, the current open premise directly determines which rule should be applied). In each of these cases, the arity of the proof-outline constructor corresponds to the number of premises in the derived rule.

`vc_arrLen` and `vc_instFld` apply Java typing rules from Section 5.8, whereas `vc_arrElem` applies the array element rule arrElem from Section 6.5.3. Likewise, `vc_ptrCong` applies the pointer congruence rule ptrCong from Section 6.5.3. Finally, the remaining cases apply the various condition-code rules from Section 6.5.3. Note that proof outlines are needed for Java typing derivations, because the machine-level type system is not naturally syntax-directed. This is unlike other approaches to machine-code certification such as TAL [MWCG98] that rely on decidable type systems.

6.6.2 Demonstration

The proof outline for the demonstration derivation of Section 6.3.5 can be constructed as follows:

$$\texttt{eval\_head}\langle 16\rangle\langle p_{\mathsf{u}_{16}} \wedge p_{\mathsf{z}_{16}}\rangle\langle 1\rangle\langle\texttt{eval\_tail}\langle\texttt{ck\_u}\langle\ldots\rangle\langle\texttt{ck\_z}\rangle\rangle\rangle\langle oe_{16}\rangle$$

$$\begin{aligned}
\text{where}\quad oe_{16} &\stackrel{\mathsf{def}}{=} \texttt{eval_step}\langle oe_{19}\rangle\\
oe_{19} &\stackrel{\mathsf{def}}{=} \texttt{eval_step}\langle oe_{20}\rangle\\
oe_{20} &\stackrel{\mathsf{def}}{=} \texttt{eval_step}\langle oe_{24}\rangle\\
oe_{24} &\stackrel{\mathsf{def}}{=} \texttt{eval_split}\langle oe'_{16}\rangle\langle oe_{26}\rangle\\
oe'_{16} &\stackrel{\mathsf{def}}{=} \texttt{eval_tail}\langle\texttt{ck_u}\langle\ldots\rangle\langle\texttt{ck_z}\rangle\rangle\\
oe_{26} &\stackrel{\mathsf{def}}{=} \texttt{eval_tail}\langle\texttt{ck_u}\langle\ldots\rangle\langle\texttt{ck_z}\rangle\rangle
\end{aligned}$$

I elide the proof outlines for residual proofs. I also assume that there is a hypothesis from an enclosing derivation that provides validity for the initial memory.

Chapter 7

Proof Construction

In this chapter, I present an algorithm for constructing a proof outline (see Section 6.6) for a program that has been compiled by the SpecialJ compiler [CLN$^+$00]. The proof outline representation enables a reasonably compact proof representation, which is difficult to obtain for formal safety properties using a explicit proof representation such as LFi [NL98b].

SpecialJ is a PCC compiler for the Java programming language [GJS96] that produces certified Intel IA-32 object code. A SpecialJ certificate is a first-order proof of a verification condition (VC) that implies that the compiled program respects a low-level encoding of the Java type system. The proofs of Java type safety are not "discovered" by the compiler. In principle, all the information needed to construct a type safety proof is already present is the Java source code. Thus, the compiler need only translate typing derivations from the source code to the corresponding object code. However, in practice, it is more expedient to only translate loop invariants and procedure specifications in the compiler itself—the safety proofs are then reconstructed from the typing specifications by a first-order theorem prover.

Note that typing information for Java type safety is inserted into the code by the programmer, and is thus already present in the source code when compilation starts. It is thus incorrect to view SpecialJ as a verification tool in the usual sense, because no new properties of the code are being synthesized. The use of a first-order theorem prover to generate type safety proofs is an essentially an engineering detail—the output of SpecialJ would be indistinguishable from its current incarnation if safety proofs were instead propagated through each stage of the compiler.

I choose to use the SpecialJ compiler because it is a relatively mature compiler that already produces certified code. Although it is possible to construct safety proofs for machine-language programs "by hand" in my framework, this approach is feasible only for small programs and/or simple safety properties. Thus, SpecialJ provides me with a basis for experimentation with larger programs that would be impractical to certify manually.

Unfortunately, the certificates produced by SpecialJ are incompatible with my logical formal system because the certificates are first-order proofs of VCs rather than being proofs of temporal-logic safety properties. Thus, I need to translate the output of SpecialJ to use these certificates with my PCC infrastructure.

7.1 Overview

Although it is probably feasible to construct a fully-explicit proof (*e.g.*, as in Bernard and Lee [BL02a]) and then extract a proof outline from the explicit proof, this approach introduces an extra level of inefficiency that I wish to avoid. Instead, the approach that I present in this chapter bypasses explicit proofs entirely and generates a proof outline directly from the SpecialJ certificate. Note that although this chapter is aimed at constructing a proof outline that demonstrates Java type safety, I expect that the approach will generalize to any invariance property that has suitable loop invariants and a valid residual proof, such as those produced by a certifying compiler.

In order to construct a complete proof outline, a tree of program-logic inference rules is constructed in which the VC proof provided by SpecialJ is used to satisfy premises that correspond to residual proofs (*i.e.*, these premises cannot be satisfied by other program-logic rules—see Figure 6.2). The VC proof is actually deconstructed by my implementation as the tree of program-logic rules is simultaneously constructed by a deterministic algorithm. Thus, the program-logic rules comprise a "proof skeleton" to which the residual VC proofs are attached. Constructing the "algorithmic" part of the proof relies on a simulation of the SpecialJ VC generator/symbolic evaluator (see Section 6.2). While it is relatively straightforward to implement the proof construction algorithm as a logic program, an implementation of the symbolic evaluator frequently executes arithmetic and array-lookup operations that are not well-suited to a logic interpreter.

To address this problem, I split the proof generation task into two phases: first, a *trace* of the symbolic evaluator is generated using a functional program that embeds the result of each arithmetic and array operation into an inductive data structure that can be easily interpreted by a logic program. In the second phase, the trace and VC proof is interpreted by a logic program that synthesizes a complete proof outline. Thus, the program-logic proof is instantiated with the VC proof, as in Figure 6.2. This two-phase approach also simplifies the presentation. The symbolic evaluator is most naturally presented in an algorithmic style, whereas the proof-outline constructor is most naturally presented as a deductive system. The choice of implementation language (functional program or logic program) is essentially a detail.

Note that this multi-phase strategy for proof-outline construction is not simply an artifact of my decision to adapt the SpecialJ compiler. It is useful for any proof-construction strategy to be able to systematically disentangle proofs that can be constructed *algorithmically* (*i.e.*, the skeleton of program-logic rules) from proofs that are *discovered* by search (*i.e.*, the VC proof), derived from source-language properties, or even constructed by hand. Note also that this strategy is also applicable even when no complete VC proof is "in hand" initially. A search for each particular fragment of the VC proof could be initiated incrementally as the proof skeleton is constructed.

Only a high-level approximation of the symbolic evaluator is needed to construct a valid proof outline. Many details that are essential to constructing a fully-explicit proof (*e.g.*, instantiations of existential variables) are unimportant when the proof is reduced to an outline, at least when the goal is Java type safety.[1] Thus, in the interest of brevity, I do not give a detailed account of the SpecialJ symbolic evaluator. Instead, I present a high-level simulation that only executes operations that are relevant to a proof-outline.

In a previous work [BL02a], we were able to demonstrate relative completeness, in the sense that any VC proof necessarily has a corresponding fully-explicit proof. Unfortunately, the high-level treatment of the symbolic evaluator and the proof-outline data structure given in this chapter makes it impossible to prove a similar result for this system. However, I conjecture that it would be possible to prove such a result given a complete formalization of the symbolic evaluator and fully-explicit proof construction algorithm.

The purpose of this chapter is to provide a detailed account of my proof-generation strategy for invariance properties. Because I have dramatically increased the number of security policies that the code consumer can enforce, it is natural to question whether anything has been lost in terms of which security policies the code producer can automatically certify in the more expressive framework. This chapter, along with the experimental results of Chapter 7, provide evidence that it is possible to adapt a current certifying compiler to the more general infrastructure, and thus that nothing is lost in terms of which safety properties can be automatically certified.

This chapter is organized as follows: in Section 7.2, I show how the SpecialJ symbolic evaluator is simulated in order to provide a "blueprint" for the tree of program-logic rules that comprise the algorithmic part of the safety proof. In Section 7.3, I show how the blueprint is combined with a SpecialJ VC proof and translated into a deductive system that corresponds to the target proof-outline representation. Finally, in Section 7.4, I show how proof outlines are extracted from the derivations of the intermediate deductive system.

7.2 Tracing the Symbolic Evaluator

In this section, I present a high-level simulation of the operation of the SpecialJ symbolic evaluator. For my proof-generation algorithm, it is not necessary to simulate SpecialJ at a detailed level, but for a more comprehensive presentation, see Necula [Nec98] and Colby, *et al.* [CLN+00].

In Section 7.2.1, I explain the syntax for the trace encoding, whereas in Section 7.2.2, I present the algorithm for generating traces.

[1] It is conceivable that more advanced security policies will require a correspondingly more faithful simulation of the symbolic evaluator.

7.2.1 Syntax

A *symbolic-evaluator trace* θ is a data structure that enumerates the actions that the symbolic evaluator takes at a high level:

$$\begin{array}{lll} \text{Traces} & \theta ::= & \texttt{head}\langle p\rangle\langle e^{\texttt{wd}}\rangle\langle\theta_1\rangle \mid \texttt{head\_dom}\langle p\rangle\langle e^{\texttt{wd}}\rangle\langle\theta_1\rangle\langle\theta_2\rangle \mid \texttt{tail}\langle e^{\texttt{wd}}\rangle \\ & & \mid \texttt{step}\langle e^{\texttt{inst}}\rangle\langle\theta_1\rangle \mid \texttt{unsafe}\langle e^{\texttt{inst}}\rangle\langle\theta_1\rangle \mid \texttt{unsafe\_mem}\langle e^{\texttt{inst}}\rangle\langle e^{\texttt{wd}}\rangle\langle\theta_1\rangle \\ & & \mid \texttt{split}\langle e^{\texttt{inst}}\rangle\langle\theta_1\rangle\langle\theta_2\rangle \mid \texttt{call}\langle p_1\rangle\langle p_2\rangle\langle e^{\texttt{wd}}\rangle\langle\theta_1\rangle \end{array}$$

Each node of the trace corresponds to the execution of a single instruction or to some other high-level operation such as instantiating a loop invariant.

As in Chapter 6, each loop invariant or procedure specification contains the free variables $x_{\texttt{sq}_0}$ and $x_{\texttt{sq}}$. $x_{\texttt{sq}_0}$ is instantiated to the symbolic machine state in effect at the start of the loop or procedure call. $x_{\texttt{sq}}$ is instantiated to the symbolic machine state in effect when the specification is used. See Section 6.3 for more details.

Each hypothetical loop invariant and procedure postcondition is assigned a unique index when a trace is generated. The hypothetical index enables a free assumption to be efficiently identified in a safety proof. Without an index, unification for specifications of free assumptions is a potentially costly part of proof reconstruction (see Chapter 8), because unification may not immediately instantiate a complete proposition. Hypothetical specifications are numbered sequentially (instead of by address, for example) to yield compact proof encodings.

A loop invariant is hypothetical in the body of the loop, whereas a procedure postcondition is hypothetical in the code following a call to the procedure (*i.e.*, the "continuation"). I call these specifications *hypothetical* because a proof over the loop body or continuation presumes that the specification to be true for the purposes of proving safety—the hypothesis is then discharged by a higher-level rule of the program logic. For example, a safety proof for a loop body *presumes* that the loop invariant is true, then shows that the invariance property holds until the loop invariant becomes true again. See Section 6.3 for a more detailed discussion of hypothetical specifications.

Each abstract state of memory is also assigned an index so that proofs of validity can be shared. Whenever the memory is changed, validity (see Section 5) for the new memory state is proven immediately and this fact is noted (hypothetically) under the assigned index. Later, a proof in the scope of the hypothesis can simply refer to the validity index to justify that the specific memory state is valid. The hypothesis is discharged by a derived rule at the instruction that changes the memory (see Chapter 8).

In Table 7.1, I give an intuitive reading of each constructor of the trace data type.

7.2.2 Construction

In this section, I formalize the algorithm for generating symbolic-evaluator traces. Essentially, the algorithm is a high-level simulation of the SpecialJ symbolic evaluator. I do not formalize the symbolic evaluator completely—interested readers should

$\mathtt{head}\langle p_\mathsf{i}\rangle\langle e_\mathsf{nh}\rangle\langle\theta\rangle$	Start a new loop at the current instruction, after instantiating a given loop invariant p_i; the hypothetical invariant will be labeled e_nh inside the loop body; θ traces execution in the body of the loop. Note that the loop "head" need not be the first instruction in the loop–in practice, a compiler can choose any address in the loop body to assign to the invariant.
$\mathtt{head\_dom}\langle p_\mathsf{i}\rangle\langle e_\mathsf{nh}\rangle\langle\theta_1\rangle\langle\theta_2\rangle$	The current instruction dominates a loop: start this new loop, after instantiating a given loop invariant p_i; share an explicit proof among all dominated instructions under hypothetical label e_nh; θ_1 traces execution up to the head of the loop, whereas θ_2 traces execution in the body of the loop.
$\mathtt{tail}\langle e_\mathsf{pc}\rangle$	Complete a pending loop (*i.e.*, the current instruction is the head of a loop with an active hypothesis), or return from the current procedure, where e_pc is the current program counter.
$\mathtt{step}\langle e_\mathsf{l}\rangle\langle\theta\rangle$	Execute the current instruction e_l; safety is established automatically by rewriting; θ traces execution for the remaining instructions.
$\mathtt{unsafe}\langle e_\mathsf{l}\rangle\langle\theta\rangle$	Execute the current instruction e_l; safety must be established by an explicit proof; θ traces execution for the remaining instructions.
$\mathtt{unsafe\_mem}\langle e_\mathsf{l}\rangle\langle e_\mathsf{nm}\rangle\langle\theta\rangle$	Execute the current instruction e_l, updating the memory to an abstract state with label e_nm; safety is established by an explicit proof; θ traces execution for the remaining instructions.
$\mathtt{split}\langle e_\mathsf{l}\rangle\langle\theta_1\rangle\langle\theta_2\rangle$	Execute the current conditional branch instruction e_l and perform a case split; safety is established by rewriting; θ_1 and θ_2 trace execution for the remaining instructions in each possible branch.
$\mathtt{call}\langle p_\mathsf{p}\rangle\langle p_\mathsf{q}\rangle\langle e_\mathsf{nh}\rangle\langle\theta\rangle$	Call a procedure using a given precondition p_p and postcondition p_q; the hypothetical postcondition will be labeled e_nh in the continuation; θ traces execution in the continuation.

Table 7.1: Trace Constructors

refer to Bernard and Lee [BL02b] for a thorough treatment of a simpler algorithm that does not encompass procedures.

Each algorithm function operates on some program Φ to be certified. Additionally, I assume that three functions are available to identify procedure specifications and loop invariants. These functions can be constructed from annotations attached to the code by the SpecialJ compiler. ψ_{proc} is a partial function mapping machine words to pairs of propositions consisting of a precondition and a postcondition. Only addresses that are procedure entry points are in the domain of this function. ψ_{inv} is a partial function mapping machine words to propositions. Only addresses that have loop invariants assigned are in the domain of this function, and the result of applying the function is the corresponding loop invariant itself. ψ_{dom} is a partial function mapping machine words to machine words. This function identifies loop dominator instructions at which VC proofs are shared. The result of applying this function is the address of the corresponding loop head.[2]

Each trace-construction function is specialized to a tuple $(\Phi, \psi_{\mathsf{proc}}, \psi_{\mathsf{inv}}, \psi_{\mathsf{dom}})$, shown as a subscript in its definition. However, because these arguments are effectively constant during trace construction, they are not instantiated explicitly where functions are used—the reader should assume that the corresponding arguments are inherited from the enclosing definition.

The result of the general trace construction algorithm is a triple, consisting of a trace, a next unused hypothetical index, and a next unused memory index. Only the first result is interesting at the top level, but the other results are used at intermediate levels. In order to express the composition of trace-construction functions succinctly, I use the following abbreviation in this section:

$$\theta \cdot \langle \theta', n_{\mathsf{h}}, n_{\mathsf{m}} \rangle \stackrel{\mathsf{def}}{=} \langle [\theta'/\circ]\, \theta, n_{\mathsf{h}}, n_{\mathsf{m}} \rangle$$

This operation substitutes the trace θ' for the "hole" $\circ$ in θ. The hole is simply a special token used to support this composition notation—the definition of substitution is absolutely straightforward, because there is no possibility of scope or binding.

Additionally, it is often necessary to construct an expression for a given machine instruction. I use the notation e_I for the expression whose value is always I:

$$\mathcal{V}_{\phi}^{\mathcal{J}}(e_I) = I \text{ for all } \phi$$

Such an expression, comprised purely of constants and applications of constant functions, exists for any instruction value.

$Eval(n_{\mathsf{pc}}, N_{\mathsf{i}}, n_{\mathsf{h}}, n_{\mathsf{m}})$ is the top-level trace-construction function. It returns a trace, hypothetical index, and abstract memory index for a symbolic evaluation run starting at address n_{pc} of the program Φ. N_{i} is a set of machine words that identify loop invariants that have already been encountered. n_{h} and n_{m} are the current hypothetical and abstract memory indices.

[2]Loop dominator annotations are used by SpecialJ to share VC proofs after a join point in the control-flow graph.

Note that *Eval* and some of the other trace-construction functions are not defined on some arguments. For example, if the program jumps outside the domain of Φ, evaluation cannot proceed and the implementation will terminate with an error message. Such cases are identified by a result value of $\perp$ in the function definitions and they trigger exceptions in the implementation. Additionally, *Eval* will fail to terminate if some loop does not have at least one invariant in its body. Note, however, that SpecialJ will never generate code that is outside the domain of *Eval*, so its partiality is only a practical consideration when it is used with other compilers or handwritten code.

I now define the trace construction functions in "bottom-up" order.

$Step(EA_{\mathsf{rd}}, EA_{\mathsf{wr}}, I, n_{\mathsf{pc}}, N_{\mathsf{i}}, n_{\mathsf{h}}, n_{\mathsf{m}})$ constructs a trace starting from an "ordinary" instruction I. n_{pc} is the address of the next instruction *after* I. Ordinary instructions do not change the control flow in interesting ways, and additionally do not require any reasoning in terms of abstract specifications. Such an instruction uses one of the three "step" constructors, depending on the degree to which the instruction accesses memory. EA_{rd} is the set of effective addresses that are read by I, whereas EA_{wr} is the set of effective addresses that are written by I. N_{i}, n_{h}, and n_{m} are as for *Eval*.

Step is defined as follows:

$$
\begin{aligned}
&Step_{\Phi,\psi_{\mathsf{proc}},\psi_{\mathsf{inv}},\psi_{\mathsf{dom}}}(EA_{\mathsf{rd}}, EA_{\mathsf{wr}}, I, n_{\mathsf{pc}}, N_{\mathsf{i}}, n_{\mathsf{h}}, n_{\mathsf{m}}) \\
&= \begin{cases}
\texttt{unsafe\_mem}\langle e_I\rangle\langle\overline{n_{\mathsf{m}}}\rangle\langle\circ\rangle \cdot Eval(n_{\mathsf{pc}}, N_{\mathsf{i}}, n_{\mathsf{h}}, n_{\mathsf{m}} \dot{+} 1) & \text{if } \mathsf{ea\_m}\langle\ldots\rangle \in EA_{\mathsf{wr}} \\
\texttt{unsafe}\langle e_I\rangle\langle\circ\rangle \cdot Eval(n_{\mathsf{pc}}, N_{\mathsf{i}}, n_{\mathsf{h}}, n_{\mathsf{m}}) & \text{if } \mathsf{ea\_m}\langle\ldots\rangle \in EA_{\mathsf{rd}} \setminus EA_{\mathsf{wr}} \\
\texttt{step}\langle e_I\rangle\langle\circ\rangle \cdot Eval(n_{\mathsf{pc}}, N_{\mathsf{i}}, n_{\mathsf{h}}, n_{\mathsf{m}}) & \text{otherwise}
\end{cases}
\end{aligned}
$$

It essentially chooses a trace constructor according to whether I writes to memory, reads from memory, or does not access memory. Additionally, if I writes to memory, the current abstract memory index is incremented. The corresponding function in an actual symbolic evaluator would simply update the symbolic state representation and emit a formal proof obligation for instructions that access memory.

$Cont(p_{\mathsf{q}}, n_{\mathsf{pc}}, N_{\mathsf{i}}, n_{\mathsf{h}}, n_{\mathsf{m}})$ constructs a trace for the continuation of a procedure call starting at address n_{pc}. p_{q} is the postcondition of the procedure that was called, and N_{i}, n_{h}, and n_{m} are as for *Eval*.

Cont is defined as follows:

$$
\begin{aligned}
&Cont_{\Phi,\psi_{\mathsf{proc}},\psi_{\mathsf{inv}},\psi_{\mathsf{dom}}}(p_{\mathsf{q}}, n_{\mathsf{pc}}, N_{\mathsf{i}}, n_{\mathsf{h}}, n_{\mathsf{m}}) \\
&= \begin{cases}
Ret(n_{\mathsf{h}}, n_{\mathsf{m}}) & \text{if } p_{\mathsf{q}} \text{ is } \mathsf{pc} = e'_{\mathsf{pc}} \wedge \perp \\
Eval(n_{\mathsf{pc}}, N_{\mathsf{i}}, n_{\mathsf{h}} \dot{+} 1, n_{\mathsf{m}}) & \text{otherwise}
\end{cases}
\end{aligned}
$$

It evaluates the continuation normally after incrementing the hypothetical index, unless the postcondition is $\perp$. When a postcondition is $\perp$, it implies that the procedure never returns normally (*e.g.*, it uses some mechanism from the run-time system that triggers a non-local exit). In such cases, the postcondition of the procedure containing the call will be implied trivially by $\perp$, so it skips directly to the "return" case. An exception generator is a typical example of a procedure with a $\perp$ postcondition.

$Call(n_{\mathsf{pc}}, n'_{\mathsf{pc}}, N_{\mathsf{i}}, n_{\mathsf{h}}, n_{\mathsf{m}})$ constructs a trace for a procedure call. n_{pc} is the address of the procedure to call, and n'_{pc} is the address of the continuation in the current procedure. N_{i}, n_{h}, and n_{m} are as for *Eval*.

Call is defined as follows:

$$
\begin{array}{l}
Call_{\Phi,\psi_{\mathsf{proc}},\psi_{\mathsf{inv}},\psi_{\mathsf{dom}}}(n_{\mathsf{pc}}, n'_{\mathsf{pc}}, N_{\mathsf{i}}, n_{\mathsf{h}}, n_{\mathsf{m}}) \\
= \begin{cases}
\mathtt{call}\langle p_{\mathsf{p}}\rangle\langle p_{\mathsf{q}}\rangle\langle\overline{n_{\mathsf{h}}}\rangle\langle\circ\rangle \cdot Cont(p_{\mathsf{q}}, n'_{\mathsf{pc}}, N_{\mathsf{i}}, n_{\mathsf{h}}, n_{\mathsf{m}}) & \text{if} \quad n_{\mathsf{pc}} \in \operatorname{dom}\psi_{\mathsf{proc}} \\
 & \quad \text{and } \psi_{\mathsf{proc}}(n_{\mathsf{pc}}) = \langle p_{\mathsf{p}}, p_{\mathsf{q}}\rangle \\
\perp & \text{otherwise}
\end{cases}
\end{array}
$$

It simply applies the "call" trace constructor to the precondition and postcondition of the procedure, and resumes evaluation in the continuation. The current hypothetical index n_{h} (identifying the postcondition) is also saved for use during proof construction.

$Ret(n_{\mathsf{h}}, n_{\mathsf{m}})$ constructs a trace that returns from the current procedure:

$$Ret_{\Phi,\psi_{\mathsf{proc}},\psi_{\mathsf{inv}},\psi_{\mathsf{dom}}}(n_{\mathsf{h}}, n_{\mathsf{m}}) = \langle \mathtt{tail}\langle x_{\mathsf{pc}}\rangle, n_{\mathsf{h}}, n_{\mathsf{m}}\rangle$$

It returns the current hypothetical and abstract memory indices n_{h} and n_{m} unchanged. *Ret* is used when the current instruction is the first instruction of the continuation in the "caller" (*i.e.*, the instruction to be executed immediately after `ret` is executed by the "callee"). The variable x_{pc} is a "placeholder" for the return address. This variable will be instantiated with a parameter during proof construction (see Section 7.3).

$Next(n_{\mathsf{pc}}, N_{\mathsf{i}}, n_{\mathsf{h}}, n_{\mathsf{m}})$ constructs a trace starting at address n_{pc}, but without considering any loop invariants. n_{pc} is the address of the next instruction to execute, and N_{i}, n_{h}, and n_{m} are as for *Eval*.

Next is defined in Table 7.2. It applies *Step* to continue ordinary evaluation for most instructions, providing it with the appropriate set of effective addresses and the address of the next instruction to execute. In the case of a conditional branch, it first constructs a trace for each possible arm of the branch (considering the "true" branch first), then combines the resulting traces under the `split` constructor. In the case of a procedure call, it first takes a step to reach the first instruction of the procedure, then applies *Call* to complete the call itself. Similarly, in the case of a procedure return, it first takes a step to reach the nominal continuation of the calling procedure (the instruction address is a parameter in this case), then applies *Ret* to complete the return itself. If a jump or call is performed on a non-immediate address, or if n_{pc} is not in the domain of Φ, then the result of *Next* is undefined.

Finally, the top-level trace construction function *Eval* is defined in Figure 7.1. It is identical to *Next*, except that explicit consideration is given to loop invariants. In the case of an already-seen invariant, evaluation stops at the current instruction address. In the case of a new regular loop invariant, a trace is constructed for the loop body from the current instruction (*Next* will ignore the current invariant) after incrementing the current hypothetical index and adding the current address to the "already-seen" set. The current loop invariant, hypothetical index, and loop-body trace are stored under the `head` constructor. The loop dominator case is similar,

$\Phi(n_{\mathsf{pc}})$	$\mathit{Next}_{\Phi,\psi_{\mathsf{proc}},\psi_{\mathsf{inv}},\psi_{\mathsf{dom}}}(n_{\mathsf{pc}}, N_{\mathsf{i}}, n_{\mathsf{h}}, n_{\mathsf{m}})$
$\texttt{mov}\langle n_{\mathsf{i}}\rangle\langle ea_1\rangle\langle ea_2\rangle$	$\mathit{Step}(\{ea_1\}, \{ea_2\}, \Phi(n_{\mathsf{pc}}), n_{\mathsf{pc}} \dotplus n_{\mathsf{i}}, N_{\mathsf{i}}, n_{\mathsf{h}}, n_{\mathsf{m}})$
$\texttt{xchg}\langle n_{\mathsf{i}}\rangle\langle ea\rangle\langle r\rangle$	$\mathit{Step}(\emptyset, \{ea\}, \Phi(n_{\mathsf{pc}}), n_{\mathsf{pc}} \dotplus n_{\mathsf{i}}, N_{\mathsf{i}}, n_{\mathsf{h}}, n_{\mathsf{m}})$
$\texttt{lea}\langle n_{\mathsf{i}}\rangle\langle ea\rangle\langle r\rangle$	$\mathit{Step}(\emptyset, \emptyset, \Phi(n_{\mathsf{pc}}), n_{\mathsf{pc}} \dotplus n_{\mathsf{i}}, N_{\mathsf{i}}, n_{\mathsf{h}}, n_{\mathsf{m}})$
$\texttt{push}\langle n_{\mathsf{i}}\rangle\langle ea\rangle$	$\mathit{Step}(\{ea\}, \emptyset, \Phi(n_{\mathsf{pc}}), n_{\mathsf{pc}} \dotplus n_{\mathsf{i}}, N_{\mathsf{i}}, n_{\mathsf{h}}, n_{\mathsf{m}})$
$\texttt{pop}\langle n_{\mathsf{i}}\rangle\langle ea\rangle$	$\mathit{Step}(\emptyset, \{ea\}, \Phi(n_{\mathsf{pc}}), n_{\mathsf{pc}} \dotplus n_{\mathsf{i}}, N_{\mathsf{i}}, n_{\mathsf{h}}, n_{\mathsf{m}})$
$\texttt{op1}\langle n_{\mathsf{i}}\rangle\langle \mathit{op1}\rangle\langle ea\rangle$	$\mathit{Step}(\emptyset, \{ea\}, \Phi(n_{\mathsf{pc}}), n_{\mathsf{pc}} \dotplus n_{\mathsf{i}}, N_{\mathsf{i}}, n_{\mathsf{h}}, n_{\mathsf{m}})$
$\texttt{op2}\langle n_{\mathsf{i}}\rangle\langle \mathit{op2}\rangle\langle ea_1\rangle\langle ea_2\rangle$	$\mathit{Step}(\{ea_1\}, \{ea_2\}, \Phi(n_{\mathsf{pc}}), n_{\mathsf{pc}} \dotplus n_{\mathsf{i}}, N_{\mathsf{i}}, n_{\mathsf{h}}, n_{\mathsf{m}})$
$\texttt{op2n}\langle n_{\mathsf{i}}\rangle\langle \mathit{op2}\rangle\langle ea_1\rangle\langle ea_2\rangle$	$\mathit{Step}(\{ea_1, ea_2\}, \emptyset, \Phi(n_{\mathsf{pc}}), n_{\mathsf{pc}} \dotplus n_{\mathsf{i}}, N_{\mathsf{i}}, n_{\mathsf{h}}, n_{\mathsf{m}})$
$\texttt{op3}\langle n_{\mathsf{i}}\rangle\langle \mathit{op3}_1\rangle\langle \mathit{op3}_2\rangle\langle ea\rangle\langle r_1\rangle\langle r_2\rangle$	$\mathit{Step}(\{ea\}, \emptyset, \Phi(n_{\mathsf{pc}}), n_{\mathsf{pc}} \dotplus n_{\mathsf{i}}, N_{\mathsf{i}}, n_{\mathsf{h}}, n_{\mathsf{m}})$
$\texttt{jmp}\langle n_{\mathsf{i}}\rangle\langle \texttt{ea\_i}\langle n'_{\mathsf{pc}}\rangle\rangle$	$\mathit{Step}(\emptyset, \emptyset, \Phi(n_{\mathsf{pc}}), n'_{\mathsf{pc}}, N_{\mathsf{i}}, n_{\mathsf{h}}, n_{\mathsf{m}})$
$\texttt{j}\langle n_{\mathsf{i}}\rangle\langle \mathit{cop}\rangle\langle n\rangle$	$\langle \texttt{split}\langle e_{\Phi(n_{\mathsf{pc}})}\rangle\langle\theta'\rangle\langle\theta''\rangle, n''_{\mathsf{h}}, n''_{\mathsf{m}}\rangle$ where $\langle\theta', n'_{\mathsf{h}}, n'_{\mathsf{m}}\rangle = \mathit{Eval}(n'_{\mathsf{pc}}, N_{\mathsf{i}}, n_{\mathsf{h}}, n_{\mathsf{m}})$ and $\langle\theta'', n''_{\mathsf{h}}, n''_{\mathsf{m}}\rangle = \mathit{Eval}(n''_{\mathsf{pc}}, N_{\mathsf{i}}, n'_{\mathsf{h}}, n'_{\mathsf{m}})$ and $n''_{\mathsf{pc}} = n_{\mathsf{pc}} \dotplus n_{\mathsf{i}}$ and $n'_{\mathsf{pc}} = n''_{\mathsf{pc}} \dotplus n$
$\texttt{call}\langle n_{\mathsf{i}}\rangle\langle \texttt{ea\_i}\langle n'_{\mathsf{pc}}\rangle\rangle$	$\texttt{step}\langle e_{\Phi(n_{\mathsf{pc}})}\rangle\langle\circ\rangle \cdot \mathit{Call}(n'_{\mathsf{pc}}, n_{\mathsf{pc}} \dotplus n_{\mathsf{i}}, N_{\mathsf{i}}, n_{\mathsf{h}}, n_{\mathsf{m}})$
$\texttt{ret}\langle n_{\mathsf{i}}\rangle$	$\texttt{step}\langle e_{\Phi(n_{\mathsf{pc}})}\rangle\langle\circ\rangle \cdot \mathit{Ret}(n_{\mathsf{h}}, n_{\mathsf{m}})$
otherwise	$\perp$

Table 7.2: Trace Next Instruction

$$
\begin{aligned}
&\mathit{Eval}_{\Phi,\psi_{\mathsf{proc}},\psi_{\mathsf{inv}},\psi_{\mathsf{dom}}}(n_{\mathsf{pc}}, N_{\mathsf{i}}, n_{\mathsf{h}}, n_{\mathsf{m}}) \\
&= \begin{cases}
\langle \texttt{tail}\langle \overline{n_{\mathsf{pc}}}\rangle, n_{\mathsf{h}}, n_{\mathsf{m}}\rangle & \text{if } n_{\mathsf{pc}} \in \operatorname{dom}\psi_{\mathsf{inv}} \cap N_{\mathsf{i}} \\
\langle \texttt{head}\langle \psi_{\mathsf{inv}}(n_{\mathsf{pc}})\rangle\langle\overline{n_{\mathsf{h}}}\rangle\langle\theta'\rangle, n'_{\mathsf{h}}, n'_{\mathsf{m}}\rangle & \text{if } n_{\mathsf{pc}} \in \operatorname{dom}\psi_{\mathsf{inv}} \setminus N_{\mathsf{i}} \\
\quad \text{where } \langle\theta', n'_{\mathsf{h}}, n'_{\mathsf{m}}\rangle = \mathit{Next}(n_{\mathsf{pc}}, N'_{\mathsf{i}}, n_{\mathsf{h}} \dotplus 1, n_{\mathsf{m}}) & \\
\quad \text{and } N'_{\mathsf{i}} = N_{\mathsf{i}} \cup \{n_{\mathsf{pc}}\} & \\
\langle \texttt{head\_dom}\langle \psi_{\mathsf{inv}}(n'_{\mathsf{pc}})\rangle\langle\overline{n_{\mathsf{h}}}\rangle\langle\theta'\rangle\langle\theta''\rangle, n''_{\mathsf{h}}, n''_{\mathsf{m}}\rangle & \text{if } n_{\mathsf{pc}} \in \operatorname{dom}\psi_{\mathsf{dom}} \\
\quad \text{where } \langle\theta', n'_{\mathsf{h}}, n'_{\mathsf{m}}\rangle = \mathit{Next}(n_{\mathsf{pc}}, N'_{\mathsf{i}}, n_{\mathsf{h}} \dotplus 1, n_{\mathsf{m}}) & \qquad \setminus \operatorname{dom}\psi_{\mathsf{inv}} \\
\quad \text{and } \langle\theta'', n''_{\mathsf{h}}, n''_{\mathsf{m}}\rangle = \mathit{Next}(n'_{\mathsf{pc}}, N'_{\mathsf{i}}, n'_{\mathsf{h}}, n'_{\mathsf{m}}) & \quad \text{and } n'_{\mathsf{pc}} \in \operatorname{dom}\psi_{\mathsf{inv}} \\
\quad \text{and } N'_{\mathsf{i}} = N_{\mathsf{i}} \cup \{n'_{\mathsf{pc}}\} & \\
\quad \text{and } n'_{\mathsf{pc}} = \psi_{\mathsf{dom}}(n_{\mathsf{pc}}) & \\
\mathit{Next}(n_{\mathsf{pc}}, N_{\mathsf{i}}, n_{\mathsf{h}}, n_{\mathsf{m}}) & \text{otherwise}
\end{cases}
\end{aligned}
$$

Figure 7.1: Trace Construction

except that a separate trace is constructed to get from the current instruction to the loop head. The two traces are combined under the `head_dom` constructor.

7.2.3 Demonstration

Let Φ_{16} be the factorial loop from Section 6.3.5, and let

$$\begin{array}{ll} \operatorname{dom}\psi_{\mathsf{proc}_{16}} = \emptyset & \\ \operatorname{dom}\psi_{\mathsf{inv}_{16}} = \{16\} & \qquad \psi_{\mathsf{inv}}(16) = p_{\mathsf{i}_{16}} \\ \operatorname{dom}\psi_{\mathsf{dom}_{16}} = \emptyset & \end{array}$$

The trace for Φ_{16} is the part of the result of $Eval_{\Phi_{16},\psi_{\mathsf{proc}_{16}},\psi_{\mathsf{inv}_{16}},\psi_{\mathsf{dom}_{16}}}(16,\emptyset,1,1)$ which, in turn, is

$$\langle \mathtt{head}\langle p_{\mathsf{i}_{16}}\rangle\langle 1\rangle\langle\theta_{16}\rangle, 2, 1\rangle$$

$$\begin{array}{lll} \text{where} & \theta_{16} \stackrel{\mathsf{def}}{=} \mathtt{step}\langle\Phi_{16}(16)\rangle\langle\theta_{19}\rangle \\ & \theta_{19} \stackrel{\mathsf{def}}{=} \mathtt{step}\langle\Phi_{16}(19)\rangle\langle\theta_{20}\rangle \\ & \theta_{20} \stackrel{\mathsf{def}}{=} \mathtt{step}\langle\Phi_{16}(20)\rangle\langle\theta_{24}\rangle \\ & \theta_{24} \stackrel{\mathsf{def}}{=} \mathtt{split}\langle\Phi_{16}(24)\rangle\langle\theta'_{16}\rangle\langle\theta_{26}\rangle \\ & \theta'_{16} \stackrel{\mathsf{def}}{=} \mathtt{tail}\langle 16\rangle \\ & \theta_{26} \stackrel{\mathsf{def}}{=} \mathtt{step}\langle\Phi_{16}(26)\rangle\langle\mathtt{tail}\langle x_{\mathtt{pc}}\rangle\rangle \end{array}$$

Note the close correspondence with the safety proof outline for the same program in Section 6.6.2. The two trees differ only in that the proof outline stops at address 26 before executing the return instruction, whereas the trace stops immediately after the procedure returns.

7.3 Constructing Proof-Outline Derivations

The obtain a complete proof outline, an outline of the VC proof must be merged with an outline of the program-logic proof. In this section, I present an algorithm that constructs an outline of the program-logic proof while it simultaneously merges in the important fragments of the VC proof.

The algorithm is not presented as a program that manipulates proofs. Rather, I present a deductive system that has a natural implementation as a logic program. The logic program does not itself manipulate proofs. Instead, a *trace* of a single run of the logic program corresponds directly to the correct proof outline. After obtaining a trace of the logic program, it is a simple transformation to extract a proof outline from the derivation (see Section 7.4). A trace of a run of the logic program is called a *proof-outline derivation*: a derivation in the deductive system that I present in the remainder of this section.

In my experience, implementing a proof-construction algorithm in this fashion is significantly more efficient than an equivalent logic program that explicitly manipulates proofs, as is advocated in Appel and Felty [AF03], and first explored by

Felty [Fel93]. The performance advantage seems to stem from the fact that unification steps can be avoided entirely for the result derivation. In a logic program that manipulates proofs, the result derivation is part of the search goal, and thus must be unified frequently with existential variables. Note that my observations are based on my experience with the Twelf [PS99] logic interpreter, which has a relatively simple compilation phase [Cer98]. It is possible that more advanced logic-program compilation phase would negate the observed performance advantage by eliminating unnecessary unification steps.

The top-level connective of a VC proof shows that some specification (*e.g.*, a postcondition) follows from another specification (*e.g.*, a precondition) by means of the implication introduction rule $\supset \mathsf{i}$. The derived program-logic rules provide an explicit (though typically hypothetical) derivation of the premise of the VC proof, so this derivation can be substituted for the hypothetical derivation that is used internally by the $\supset \mathsf{i}$ rule. This example illustrates the primary operation of the process that "merges" a VC proof into a program-logic proof skeleton. The precise structure of the VC proof is defined by the SpecialJ compiler, and this structure is anticipated by the proof-outline rules.

Note that there are several details of the logic program implementation that are not exposed by the presentation as a deductive system. First, the VC proofs are mapped onto outlines themselves to avoid expensive unification steps. This is safe to do, because precise information is only needed where it will affect the overall structure of the resulting proof outline. In the case of VC proofs for Java type safety, very little detailed information is relevant. This optimization provides a tremendous performance improvement, although it may not be applicable to cases where more detailed information is important.

Additionally, the order of specification elements is permuted by the implementation to obtain a more efficient ordering for proof reconstruction (see Chapter 8). Putting frequently-used elements at the head of a long conjunction simplifies unification, and thus results in improved proof-reconstruction times. These permutation steps are not shown here in the interest of simplifying the presentation.

7.3.1 Syntax

The syntax of the language for proof-outline generation is defined in Figure 7.2. A *proof classifier* Θ denotes an informal safety property for some fragment of code (based on a trace θ) or some informal local property. The proof classifier is ascribed to zero or more VC proofs that are expected to formally establish the property during proof reconstruction. The proof classifier implicitly uses program-logic rules to establish the safety property. It is not incorrect to think of the proof classifier as a *tactical* [GMW79] for constructing a proof outline, except that the state of the tactical is itself threaded through the judgments established by the derivation, rather than being a distinct program that manipulates derivations.

Safety properties are only established informally in this chapter, because at this point I am only interested in constructing an *outline* of a safety proof, as opposed to a fully explicit proof in the sense of Bernard and Lee [BL02a]. Because only the

$$
\begin{array}{lll}
\text{Proof Classifiers} & \Theta ::= & \mathtt{callee}\langle p\rangle\langle\theta\rangle \\
& & \mid \mathtt{eval}\langle\theta\rangle \\
& & \mid \mathtt{check}\langle p\rangle \mid \mathtt{loop}\langle e_1^{\mathtt{wd}}\rangle\langle p\rangle\langle e_2^{\mathtt{wd}}\rangle\langle\theta\rangle \mid \mathtt{cont}\langle p\rangle\langle e^{\mathtt{wd}}\rangle\langle\theta\rangle \\
& & \mid \mathtt{dis\_uz}\langle p\rangle \mid \mathtt{dis}\langle p\rangle \\
& & \mid \mathtt{vc} \\
& & \mid \mathtt{goal}\langle e^{\mathtt{wd}}\rangle\langle p\rangle \\
& & \mid \mathtt{hyp\_cont}\langle e^{\mathtt{wd}}\rangle \mid \mathtt{hyp\_pre} \mid \mathtt{hyp\_cop} \\
\text{Search Goals} & \omega ::= & \mathcal{D}_1,\, \ldots,\, \mathcal{D}_k :: \Theta \mid \langle\!\langle J\rangle\!\rangle_O \\
\text{Search Contexts} & \Omega ::= & \cdot \mid \Omega_1,\, (\Pi y_1, \ldots, y_k.\omega_1, \ldots, \omega_{k'} \to \omega')
\end{array}
$$

Figure 7.2: Abstract Syntax for Proof Generation

outline of the proof is stored in the certificate, there is nothing to be gained by generating fully-explicit proofs at this stage. Although it is feasible to construct a fully-explicit proof from a proof outline by tracing the logic program of Chapter 8, I do not explore this approach further because fully-explicit proofs are simply too costly to be of practical use in my infrastructure.

A *search goal* ω is the target of either proof-outline generation (the subject of this chapter), or proof reconstruction (the subject of Chapter 8). Search goals are most naturally implemented as logic-program goals, although other implementations are possible. A *search context* Ω contains a set of goals that are presumed to succeed.

The general search affirmation is

$$\Omega \rhd \omega$$

which asserts that the goal ω succeeds, under the assumption that all goals in Ω succeed. Thus, the affirmation

$$\Omega \rhd \mathcal{D}_1, \ldots, \mathcal{D}_k :: \Theta$$

asserts that the derivations $\mathcal{D}_1, \ldots, \mathcal{D}_k$ are expected to establish the property denoted by the classifier Θ, assuming that all goals in Ω succeed. The search goal $\langle\!\langle J\rangle\!\rangle_O$ is described in the next chapter.

A hypothesis

$$\Pi y_1, \ldots, y_k.\omega_1, \ldots, \omega_{k'} \to \omega'$$

is a *conditional search hypothesis*: a hypothesis that is expected to establish ω' for some instantiation of time variables y_1 through y_k, given an explicit derivation of ω_1 through $\omega_{k'}$ with the same instantiation. A time variable y is like an expression variable x, except that a time variable can only appear directly in a judgment with an explicit time (*i.e.*, $y_1 \leq y_2$, $p \,@\, y$, $p \,@[y_1, y_2)$, and $p \,\underline{@}\, y)$, and times t are substituted for time variables rather than expressions. I abbreviate such a hypothesis as

$$\omega_1, \ldots, \omega_{k'} \to \omega'$$

$\mathcal{D} :: \texttt{callee}\langle p\rangle\langle\theta\rangle$	The derivation $\mathcal{D}$ is expected to establish the safety of a procedure with specification p, where θ is a trace of the procedure body.
$\mathcal{D} :: \texttt{eval}\langle\theta\rangle$	The derivation $\mathcal{D}$ is expected to establish safety over the trace θ.
$\mathcal{D}, \mathcal{D}' :: \texttt{check}\langle p_{\mathsf{uz}}\rangle$	Some parts of the derivation $\mathcal{D}$ are expected to establish the proposition p_{uz}—the derivation $\mathcal{D}'$ contains the "unused" parts of $\mathcal{D}$.
$\mathcal{D} :: \texttt{loop}\langle e_{\mathsf{pc}}\rangle\langle p_{\mathsf{iuz}}\rangle\langle e_{\mathsf{nh}}\rangle\langle\theta\rangle$	The derivation $\mathcal{D}$ is expected to establish safety over a loop with trace θ, using e_{pc} as the address of the loop invariant p_{iuz}.
$\mathcal{D} :: \texttt{cont}\langle p_{\mathsf{uz}}\rangle\langle e_{\mathsf{nh}}\rangle\langle\theta\rangle$	The derivation $\mathcal{D}$ is expected to establish safety over a continuation trace θ (*i.e.*, a loop body or the continuation of a procedure call), where e_{nh} is the address at which the specification p_{uz} holds.
$\mathcal{D}_{\mathsf{uz}}, \mathcal{D}, \mathcal{D}' :: \texttt{dis\_uz}\langle p_{\mathsf{uz}}\rangle$	Use the derivation $\mathcal{D}_{\mathsf{uz}}$ to discharge implications in the derivation $\mathcal{D}$, resulting in the derivation $\mathcal{D}'$ (this is a wrapper for $\texttt{dis}\langle p_{\mathsf{u}}\rangle$).
$\mathcal{D}_{\mathsf{u}}, \mathcal{D}, \mathcal{D}' :: \texttt{dis}\langle p_{\mathsf{u}}\rangle$	Use the derivation $\mathcal{D}_{\mathsf{u}}$ (a proof of the proposition p_{u}) to discharge implications in the derivation $\mathcal{D}$, resulting in the derivation $\mathcal{D}'$.
$\mathcal{D} :: \texttt{vc}$	Deconstruct the derivation $\mathcal{D}$ into a VC proof outline.
$:: \texttt{goal}\langle e_{\mathsf{pc}}\rangle\langle p_{\mathsf{uz}}\rangle$	The goal specification p_{uz} (*i.e.*, a target loop invariant or procedure postcondition) is tagged by the address e_{pc}.
$:: \texttt{hyp\_cont}\langle e_{\mathsf{nh}}\rangle$	Use a continuation hypothesis with tag e_{nh}.
$:: \texttt{hyp\_pre}$	Use a precondition hypothesis.
$:: \texttt{hyp\_cop}$	Use a conditional branch hypothesis.

Table 7.3: Proof Classifiers

when k is zero and simply as

$$\omega'$$

when both k and k' are zero.

In Table 7.3, I give an intuitive reading of each of the proof classifiers.

In Section 7.3.2 through Section 7.3.11, I present the deductive system for proof-outline construction. Because the search affirmations are "higher-order" judgments on derivations, the notation becomes somewhat awkward when I must specify the shape of a VC proof. The solution I use is to identify an existential derivation variable (*e.g.*, $\mathcal{D}$) appearing in a search affirmation with an explicit VC proof derivation as a side condition of the inference rule. For example,

$$\frac{\Omega \rhd \mathcal{D}' :: \texttt{vc}}{\Omega \rhd \mathcal{D} :: \texttt{vc}}\ \mathsf{vc\_and\_el}$$

$$\text{where } \mathcal{D} \stackrel{\mathsf{def}}{=} \dfrac{\begin{array}{c}\mathcal{D}' \\ \cdot \vdash p_1 \wedge p_2 \mathbin{@} 0\end{array}}{\cdot \vdash p_1 \mathbin{@} 0}\ \wedge\mathsf{el}$$

should be understood to mean that the derivation

$$\dfrac{\begin{array}{c}\mathcal{D}' \\ \cdot \vdash p_1 \wedge p_2 \mathbin{@} 0\end{array}}{\cdot \vdash p_1 \mathbin{@} 0}\ \wedge\mathsf{el}$$

is the subject $\mathcal{D}$ of the conclusion of $\mathsf{vc\_and\_el}$.

There is a direct correspondence between the proof-outline rules and the proof-outline data structure I introduced in Section 6.6.1. In fact, each proof-outline constructor has an associated proof-outline rule (this correspondence is formalized in Section 7.4.2). Although one can read the proof-outline rules operationally as "Given the execution trace θ, how do I construct a proof of safety?", it is equally valid to read them deductively as "Given a proof-outline o, in what case does this outline apply?" The order of the rules follows this latter orientation, although the rules are normally read operationally.

The deductive system has a natural implementation as a logic program. To see the evaluation strategy, read the rules from conclusion to premises. The premises are normally evaluated in left-to-right order. Essentially, the object is to imitate the structure of a fully explicit proof, where important details of the proof are remembered as indices on rule labels. For example, loop invariants cannot be reconstructed automatically from the object code, so the loop invariant label $\mathsf{eval\_head}$ is indexed by the loop invariant p_{iuz} that the rule is applied to. This notation greatly simplifies the presentation of proof-outline extraction (see Section 7.4).

7.3.2 Hypothesis Rule

The conditional hypothesis rule hyp allows a conditional hypothesis to succeed if its premises ω_1 through ω_k can be established directly:

$$\frac{\Omega \rhd [t_1/y_1]\ \ldots [t_k/y_k]\,\omega_1 \quad \ldots \quad \Omega \rhd [t_1/y_1]\ \ldots [t_k/y_k]\,\omega_{k'}}{\Omega \rhd [t_1/y_1]\ \ldots [t_k/y_k]\,\omega'}\ \mathsf{hyp}$$

$$\text{where } \Omega \stackrel{\mathsf{def}}{=} \Omega_1,\ (\Pi y_1, \ldots, y_k.\omega_1, \ldots, \omega_{k'} \rightarrow \omega'),\ \Omega_2$$

The free time variables in the goals $(y_1, \ldots, y_k)$ must be instantiated consistently.

7.3.3 Callee Rule

$\boxed{\mathcal{D} :: \texttt{callee}\langle p\rangle\langle\theta\rangle}$

callee is the "top-level" rule that constructs a proof outline demonstrating the safety of a procedure at address e_{pc} with precondition p_{p} and postcondition p_{q}:

$$\frac{\begin{array}{l}\Omega \rhd \mathcal{D}_{\mathsf{pu}},\, \mathcal{D},\, \mathcal{D}' :: \mathtt{dis\_uz}\langle [a_{\mathsf{sq}_0}/x_{\mathsf{sq}_0}]\, [a_{\mathsf{sq}}/x_{\mathsf{sq}}]\, p_{\mathsf{puz}}\rangle \\ \Omega' \rhd \mathcal{D}' :: \mathtt{eval}\langle [e''_{\mathtt{pc}}/x_{\mathtt{pc}}]\, \theta\rangle\end{array}}{\Omega \rhd \mathcal{D} :: \mathtt{callee}\langle \begin{pmatrix}\forall x_{\mathsf{sp}}\!:\!\mathsf{fl}.\, \Box(\forall x_{\mathsf{sq}_0}\!:\!\mathsf{ri}.\, \forall x_{\mathsf{sq}}\!:\!\mathsf{fl}. \\ \mathtt{sq} = x_{\mathsf{sq}_0} \supset \Box(\mathtt{sq} = x_{\mathsf{sq}}) \supset p_{\mathsf{p}} \supset \mathtt{safe}\,\mathcal{U}\, p_{\mathsf{q}})\end{pmatrix}\rangle\langle\theta\rangle}\ \mathsf{callee}^{a_{\mathsf{sq}_0}, a_{\mathsf{sq}}, \mathcal{D}_{\mathsf{pu}}}$$

where $\Omega' \overset{\mathsf{def}}{\equiv} \Omega,\, :: \mathtt{goal}\langle e''_{\mathtt{pc}}\rangle\langle [a_{\mathsf{sq}_0}/x_{\mathsf{sq}_0}]\, p_{\mathsf{quz}}\rangle,\, (:: \mathtt{hyp\_pre} \to \mathcal{D}_{\mathsf{pu}} :: \mathtt{vc})$

and $p_{\mathsf{p}} \overset{\mathsf{def}}{\equiv} \mathtt{pc} = e_{\mathtt{pc}} \wedge \mathtt{gsp} = x_{\mathsf{sp}} \wedge p_{\mathsf{puz}}$

and $p_{\mathsf{q}} \overset{\mathsf{def}}{\equiv} \mathtt{pc} = e'_{\mathtt{pc}} \wedge p_{\mathsf{quz}}$

and $e''_{\mathtt{pc}} \overset{\mathsf{def}}{\equiv} [a_{\mathsf{sq}_0}/x_{\mathsf{sq}_0}]\, e'_{\mathtt{pc}}$

This rule corresponds to the proof-outline constructor `callee`. It is the most complex proof-outline rule. It introduces two parameters a_{sq_0} and a_{sq} to represent an abstraction of the current state and instantiate the state variables in the procedure specification. Additionally, it introduces a derivation *parameter* $\mathcal{D}_{\mathsf{pu}}$ to represent the hypothetical precondition of the procedure. This derivation parameter replaces the hypothetical precondition in the VC proof $\mathcal{D}$. Later, when an outline for the parameter is to be constructed, the outline `hyp_pre` is used instead (this is the effect of the second hypothetical goal in Ω').

The first premise of this rule discharges the hypothetical use of the precondition in the VC proof $\mathcal{D}$ by applying it to the derivation parameter $\mathcal{D}_{\mathsf{pu}}$. The second premise continues evaluation in the body of the procedure, replacing the placeholder $x_{\mathtt{pc}}$ with the correct return address in the body trace. The extended context Ω' identifies the postcondition as a successful goal when the return address is reached, and additionally associates the derivation parameter $\mathcal{D}_{\mathsf{pu}}$ with the hypothetical precondition proof outline. Note that the rule label is not indexed by the procedure specification, because procedure specifications are stored separately under the address at which the procedure is located, and thus need not be stored explicitly in the proof outline.

Note also that instead of substituting `sq` directly for x_{sq} in the specifications, I introduce an intermediate condition, $\Box(\mathtt{sq} = x_{\mathsf{sq}})$ to avoid a direct substitution. This is a cosmetic change that simplifies higher-order unification when the proof-outline rules are executed by a logic interpreter—avoiding a direct substitution keeps the specifications within the so-called *pattern fragment* [Mil91].

7.3.4 Evaluation Rules

$\boxed{\mathcal{D} :: \mathtt{eval}\langle\theta\rangle}$

In this section, I present a proof-outline construction rule for each possible constructor of a symbolic evaluation trace. Essentially, the trace is used as a "guide" for constructing the proof outline.

The ordinary loop-head rule $\mathsf{eval\_head}$ evaluates loop invariant p_{iuz} at address $e_{\mathtt{pc}}$:

$$\frac{\begin{array}{l}\Omega \rhd \mathcal{D},\, \mathcal{D}' :: \mathtt{check}\langle [a_{\mathsf{sq}_0}/x_{\mathsf{sq}_0}]\, p_{\mathsf{iuz}}\rangle \\ \Omega \rhd \mathcal{D}' :: \mathtt{loop}\langle e_{\mathtt{pc}}\rangle\langle [a_{\mathsf{sq}_0}/x_{\mathsf{sq}_0}]\, p_{\mathsf{iuz}}\rangle\langle e_{\mathsf{nh}}\rangle\langle\theta\rangle\end{array}}{\Omega \rhd \mathcal{D} :: \mathtt{eval}\langle \mathtt{head}\langle \mathtt{pc} = e_{\mathtt{pc}} \wedge p_{\mathsf{iuz}}\rangle\langle e_{\mathsf{nh}}\rangle\langle\theta\rangle\rangle}\ \mathsf{eval\_head}^{a_{\mathsf{sq}_0}}_{e_{\mathtt{pc}}, p_{\mathsf{iuz}}}$$

This rule corresponds to the proof-outline constructor `eval_head`. It introduces a fresh parameter a_{sq_0} to abstract over the machine state. The rule label is indexed by the program counter $e_{\mathtt{pc}}$ and loop invariant p_{iuz}. These indices are a notational device to simplify the presentation of the proof-outline extraction rules in Section 7.4. The first premise uses part of the VC proof $\mathcal{D}$ to show that the loop invariant holds for the current state. The remainder of the VC proof $\mathcal{D}'$ is then used in the evaluation of the loop body, which is accomplished by the `loop` classifier in the second premise.

eval_head_dom is a more complex version of the loop-head rule that enables a VC proof to be shared among instructions dominated by the current instruction:

$$\frac{\begin{array}{c}\Omega, :: \mathtt{goal}\langle e'_{\mathtt{pc}}\rangle\langle[a_{\mathsf{sq}_0}/x_{\mathsf{sq}_0}]\,p_{\mathsf{iuz}}\rangle \rhd \mathcal{D}' :: \mathtt{eval}\langle\theta_1\rangle \\ \Omega \rhd \mathcal{D}_2 :: \mathtt{loop}\langle e'_{\mathtt{pc}}\rangle\langle[a_{\mathsf{sq}_0}/x_{\mathsf{sq}_0}]\,p_{\mathsf{iuz}}\rangle\langle e_{\mathsf{nh}}\rangle\langle\theta_2\rangle\end{array}}{\Omega \rhd \mathcal{D} :: \mathtt{eval}\langle\mathtt{head\_dom}\langle\mathtt{pc} = e'_{\mathtt{pc}} \wedge p_{\mathsf{iuz}}\rangle\langle e_{\mathsf{nh}}\rangle\langle\theta_1\rangle\langle\theta_2\rangle\rangle}\ \mathsf{eval\_head\_dom}^{a_{\mathsf{sq}0}}_{e'_{\mathtt{pc}},p_{\mathsf{iuz}}}$$

$$\text{where } \mathcal{D} \stackrel{\mathsf{def}}{=} \frac{\begin{array}{c}\mathcal{D}_1\\ \cdot \vdash p_1 \,@\, 0\end{array} \quad \dfrac{\begin{array}{c}\mathcal{D}'_1\\ \cdot \vdash p'_1 \,@\, 0\end{array} \quad \begin{array}{c}\mathcal{D}_2\\ \cdot \vdash p_2 \,@\, 0\end{array}}{\cdot \vdash p'_1 \wedge p_2 \,@\, 0}\ \wedge\mathsf{i}}{\cdot \vdash p_1 \wedge (p'_1 \wedge p_2) \,@\, 0}\ \wedge\mathsf{i}$$

$$\text{and } \mathcal{D}' \stackrel{\mathsf{def}}{=} \frac{\begin{array}{c}\mathcal{D}_1\\ \cdot \vdash p_1 \,@\, 0\end{array} \quad \begin{array}{c}\mathcal{D}'_1\\ \cdot \vdash p'_1 \,@\, 0\end{array}}{\cdot \vdash p_1 \wedge p'_1 \,@\, 0}\ \wedge\mathsf{i}$$

This rule also corresponds to the proof-outline constructor `eval_head`. It only succeeds if the VC proof $\mathcal{D}$ has the correct structure.[3] The first premise uses a restructured fragment of the VC proof $\mathcal{D}'$ to continue evaluation until the loop invariant is reached for the first time. Note that the loop invariant will be established for the appropriate state by eval_tail if the loop-invariant goal is ever used. The second premise evaluates the loop body with the remainder of the VC proof, as for eval_head.

eval_tail establishes a search goal p_{uz}:

$$\frac{\Omega\rhd :: \mathtt{goal}\langle e_{\mathtt{pc}}\rangle\langle p_{\mathsf{uz}}\rangle \quad \Omega \rhd \mathcal{D}, \mathcal{D}' :: \mathtt{check}\langle p_{\mathsf{uz}}\rangle}{\Omega \rhd \mathcal{D} :: \mathtt{eval}\langle\mathtt{tail}\langle e_{\mathtt{pc}}\rangle\rangle}\ \mathsf{eval\_tail}$$

This rule corresponds to the proof-outline constructor `eval_tail`. The first premise retrieves the goal proposition p_{uz} from the search context where it is stored under the current program counter $e_{\mathtt{pc}}$. The second premise uses the VC proof $\mathcal{D}$ to show that this goal holds for the current state.

eval_step evaluates an ordinary instruction e_{I}:

$$\frac{\Omega \rhd \mathcal{D} :: \mathtt{eval}\langle\theta\rangle}{\Omega \rhd \mathcal{D} :: \mathtt{eval}\langle\mathtt{step}\langle e_{\mathsf{I}}\rangle\langle\theta\rangle\rangle}\ \mathsf{eval\_step}$$

[3] This particular structure is an artifact of how the SpecialJ symbolic evaluator operates.

The premise simply continues evaluation for the remainder of the trace. This rule corresponds to the proof-outline constructor `eval_step`.

eval_unsafe evaluates an *unsafe* instruction e_l:

$$\frac{\Omega \rhd \mathcal{D}_1 :: \mathtt{vc} \quad \Omega \rhd \mathcal{D}_2 :: \mathtt{eval}\langle\theta\rangle}{\Omega \rhd \mathcal{D} :: \mathtt{eval}\langle\mathtt{unsafe}\langle e_\mathsf{l}\rangle\langle\theta\rangle\rangle}\ \mathsf{eval\_unsafe}$$

$$\text{where } \mathcal{D} \stackrel{\mathsf{def}}{=} \frac{\begin{matrix}\mathcal{D}_1 \\ \cdot \vdash p_1 \mathbin{@} 0\end{matrix} \quad \begin{matrix}\mathcal{D}_2 \\ \cdot \vdash p_2 \mathbin{@} 0\end{matrix}}{\cdot \vdash p_1 \wedge p_2 \mathbin{@} 0}\ \wedge\mathsf{i}$$

This rule corresponds to the proof-outline constructor `eval_unsafe`. It only succeeds if the VC proof $\mathcal{D}$ has the correct structure. The first premise constructs a proof outline for the left-hand branch of the VC proof, which demonstrates the safety of e_l for the current state. The second premise continues evaluation for the remainder of the trace.

eval_unsafe_mem is a slight extension of eval_unsafe:

$$\frac{\Omega \rhd \mathcal{D}_1 :: \mathtt{vc} \quad \Omega \rhd \mathcal{D}_2 :: \mathtt{eval}\langle\theta\rangle}{\Omega \rhd \mathcal{D} :: \mathtt{eval}\langle\mathtt{unsafe\_mem}\langle e_\mathsf{l}\rangle\langle e_\mathsf{nm}\rangle\langle\theta\rangle\rangle}\ \mathsf{eval\_unsafe\_mem}_{e_\mathsf{nm}}$$

$$\text{where } \mathcal{D} \stackrel{\mathsf{def}}{=} \frac{\begin{matrix}\mathcal{D}_1 \\ \cdot \vdash p_1 \mathbin{@} 0\end{matrix} \quad \begin{matrix}\mathcal{D}_2 \\ \cdot \vdash p_2 \mathbin{@} 0\end{matrix}}{\cdot \vdash p_1 \wedge p_2 \mathbin{@} 0}\ \wedge\mathsf{i}$$

This rule corresponds to the proof-outline constructor `eval_unsafe_mem`. The rule label is indexed by the new abstract-memory index e_nm so that it can be easily extracted into a proof outline.

eval_split evaluates a conditional branch by considering both possible outcomes of the conditional:

$$\frac{\begin{matrix}\Omega, (:: \mathtt{hyp\_cop} \to \mathcal{D}_{\mathsf{cop}_1} :: \mathtt{vc}) \rhd \mathcal{D}'_1 :: \mathtt{eval}\langle\theta_1\rangle \\ \Omega, (:: \mathtt{hyp\_cop} \to \mathcal{D}_{\mathsf{cop}_2} :: \mathtt{vc}) \rhd \mathcal{D}'_2 :: \mathtt{eval}\langle\theta_2\rangle\end{matrix}}{\Omega \rhd \mathcal{D} :: \mathtt{eval}\langle\mathtt{split}\langle e_\mathsf{l}\rangle\langle\theta_1\rangle\langle\theta_2\rangle\rangle}\ \mathsf{eval\_split}^{\mathcal{D}_{\mathsf{cop}_1}, \mathcal{D}_{\mathsf{cop}_2}}$$

$$\text{where } \mathcal{D} \stackrel{\mathsf{def}}{=} \frac{\dfrac{\begin{matrix}\mathcal{D}_1 \\ \cdot, p_{\mathsf{cop}_1} \mathbin{@} 0 \vdash p_1 \mathbin{@} 0\end{matrix}}{\cdot \vdash p_{\mathsf{cop}_1} \supset p_1 \mathbin{@} 0}\ {\supset}\mathsf{i} \quad \dfrac{\begin{matrix}\mathcal{D}_2 \\ \cdot, p_{\mathsf{cop}_2} \mathbin{@} 0 \vdash p_2 \mathbin{@} 0\end{matrix}}{\cdot \vdash p_{\mathsf{cop}_2} \supset p_2 \mathbin{@} 0}\ {\supset}\mathsf{i}}{\cdot \vdash (p_{\mathsf{cop}_1} \supset p_1) \wedge (p_{\mathsf{cop}_2} \supset p_2) \mathbin{@} 0}\ \wedge\mathsf{i}$$

$$\text{and } \begin{matrix}\mathcal{D}_{\mathsf{cop}_1} \\ \cdot \vdash p_{\mathsf{cop}_1} \mathbin{@} 0\end{matrix} \text{ and } \begin{matrix}\mathcal{D}_{\mathsf{cop}_2} \\ \cdot \vdash p_{\mathsf{cop}_2} \mathbin{@} 0\end{matrix}$$

$$\text{and } \begin{matrix}\mathcal{D}'_1 \\ \cdot \vdash p_1 \mathbin{@} 0\end{matrix} \text{ and } \begin{matrix}\mathcal{D}'_2 \\ \cdot \vdash p_2 \mathbin{@} 0\end{matrix}$$

This rule corresponds to the proof-outline constructor `eval_split`. The VC proof is structured as a pair of implications that are each hypothetical in the corresponding outcome of the conditional. This structure arises because the VC proof is entitled to presume that the conditional succeeded or failed during evaluation of the corresponding arm of the branch. The hypothetical conditional outcomes are replaced by derivation parameters $\mathcal{D}_{\mathsf{cop}_1}$ and $\mathcal{D}_{\mathsf{cop}_2}$ in the VC proof. When used in an outline of the VC proof, these parameters are identified by the constructor `hyp_cop`. This is the function of the additional hypotheses in the search contexts of the premises. The derivations $\mathcal{D}'_1$ and $\mathcal{D}'_2$ do not exist as such in the VC proof, but their existence can be inferred by applying the substitution principle [Pfe99]. Operationally, $\mathcal{D}'_1$ and $\mathcal{D}'_2$ can be constructed by replacing uses of the free hypotheses ($p_1 \mathbin{@} 0$ and $p_2 \mathbin{@} 0$) in $\mathcal{D}_1$ and $\mathcal{D}_2$ with $\mathcal{D}_{\mathsf{cop}_1}$ and $\mathcal{D}_{\mathsf{cop}_2}$, respectively.

eval_call evaluates a call to a procedure with precondition p_p and postcondition p_q:

$$\frac{\begin{array}{c}\Omega \rhd \mathcal{D}, \mathcal{D}' :: \mathtt{check}\langle [a_{\mathsf{sq}_0}/x_{\mathsf{sq}_0}]\, p_{\mathsf{puz}}\rangle \\ \Omega \rhd \mathcal{D}' :: \mathtt{cont}\langle [a_{\mathsf{sq}_0}/x_{\mathsf{sq}_0}]\, p_{\mathsf{quz}}\rangle\langle e_{\mathsf{nh}}\rangle\langle\theta\rangle\end{array}}{\Omega \rhd \mathcal{D} :: \mathtt{eval}\langle\mathtt{call}\langle p_\mathsf{p}\rangle\langle p_\mathsf{q}\rangle\langle e_{\mathsf{nh}}\rangle\langle\theta\rangle\rangle}\ \mathsf{eval\_call}^{a_{\mathsf{sq}0}}$$

$$\text{where } p_\mathsf{p} \stackrel{\mathsf{def}}{\equiv} \mathtt{pc} = e_{\mathsf{pc}} \wedge \mathtt{gsp} = e_{\mathsf{sp}} \wedge p_{\mathsf{puz}}$$
$$\text{and } p_\mathsf{q} \stackrel{\mathsf{def}}{\equiv} \mathtt{pc} = e'_{\mathsf{pc}} \wedge p_{\mathsf{quz}}$$

This rule corresponds to one of the proof-outline constructors `eval_call_and` or `eval_call_false`, depending on the postcondition. The first premise uses the VC proof to show that the precondition holds. The second premise resumes evaluation in the continuation of the call. The postcondition will be accumulated into the set of available hypotheses. Note that the rule label is not indexed by the procedure specification, because procedure specifications are stored separately under the address at which the procedure is located, and thus need not be stored explicitly in the proof outline.

7.3.5 Checking Rules

$\boxed{\mathcal{D}, \mathcal{D}' :: \mathtt{check}\langle p_{\mathsf{uz}}\rangle}$

This section contains proof-outline rules for establishing specifications with some part of a VC proof. The specification is always a right-associative nested series of conjunctions.

check_z is the base case, when the specification is empty:

$$\frac{}{\Omega \rhd \mathcal{D}, \mathcal{D} :: \mathtt{check}\langle\top \wedge p_\mathsf{z}\rangle}\ \mathsf{check\_z}$$

This rule corresponds to the proof-outline constructor `ck_z`.

check_u establishes an individual specification element p_u by extracting a fragment of the VC proof $\mathcal{D}$:

$$\frac{\Omega \rhd \mathcal{D}_1 :: \mathtt{vc} \quad \Omega \rhd \mathcal{D}_2, \mathcal{D}' :: \mathtt{check}\langle p'_\mathsf{u} \wedge p_\mathsf{z}\rangle}{\Omega \rhd \mathcal{D}, \mathcal{D}' :: \mathtt{check}\langle (p_\mathsf{u} \wedge p'_\mathsf{u}) \wedge p_\mathsf{z}\rangle}\ \mathsf{check\_u}_{p_\mathsf{u}}$$

$$\text{where } \mathcal{D} \stackrel{\mathsf{def}}{=} \frac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \cdot \vdash p_1 \mathbin{@} 0 & \cdot \vdash p_2 \mathbin{@} 0\end{array}}{\cdot \vdash p_1 \wedge p_2 \mathbin{@} 0}\ \wedge\mathsf{i}$$

This rule corresponds to the proof-outline constructor `ck_u`. It only succeeds if the VC proof $\mathcal{D}$ has the correct structure. This structure is an artifact of symbolic evaluation strategy used by SpecialJ. The first premise constructs a proof outline for the left-hand branch of the VC proof, which establishes p_{u} for the current state. The second premise establishes the rest of the specification p'_{u} with the rest of the VC proof $\mathcal{D}_2$.

7.3.6 Loop Rule

$\boxed{\mathcal{D} :: \texttt{loop}\langle e_{\mathsf{pc}}\rangle\langle p_{\mathsf{iuz}}\rangle\langle e_{\mathsf{nh}}\rangle\langle\theta\rangle}$

loop evaluates a loop body starting at address e_{pc} with loop invariant p_{iuz}:

$$\frac{\Omega, :: \texttt{goal}\langle e_{\mathsf{pc}}\rangle\langle p_{\mathsf{iuz}}\rangle \rhd \mathcal{D} :: \texttt{cont}\langle p_{\mathsf{iuz}}\rangle\langle e_{\mathsf{nh}}\rangle\langle\theta\rangle}{\Omega \rhd \mathcal{D} :: \texttt{loop}\langle e_{\mathsf{pc}}\rangle\langle p_{\mathsf{iuz}}\rangle\langle e_{\mathsf{nh}}\rangle\langle\theta\rangle}\ \mathsf{loop}$$

The premise resumes evaluation in the body of the loop with the loop invariant as a valid goal. The invariant will also be accumulated into the set of available hypotheses.

7.3.7 Continuation Rules

$\boxed{\mathcal{D} :: \texttt{cont}\langle p_{\mathsf{uz}}\rangle\langle e_{\mathsf{nh}}\rangle\langle\theta\rangle}$

The continuation rules discharge implications in the VC proof that are hypothetical in a given specification—the term "continuation" here is generalized to include loop bodies as well as procedure-call continuations. The VC proof is structured as a series of implications for each individual component of the specification. This structure arises because the VC proof is entitled to presume that the specification (a postcondition or loop invariant) holds during evaluation of the continuation or loop body.

cont_and discharges implications in the VC proof for a given specification p_{uz}:

$$\frac{\begin{array}{l}\Omega \rhd \mathcal{D}_{\mathsf{u}}, \mathcal{D}, \mathcal{D}' :: \texttt{dis\_uz}\langle [a_{\mathsf{sq}}/x_{\mathsf{sq}}]\, p_{\mathsf{uz}}\rangle \\ \Omega, (:: \texttt{hyp\_cont}\langle e_{\mathsf{nh}}\rangle \to \mathcal{D}_{\mathsf{u}} :: \texttt{vc}) \rhd \mathcal{D}' :: \texttt{eval}\langle\theta\rangle\end{array}}{\Omega \rhd \mathcal{D} :: \texttt{cont}\langle p_{\mathsf{uz}}\rangle\langle e_{\mathsf{nh}}\rangle\langle\theta\rangle}\ \mathsf{cont\_and}^{a_{\mathsf{sq}},\mathcal{D}_{\mathsf{u}}}_{e_{\mathsf{nh}}}$$

The hypothetical specification is replaced by a derivation parameter $\mathcal{D}_{\mathsf{u}}$ in the VC proof. When used in an outline of the VC proof, this parameter is identified by the constructor $\texttt{hyp\_cont}\langle e_{\mathsf{nh}}\rangle$. This is the function of the additional hypothesis in the search context of the second premise. e_{nh} is the index assigned to the specification being discharged. Once the implications in the VC proof are discharged by the first premise, the second premise uses the resulting VC proof $\mathcal{D}'$ is used to certify the continuation code.

cont_false is a special case when the continuation code is never reached:

$$\frac{}{\Omega \rhd \mathcal{D} :: \texttt{cont}\langle\bot\rangle\langle e_{\mathsf{nh}}\rangle\langle\theta\rangle}\ \mathsf{cont\_false}$$

This rule is used when a procedure is called whose postcondition is $\bot$ (*e.g.*, an exception generator).

7.3.8 UZ Discharge Rule

$\boxed{\mathcal{D}_{\mathsf{u}},\, \mathcal{D},\, \mathcal{D}' :: \mathtt{dis\_uz}\langle p_{\mathsf{uz}}\rangle}$

dis_uz is a notational convenience rule that breaks down a conjunction specification into its component specifications:

$$\frac{\Omega \rhd \mathcal{D}_{\mathsf{u}},\, \mathcal{D},\, \mathcal{D}' :: \mathtt{dis}\langle p_{\mathsf{u}}\rangle}{\Omega \rhd \mathcal{D}_{\mathsf{u}},\, \mathcal{D},\, \mathcal{D}' :: \mathtt{dis\_uz}\langle p_{\mathsf{u}} \wedge p_{\mathsf{z}}\rangle}\ \mathsf{dis\_uz}$$

7.3.9 Discharge Rules

$\boxed{\mathcal{D}_{\mathsf{u}},\, \mathcal{D},\, \mathcal{D}' :: \mathtt{dis}\langle p_{\mathsf{u}}\rangle}$

The discharge rules transform a VC proof by instantiating its internal implications with explicit proofs. During proof reconstruction (see Section 8), the explicit proofs are part of the proof skeleton that is constructed algorithmically from derived rules.

Each complete specification to be discharged is a right-associative conjunction of component specifications, each of which is discharged individually. In a VC proof, the component specification hypotheses are nested in a sequence of implication introduction rules. Additionally, the VC proof may contain universal quantifiers that can essentially be discarded because the proof outline does not depend on how the bound variables are instantiated. For more expressive security policies and/or proof outlines, it may be necessary to determine bound-variable instantiations via unification.

dis_true is the base case for a conjunction of specification elements:

$$\frac{}{\Omega \rhd \mathcal{D}_{\mathsf{u}},\, \mathcal{D},\, \mathcal{D} :: \mathtt{dis}\langle \top \rangle}\ \mathsf{dis\_true}$$

No implications are discharged.

dis_imp discharges a single specification element p_{u} in a VC proof $\mathcal{D}$:

$$\frac{\Omega \rhd \mathcal{D}'_{\mathsf{u}},\, \mathcal{D}'_1,\, \mathcal{D}' :: \mathtt{dis}\langle p'_{\mathsf{u}}\rangle}{\Omega \rhd \mathcal{D}_{\mathsf{u}},\, \mathcal{D},\, \mathcal{D}' :: \mathtt{dis}\langle p_{\mathsf{u}} \wedge p'_{\mathsf{u}}\rangle}\ \mathsf{dis\_imp}$$

$$\text{where } \mathcal{D} \stackrel{\mathsf{def}}{=} \frac{\begin{array}{c}\mathcal{D}_1 \\ \cdot,\, p_{\mathsf{u}} \mathbin{@} 0 \vdash p_1 \mathbin{@} 0\end{array}}{\cdot \vdash p_{\mathsf{u}} \supset p_1 \mathbin{@} 0}\ \supset\mathsf{i}$$

$$\text{and } \frac{\begin{array}{c}\mathcal{D}_{\mathsf{u}} \\ \cdot \vdash p_{\mathsf{u}} \wedge p'_{\mathsf{u}} \mathbin{@} 0\end{array}}{\cdot \vdash p_{\mathsf{u}} \mathbin{@} 0}\ \wedge\mathsf{el} \quad \text{and } \begin{array}{c}\mathcal{D}'_1 \\ \cdot \vdash p_1 \mathbin{@} 0\end{array}$$

$$\text{and } \mathcal{D}'_{\mathsf{u}} \stackrel{\mathsf{def}}{=} \frac{\begin{array}{c}\mathcal{D}_{\mathsf{u}} \\ \cdot \vdash p_{\mathsf{u}} \wedge p'_{\mathsf{u}} \mathbin{@} 0\end{array}}{\cdot \vdash p'_{\mathsf{u}} \mathbin{@} 0}\ \wedge\mathsf{er}$$

This rule fails if the VC proof $\mathcal{D}$ does not have the correct structure. The derivation $\mathcal{D}'_1$ is obtained from $\mathcal{D}$ by substituting a derivation of p_{u} for the open hypothesis in $\mathcal{D}_1$. The premise of this rule continues to discharge hypotheses in the remainder of the VC proof by constructing a proof of p'_{u} from the current proof of $p_{\mathsf{u}} \wedge p'_{\mathsf{u}}$.

dis_all instantiates a universal quantifier in a VC proof $\mathcal{D}$:

$$\frac{\Omega \rhd \mathcal{D}_{\mathsf{u}},\, \mathcal{D}'_1,\, \mathcal{D}' :: \mathtt{dis}\langle p_{\mathsf{u}} \wedge p'_{\mathsf{u}}\rangle}{\Omega \rhd \mathcal{D}_{\mathsf{u}},\, \mathcal{D},\, \mathcal{D}' :: \mathtt{dis}\langle p_{\mathsf{u}} \wedge p'_{\mathsf{u}}\rangle}\ \mathsf{dis\_all}^{\mathcal{D}_{\mathsf{ri}}}$$

$$\text{where } \mathcal{D} \stackrel{\mathsf{def}}{=} \dfrac{\begin{array}{c}[a/x]\,\mathcal{D}_1\\ \cdot,\, a\!:\!\mathsf{ri} \vdash [a/x]\,p_1 \mathbin{@} 0\end{array}}{\cdot \vdash \forall x\!:\!\mathsf{ri}.\, p_1 \mathbin{@} 0}\ \forall\mathsf{i}^a$$

$$\text{and } \begin{array}{c}\mathcal{D}_{\mathsf{ri}}\\ \cdot \vdash e\!:\!\mathsf{ri}\end{array} \ \text{ and } \begin{array}{c}\mathcal{D}'_1\\ \cdot \vdash [e/x]\,p_1 \mathbin{@} 0\end{array}$$

This rule fails if the VC proof $\mathcal{D}$ does not have the correct structure. In general, the instantiating expression e must be found by unification, but in the case of Java type safety, proof outlines do not depend on such instantiations, so the choice does not matter. $\mathcal{D}'_1$ is obtained by substituting e for x in $\mathcal{D}_1$: $[e/x]\,\mathcal{D}_1$.

7.3.10 VC Rules $\boxed{\mathcal{D} :: \mathtt{vc}}$

The VC rules are responsible for duplicating the structure of a VC proof in the proof-outline derivation. I only show a few representative cases here, because the rules follow the proof-outline data structure quite closely.

vc_and_el mimics the left conjunction elimination rule:

$$\frac{\Omega \rhd \mathcal{D}' :: \mathtt{vc}}{\Omega \rhd \mathcal{D} :: \mathtt{vc}}\ \mathsf{vc\_and\_el}$$

$$\text{where } \mathcal{D} \stackrel{\mathsf{def}}{=} \dfrac{\begin{array}{c}\mathcal{D}'\\ \cdot \vdash p_1 \wedge p_2 \mathbin{@} 0\end{array}}{\cdot \vdash p_1 \mathbin{@} 0}\ \wedge\mathsf{el}$$

This rule corresponds to the proof-outline constructor `vc_and_el`. The premise simply establishes the corresponding premise of the VC proof.

The right rule vc_and_er is similar:

$$\frac{\Omega \rhd \mathcal{D}' :: \mathtt{vc}}{\Omega \rhd \mathcal{D} :: \mathtt{vc}}\ \mathsf{vc\_and\_er}$$

$$\text{where } \mathcal{D} \stackrel{\mathsf{def}}{=} \dfrac{\begin{array}{c}\mathcal{D}'\\ \cdot \vdash p_1 \wedge p_2 \mathbin{@} 0\end{array}}{\cdot \vdash p_2 \mathbin{@} 0}\ \wedge\mathsf{er}$$

This rule corresponds to the proof-outline constructor `vc_and_er`.

The expression normalization rule vc_normE_e is more interesting:

$$\frac{}{\Omega \rhd \mathcal{D} :: \mathtt{vc}}\ \mathsf{vc\_normE\_e}$$

$$\text{where } \mathcal{D} \stackrel{\mathsf{def}}{=} \dfrac{\begin{array}{c}\mathcal{D}'\\ \cdot \vdash e \Longrightarrow^* e'\end{array}}{\cdot \vdash e = e' \mathbin{@} 0} \Longrightarrow^* \mathsf{e_{exp}}$$

This rule corresponds to the proof-outline constructor `vc_normE_e`. The remainder of the VC proof $\mathcal{D}'$ is discarded here, because the derivation can be reconstructed within the decidable rewriting system. Note that I also choose not to index the label of the inference rule by the expression to be rewritten e. For Java type safety, e is always sufficiently constrained such that its value need not be stored explicitly in a proof outline.

The rest of the VC proof rules follow a similar strategy—I do not present them here in the interest of brevity.

7.3.11 Hypothetical Rules

The three "hypothetical" rules always succeed. Their function is to provide an identifying label in the proof-outline derivation so that a suitable proof-outline constructor can be extracted (see Section 7.4.5). Note that these rules are only used after a conditional hypothesis (*e.g.*, $(:: \texttt{hyp\_cont}\langle e_{\mathsf{nh}}\rangle \rightarrow \mathcal{D}_{\mathsf{u}} :: \texttt{vc}))$ has been used by hyp, but because hyp does not contain sufficient information to show which hypothesis was used, an explicit identifying rule is needed.

$$\frac{}{\Omega\rhd :: \texttt{hyp\_cont}\langle e_{\mathsf{nh}}\rangle}\ \mathsf{hyp\_cont}_{e_{\mathsf{nh}}} \qquad \frac{}{\Omega\rhd :: \texttt{hyp\_pre}}\ \mathsf{hyp\_pre} \qquad \frac{}{\Omega\rhd :: \texttt{hyp\_cop}}\ \mathsf{hyp\_cop}$$

These rules correspond to the proof-outline constructors `vc_hyp_cont`, `vc_hyp_pre`, and `vc_hyp_cop`.

7.4 Extracting Proof Outlines

I now show how proof outlines can be systematically extracted from proof-outline derivations. The operator $|\cdot|$ maps a proof-outline derivation to a proof outline. The defining equations are in the form

$$|\mathcal{S}| = o$$

where $\mathcal{S}$ is a proof-outline derivation and o is the corresponding proof outline.

In most cases, extracting a proof outline is straightforward. It is important to note that discharge rules and goal rules are never extracted into proof outlines, because these rules are used solely to administer the search (by transforming VC proofs, for example), and have no proof "content" of their own.

Note that in my PCC system proof outline extraction is not implemented as a distinct transformation, but is rather an implicit part of the encoding and decoding rules (see Section 8.2.2).

7.4.1 Callee Rule

A callee outline is extracted by discarding the discharge derivation $\mathcal{S}_1$ and extracting an outline from the evaluation derivation $\mathcal{S}_2$:

$$\left| \frac{\begin{array}{cc} \mathcal{S}_1 & \mathcal{S}_2 \\ \Omega_1 \rhd \omega_1 & \Omega_2 \rhd \omega_2 \end{array}}{\Omega' \rhd \omega'}\ \mathsf{callee} \right| = \texttt{callee}\langle|\mathcal{S}_2|\rangle$$

$$\left|\;\dfrac{\begin{array}{c}\mathcal{S}_1\\ \Omega_1 \rhd \omega_1\end{array} \quad \dfrac{\dfrac{\begin{array}{c}\mathcal{S}_2\\ \Omega''_2 \rhd \omega''_2\end{array} \quad \begin{array}{c}\mathcal{S}_3\\ \Omega''_3 \rhd \omega''_3\end{array}}{\Omega'_2 \rhd \omega'_2}\,\mathsf{cont\_and}_{e_{\mathsf{nh}}}}{\Omega_2 \rhd \omega_2}\,\mathsf{loop}}{\Omega' \rhd \omega'}\,\mathsf{eval\_head}_{e_{\mathsf{pc}},p_{\mathsf{iuz}}}\;\right| = \mathtt{eval\_head}\langle e_{\mathtt{pc}}\rangle\langle p_{\mathsf{iuz}}\rangle\langle e_{\mathsf{nh}}\rangle\langle \mathtt{eval\_tail}\langle|\mathcal{S}_1|\rangle\rangle\langle|\mathcal{S}_3|\rangle$$

$$\left|\;\dfrac{\begin{array}{c}\mathcal{S}_1\\ \Omega_1 \rhd \omega_1\end{array} \quad \dfrac{\dfrac{\begin{array}{c}\mathcal{S}_2\\ \Omega''_2 \rhd \omega''_2\end{array} \quad \begin{array}{c}\mathcal{S}_3\\ \Omega''_3 \rhd \omega''_3\end{array}}{\Omega'_2 \rhd \omega'_2}\,\mathsf{cont\_and}_{e_{\mathsf{nh}}}}{\Omega_2 \rhd \omega_2}\,\mathsf{loop}}{\Omega' \rhd \omega'}\,\mathsf{eval\_head\_dom}_{e'_{\mathsf{pc}},p_{\mathsf{iuz}}}\;\right| = \mathtt{eval\_head}\langle e'_{\mathtt{pc}}\rangle\langle p_{\mathsf{iuz}}\rangle\langle e_{\mathsf{nh}}\rangle\langle|\mathcal{S}_1|\rangle\langle|\mathcal{S}_3|\rangle$$

$$\left|\;\dfrac{\begin{array}{c}\mathcal{S}_1\\ \Omega_1 \rhd \omega_1\end{array} \quad \begin{array}{c}\mathcal{S}_2\\ \Omega_2 \rhd \omega_2\end{array}}{\Omega' \rhd \omega'}\,\mathsf{eval\_tail}\;\right| = \mathtt{eval\_tail}\langle|\mathcal{S}_2|\rangle$$

$$\left|\;\dfrac{\begin{array}{c}\mathcal{S}\\ \Omega \rhd \omega\end{array}}{\Omega' \rhd \omega'}\,\mathsf{eval\_step}\;\right| = \mathtt{eval\_step}\langle|\mathcal{S}|\rangle$$

$$\left|\;\dfrac{\begin{array}{c}\mathcal{S}_1\\ \Omega_1 \rhd \omega_1\end{array} \quad \begin{array}{c}\mathcal{S}_2\\ \Omega_2 \rhd \omega_2\end{array}}{\Omega' \rhd \omega'}\,\mathsf{eval\_unsafe}\;\right| = \mathtt{eval\_unsafe}\langle|\mathcal{S}_1|\rangle\langle|\mathcal{S}_2|\rangle$$

$$\left|\;\dfrac{\begin{array}{c}\mathcal{S}_1\\ \Omega_1 \rhd \omega_1\end{array} \quad \begin{array}{c}\mathcal{S}_2\\ \Omega_2 \rhd \omega_2\end{array}}{\Omega' \rhd \omega'}\,\mathsf{eval\_unsafe\_mem}_{e_{\mathsf{nm}}}\;\right| = \mathtt{eval\_unsafe\_mem}\langle|\mathcal{S}_1|\rangle\langle e_{\mathsf{nm}}\rangle\langle|\mathcal{S}_2|\rangle$$

$$\left|\;\dfrac{\begin{array}{c}\mathcal{S}_1\\ \Omega_1 \rhd \omega_1\end{array} \quad \begin{array}{c}\mathcal{S}_2\\ \Omega_2 \rhd \omega_2\end{array}}{\Omega' \rhd \omega'}\,\mathsf{eval\_split}\;\right| = \mathtt{eval\_split}\langle|\mathcal{S}_1|\rangle\langle|\mathcal{S}_2|\rangle$$

$$\left|\;\dfrac{\begin{array}{c}\mathcal{S}_1\\ \Omega_1 \rhd \omega_1\end{array} \quad \dfrac{\begin{array}{c}\mathcal{S}_2\\ \Omega'_2 \rhd \omega'_2\end{array} \quad \begin{array}{c}\mathcal{S}_3\\ \Omega'_3 \rhd \omega'_3\end{array}}{\Omega_2 \rhd \omega_2}\,\mathsf{cont\_and}_{e_{\mathsf{nh}}}}{\Omega' \rhd \omega'}\,\mathsf{eval\_call}\;\right| = \mathtt{eval\_call\_and}\langle e_{\mathsf{nh}}\rangle\langle|\mathcal{S}_1|\rangle\langle|\mathcal{S}_3|\rangle$$

$$\left|\;\dfrac{\begin{array}{c}\mathcal{S}_1\\ \Omega_1 \rhd \omega_1\end{array} \quad \dfrac{}{\Omega_2 \rhd \omega_2}\,\mathsf{cont\_false}}{\Omega' \rhd \omega'}\,\mathsf{eval\_call}\;\right| = \mathtt{eval\_call\_false}\langle|\mathcal{S}_1|\rangle$$

Figure 7.3: Extracting Evaluation Outlines

7.4.2 Evaluation Rules

Figure 7.3 shows how proof outlines are extracted from evaluation derivations. eval_head and eval_head_dom are not distinguished at the level of proof outlines. Their only important difference is in how the incoming VC proof is processed. Observe that indices on inference-rule labels are carried directly into the proof outline (these details are essential to efficient proof reconstruction).

Discharge derivations (*i.e.*, $\mathcal{S}_2$ in eval_head, eval_head_dom, and eval_call) are discarded completely, because their only function is to transform the VC proof. The goal derivation $\mathcal{S}_1$ in eval_tail is similarly discarded.

7.4.3 Checking Rules

Extracting a proof outline from a "checking" derivation is absolutely straightforward, because the two are in direct correspondence:

$$\left| \frac{}{\Omega \rhd \omega}\ \mathsf{check\_z} \right| = \mathtt{ck\_z}$$

$$\left| \frac{\begin{array}{cc}\mathcal{S}_1 & \mathcal{S}_2 \\ \Omega_1 \rhd \omega_1 & \Omega_2 \rhd \omega_2\end{array}}{\Omega' \rhd \omega'}\ \mathsf{check\_u} \right| = \mathtt{ck\_u}\langle|\mathcal{S}_1|\rangle\langle|\mathcal{S}_2|\rangle$$

7.4.4 VC Rules

Extracting a proof outline from a VC derivation is also straightforward:

$$\left| \frac{\begin{array}{c}\mathcal{S} \\ \Omega \rhd \omega\end{array}}{\Omega' \rhd \omega'}\ \mathsf{vc\_and\_el} \right| = \mathtt{vc\_and\_el}\langle|\mathcal{S}|\rangle$$

$$\left| \frac{\begin{array}{c}\mathcal{S} \\ \Omega \rhd \omega\end{array}}{\Omega' \rhd \omega'}\ \mathsf{vc\_and\_er} \right| = \mathtt{vc\_and\_er}\langle|\mathcal{S}|\rangle$$

$$\left| \frac{}{\Omega \rhd \omega}\ \mathsf{vc\_normE\_e} \right| = \mathtt{vc\_normE\_e}$$

I only show a representative sample of these rules.

7.4.5 Hypothetical Rules

Extracting a proof outline for the use of a hypothesis is driven by the artificial hypothetical rule that identifies which hypothesis was used:

$$\left| \frac{\frac{}{\Omega \rhd \omega}\ \mathsf{hyp\_cont}_{e_{\mathsf{nh}}}}{\Omega' \rhd \omega'}\ \mathsf{hyp} \right| = \mathtt{vc\_hyp\_cont}\langle e_{\mathsf{nh}}\rangle$$

$$\left| \frac{\frac{}{\Omega \rhd \omega}\ \mathsf{hyp\_pre}}{\Omega' \rhd \omega'}\ \mathsf{hyp} \right| = \mathtt{vc\_hyp\_pre}$$

$$\left| \ \dfrac{\overline{\Omega \rhd \omega}\ \mathsf{hyp\_cop}}{\Omega' \rhd \omega'}\ \mathsf{hyp} \ \right| = \mathtt{vc\_hyp\_cop}$$

Chapter 8

Proof Engineering

In this chapter, I describe my approach to proof engineering for my PCC implementation. In Section 8.1, I illustrate how proofs are reconstructed from minimal outlines, whereas in Section 8.2, I briefly survey how proofs are encoded in certificates.

8.1 Proof Reconstruction

The *proof reconstruction* problem is to establish a particular judgment, given a possible outline of a proof of that judgment. I approach this problem by localizing the search for the judgment to a particular *reconstruction scope* that is determined by the proof outline. Intuitively, the reconstruction scope is a restriction of the total set of inference rules that makes it possible to derive the judgment efficiently. A reconstruction scope can also be thought of as a *name* for a particular set of inference rules. In some cases, the reconstruction scope will only allow one particular rule to be applied; in other cases, some small number of rules will be available, and one will be selected according to the shape of the particular judgment. The core idea behind this approach can be traced to Pfenning [Pfe01], in which a formal system for proof irrelevance is proposed as a possible foundation for a generalization of an oracle-based proof encoding [NR01].

This approach is reminiscent of the use of *tactics* and *tacticals* [GMW79, Fel93] in logic programming, except that here the implementation of the search procedure is "threaded" through the formal system itself, rather than being coded as a separate procedure that manipulates an explicit derivation (*i.e.*, a trace of a run of the reconstruction "function" contains the desired derivation). To adopt a logic-programming viewpoint, a clause of a logic program corresponds to a tactic, and the state of a tactical is embedded into the logic program by specializing the relevant clauses. This orientation enables me to use a relatively simple logic interpreter to carry out the reconstruction strategy from within the formal system—no additional language implementation is needed. Additionally, because proof reconstruction is part of the formal system, it can be derived formally if it is specified correctly. Thus, the entire content of this section is an *untrusted*, derived system that is supplied by the code producer and checked for correctness by the code consumer. I describe how the formal system is actually encoded in Section 8.2. The formal system in this section is essentially a "blueprint" for this implementation.

One can imagine the proof outline and its attendant reconstruction scope as a constraint on the search for a type safety proof that essentially "directs" the interpreter to apply a given rule at a given point. The reconstruction scope and proof outline are necessary because the inference rules of the program logic and type system are not naturally syntax directed. This is unlike other approaches to object-code certification such as TAL [MWCG98] that enable typing derivations to be reconstructed automatically.

By viewing the reconstruction scope as constraints on proof search, one is led to see many inference rules in this section as a *specializations* of the derived program-logic rules from Chapter 6. In fact, the program-logic rules are specialized to the point where they comprise a deterministic logic program, given a valid proof outline to interpret. In the following sections, I will note where a given proof-reconstruction judgment or rule is related to one or more program-logic judgments or rules.

8.1.1 Syntax

The proof reconstruction scopes are partitioned as follows:

$$
\begin{array}{lrl}
\text{Reconstruction Scopes} & O ::= & \texttt{callee}\langle oc\rangle \\
& & \mid \texttt{eval\_next}\langle oe\rangle \mid \texttt{eval}\langle oe\rangle \\
& & \mid \texttt{check}\langle ock\rangle \mid \texttt{checku}\langle ock\rangle \mid \texttt{vc}\langle ovc\rangle \\
& & \mid \texttt{caller} \mid \texttt{checkz} \mid \texttt{initz} \\
& & \mid \texttt{ande1\_mem} \mid \texttt{ande1\_ts} \mid \texttt{ande4\_ts} \mid \texttt{ori} \\
& & \mid \texttt{leq\_ts} \mid \texttt{lt\_ts} \mid \texttt{next} \mid \texttt{fetch} \\
& & \mid \texttt{sub0} \mid \texttt{sub} \mid \texttt{sub0u} \mid \texttt{subu} \mid \texttt{sub0z} \mid \texttt{subz} \\
& & \mid \texttt{sub0u1} \mid \texttt{subu1} \mid \texttt{sub0z1} \mid \texttt{subz1} \\
& & \mid \texttt{normE1} \\
& & \mid \texttt{hyp\_cont}\langle e^{\texttt{wd}}\rangle \mid \texttt{hyp\_mem}\langle e^{\texttt{wd}}\rangle \\
& & \mid \texttt{hyp\_pre} \mid \texttt{hyp\_cop} \mid \texttt{hyp\_lt\_ts} \\
& & \mid \texttt{goal}\langle e^{\texttt{wd}}\rangle\langle p\rangle \\
& & \mid \texttt{pf}
\end{array}
$$

The reconstruction scope `pf` is special in that is has the effect of "lifting" the judgment back into the standard formal system.

The affirmation

$$\Omega \rhd \langle\!\langle J \rangle\!\rangle_O$$

indicates that a proof of the judgment J can be reconstructed automatically from information provided by the reconstruction scope O, assuming that all goals in Ω succeed.

I now give an intuitive reading of each of the reconstruction scopes.

The callee reconstruction scope is a specialization of the procedure introduction rule of the program logic (see Section 6.3.4). The callee scope indicates that the proof outline *oc* contains sufficient information to reconstruct a safety proof for a procedure with the given specification:

$$\langle\!\langle \forall x_{\mathsf{sp}}\!:\!\mathsf{ri}.\ \Box(\forall x_{\mathsf{sq}_0}\!:\!\mathsf{ri}.\ \forall x_{\mathsf{sq}}\!:\!\mathsf{fl}.\ \mathtt{sq} = x_{\mathsf{sq}_0} \supset \Box(\mathtt{sq} = x_{\mathsf{sq}}) \supset p_{\mathsf{p}} \supset \mathtt{safe}\,\mathcal{U}\,p_{\mathsf{q}}) \mathbin{@} 0 \rangle\!\rangle_{\mathtt{callee}\langle oc\rangle}$$
$$\text{where } p_{\mathsf{p}} \overset{\mathsf{def}}{\equiv} \mathtt{pc} = e_{\mathtt{pc}} \wedge \mathtt{gsp} = x_{\mathsf{sp}} \wedge p_{\mathsf{puz}}$$
$$\text{and } p_{\mathsf{q}} \overset{\mathsf{def}}{\equiv} \mathtt{pc} = e'_{\mathtt{pc}} \wedge p_{\mathsf{quz}}$$

$\mathtt{eval\_next}\langle oe\rangle$ indicates that the proof outline *oe* contains sufficient information to reconstruct a proof that shows safety until the goal p_{gl}, after starting in state $e_{\mathtt{sq}}$, and taking at least one step:

$$\langle\!\langle \mathtt{sq} = e_{\mathtt{sq}} \supset \mathtt{safe} \wedge \bigcirc(\mathtt{safe}\,\mathcal{U}\,p_{\mathsf{gl}}) \mathbin{@} t \rangle\!\rangle_{\mathtt{eval\_next}\langle oe\rangle}$$

This reconstruction scope is a specialization of the strict evaluation judgment of the program logic (see Section 6.3.2).

$\mathtt{eval}\langle oe\rangle$ indicates that the proof outline *oe* contains sufficient information to reconstruct a proof that shows safety until the goal p_{gl}, after starting in state $e_{\mathtt{sq}}$:

$$\langle\!\langle \mathtt{sq} = e_{\mathtt{sq}} \supset \mathtt{safe}\,\mathcal{U}\,p_{\mathsf{gl}} \mathbin{@} t \rangle\!\rangle_{\mathtt{eval}\langle oe\rangle}$$

This reconstruction scope is a specialization of the standard evaluation judgment of the program logic (see Section 6.3.3).

$\mathtt{check}\langle ock\rangle$ indicates that the proof outline *ock* contains sufficient information to reconstruct a proof of the instantiated specification p_{uz}:

$$\langle\!\langle p_{\mathsf{uz}} \mathbin{@} t \rangle\!\rangle_{\mathtt{check}\langle ock\rangle}$$

$\mathtt{checku}\langle ock\rangle$ indicates that the proof outline *ock* contains sufficient information to reconstruct a proof of the instantiated specification p_{u}:

$$\langle\!\langle p_{\mathsf{u}} \mathbin{@} t \rangle\!\rangle_{\mathtt{checku}\langle ock\rangle}$$

$\mathtt{vc}\langle ovc\rangle$ indicates that the proof outline *ovc* contains sufficient information to reconstruct a proof of the proposition p_{vc}:

$$\langle\!\langle p_{\mathsf{vc}} \mathbin{@} t \rangle\!\rangle_{\mathtt{vc}\langle ovc\rangle}$$

`caller` indicates that a safety proof for a procedure with a given specification has been established:

$$\langle\!\langle \Box(\forall x_{\mathsf{sq}_0}\!:\!\mathsf{ri}.\ \forall x_{\mathsf{sq}}\!:\!\mathsf{fl}.\ \mathtt{sq} = x_{\mathsf{sq}_0} \supset \Box(\mathtt{sq} = x_{\mathsf{sq}}) \supset p_{\mathsf{p}} \supset \mathtt{safe}\,\mathcal{U}\,p_{\mathsf{q}}) \mathbin{@} 0 \rangle\!\rangle_{\mathtt{caller}}$$
$$\text{where } p_{\mathsf{p}} \overset{\mathsf{def}}{\equiv} \mathtt{pc} = e_{\mathtt{pc}} \wedge \mathtt{gsp} = e_{\mathsf{sp}} \wedge p_{\mathsf{puz}}$$
$$\text{and } p_{\mathsf{q}} \overset{\mathsf{def}}{\equiv} \mathtt{pc} = e'_{\mathtt{pc}} \wedge p_{\mathsf{quz}}$$

This reconstruction scope is a specialization of procedure elimination rule of the program logic (see Section 6.3.4). Note that there are no general rules for this reconstruction scope. The safety of each procedure is established individually by the code producer.

`checkz` indicates that a proof of the preservation specification $p_{\mathtt{z}}$ can be constructed automatically:

$$\langle\langle p_{\mathtt{z}} \mathbin{@} t\rangle\rangle_{\mathtt{checkz}}$$

`initz` indicates that a proof that the current state is $e''_{\mathtt{sq}}$ can be constructed automatically from the proposition $p_{\mathtt{z}}$, given that the current state is also $e'_{\mathtt{sq}}$:

$$\langle\langle e_{\mathtt{sq}} = e'_{\mathtt{sq}} \wedge p_{\mathtt{z}} \supset e_{\mathtt{sq}} = e''_{\mathtt{sq}} \mathbin{@} t\rangle\rangle_{\mathtt{initz}}$$

$e''_{\mathtt{sq}}$ is a syntactic transformation of $e'_{\mathtt{sq}}$ that contains explicit assignments for each location that is preserved by the specification $p_{\mathtt{z}}$. By transforming the symbolic state representation, I remove the need to reason about the preservation specification directly. All relevant properties are internalized in the state representation. For example, given the instantiated preservation elements

$$\mathtt{q\_ta}(e'_{\mathtt{sq}}) = \mathtt{q\_ta}(e_{\mathtt{sq}_0}) \wedge \top$$

the state $e'_{\mathtt{sq}}$ is transformed into the state

$$e''_{\mathtt{sq}} = \mathtt{upds}(\mathtt{q\_q}(e'_{\mathtt{sq}}), \overline{\mathtt{ta}}, \mathtt{q\_ta}(e_{\mathtt{sq}_0}))$$

Although $e'_{\mathtt{sq}}$ and $e''_{\mathtt{sq}}$ have identical valuations, the syntactic form of $e''_{\mathtt{sq}}$ enables an expression such as $\mathtt{q\_ta}(e''_{\mathtt{sq}})$ to be transformed into $\mathtt{q\_ta}(e_{\mathtt{sq}_0})$ by an automatic rewriting strategy based on the McCarthy rules.

The following reconstruction scopes imply that a proof of the proposition p' can be constructed automatically from the proposition p via *and elimination*:

$$\langle\langle p \supset p' \mathbin{@} t\rangle\rangle_{\mathtt{ande1\_mem}}$$

$$\langle\langle p \supset p' \mathbin{@} t\rangle\rangle_{\mathtt{ande1\_ts}}$$

$$\langle\langle p \supset p' \mathbin{@} t\rangle\rangle_{\mathtt{ande4\_ts}}$$

The three distinct scopes identify various special cases of this strategy.

`ori` indicates that a proof of the proposition p' can be constructed automatically from the proposition p via *or introduction*:

$$\langle\langle p \supset p' \mathbin{@} t\rangle\rangle_{\mathtt{ori}}$$

`leq_ts` indicates that a proof that type environment $e_{\mathtt{jts}_1}$ is contained in type environment $e_{\mathtt{jts}_2}$ can be constructed automatically:

$$\langle\langle \mathtt{js\_leq}(e_{\mathtt{jts}_1}, e_{\mathtt{jts}_2}) \mathbin{@} t\rangle\rangle_{\mathtt{leq\_ts}}$$

`lt_ts` indicates that a proof that type environment $e_{\mathtt{jts}_1}$ is *strictly* contained in type environment $e_{\mathtt{jts}_2}$ can be constructed automatically:

$$\langle\langle \mathtt{js\_leq}(e_{\mathtt{jts}_1}, e_{\mathtt{jts}_2}) \mathbin{@} t\rangle\rangle_{\mathtt{lt\_ts}}$$

$\texttt{next}$ indicates that executing the instruction e_l in the symbolic state e_sq results in symbolic state e'_sq:

$$\langle\!\langle \texttt{q\_next}(e_\mathsf{sq}, e_\mathsf{l}) = e'_\mathsf{sq} \mathbin{@} t \rangle\!\rangle_\texttt{next}$$

This reconstruction scope is a specialization of the transition judgment of the program logic (see Section 6.3.1).

$\texttt{fetch}$ indicates that the instruction at the program counter of state e_sq is e_l:

$$\langle\!\langle \texttt{fetch}(\texttt{pm}, \texttt{q\_pc}(e_\mathsf{sq}), e_\mathsf{l}) \mathbin{@} t \rangle\!\rangle_\texttt{fetch}$$

This is primarily a "wrapper" judgment that avoids the need to explicitly deconstruct the current state to retrieve the program counter.

The various substitution scopes indicate that substituting e'_sq for e_sq in the proposition p results in the proposition p':

$$\begin{array}{l}
\langle\!\langle e_\mathsf{sq} = e'_\mathsf{sq} \supset ([e_\mathsf{sq}/x_\mathsf{sq}]\, p \equiv p') \mathbin{@} t \rangle\!\rangle_\texttt{sub0} \\
\langle\!\langle e_\mathsf{sq} = e'_\mathsf{sq} \supset ([e_\mathsf{sq}/x_\mathsf{sq}]\, p \equiv p') \mathbin{@} t \rangle\!\rangle_\texttt{sub} \\
\langle\!\langle e_\mathsf{sq} = e'_\mathsf{sq} \supset ([e_\mathsf{sq}/x_\mathsf{sq}]\, p \equiv p') \mathbin{@} t \rangle\!\rangle_\texttt{sub0u} \\
\langle\!\langle e_\mathsf{sq} = e'_\mathsf{sq} \supset ([e_\mathsf{sq}/x_\mathsf{sq}]\, p \equiv p') \mathbin{@} t \rangle\!\rangle_\texttt{subu} \\
\langle\!\langle e_\mathsf{sq} = e'_\mathsf{sq} \supset ([e_\mathsf{sq}/x_\mathsf{sq}]\, p \equiv p') \mathbin{@} t \rangle\!\rangle_\texttt{sub0z} \\
\langle\!\langle e_\mathsf{sq} = e'_\mathsf{sq} \supset ([e_\mathsf{sq}/x_\mathsf{sq}]\, p \equiv p') \mathbin{@} t \rangle\!\rangle_\texttt{subz} \\
\langle\!\langle e_\mathsf{sq} = e'_\mathsf{sq} \supset ([e_\mathsf{sq}/x_\mathsf{sq}]\, p \equiv p') \mathbin{@} t \rangle\!\rangle_\texttt{sub0u1} \\
\langle\!\langle e_\mathsf{sq} = e'_\mathsf{sq} \supset ([e_\mathsf{sq}/x_\mathsf{sq}]\, p \equiv p') \mathbin{@} t \rangle\!\rangle_\texttt{subu1} \\
\langle\!\langle e_\mathsf{sq} = e'_\mathsf{sq} \supset ([e_\mathsf{sq}/x_\mathsf{sq}]\, p \equiv p') \mathbin{@} t \rangle\!\rangle_\texttt{sub0z1} \\
\langle\!\langle e_\mathsf{sq} = e'_\mathsf{sq} \supset ([e_\mathsf{sq}/x_\mathsf{sq}]\, p \equiv p') \mathbin{@} t \rangle\!\rangle_\texttt{subz1}
\end{array}$$

The individual cases discriminate on whether p has the shape of a list of specification elements or preservation elements, *etc.*[1]

The next reconstruction scope implements a form of "shallow" normalization. $\texttt{normE1}$ indicates that a proof that expression e is equal to expression e' can be constructed automatically via rewriting:

$$\langle\!\langle e = e' \mathbin{@} t \rangle\!\rangle_\texttt{normE1}$$

Unlike full normalization, $\texttt{normE1}$ only normalizes the outermost function application of an expression—the rest of the expression is presumed to be in normal form already.

$\texttt{hyp\_cont}\langle e_\mathsf{nh}\rangle$ indicates that a proof of the specification p_uz is available as a continuation hypothesis tagged by index e_nh:

$$\langle\!\langle p_\mathsf{uz} \mathbin{@} t \rangle\!\rangle_{\texttt{hyp\_cont}\langle e_\mathsf{nh}\rangle}$$

The tag e_nh is essential to discriminate between many possible hypotheses in the current search context.

[1] Note that substitution operations are performed indirectly in these cases to simplify unification steps when the proof-reconstruction logic program is run. The proposition $[e'_\mathsf{sq}/x_\mathsf{sq}]\, p$ in particular tends to be so large that it causes a measurable slowdown when it is instantiated directly.

$\texttt{hyp\_mem}\langle e_{\mathsf{nm}}\rangle$ indicates that a proof that $e_{\mathtt{m}}$ is a valid memory in type environment $e_{\mathtt{jts}}$ is available as a memory hypothesis tagged by index e_{nm}:

$$\langle\!\langle \texttt{js\_mem}(e_{\mathtt{jts}}, e_{\mathtt{m}}) \mathbin{@} t \rangle\!\rangle_{\texttt{hyp\_mem}\langle e_{\mathsf{nm}}\rangle}$$

$\texttt{hyp\_pre}$ indicates that a proof of the specification p_{puz} is available as a precondition hypothesis:

$$\langle\!\langle p_{\mathsf{puz}} \mathbin{@} t \rangle\!\rangle_{\texttt{hyp\_pre}}$$

$\texttt{hyp\_cop}$ indicates that a proof that the conditional operator $e_{\mathtt{cop}}$ holds is available as a conditional-branch hypothesis:

$$\langle\!\langle \texttt{self}(e_{\mathtt{cop}}, e_{\mathtt{f}}) \neq 0 \mathbin{@} t \rangle\!\rangle_{\texttt{hyp\_cop}}$$

$\texttt{hyp\_lt\_ts}$ indicates that a proof that type environment $e_{\mathtt{jts}_1}$ is contained in type environment $e_{\mathtt{jts}_2}$ is available as a hypothesis:

$$\langle\!\langle \texttt{js\_leq}(e_{\mathtt{jts}_1}, e_{\mathtt{jts}_2}) \mathbin{@} t \rangle\!\rangle_{\texttt{hyp\_lt\_ts}}$$

$\texttt{goal}\langle e_{\mathsf{pc}}\rangle\langle p_{\mathsf{uz}}\rangle$ indicates that the goal specification for address e_{pc} is p_{uz}:

$$\langle\!\langle \top \mathbin{@} 0 \rangle\!\rangle_{\texttt{goal}\langle e_{\mathsf{pc}}\rangle\langle p_{\mathsf{uz}}\rangle}$$

$\texttt{pf}$ indicates that the judgment J is decidable in the current context:

$$\langle\!\langle J \rangle\!\rangle_{\texttt{pf}}$$

Note that some care must be exercised when using this reconstruction space. If J is not in fact decidable (or hypothetical in the current context), nontermination will result.

8.1.2 Hypothesis Rule

The conditional hypothesis rule hyp allows a conditional hypothesis to succeed if its premises ω_1 through ω_k can be established directly:

$$\frac{\Omega \rhd [t_1/y_1]\ \ldots [t_k/y_k]\,\omega_1 \quad \ldots \quad \Omega \rhd [t_1/y_1]\ \ldots [t_k/y_k]\,\omega_{k'}}{\Omega \rhd [t_1/y_1]\ \ldots [t_k/y_k]\,\omega'}\ \mathsf{hyp}$$

$$\text{where } \Omega \stackrel{\mathsf{def}}{=} \Omega_1,\ (\Pi y_1, \ldots, y_k.\omega_1, \ldots, \omega_{k'} \to \omega'),\ \Omega_2$$

The free variables in the goals $(y_1, \ldots, y_k)$ must be instantiated consistently. This rule was first introduced in Section 7.3.2.

8.1.3 Callee Rules

The callee rule incrementally refines the proof goal by introducing parameters for variables in a procedure specification. This rule is a complex specialization of the procedure introduction rule—refer to the LF implementation for its precise construction.

8.1.4 Evaluation Rules

$$\boxed{\langle\!\langle \mathtt{sq} = e_{\mathtt{sq}} \supset \mathtt{safe}\,\mathcal{U}\, p_{\mathsf{gl}} \mathbin{@} t \rangle\!\rangle_{\mathtt{eval}\langle oe \rangle}}$$

The evaluation rules show that safety property holds from some current state until some goal proposition holds. These rules are based on the derived program-logic rules in Section 6.4.3 and Section 6.4.4. I only sample three rules here to illustrate how the program logic is refined into a logic program—the other cases follow a similar pattern. Additionally, I only show rules in the `eval` scope—the rules in the `eval_next` scope are very similar.

In each, case the proof outline indicates which rule should be applied. For example, $\mathtt{eval\_tail}\langle ock \rangle$ is associated with $\mathsf{eval\_tail}$. The logic-program rules are encoded in such a way as to make only one rule applicable for a given proof outline (see Section 8.2.1.

$\mathsf{eval\_tail}$ establishes the goal proposition p_{gl} immediately by showing that some component specification p_{uz} holds for the current state:

$$\frac{\begin{array}{l}\Omega \rhd \langle\!\langle \mathtt{q\_pc}(e_{\mathtt{sq}}) = e_{\mathtt{pc}} \mathbin{@} t_1 \rangle\!\rangle_{\mathtt{normE1}} \\ \Omega \rhd \langle\!\langle \top \mathbin{@} 0 \rangle\!\rangle_{\mathtt{goal}\langle e_{\mathtt{pc}}\rangle\langle p_{\mathsf{uz}}\rangle} \\ \Omega \rhd \langle\!\langle a_{\mathtt{sq}} = e_{\mathtt{sq}} \supset ([a_{\mathtt{sq}}/x_{\mathtt{sq}}]\, p_{\mathsf{uz}} \equiv p'_{\mathsf{uz}}) \mathbin{@} t_1 \rangle\!\rangle_{\mathtt{sub}} \\ \Omega \rhd \langle\!\langle p'_{\mathsf{uz}} \mathbin{@} t \rangle\!\rangle_{\mathtt{check}\langle ock \rangle} \\ \Omega \rhd \langle\!\langle p_{\mathsf{gl}} : \mathsf{lo}\ (a_{\mathtt{sq}}) \rangle\!\rangle_{\mathtt{pf}} \\ \Omega \rhd \langle\!\langle \mathtt{pc} = e_{\mathtt{pc}} \wedge [\mathtt{sq}/x_{\mathtt{sq}}]\, p_{\mathsf{uz}} \supset p_{\mathsf{gl}} \mathbin{@} t \rangle\!\rangle_{\mathtt{ori}}\end{array}}{\Omega \rhd \langle\!\langle \mathtt{sq} = e_{\mathtt{sq}} \supset \mathtt{safe}\,\mathcal{U}\, p_{\mathsf{gl}} \mathbin{@} t \rangle\!\rangle_{\mathtt{eval}\langle \mathtt{eval\_tail}\langle ock \rangle\rangle}}\ \mathsf{eval\_tail}^{a_{\mathtt{sq}}, t_1}$$

The second premise locates the component specification that is associated with the current program counter. Goal hypotheses in this form are introduced into the current context by the loop and "callee" rules. The fourth premise verifies that the specification does indeed hold for this state, whereas the last premise verifies that the specification is one element of the disjunctive goal proposition.

$\mathsf{eval\_unsafe}$ establishes safety over an interval of time by incrementally showing that a single potentially *unsafe* instruction is safe to execute:

$$\frac{\begin{array}{l}\Omega \rhd \langle\!\langle \mathtt{fetch}(\mathtt{pm}, \mathtt{q\_pc}(e_{\mathtt{sq}}), e_{\mathsf{I}}) \mathbin{@} t \rangle\!\rangle_{\mathtt{fetch}} \\ \Omega \rhd \langle\!\langle \mathtt{safe\_inst}(e_{\mathtt{sq}}, e_{\mathsf{I}}) \Longrightarrow^{**} p_{\mathsf{vc}} \rangle\!\rangle_{\mathtt{pf}} \\ \Omega \rhd \langle\!\langle \mathtt{q\_next}(e_{\mathtt{sq}}, e_{\mathsf{I}}) = e'_{\mathtt{sq}} \mathbin{@} t_1 \rangle\!\rangle_{\mathtt{next}} \\ \Omega \rhd \langle\!\langle e_{\mathsf{I}} : \mathsf{ri} \rangle\!\rangle_{\mathtt{pf}} \quad \Omega \rhd \langle\!\langle \mathtt{ri}(e_{\mathtt{sq}}) \mathbin{@} 0 \rangle\!\rangle_{\mathtt{pf}} \\ \Omega \rhd \langle\!\langle p_{\mathsf{vc}} \mathbin{@} t \rangle\!\rangle_{\mathtt{vc}\langle ovc \rangle} \\ \Omega, \langle\!\langle \mathtt{ri}(e'_{\mathtt{sq}}) \mathbin{@} 0 \rangle\!\rangle_{\mathtt{pf}} \rhd \langle\!\langle \mathtt{sq} = e'_{\mathtt{sq}} \supset \mathtt{safe}\,\mathcal{U}\, p_{\mathsf{gl}} \mathbin{@} t+1 \rangle\!\rangle_{\mathtt{eval}\langle oe \rangle}\end{array}}{\Omega \rhd \langle\!\langle \mathtt{sq} = e_{\mathtt{sq}} \supset \mathtt{safe}\,\mathcal{U}\, p_{\mathsf{gl}} \mathbin{@} t \rangle\!\rangle_{\mathtt{eval}\langle \mathtt{eval\_unsafe}\langle ovc \rangle\langle oe \rangle\rangle}}\ \mathsf{eval\_unsafe}^{t_1}$$

The first premise retrieves the current instruction e_{I} from the program according to the program counter of the current state $e_{\mathtt{sq}}$. The second premise rewrites the instruction-level safety condition to a single proof obligation p_{vc}. This is the proof obligation that would be emitted by a symbolic evaluator for e_{I}. The third premise constructs a symbolic representation of the next state $e'_{\mathtt{sq}}$ from the current state and instruction. The fourth and fifth premises establish the rigidity of the current state and instruction. These premises together with the third premise imply that $e'_{\mathtt{sq}}$ is rigid, which is needed to continue evaluation. The sixth premise reconstructs a proof of the safety obligation p_{vc} from the proof outline ovc. Finally, the last premise continues evaluation at the next time instant from the new state $e'_{\mathtt{sq}}$ using the rest of the proof outline oe.

eval_split establishes safety over an interval of time by incrementally showing that a single branch instruction is safe to execute:

$$\frac{\begin{array}{l}\Omega \rhd \langle\!\langle \mathtt{fetch}(\mathtt{pm}, e_{\mathtt{pc}}, f_{\mathtt{j}}(e_{\mathsf{i}}, e_{\mathtt{cop}_1}, e_{\mathsf{n}})) \mathbin{@} t\rangle\!\rangle_{\mathtt{pf}} \\ \Omega \rhd \langle\!\langle \mathtt{safe\_inst}(e_{\mathtt{sq}}, f_{\mathtt{j}}(e_{\mathsf{i}}, e_{\mathtt{cop}_1}, e_{\mathsf{n}})) \Longrightarrow^{**} \top\rangle\!\rangle_{\mathtt{pf}} \\ \Omega \rhd \langle\!\langle \mathtt{addw}(e_{\mathtt{pc}}, e_{\mathsf{i}}) \Longrightarrow^{**} e'_{\mathtt{pc}_2}\rangle\!\rangle_{\mathtt{pf}} \quad \Omega \rhd \langle\!\langle \mathtt{addw}(e'_{\mathtt{pc}_2}, e_{\mathsf{n}}) \Longrightarrow^{**} e'_{\mathtt{pc}_1}\rangle\!\rangle_{\mathtt{pf}} \\ \Omega \rhd \langle\!\langle \mathtt{not}(e_{\mathtt{cop}_1}) \Longrightarrow^{**} e_{\mathtt{cop}_2}\rangle\!\rangle_{\mathtt{pf}} \\ \Omega \rhd \langle\!\langle f_{\mathtt{j}}(e_{\mathsf{i}}, e_{\mathtt{cop}_1}, e_{\mathsf{n}}) : \mathsf{ri}\rangle\!\rangle_{\mathtt{pf}} \quad \Omega \rhd \langle\!\langle \mathtt{ri}(e_{\mathtt{sq}}) \mathbin{@} 0\rangle\!\rangle_{\mathtt{pf}} \\ \Omega'_1 \rhd \langle\!\langle \mathtt{sq} = e'_{\mathtt{sq}_1} \supset \mathtt{safe}\, \mathcal{U}\, p_{\mathsf{gl}} \mathbin{@} t+1\rangle\!\rangle_{\mathtt{eval}\langle oe_1\rangle} \\ \Omega'_2 \rhd \langle\!\langle \mathtt{sq} = e'_{\mathtt{sq}_2} \supset \mathtt{safe}\, \mathcal{U}\, p_{\mathsf{gl}} \mathbin{@} t+1\rangle\!\rangle_{\mathtt{eval}\langle oe_2\rangle}\end{array}}{\Omega \rhd \langle\!\langle \mathtt{sq} = e_{\mathtt{sq}} \supset \mathtt{safe}\, \mathcal{U}\, p_{\mathsf{gl}} \mathbin{@} t\rangle\!\rangle_{\mathtt{eval}\langle \mathtt{eval\_split}\langle oe_1\rangle\langle oe_2\rangle\rangle}}\ \mathsf{eval\_split}$$

$$\begin{aligned}\text{where } \Omega'_1 &\stackrel{\mathsf{def}}{=} \Omega,\ (\Pi y_1.\langle\!\langle \mathtt{self}(e_{\mathtt{cop}_1}, e_{\mathtt{f}}) \neq 0 \mathbin{@} y_1\rangle\!\rangle_{\mathtt{hyp\_cop}}),\ \langle\!\langle \mathtt{ri}(e'_{\mathtt{sq}_1}) \mathbin{@} 0\rangle\!\rangle_{\mathtt{pf}} \\ \text{and } \Omega'_2 &\stackrel{\mathsf{def}}{=} \Omega,\ (\Pi y_1.\langle\!\langle \mathtt{self}(e_{\mathtt{cop}_2}, e_{\mathtt{f}}) \neq 0 \mathbin{@} y_1\rangle\!\rangle_{\mathtt{hyp\_cop}}),\ \langle\!\langle \mathtt{ri}(e'_{\mathtt{sq}_2}) \mathbin{@} 0\rangle\!\rangle_{\mathtt{pf}} \\ \text{and } e_{\mathtt{sq}} &\stackrel{\mathsf{def}}{=} \mathtt{q\_mk}(e_{\mathtt{pc}}, e_{\mathtt{f}}, e_{\mathtt{g}}, e_{\mathtt{s}}, e_{\mathtt{m}}, e_{\mathtt{q}}) \\ \text{and } e'_{\mathtt{sq}_1} &\stackrel{\mathsf{def}}{=} \mathtt{q\_mk}(e'_{\mathtt{pc}_1}, e_{\mathtt{f}}, e_{\mathtt{g}}, e_{\mathtt{s}}, e_{\mathtt{m}}, e_{\mathtt{q}}) \\ \text{and } e'_{\mathtt{sq}_2} &\stackrel{\mathsf{def}}{=} \mathtt{q\_mk}(e'_{\mathtt{pc}_2}, e_{\mathtt{f}}, e_{\mathtt{g}}, e_{\mathtt{s}}, e_{\mathtt{m}}, e_{\mathtt{q}})\end{aligned}$$

This rule follows an approach that resembles eval_unsafe, except that the rule is specialized to the jump instruction (the only branch instruction in the instruction set), and the rule must consider the two possible outcomes of the branch. Because branch instructions are always safe, the instruction-level safety condition is rewritten to "true." Other premises compute the next symbolic state according to the semantics of the jump instruction. The last two premises continue evaluation at the next time instant according to the two possible outcomes of the branch. Along each possible outcome, the result of the conditional governing the branch is accumulated into the hypothetical context.

8.1.5 Checking Rule

$\boxed{\langle\!\langle p_{\mathsf{uz}} \mathbin{@} t\rangle\!\rangle_{\mathtt{check}\langle ock\rangle}}$

The checking rule establishes a pair consisting of a list of specification elements and a list of preservation elements:

$$\frac{\Omega \rhd \langle\!\langle p_{\mathsf{u}} \mathbin{@} t\rangle\!\rangle_{\mathtt{checku}\langle ock\rangle} \quad \Omega \rhd \langle\!\langle p_{\mathsf{z}} \mathbin{@} t\rangle\!\rangle_{\mathtt{checkz}}}{\Omega \rhd \langle\!\langle p_{\mathsf{u}} \wedge p_{\mathsf{z}} \mathbin{@} t\rangle\!\rangle_{\mathtt{check}\langle ock\rangle}}\ \mathsf{check}$$

8.1.6 Specification Rules

$$\boxed{\langle\!\langle p_{\mathsf{u}} \mathbin{@} t \rangle\!\rangle_{\mathtt{checku}\langle ock\rangle}}$$

The specification rules show that some list of specification elements holds. The list p_{u} is represented as a right-associative conjunction of specification elements $p_1, \ldots, p_k$: $p_1 \wedge (\ldots \wedge (p_k \wedge \top))$. The rules in this section are specializations of program-logic support rules in Section 6.5.1.

checku_z is the base case:

$$\frac{}{\Omega \rhd \langle\!\langle \top \mathbin{@} t \rangle\!\rangle_{\mathtt{checku}\langle\mathtt{ck\_z}\rangle}}\ \mathsf{checku\_z}$$

checku_u establishes a single element of the specification list:

$$\frac{\Omega \rhd \langle\!\langle p_{\mathsf{u}} \mathbin{@} t \rangle\!\rangle_{\mathtt{vc}\langle ock_1\rangle} \quad \Omega \rhd \langle\!\langle p'_{\mathsf{u}} \mathbin{@} t \rangle\!\rangle_{\mathtt{checku}\langle ock_2\rangle}}{\Omega \rhd \langle\!\langle p_{\mathsf{u}} \wedge p'_{\mathsf{u}} \mathbin{@} t \rangle\!\rangle_{\mathtt{checku}\langle\mathtt{ck\_u}\langle ock_1\rangle\langle ock_2\rangle\rangle}}\ \mathsf{checku\_u}$$

The first premise reconstructs the VC proof from the proof outline ock_1. The second premise checks the remainder of the specification p'_{u} by passing in the remainder of the proof outline ock_2.

8.1.7 VC Rules

$$\boxed{\langle\!\langle p_{\mathsf{vc}} \mathbin{@} t \rangle\!\rangle_{\mathtt{vc}\langle ovc\rangle}}$$

The VC rules reconstruct a proof of a VC using a proof outline. These rules are straightforward adaptations of derived inference rules from Section 6.5.3, so I will only show three examples—the other cases follow a similar approach.

and_el and and_er eliminate a conjunction on the left and right, respectively:

$$\frac{\Omega \rhd \langle\!\langle p_{\mathsf{vc}_1} \wedge p_{\mathsf{vc}_2} \mathbin{@} t \rangle\!\rangle_{\mathtt{vc}\langle ovc\rangle}}{\Omega \rhd \langle\!\langle p_{\mathsf{vc}_1} \mathbin{@} t \rangle\!\rangle_{\mathtt{vc}\langle\mathtt{vc\_and\_el}\langle ovc\rangle\rangle}}\ \mathsf{and\_el} \qquad \frac{\Omega \rhd \langle\!\langle p_{\mathsf{vc}_1} \wedge p_{\mathsf{vc}_2} \mathbin{@} t \rangle\!\rangle_{\mathtt{vc}\langle ovc\rangle}}{\Omega \rhd \langle\!\langle p_{\mathsf{vc}_2} \mathbin{@} t \rangle\!\rangle_{\mathtt{vc}\langle\mathtt{vc\_and\_er}\langle ovc\rangle\rangle}}\ \mathsf{and\_er}$$

The premise uses the remainder of the proof outline ovc to reconstruct a proof of the conjunction itself.

normE_e normalizes an expression in the rewriting system:

$$\frac{\Omega \rhd \langle\!\langle e \Longrightarrow^* e' \rangle\!\rangle_{\mathtt{pf}}}{\Omega \rhd \langle\!\langle e = e' \mathbin{@} t \rangle\!\rangle_{\mathtt{vc}\langle\mathtt{vc\_normE\_e}\rangle}}\ \mathsf{normE\_e}$$

e is the expression to normalize, and e' is its normal form. The expression e is constrained sufficiently in this system so that its value need not be stored in the proof outline. e' is derived directly from e, so it also need not be stored in the proof outline. The premise of this rule searches for a rewriting derivation using the standard rewriting judgment. Because this system is decidable, the proof is reconstructed automatically.

8.1.8 Preservation Rules

$$\boxed{\langle\!\langle p_{\mathsf{z}} \mathbin{@} t \rangle\!\rangle_{\mathtt{checkz}}}$$

The preservation rules show that some list of preservation specifications $p_1, \ldots, p_k$ holds. The list p_{u} is represented as a right-associative conjunction: $p_1 \wedge (\ldots \wedge (p_k \wedge \top))$. Each preservation specification is an equality that asserts that some location is unchanged throughout a procedure or loop body. The rules in this section are specializations of program-logic support rules in Section 6.5.2.

checkz_none is the base case:

$$\frac{}{\Omega \rhd \langle\!\langle \top \mathbin{@} t \rangle\!\rangle_{\texttt{checkz}}}\ \textsf{checkz\_none}$$

checkz_eq establishes a single equality specification at the head of the specification list:

$$\frac{\Omega \rhd \langle\!\langle p'_{\mathsf{z}} \mathbin{@} t \rangle\!\rangle_{\texttt{checkz}}}{\Omega \rhd \langle\!\langle e = e \wedge p'_{\mathsf{z}} \mathbin{@} t \rangle\!\rangle_{\texttt{checkz}}}\ \textsf{checkz\_eq}$$

The premise establishes the rest of the preservation specification p'_{z}.

checkz_leq_ts establishes a single type-environment containment at the head of the specification list:

$$\frac{\Omega \rhd \langle\!\langle \texttt{js\_leq}(e_{\texttt{jts}_1}, e_{\texttt{jts}_2}) \mathbin{@} t \rangle\!\rangle_{\texttt{leq\_ts}} \quad \Omega \rhd \langle\!\langle p'_{\mathsf{z}} \mathbin{@} t \rangle\!\rangle_{\texttt{checkz}}}{\Omega \rhd \langle\!\langle \texttt{js\_leq}(e_{\texttt{jts}_1}, e_{\texttt{jts}_2}) \wedge p'_{\mathsf{z}} \mathbin{@} t \rangle\!\rangle_{\texttt{checkz}}}\ \textsf{checkz\_leq\_ts}$$

The first premise establishes the containment. The second premise establishes the rest of the preservation specification p'_{z}.

8.1.9 Initialization Rules $\boxed{\langle\!\langle e_{\texttt{sq}} = e'_{\texttt{sq}} \wedge p_{\mathsf{z}} \supset e_{\texttt{sq}} = e''_{\texttt{sq}} \mathbin{@} t \rangle\!\rangle_{\texttt{initz}}}$

The purpose of the initialization rules is to use a list of preservation specifications $p_1 \wedge (\ldots \wedge (p_k \wedge \top))$ to transform a symbolic state representation so that the equalities are internal properties of the symbolic state. This representation makes it possible to later establish many properties by rewriting rather than by applying a substitution rule (Section 8.1.1 contains an example of how a state is initialized). I only show two sample initialization rules here—the remaining cases are similar to the second example.

initz_none is the base case:

$$\frac{}{\Omega \rhd \langle\!\langle e_{\texttt{sq}} = e'_{\texttt{sq}} \wedge \top \supset e_{\texttt{sq}} = e'_{\texttt{sq}} \mathbin{@} t \rangle\!\rangle_{\texttt{initz}}}\ \textsf{initz\_none}$$

No specifications are left to process, so the symbolic state is not transformed.

initz_selg initializes a single machine register e_{r}:

$$\frac{\Omega \rhd \langle\!\langle e_{\texttt{sq}} = \texttt{q\_mk}(e'_{\texttt{pc}}, e'_{\texttt{f}}, e''_{\texttt{g}}, e'_{\texttt{s}}, e'_{\texttt{m}}, e'_{\texttt{q}}) \wedge p'_{\mathsf{z}} \supset e_{\texttt{sq}} = e''_{\texttt{sq}} \mathbin{@} t \rangle\!\rangle_{\texttt{initz}}}{\Omega \rhd \langle\!\langle e_{\texttt{sq}} = \texttt{q\_mk}(e'_{\texttt{pc}}, e'_{\texttt{f}}, e'_{\texttt{g}}, e'_{\texttt{s}}, e'_{\texttt{m}}, e'_{\texttt{q}}) \wedge p_{\mathsf{z}} \supset e_{\texttt{sq}} = e''_{\texttt{sq}} \mathbin{@} t \rangle\!\rangle_{\texttt{initz}}}\ \textsf{initz\_selg}$$

$$\text{where } p_{\mathsf{z}} \overset{\mathsf{def}}{\equiv} \texttt{selg}(\texttt{q\_g}(e_{\texttt{sq}}), e_{\mathsf{r}}) = e_{\mathsf{n}} \wedge p'_{\mathsf{z}}$$
$$\text{and } e''_{\texttt{g}} \overset{\mathsf{def}}{=} \texttt{updg}(e'_{\texttt{g}}, e_{\mathsf{r}}, e_{\mathsf{n}})$$

The register file of the transformed state has an explicit assignment for the value of e_{r}: $\texttt{updg}(e'_{\texttt{g}}, e_{\mathsf{r}}, e_{\mathsf{n}})$. The premise initializes the rest of the state using the remainder of the preservation specification p'_{z}.

8.1.10 Or-Introduction Rules

$\boxed{\langle\!\langle p \supset p' \mathbin{@} t \rangle\!\rangle_{\texttt{ori}}}$

The *or introduction* reconstruction scope uses a given proposition p_j to establish one case of a disjunctive list of propositions. By establishing one case p_j, the entire list $p_1 \vee (\ldots \vee (p_k \vee \bot))$ is also established. The strategy of the or-introduction rules is to simply try each case in order:

$$\frac{}{\Omega \rhd \langle\!\langle p \supset p \vee p' \mathbin{@} t \rangle\!\rangle_{\texttt{ori}}}\ \textsf{ori\_left} \qquad \frac{\Omega \rhd \langle\!\langle p \supset p'' \mathbin{@} t \rangle\!\rangle_{\texttt{ori}}}{\Omega \rhd \langle\!\langle p \supset p' \vee p'' \mathbin{@} t \rangle\!\rangle_{\texttt{ori}}}\ \textsf{ori\_right}$$

The *and elimination* reconstruction scopes similarly show that some target proposition p_j follows from a conjunctive list of propositions $p_1 \wedge (\ldots \wedge (p_k \wedge \top))$. The and elimination rules follow a more ad-hoc strategy based on specialized cases, so I do not show them here. An ad-hoc strategy is more efficient here, because the general case of and elimination is not needed by my implementation.

8.1.11 Type-State Rules

$\boxed{\langle\!\langle \texttt{js\_leq}(e_{\texttt{jts}_1}, e_{\texttt{jts}_2}) \mathbin{@} t \rangle\!\rangle_{\texttt{leq\_ts}}}$

The type environment rules automatically show containment for type environments. Because the type environment changes during storage allocation, some concept of containment is need to show that typing ascriptions that were in effect before the allocation are still valid in the new type environment. Each storage allocation procedure provided by the run-time system has a postcondition that asserts that the type environment in effect at the time of the call is contained in the type environment in effect after the call. Each of these specifications becomes part of the search context when the procedure-call rule is applied. The role of the type-environment rules in this section is to incrementally compose these hypothetical specifications to show that some arbitrarily old type environment is contained in the current type environment by transitive closure.

There are two distinct reconstruction scopes for containment: the *non-strict* scope `leq_ts` admits reflexivity, whereas the *strict* scope `lt_ts` does not. Reflexivity is needed because in many cases there has been no storage allocation since a typing ascription was established, and thus the type environment has not changed.

The rules for non-strict containment have a case for reflexivity:

$$\frac{}{\Omega \rhd \langle\!\langle \texttt{js\_leq}(e_{\texttt{jts}}, e_{\texttt{jts}}) \mathbin{@} t \rangle\!\rangle_{\texttt{leq\_ts}}}\ \textsf{leq\_ts\_ref}$$

Otherwise, strict containment must be established:

$$\frac{\Omega \rhd \langle\!\langle \texttt{js\_leq}(e_{\texttt{jts}_1}, e_{\texttt{jts}_2}) \mathbin{@} t \rangle\!\rangle_{\texttt{lt\_ts}}}{\Omega \rhd \langle\!\langle \texttt{js\_leq}(e_{\texttt{jts}_1}, e_{\texttt{jts}_2}) \mathbin{@} t \rangle\!\rangle_{\texttt{leq\_ts}}}\ \textsf{leq\_ts\_lt}$$

Strict containment can be shown by one of two cases. Either the containment was established directly by a procedure postcondition, and the containment is now a hypothesis:

$$\frac{\Omega \rhd \langle\!\langle \texttt{js\_leq}(e_{\texttt{jts}_1}, e_{\texttt{jts}_2}) \mathbin{@} t \rangle\!\rangle_{\texttt{hyp\_lt\_ts}}}{\Omega \rhd \langle\!\langle \texttt{js\_leq}(e_{\texttt{jts}_1}, e_{\texttt{jts}_2}) \mathbin{@} t \rangle\!\rangle_{\texttt{lt\_ts}}}\ \textsf{lt\_ts\_hyp}$$

Or there is some direct containment hypothesis available that is one step closer to the goal environment, and transitivity is used to get the rest of the way:

$$\frac{\Omega \rhd \langle\!\langle \mathtt{js\_leq}(e_{\mathtt{jts}_1}, e'_{\mathtt{jts}_1}) \mathbin{@} t \rangle\!\rangle_{\mathtt{hyp\_lt\_ts}} \quad \Omega \rhd \langle\!\langle \mathtt{js\_leq}(e'_{\mathtt{jts}_1}, e_{\mathtt{jts}_2}) \mathbin{@} t \rangle\!\rangle_{\mathtt{lt\_ts}}}{\Omega \rhd \langle\!\langle \mathtt{js\_leq}(e_{\mathtt{jts}_1}, e_{\mathtt{jts}_2}) \mathbin{@} t \rangle\!\rangle_{\mathtt{lt\_ts}}}\ \mathsf{lt\_ts\_trans}$$

Note that some care must be taken here to avoid infinite loops when the rules are interpreted as a search strategy.

8.1.12 Next-State Rule $\boxed{\langle\!\langle \mathtt{q\_next}(e_{\mathtt{sq}}, e_{\mathsf{l}}) = e'_{\mathtt{sq}} \mathbin{@} t \rangle\!\rangle_{\mathtt{next}}}$

next is a composition of six individual rewriting rules:

$$\frac{\begin{array}{ll} \Omega \rhd \langle\!\langle \mathtt{nextpc}(e_{\mathtt{ss}}, e_{\mathsf{l}}) \Longrightarrow^{**} e'_{\mathtt{pc}} \rangle\!\rangle_{\mathtt{pf}} & \Omega \rhd \langle\!\langle \mathtt{nextf}(e_{\mathtt{ss}}, e_{\mathsf{l}}) \Longrightarrow^{**} e'_{\mathtt{f}} \rangle\!\rangle_{\mathtt{pf}} \\ \Omega \rhd \langle\!\langle \mathtt{nextg}(e_{\mathtt{ss}}, e_{\mathsf{l}}) \Longrightarrow^{**} e'_{\mathtt{g}} \rangle\!\rangle_{\mathtt{pf}} & \Omega \rhd \langle\!\langle \mathtt{nexts}(e_{\mathtt{ss}}, e_{\mathsf{l}}) \Longrightarrow^{**} e'_{\mathtt{s}} \rangle\!\rangle_{\mathtt{pf}} \\ \Omega \rhd \langle\!\langle \mathtt{nextm}(e_{\mathtt{ss}}, e_{\mathsf{l}}) \Longrightarrow^{**} e'_{\mathtt{m}} \rangle\!\rangle_{\mathtt{pf}} & \Omega \rhd \langle\!\langle \mathtt{nextq}(\mathtt{mkp}(e_{\mathtt{ss}}, e_{\mathtt{q}}), e_{\mathsf{l}}) \Longrightarrow^{**} e'_{\mathtt{q}} \rangle\!\rangle_{\mathtt{pf}} \end{array}}{\Omega \rhd \langle\!\langle \mathtt{q\_next}(\mathtt{mkp}(e_{\mathtt{ss}}, e_{\mathtt{q}}), e_{\mathsf{l}}) = \mathtt{q\_mk}(e'_{\mathtt{pc}}, e'_{\mathtt{f}}, e'_{\mathtt{g}}, e'_{\mathtt{s}}, e'_{\mathtt{m}}, e'_{\mathtt{q}}) \mathbin{@} t \rangle\!\rangle_{\mathtt{next}}}\ \mathsf{next}$$

It computes a symbolic representation of the next machine state given a symbolic representation of the current machine state.

8.1.13 Fetch Rule $\boxed{\langle\!\langle \mathtt{fetch}(\mathtt{pm}, \mathtt{q\_pc}(e_{\mathtt{sq}}), e_{\mathsf{l}}) \mathbin{@} t \rangle\!\rangle_{\mathtt{fetch}}}$

fetch is essentially a "wrapper" that retrieves the program counter from the current state and fetches the current instruction in one judgment:

$$\frac{\Omega \rhd \langle\!\langle \mathtt{fetch}(\mathtt{pm}, e_{\mathtt{pc}}, e_{\mathsf{l}}) \mathbin{@} t \rangle\!\rangle_{\mathtt{pf}}}{\Omega \rhd \langle\!\langle \mathtt{fetch}(\mathtt{pm}, \mathtt{q\_pc}(\mathtt{q\_mk}(e_{\mathtt{pc}}, e_{\mathtt{f}}, e_{\mathtt{g}}, e_{\mathtt{s}}, e_{\mathtt{m}}, e_{\mathtt{q}})), e_{\mathsf{l}}) \mathbin{@} t \rangle\!\rangle_{\mathtt{fetch}}}\ \mathsf{fetch}$$

8.1.14 Proof-Embedding Rule $\boxed{\langle\!\langle J \rangle\!\rangle_{\mathtt{pf}}}$

The proof-embedding rule is used to trigger a search for a decidable (or hypothetical) judgment, independent of a particular reconstruction scope. To make any $\mathtt{pf}$ hypotheses of the search context available to the search, I define an operation $|\cdot|$ that maps a search context Ω to an ordinary context Γ:

$$\begin{aligned} |\cdot| &= \cdot \\ |\Omega,\, \omega| &= \begin{cases} |\Omega|,\ J & \text{if } \omega = \langle\!\langle J \rangle\!\rangle_{\mathtt{pf}} \\ |\Omega| & \text{otherwise} \end{cases} \end{aligned}$$

Any non-$\mathtt{pf}$ hypotheses are not relevant and are thus discarded.

Given this definition, the realization of the proof-embedding rule is straightforward:

$$\frac{|\Omega| \vdash J}{\Omega \rhd \langle\!\langle J \rangle\!\rangle_{\mathtt{pf}}}\ \mathsf{embed}$$

8.2 Proof Encoding

In this section, I give an overview of how proofs are encoded. In Section 8.2.1, I show how reconstruction scopes and proof outlines are derived in the standard formal system. In Section 8.2.2, I show how proof outlines are encoded as binary certificates.

8.2.1 Reconstruction-Scope Encoding

The general strategy for deriving reconstruction scopes is based on leveraging the restricted formal system that governs the judgment

$$\Gamma \vdash p \mathrel{\underline{@}} t$$

This judgment is defined by the following inference rules (from Section 3.3):

$$\frac{\Gamma \vdash p \mathrel{@} t}{\Gamma \vdash p \mathrel{\underline{@}} t}\ \underline{@}\mathsf{i} \qquad \frac{\Gamma \vdash p \mathrel{\underline{@}} t}{\Gamma \vdash p \mathrel{@} t}\ \underline{@}\mathsf{e}$$

which enable the code producer to determine precisely when the judgment holds using derived rules. It is crucial to note that the introduction rule $\underline{@}\mathsf{i}$ is given special treatment by the logic interpreter. This rule is ignored during proof search, and only the rules *derived* from this rule are considered. If not given this treatment, the $\underline{@}$ judgment would be exactly equivalent to $@$, and would thus be much more difficult to search over.[2]

The reconstruction-scope formal system of Section 8.1 can now be encoded in terms of ordinary propositions under the $\underline{@}$ judgment. As long as each reconstruction scope is isolated in a distinct "partition" of the total set of propositions, then the partition can be treated just like a new judgment for the purpose of logic programming. Any such partition of the propositions will work, but in my derived system, I select an arbitrary number to tag each reconstruction scope. For example, the judgment

$$\langle\!\langle p \supset p' \mathrel{@} t \rangle\!\rangle_{\mathtt{ori}}$$

is encoded as

$$6 = 6 \supset p \supset p' \mathrel{\underline{@}} t$$

whereas

$$\langle\!\langle e = e' \mathrel{@} t \rangle\!\rangle_{\mathtt{normE1}}$$

is encoded as

$$32 = 32 \supset e = e' \mathrel{\underline{@}} t$$

The trivial equalities $6 = 6$ and $32 = 32$ only serve to distinguish the reconstruction scopes in which the judgments are derived. Any such tokens are suitable for distinguishing reconstruction scopes, but the tokens themselves must also be *derivable* if they are to be used in derived rules. For example, the inference rule

[2]Some sort of general-purpose theorem-proving approach would be needed to develop an effective search strategy.

$$\frac{\Omega \rhd \langle\!\langle p \supset p'' \mathbin{@} t \rangle\!\rangle_{\texttt{ori}}}{\Omega \rhd \langle\!\langle p \supset p' \vee p'' \mathbin{@} t \rangle\!\rangle_{\texttt{ori}}}\ \textsf{ori\_right}$$

is derived as

$$\frac{\Gamma \vdash 6 = 6 \supset p \supset p'' \mathbin{\underline{@}} t}{\Gamma \vdash 6 = 6 \supset p \supset p' \vee p'' \mathbin{\underline{@}} t}\ \textsf{ori\_right}'$$

In deriving this rule, one wants to be able to easily discharge the left-hand side of the implication $(6 = 6)$ in the premise to get at the real content of the judgment—the reconstruction scopes are just superficial tags.

Because judgments are distinguished internally, an ordinary context Γ can represent the search context Ω. Also, in my implementation, the `pf` reconstruction scope can be given a trivial encoding (*e.g.*, J is encoded as J), because the specific uses of `pf` never intersect the encodings of other reconstruction scopes. Thus, the context-conversion operation $|\cdot|$ from Section 8.1.14 is the identity under this encoding.

Proof outlines can also be encoded as (trivially true) propositions. For example, the proof outline

$$\texttt{eval\_tail}\langle\texttt{ck\_z}\rangle$$

becomes

$$2 = 2 \wedge \top$$

An encoding for proof outlines is needed because they are part of several reconstruction scopes, and same criteria apply to the selection of a "good" encoding.

I do not present the complete encoding here, because it is a rather superficial detail (many equivalent encodings will work), but see the LF implementation for details.

8.2.2 Binary Encoding

The code producer attaches a fixed *prelude* to the front of each safety proof. The bulk of the prelude is made up of definitions and derived rules, but it also contains a *binary decoding* that specifies how binary strings in the certificate are decoded into proof outlines. This decoding is developed by the code producer, and can be tuned to the particular proof outline representation that is derived in the rest of the prelude. Thus, the decoding yields many of the information-density benefits of an "oracle string" [NR01], but the oracle string interpreter need not be built into the logic-program interpreter.

Note that because the decoding operation can be performed in a separate step before the logic-program interpreter is run, the decoder need not be correct in order to preserve the integrity of the checker. The decoder can be treated as an untrusted component that "usually" produces a valid proof outline. If the proof outline happens to be invalid because of a bug in the decoder, then the logic-program interpreter will reject it just as if the code producer supplied an invalid proof. The decoder must still be *safe* in the sense that it must not compromise the integrity of the proof checker. This is one reason not to allow the code producer to supply an arbitrary decoding program, but see Necula and Schneck [NS03] for an exploration of such an approach.

The proof decoding is specified as a series of grammatical rules that determine how a binary string is decoded into an LF term. Because the proof decoder is essentially an untrusted component, and because the decoding language operates at the level of LF terms, I do not develop the decoding language formally here.

Chapter 9

Experimental Results

In this chapter, I present the results of my experiments with compiling and checking various small Java programs with a prototype PCC implementation. The benchmark results show that proof sizes and proof checking times are both in linear proportion to the size of the corresponding program.

The benchmarks are small because the proof checking overhead is still substantial enough to make it impractical for larger programs. Additionally, because I only implement a core "C-like" subset of the Java language, I generally cannot use publicly available Java code for my experiments. For example, my implementation does not currently support jump tables (*i.e.*, switch statements), subtyping, virtual methods, exception handlers, or floating-point numbers.

However, the benchmarks do provide sufficient information for me to set a reasonable expectation as to the proof sizes and proof checking times for some larger programs. The benchmarks contain a mix of Java methods that implement standard computer science algorithms in an attempt to profile the code in an "ordinary" application. Because the proof of each method is checked independently, the benchmark results for a larger program that is made up of largely similar methods will be a simple multiple of the results presented here. Thus, it is reasonable to expect continued linear growth for "ordinary" larger programs—more precisely, those applications that are composed of additional methods that are not too different from the methods examined here.

9.1 Performance Analysis

In this section, I analyze the cost of my implementation in terms of proof size and proof checking overhead. These costs do not include the size of a fixed *prelude* that contains definitions, derived rules, and the binary encoding (see Chapter 8). Because the prelude is fixed by the compiler and the certification strategy, it is identical for all programs that are certified by the same implementation. Thus, there is some additional engineering flexibility in terms of how the prelude is transmitted to the code consumer. For example, it could be downloaded "on demand" over the Internet by the code consumer, cached locally, and only checked once, even though many

Program	Procedures	Source Lines	Object Size
Alloc	1	8	44
Binary Search	1	29	124
Bubble Sort	1	32	140
Checksum	1	13	64
Clone	1	14	104
Dec	1	13	16
Fact	1	13	28
Fib	1	22	56
Filter	1	8	24
Heap Sort	6	99	768
Huffman	5	102	1168
Loop	2	13	12
Matrix	10	91	1488
Matrix Multiply	2	21	292
Matrix Transpose	1	12	140
Merge Sort	3	76	648
Min	1	8	40
Negative Abs	1	8	40
Nop	1	7	8
Not	1	8	24
N Queens	2	52	604
Packet	1	8	64
Quicksort	1	53	328
Reverse	1	11	88
Swap	1	11	76

Table 9.1: Benchmark Programs

programs may depend on it. Because the prelude is a fixed overhead, I do not include it in the cost measurements of the benchmark programs. The measurements here only reflect costs that are proportional to the size of the program being certified.

I summarize the benchmark programs I used for my experiments in Table 9.1. See Appendix C for the complete source code for these programs.

In Figure 9.1, I plot proof size (in bytes) against code size (also in bytes). As is evident from the graph, proof size is roughly in linear proportion to code size. I expect this trend to continue for larger program sizes with a similar mix of methods. Because each method is verified independently, increasing the program size simply adds the size of the additional safety proofs to the total proof size. The effect of adding large and/or atypical methods to a program is less predictable. My approach to certification is based on the SpecialJ [CLN+00] symbolic evaluator, which has an exponential worst case. However, exponential proof sizes or checking times are rarely seen in practice, and techniques have been developed to mitigate this potential liability [FS01].

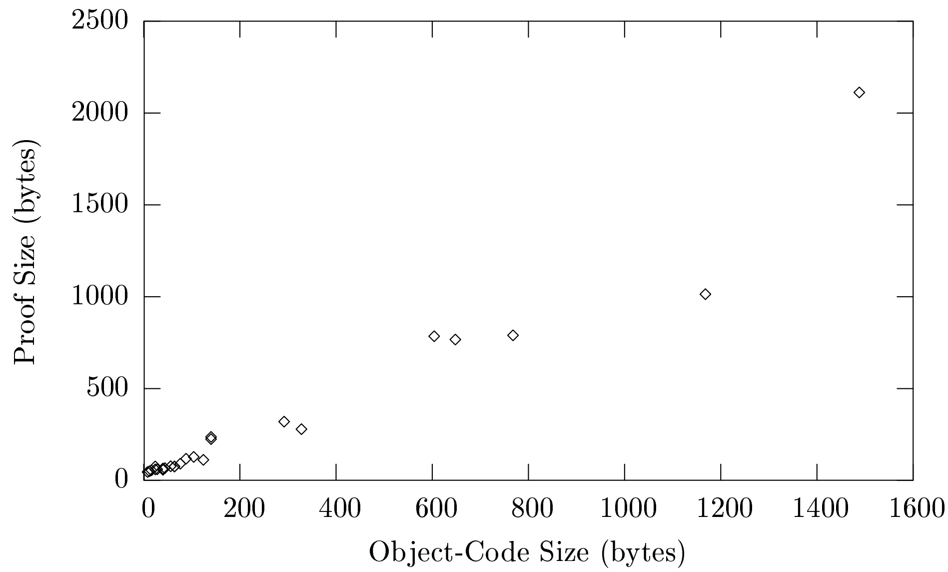

Figure 9.1: Proof Size

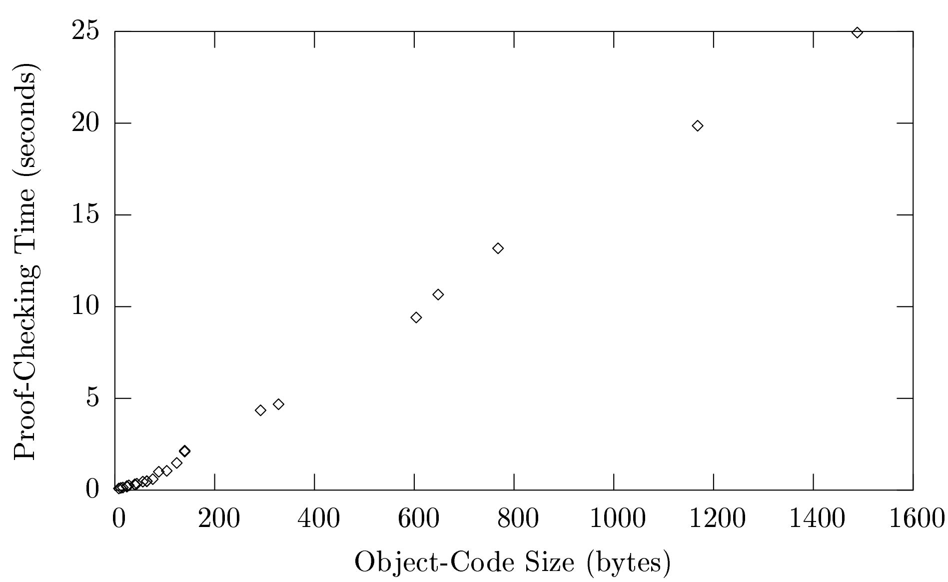

Figure 9.2: Proof-Checking Time

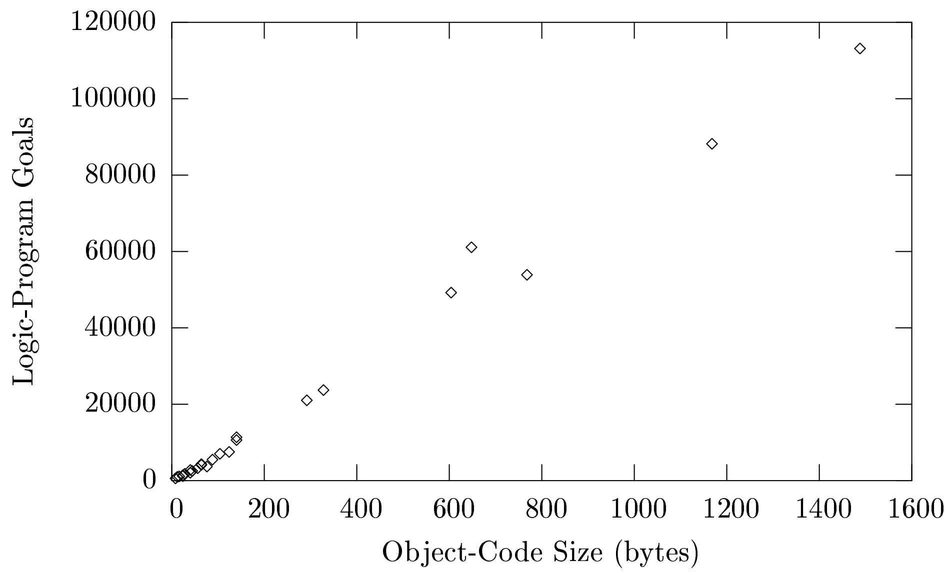

Figure 9.3: Logic-Program Goals

In Figure 9.2, I plot proof checking time (in seconds) against code size (in bytes). Proof checking times were measured on a 1.66 gigagertz AMD Athlon XP 2000+ PC with 512 megabytes of RAM[1] running under Debian GNU/Linux 3.0r2 with kernel 2.4.18. I implemented my own LF logic interpreter (see Section 9.2) for proof checking. The logic interpreter was compiled by The Objective Caml System release 3.04.

Proof-checking time appears to be linear in the size of the object code. Although this is a promising indicator, the total cost of proof checking is much higher than for a conventional PCC implementation. For example, SpecialJ takes at most 130 *milliseconds* to check any of the programs in my sample set. Currently, this is the most serious barrier to general practicality for my implementation.

In Figure 9.3, I plot the number of logic-program goals required to check a program against the code size of the program (in bytes). As is apparent from the graph, the number of goals is also a linear function of code size.

Finally, in Figure 9.4, I plot the time required to translate a SpecialJ safety proof into a fully-explicit safety proof.[2] The times here are relatively small when compared with proof-checking time—this suggests that my implementation would benefit by shifting additional work from the code consumer to the code producer. However, it is not presently clear how this can be accomplished without making the proof checker unacceptably complex or without making proofs unacceptably large. The apparent 2 second "start up" time is the time required to load the proof-

[1]Note that the logic interpreter itself never uses more than 100 megabytes of RAM.

[2]These times do not include the SpecialJ compilation or certification time—certifying compilation typically takes slightly less than 1 second, irrespective of which particular benchmark is compiled.

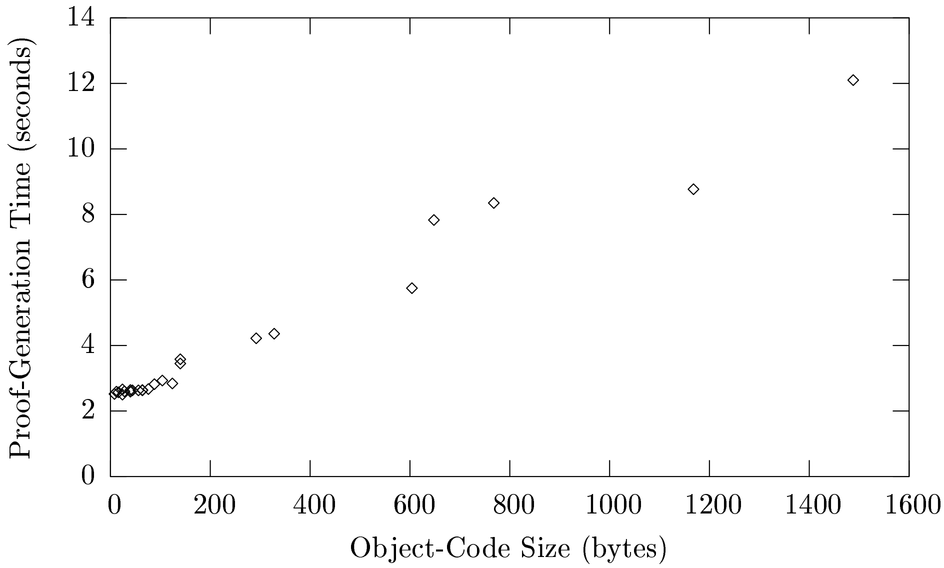

Figure 9.4: Proof-Construction Time

generation logic program into Twelf. I used Twelf [PS99] version 1.3 to generate proof outlines, and as a development environment for the LF representation and derived rules. Twelf is an implementation of the LF logical framework that also contains an interpreter for the Elf constraint logic programming language.

In Figure 9.5, I compare the proof sizes obtained by my implementation with each of two possible proof encodings supported by SpecialJ. The graph labelled "LFi" shows the proof sizes obtained for my benchmarks by the original implicit LF proof representation [NL98b]. The implicit LF representation condenses an explicit proof of a VC by removing redundant type information from the LF encoding of the proof. The graph labelled "Oracle" shows the proof sizes obtained for my benchmarks by the newer "oracle-based" proof representation [NR01]. The oracle-based proof representation operates by encoding a certificate as the set of choices that a nondeterministic logic interpreter makes in reconstructing an explicit VC proof. Note that neither of the SpecialJ proof representations supports foundational proofs, so the code producer in my PCC infrastructure has a significantly more substantial proof obligation. As is apparent from the graph, the certificates produced by my implementation are significantly smaller than those produced by the implicit-LF representation, and are usually within a factor of two of oracle-based proofs.

Note that I include the cost of "code annotations" in the proof sizes for the LFi and oracle-based representations. These code annotations contain procedure specifications and loop invariants that are needed by the SpecialJ symbolic evaluator. Because these annotations take up space in the certificate, and because my certificates need no such annotations, it is reasonable to include them in the proof-size

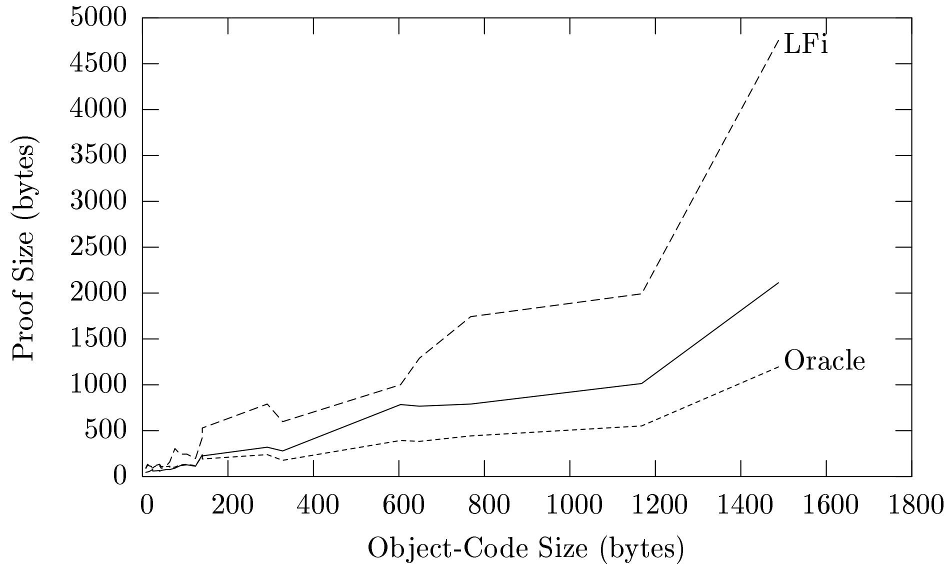

Figure 9.5: Proof-Size Comparison

measurements for the two SpecialJ representations.

9.2 The Logic Interpreter

As I explain in Section 8.1, a logic interpreter reconstructs proofs of safety from minimal outlines. The logic interpreter is thus a key component of the proof-checking implementation. Any derivation that is omitted by the code producer triggers a search for the necessary judgment using the bottom-up proof-search strategy of the logic interpreter. If the search succeeds, then type checking succeeds for the omitted derivation, otherwise the (partial) proof is rejected. Thus, the code consumer expects to see an explicit safety proof with some derivations omitted—in practice, the code producer will omit almost the entire proof, but my approach gives the code producer the flexibility to choose any agreeable balance between explicit and omitted proofs.

Note that there is a natural tension between the sophistication of the logic interpreter and the trustworthiness of the proof checker. One would like the fastest possible logic interpreter to minimize proof checking time, but faster interpreters tend to be more complex than their slower, but simpler counterparts. Thus, the system designer must choose an appropriate tradeoff between speed and complexity for the proof checking infrastructure. For my implementation, I have chosen a particularly conservative point on the complexity-speed scale. The logic interpreter adds less than two thousand lines of ML source code to my LF type checker, but the resulting implementation is several hundred times slower than the SpecialJ proof checker. It remains to be seen whether there is a better engineering “sweet spot" on the scale that brings proof checking times closer to SpecialJ, yet without compromising the trustworthiness of the proof checker.

The logical syntax and inference rules are themselves encoded in the LF logical framework [HHP93] in order to preserve a certain amount of flexibility in the implementation. The LF *signature* is the formal specification for the logical syntax and inference rules, and is also specified in terms of LF language elements. The LF signature must be trusted by the code consumer in my implementation—see Appendix B for a complete listing of the trusted LF signature.

The untrusted logic program is constructed from derived rules to ensure that the trace of any logic-program run has a derivation in the original trusted signature. Thus, the code producer effectively extends the original signature by defining new rules in terms of old ones. My proof checker supports two kinds of definitions to extend the signature: abbreviations and derived rules. Abbreviations are simply a notational device that are expanded immediately when they are encountered. Derived rules, in contrast, are never expanded. In LF terminology, they only attest to the inhabitation of a given type, and the identity of the original term is lost and replaced by a fresh constant. It is also possible to treat all definitions uniformly as fresh constants, and only expand these constants when the precise identity of a particular constant is relevant [PS98b]. However, this approach to definitions requires an additional *strictness check* when the constant is defined, which adds some complexity to the proof checker, in addition to the additional logic required to implement equality testing correctly. Because I want to simplify my implementation as much as is practical, I use a simpler, but less general approach.

The logic-program interpreter embedded in my proof checker is based on a deterministic search over LF constant declarations. The semantics is based on Cervesato [Cer98], except that clauses are tried in an unspecified order, and the logic interpreter never considers an alternative once a given goal succeeds (*i.e.*, it never backtracks). A deterministic semantics enables me to aggressively index the signature as in Necula and Rahul [NR01]. Additionally, I can avoid a significant source of overhead in standard logic interpreters, because my interpreter need not keep track of where existential variables are instantiated to support backtracking. Unification is based on Pfenning [Pfe91], and is thus decidable for higher-order patterns [Mil91]. A *higher order pattern* is a higher-order term in which each function variable is applied to distinct bound variables (only). Because unification is supported only for a decidable fragment, no additional certificate information is needed to guide unification (*e.g.*, as in Necula and Rahul [NR01]).

Note that, in general, it is not sound to search for an arbitrary LF term using a logic program semantics unless the missing term is in an *irrelevant* position. A term is used irrelevantly when no other type can depend on the precise identity of the missing term [Pfe01]. I take a simple approach to proof irrelevance by only enabling logic program searches for a distinguished type of "proofs," on which no other terms can depend.[3] This is effectively the same restriction that Necula and Rahul [NR01] use in their oracle-based logic interpreter. Pfenning [Pfe01] has developed a more

[3]The signature is constructed in such a way that a term of type "proof" can never appear in a normal-form type.

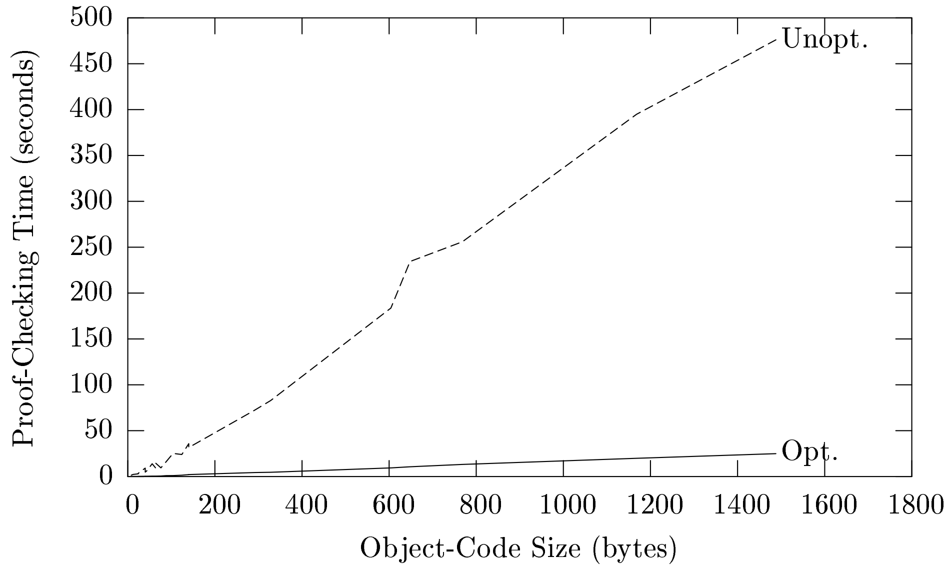

Figure 9.6: Signature Indexing

general framework for treating proof irrelevance in which irrelevant terms are given distinct LF types from relevant ones. Although this is clearly a more systematic approach to the proof irrelevance problem, it also requires a significantly more complex proof checker, and the effect of the new type system on the implementation of unification is still the subject of active research [Ree03].

The effect of indexing the signature is quite dramatic, as I illustrate in Figure 9.6 (the proof-checking times reported in Figure 9.2 include the effect of this optimization). In this figure, the "Opt." graph plots proof-checking time against code size for an indexed signature, whereas "Unopt." plots the same variables for an unindexed signature. I use an indexing strategy that is very similar to Necula and Rahul [NR01]—although this strategy is unsound in the sense that it requires a second, full unification pass to filter out clauses that do not match, it appears to be quite effective in quickly reducing the number of clauses that must be considered by the general unification algorithm. Note that the vast majority of my logic-program goals fall into the so-called "first-order" fragment (*i.e.*, unification is not actually performed on terms with a function type). A notable weakness of the Necula and Rahul strategy is that it does not effectively filter functional terms, so my interpreter may not perform as well for a more complex signature. Note that a more comprehensive approach to signature indexing has been developed recently [Pie03a, Pie03b], so it would be instructive to implement this approach in my system to determine whether proof-checking times can be improved.

A less substantial, but still significant, optimization is obtained by *hash consing* [AdRG93] the application of an LF constant to another constant (again, the proof-checking times reported in Figure 9.2 include the effect of this optimization).

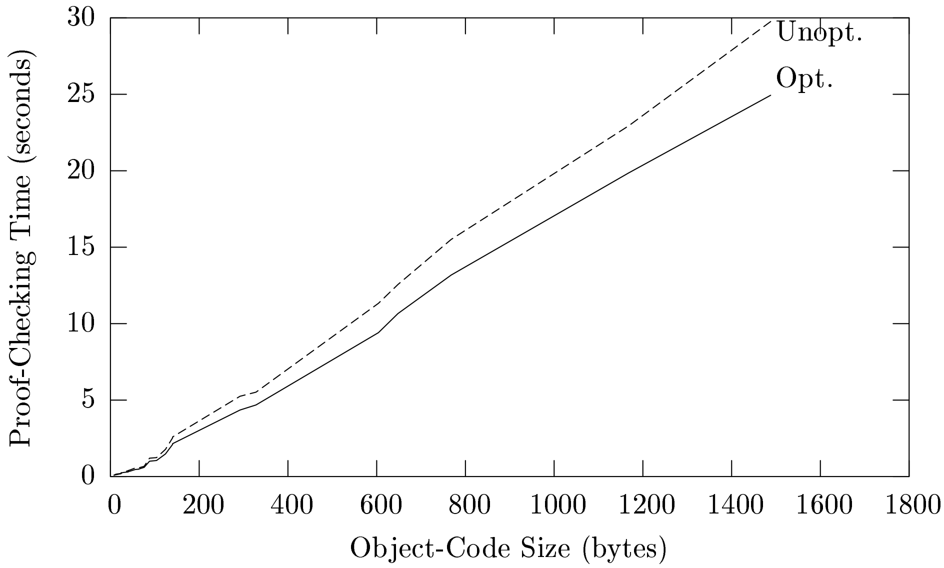

Figure 9.7: Hash-Consed Applications

This optimization operates by keeping a table of all constant applications that have been seen by the proof checker. Whenever a new LF term $(c_1 c_2)$ is to be created, the table is first checked for an existing copy of $(c_1 c_2)$. If such an application is found, then the existing term is used instead of creating a fresh term. If this approach is followed systematically, then any two constant applications $(c_1 c_2)$ and $(c_1' c_2')$ can be compared for equality in constant time by checking the *physical* (*i.e.* address) equality of the terms $(c_1 c_2)$ and $(c_1' c_2')$. This approach generalizes naturally, so that arbitrarily large terms can be compared in constant time if they are composed solely of LF constants and LF applications. The effect of this optimization is summarized in Figure 9.7: "Opt." plots proof-checking time against code size when hash-consing is used for constant applications, whereas "Unopt." plots the same variables for an unoptimized interpreter.

The hash-consing optimization results in an average percentage gain of 13.6% (geometric mean) on proof-checking time for my benchmarks. This gain can be partially attributed to the fact that my LF signature contains a large number of constant applications. For example, the temporal-logic expression

$$\texttt{addw}(3, 4)$$

is represented internally as the LF term

```
fun/app2 wd wd wd wd/#add (con/‘ wd (wd/# 3)) (con/‘ wd (wd/# 4))
```

which is collapsed down to a single constant by the hash-consing optimization. Constants, parameters, functions, and relations are "stratified" in my LF encoding to facilitate syntactic checks such as rigidity and locality.

Chapter 10

Conclusion

I have developed a PCC system in which the software infrastructure is based on checking a security proof against an explicit temporal-logic security-policy specification.

The contributions of this research are as follows:

Enforcement for Reactivity Properties My enforcement mechanism enables the code consumer to specify an arbitrary temporal-logic security property as a proof obligation. These properties are known as the reactivity properties [MP90] and include all safety properties as well as the most familiar liveness properties. Additionally, I have developed a thorough formalization of memory safety and the core of the Java type system that can be used directly by my implementation.

A Program Logic for Invariance Properties My program logic is suitable for proving arbitrary invariance properties of machine language programs. The inference rules of the logic are themselves derived rules in my temporal-logic framework, so their soundness can be established without an appeal to an informal argument. Additionally, I have developed an automatic proof construction algorithm that can be applied to any invariance property for which appropriate loop invariants and residual proofs can be found. Thus, my proof construction algorithm addresses the *domain independent* part of the proof-construction problem—once a suitable type system or verification tool is available for a particular domain of security properties, the code producer need only adapt this tool to the generic program logic. Rules for loop invariants, conditional branches, and procedure calling conventions are provided by the program logic. I have demonstrated that this approach is practical by applying the certification algorithm to Java type-safety proofs emitted by the SpecialJ compiler [CLN$^+$00].

Proof Engineering for End-to-End Proofs I have developed an approach to proof engineering that enables a PCC system that is based on foundational principles[1] to yield proof sizes that are competitive with conventional PCC

[1] Such systems [App01] are characterized by their reliance only on a generic set of logical inference

systems. My enforcement mechanism obliges the code producer to construct an *end-to-end security proof*, in which a specific security property is derived from a generic set of logical inference rules and a formal encoding of a machine semantics. Although proof checking is significantly more time consuming, compact proofs require very little in terms of additional trusted code—in fact, my PCC system requires significantly less trusted executable code than a conventional PCC system such as SpecialJ, because there is no trusted VC generator.

A Temporal-Logic Framework for PCC I have developed an extensive framework for temporal logic (including a model-theoretic semantics and an informal proof of soundness) that is suitable for use in PCC applications. The temporal-logic framework includes an implementation in the LF logical framework [HHP93] that supports machine-checkable proofs. Additionally, I have developed a formal machine model within this framework that supports a useful fragment of the IA-32 instruction set [Int01].

A Foundation for SpecialJ A significant part of the development effort has gone into deconstructing the logical formalization of the SpecialJ infrastructure such that it can be justified on more foundational grounds. For example, the encoding of type safety in my implementation (see Chapter 5) exposes an explicit security register for the type assignment that is left implicit in the SpecialJ implementation as well as in Touchstone [Nec98]—this treatment makes the argument for type safety more transparent, and makes it less likely that the code consumer will be compromised due to an incorrect informal argument. As an additional example, the well-formedness condition on the memory is now a derived concept that clarifies which heap-structure invariants must be preserved by the trusted run-time procedures. Finally, many trusted inference rules in the SpecialJ implementation (*e.g.*, conditional operators) have formal derivations in my implementation that are based directly on the IA-32 machine semantics, and thus need no longer be justified informally.

I view the current implementation as a *platform* for expressive security policies in PCC. The temporal-logic framework and the derived program logic are implemented in such a way that they will be stable under various extensions such as new security policies, type systems, machine instructions, and source-language features. This modularity means that existing derivations and meta-theoretic results will not need to be reproved as new features are added. The motivation for this approach to modularity is well articulated in Wright and Felleisen [WF94] in the context of a syntactic programming-language type soundness argument.

rules and a formal encoding of a machine semantics, as opposed to a program analysis tool that derives an intermediate property, such as a VC generator.

10.1 Related Work

There has been a great deal of interest in certifying object code for particular safety properties. In this section, I review the work to date in this area and show where my approach fits into the current spectrum of technologies.

For the purpose of this review, I categorize the current research as either *proof-carrying code* (PCC) or *typed assembly language* (TAL). In PCC, the code producer provides the code consumer with an explicit derivation of safety in an undecidable formal system. In TAL, the code producer supplies type information that is sufficient for the code consumer to reconstruct a derivation of safety in a decidable formal system. Both approaches normally rely on type systems to establish safety properties. However, in a typical PCC type system, types are assigned only to data, not to both code and data.

Either PCC or TAL can be *foundational* in the sense that the code consumer is equipped with only a minimal standard formal system (*e.g.*, higher-order logic) that is not specifically engineered to enable efficient safety checking. Foundational systems provide a flexibility benefit because safety proofs can be developed in a more general framework, and a trustworthiness benefit because the trusted computing base (TCB) is smaller and more standard. I do *not* consider the original PCC [NL96] and TAL [MWCG98] research to be foundational in this sense.

10.1.1 PCC

Necula and Lee [NL96, Nec97] developed the first PCC system, capable of certifying object code for memory safety. They later extended this approach to enforce resource bounds [NL98c], and Necula developed the first certifying compiler as part of his dissertation research [Nec98]. The development of the conventional PCC architecture culminated in the work of Necula and Rahul [NR01], in which proof sizes were dramatically reduced through the use of an "oracle-based" theorem prover. Recently, Necula and Schneck [NS03] have proposed a new architecture for PCC in which most of the VC generator component is supplied by the code producer. This design promises to provide many of the benefits of the foundational approach to conventional PCC.

Appel and Felty [AF00] were the first to argue that the VC generator is a significant liability in a conventional PCC system—their alternative is foundational proof-carrying code (F-PCC). They proposed instead to derive a formal proof obligation in higher-order logic based only on the program text and a fixed safety policy. The type system is an untrusted, derived component in this framework, and the proof checker is the only trusted component. The soundness of the type system is established by a formal argument. The original such proofs were semantic, but Hamid, *et al.* [HST$^+$02] developed a simpler syntactic argument. Chen, *et al.* [CWAF03] have recently developed an improved type system for F-PCC that may lead to a practical implementation. Current F-PCC type systems resemble TAL type systems. The key departure from foundational TAL (see below) is that the underlying trusted formal system is higher-order logic, rather than another TAL.

10.1.2 TAL

The original TAL was developed by Morrisett, *et al.* [MWCG98, MCGW98] for an abstract assembly language; soundness was established by an informal syntactic argument. TAL was later adapted to a more realistic machine architecture [MCG+99], and extended to cover more "expressive" safety policies. Walker [Wal00] developed a TAL based on security automata. This version of TAL is novel because, like my system, the safety policy is a parameter of the enforcement mechanism. Security automata were introduced by Schneider [Sch99] as a formal representation for safety properties. Crary and Weirich [CW00] developed a TAL that to enforce resource bounds. Crary, *et al.* [CWM99] developed a TAL to enforce safety policies based on a capability calculus which can ensure the safety of explicit deallocation.

The foundational approach to PCC has been recently applied to TAL. In such a system, the TAL that is used for type checking is derived formally from a simpler, undecidable TAL. Crary [Cra03] was the first to pursue this line of research, and it may soon result in a practical implementation [CS03].

10.1.3 TL-PCC

I see TL-PCC as an intermediate point between F-PCC and conventional PCC—the key departures from F-PCC are in motivation. For TL-PCC, I am more interested in additional flexibility, rather than additional trustworthiness. Additionally, practicality and scalability are more important than absolute trustworthiness. Essentially, I am adapting conventional PCC tools to a more foundational infrastructure.

A key distinction between F-PCC and current TL-PCC is that the type system in F-PCC is *derived*, as opposed to being trusted. However, a trusted type system is not an necessary attribute of a TL-PCC system. Essentially, I view the foundational justification of the type system as a separate problem from developing an infrastructure for end-to-end safety proofs that does not require a VC generator. A TL-PCC system could also be based on a derived type system, given a sufficiently expressive logic such as higher-order logic. Note, however, that a trusted type system is needed if the untrusted program is to use a run-time system with a typed interface, as is the case for SpecialJ.

Appel, *et al.* [AMSV02] originally suggested that a simple Prolog interpreter might be suitable for reconstructing derived typing derivations in F-PCC; each such rule infers the type of an instruction, as in TAL. This approach was implemented for F-PCC recently [CWAF03] using a commercial Prolog compiler. Proof-checking times for this implementation are promising, but the size and complexity of the Prolog compiler makes it unsuitable for deployment in an actual PCC system.

I expect that the proof outline technique could also be applied to TAL typing derivations, and that my proof-checking times would improve as a result. One should expect better checking times because TAL typing rules carry less state information than does a symbolic evaluator. However, instruction-based typing rules must be tailored to a specific safety policy to obtain their efficiency advantages, and therefore some of the flexibility benefits of TL-PCC would be lost by using this approach.

Necula and Rahul [NR01] first experimented with using a logic-program interpreter for PCC. My work takes this approach a step further by interpreting an external, *untrusted* logic program, rather than an internal, *trusted* logic program. Proof outlines resemble oracle strings, except that in the case of proof outlines, the interpretation of the oracle string is performed by untrusted derived rules, rather than by the logic interpreter itself. Aside from reducing the complexity of the logic interpreter, I also obtain an important flexibility advantage, because one can customize the constraint representation according to the proof generation strategy, and because one has direct control over how many bits are "spent" for any given choice. An unconstrained search can also be invoked on a judgment that is known to be decidable, and thus ensure that no bits are needed to reconstruct the derivation.

10.1.4 Other Related Work

In Section 2.1.1, Section 2.1.2, and Section 2.1.3, I introduced Schneider's security policy formalism and surveyed approaches to security-policy specification, including PCC, TAL, SFI, and safe interpreters. In this section, I discuss additional related work.

Efficient code certification (ECC) [Koz98] is primarily concerned with minimizing the overhead of an enforcement mechanism for self-certified code. The ECC enforcement mechanism relies on annotations to the object code that provide basic type information and describe calling conventions. These annotations are inserted by a compiler for a type-safe language. The ECC safety property is designed to be as simple as possible, and is essentially defined by the implementation. It provides a basic level of control-flow safety, memory safety, and stack safety.

The SwitchWare active network architecture [AAH+98] stratifies the enforcement mechanism into two levels. At the top level, agents carried by SwitchWare packets are written in the PLAN programming language [HKM+98, HK99] and are checked against a type system that ensures basic type safety and termination. PLAN programs are also permitted to invoke more general *service routines* that are written in a dialect of Objective Caml. Service routines are checked against a type system, and the visibility of sensitive services is controlled through name-space management. Additionally, access control and other security policy constraints may entail run-time checks. Run-time checks are based on a cryptographic certification system that verifies the principal on whose behalf the service routine is invoked. This two-level implementation processes many kinds of network packets efficiently, because they do not invoke the relatively expensive authentication mechanisms.

Many services in the SPIN operating system [BSP+96] are implemented by extensions that dynamically link with the kernel. Such extensions are compiled from the type-safe Modula-3 programming language and thus satisfy a basic type-safety property. Service constraints are imposed by controlling the visibility of sensitive entry points through an explicit representation of interfaces [SFPB95]. Grimm and Bershad [GB99, GB97] developed an access control system for SPIN that enforces relatively fine-grained security policies. Their access control system is based on distinct protection domains and features a strong separation of policy from enforcement. The enforcement mechanism is based on run-time checking.

The J-Kernel [HCC+97] extends the Java security model with multiple protection domains within a single virtual machine. The J-Kernel implements a capability system in which system classes are replaced by dynamically-generated stub classes. The J-Kernel automatically instruments the byte code of the agent. Unfortunately, the J-Kernel imposes significant run-time overhead on cross-domain calls. This overhead comes in part from object argument duplication.

Czajkowski and von Eicken [CvE98] developed a class library called JRes that monitors the resource consumption (*e.g.* heap allocation, processor time) of Java programs. JRes can also control the resource consumption of database extensions [CMSvE98]. JRes employs byte-code instrumentation and run-time checks, and can terminate a thread once its resource limit is exceeded. The resource bound parameters of JRes are specified by Java code that calls the JRes API. This code configures the run-time environment and determines the corrective action to take when a resource limit is exceeded. JRes employs a limited form of abstract security policy specification, because the parameters (although not the overall character) of the security policy can be changed without re-implementing the enforcement mechanism.

Naccio [ET99] is an enforcement mechanism for Java and Win32 programs that is based on a concrete safety-property language. The language interpreted by Naccio resembles Java and is similar in intention to the language proposed in this document—it is limited, however, to safety properties that can be expressed as constraints on system calls. Safety properties are stratified into platform-independent *resource descriptions* and *safety policies*, in addition to platform-dependent *platform descriptions*, which map system procedures onto resource descriptions. Naccio is implemented by code instrumentation: all protected system procedures are replaced by procedures that contain the necessary run-time checks. The agent is additionally rewritten to call the wrapper libraries and to prevent it from subverting the enforcement mechanism.

Myers and Liskov [ML97, ML98] developed a type system for statically verifying information flow constraints. This research augments a basic type system with features (*e.g.*, declassification) to make it a practical extension of a programming language—Myers [Mye99] implemented such an extension (JFlow) to the Java programming language. Information flow constraints are statically verified, so JFlow requires little run-time overhead, and can enforce security policies that cannot be enforced by run-time checks. Volpano, Smith, and Irvine [VSI96] formalized the lattice model of Denning [Den76, DD77] as a type system for an imperative language. Smith and Volpano [SV98] generalized this type system to concurrent systems.

The SLam calculus [HR98] is an extension of the lambda calculus that tracks security information as well as type information. The security information allows information flow and access control security properties to be checked by the type system. Abadi, Banerjee, Heintze, and Riecke [ABHR99] developed a calculus of dependency that generalizes many information flow calculi, including the SLam calculus. The spi calculus [AG97, Aba97] is an extension of Milner's π calculus that has cryptographic language primitives—a type system verifies cryptographic protocols for secure information flow.

10.2 Future Work

There are currently eight judgments in the trusted signature to support term rewriting. It is plausible that these could be converted into derived forms based on the single $\underline{@}$ judgment, as for proof outlines. However, there may be a noticeable performance cost to adopting this approach because rewriting is used so heavily by my implementation.

In order to enable a simple logic program to search for an omitted term of a given LF type, one must know in advance that any term of the desired type is acceptable. In the terminology of Pfenning [Pfe01], the particular identity of the omitted term is *irrelevant*, and one is only interested in whether or not the type is inhabited. It is possible to extend the LF type system to track proof irrelevance explicitly [Pfe01], and thus ensure that a search is never triggered for a "relevant" term. Currently, search is restricted to a built-in proof type, and I must manually inspect the trusted signature to ensure that it is not possible for a proof to appear in a relevant position of the normal form of any type. A type system with a built-in notion of proof irrelevance would enable me to automate this check and to generalize searches, but at the cost of some complexity.

10.2.1 Certification

In this section, I suggest possible future approaches to certifying security properties in my PCC infrastructure.

Instrumentation techniques for security automata [Sch99] are applicable to the proof-generation problem. Security automata can specify all safety properties, and program transformations exist [ES00, Wal00] that will guarantee in many cases that such properties hold. A security automaton that has been threaded through a program by instrumentation is known is an *inline reference monitor* (IRM). Adding an IRM transformation to a certifying compiler would considerably broaden the class of safety policies that can be automatically certified, because the run-time checks inserted by instrumentation make it relatively straightforward to satisfy proof obligations. In this architecture, the IRM tool becomes an *untrusted* component that the code producer uses to adapt the program to the safety policy—the code consumer need only verify that sufficient run-time checks were inserted.

An alternative approach to certification might rely on a *type and effect* system [MWH03, FD02, FA99, LG88] at the source-code level. In such a system, the certifying compiler only accepts programs that are first checked by the type and effect system. The type and effect information is propagated through the compiler such that a well-typed source program produces a well-typed object program. In such a system, the programmer effectively constructs the proof of safety by exhibiting a well-formed program in a decidable formal system. The code consumer can either adopt the object-level type and effect system as a new trusted formal system, or, alternatively, the type and effect system might be derived (as in foundational PCC [App01]) such that the desired safety properties are a consequence of the encoding of the derived system (*e.g*, see Section 6.4).

A third approach to certification might rely on existing automatic or interactive program-verification tools (*e.g.*, SLAM [BR02, BMMR01]). This is a well-established field of study, so it is appealing to leverage the successful research in this area, rather than develop a new tool that largely repeats existing work. In order to use an automatic verification tool to certify a program for PCC, the tool must be able to generate a formal proof or "witness" that the desired security property is satisfied. A *witness* provides a verification tool with sufficient information to efficiently check that a property holds. Researchers have been successful in adding witness-generation capabilities to existing model-checking tools [SD99], but engineering a compact representation for the proof remains a challenge in some cases.

10.2.2 Flat Address Spaces

In the interest of simplicity, the abstract IA-32 machine model of Chapter 4 is based on a "segmented" address space in which a write to the stack cannot overwrite a location in the heap, and *vice versa*. However, many processors and operating systems employ a "flat" address space in which the stack and heap share a common segment (or no segmentation is possible). My formalization of the abstract machine model will also be sound for such systems if the code producer can ensure in advance that stack accesses will be disjoint from heap accesses, and that the stack will not "overflow" into the heap.[2] The memory-safety policy from Chapter 5 will in fact ensure that the stack will not overflow into the heap (given an appropriate boundary page, as in Section 5.6.4), and that all stack accesses are within one page of the top of the stack. As long as the stack has not overflowed into the heap, all such accesses will not touch locations in the heap.

Therefore, the memory-safety policy does in fact ensure that the necessary run-time properties will hold for the abstract machine formalization to be sound for a flat address space. It would be attractive to use the proof of memory safety to justify the soundness of the abstract machine formalization. However, because the memory-safety proof itself is based on the abstract machine formalization, this argument employs circular reasoning as stated.

One possible solution to this dilemma might be to index the memory-safety proof by some parameter such as a maximum execution length or a maximum stack depth, perhaps following the informal indexed recursion soundness argument in Necula [Nec98]. To accommodate such an argument, the inference rules for state transitions could include an additional premise that the heap has not yet collided with the stack. To take a single step, the code producer would need to show that no collision has yet occurred. This could be added to each procedure precondition to certify the first instruction, and then proved by induction for subsequent instruc-

[2] I presume a conventional address-space layout in which the stack grows downwards and the heap grows upwards.

tions. This argument might also be made on a meta-theoretic level, perhaps at the expense of some trustworthiness.

Another possible solution would require the code producer to attach a separate proof that each memory access respects the aliasing constraints. Such a proof would be constructed in a simpler, special-purpose formalism that does not encompass the general-purpose machine model. Once the absence of aliasing is established, then the more general proof of memory safety could be checked against the segmented machine semantics of Chapter 4.

Bibliography

[AAH+98] D. Scott Alexander, William A. Arbaugh, Michael W. Hicks, Pankaj Kakkar, Angelos D. Keromytis, Jonathan T. Moore, Carl A. Gunter, Scott M. Nettles, and Jonathan M. Smith. The SwitchWare active network architecture. *IEEE Network Special Issue: Active and Programmable Networks*, 12(3):29–36, 1998.

[Aba97] Martín Abadi. Secrecy by typing in security protocols. In *Theoretical Aspects of Computer Software: Third International Symposium, Proceedings*, volume 1281 of *Lecture Notes in Computer Science*, pages 611–638, September 1997.

[ABHR99] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 147–160, San Antonio, TX, January 1999.

[ABLP93] Martín Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.

[AdRG93] Andrew W. Appel and Marcelo Jose de Rezende Goncalves. Hash-consing garbage collection. Technical Report TR-412-93, Princeton University, Computer Science Department, February 1993.

[AF00] Andrew W. Appel and Amy 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*, pages 243–253, Boston, MA, January 2000.

[AF03] Andrew W. Appel and Amy P. Felty. Dependent types ensure partial correctness of theorem provers. *Journal of Functional Programming*, 2003. To appear.

[AG97] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The Spi calculus. In *Proceedings of the 4th ACM conference on Computer and communications security*, pages 36–47, Zurich Switzerland, April 1997.

[ALLW96] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe. Efficient and language-independent mobile programs. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 127–136, Philadelphia, PA, May 1996.

[AMSV02] Andrew W. Appel, Neophytos G. Michael, Aaron Stump, and Roberto Virga. A trustworthy proof checker. In Serve Autexier and Heiko Mantel, editors, *Verification Workshop*, volume 02-07 of *DIKU technical reports*, pages 41–52, July 25–26 2002.

[And86] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Computer Science and Applied Mathematics. Academic Press, Orlando, FL, 1986.

[App92] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[App01] Andrew W. Appel. Foundational proof-carrying code. In *Proceedings, 16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*, June 2001.

[AS85] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[AS86] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. Technical Report TR86-727, Cornell University, Computer Science Department, January 1986.

[AS94] Andrew W. Appel and Zang Shao. Empirical and analytic study of stack vs. heap cost for languages with closures. Technical Report CS-TR-450-94, Princeton University, March 1994.

[BL01] Andrew Bernard and Peter Lee. Enforcing formal security properties. Technical Report CMU-CS-01-121, Carnegie Mellon University, School of Computer Science, April 2001.

[BL02a] Andrew Bernard and Peter Lee. Temporal logic for proof-carrying code. In *Proceedings of the 18th International Conference on Automated Deduction (CADE-18)*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 31–46, Copenhagen, Denmark, July 2002.

[BL02b] Andrew Bernard and Peter Lee. Temporal logic for proof-carrying code. Technical Report CMU-CS-02-130, Carnegie Mellon University, School of Computer Science, August 2002.

[BMMR01] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram Rajamani. Automatic predicate abstraction of C programs. In Cindy Norris and Jr. James 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 203–213, N.Y., June 20–22 2001. ACMPress.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, New York, 1998.

[BPW02] Andrew Bernard, Frank Pfenning, and M. Angela Weiss. Natural deduction for temporal logic. Unpublished manuscript, 2002.

[BR02] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. *ACM SIGPLAN Notices*, 37(1):1–3, January 2002.

[BSP+96] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–283, Copper Mountain Resort, CO, December 1996.

[Cer98] Iliano Cervesato. Proof-theoretic foundation of compilation in logic programming languages. In J. Jaffar, editor, *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming (JICSLP'98)*, pages 115–129, Manchester, UK, June 1998. MIT Press.

[CGP+97] A. Cimatti, F. Giunchiglia, P. Pecchiari, B. Pietra, J. Profeta, D. Romano, P. Traverso, and B. Yu. A provably correct embedded verifier for the certification of safety critical software. In *Computer Aided Verification: 9th International Conference, Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, Haifa, Israel, June 1997.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* The MIT Press, Cambridge, Massachusetts, 1999.

[Cla84] Allan Clark. *Elements of Abstract Algebra.* Dover Publications, Inc., New York, NY, 1984.

[CLN+00] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for Java. In *Proceedings of the ACM SIGPLAN '00 conference on programming language design and implementation*, pages 95–107, Vancouver, BC Canada, June 2000.

[CM99] Karl Crary and Greg Morrisett. Type structure for low-level programming languages. In *Automata, Languages and Programming: 26th International Colloquium, Proceedings*, volume 1644 of *Lecture Notes in Computer Science*, Prague, Czech Republic, July 1999.

[CMSvE98] Grzegorz Czajkowski, Tobias Mayr, Praveen Seshadri, and Thorsten von Eicken. Resource control for database extensions. Technical Report TR98-1718, Cornell University, Computer Science Department, November 1998.

[Com02] ComputerWire. Really critical hole in microsoft web software. *The Register*, November 2002.

[Cra03] Karl Crary. Toward a foundational typed assembly language. In *Proceedings of the 2003 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 198–211, New Orleans, LA, January 2003.

[CS03] Karl Crary and Susmit Sarkar. A metalogical approach to foundational certified code. In *Proceedings of the 19th International Conference on Automated Deduction (CADE-19)*, Miami Beach, FL, July 2003. To appear.

[CvE98] Grzegorz Czajkowski and Thorsten von Eicken. JRes: A resource accounting interface for Java. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 21–35, Vancouver, Canada, October 1998.

[CW00] Karl Crary and Stephnie Weirich. Resource bound certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 184–198, Boston, MA, January 2000.

[CWAF03] Juan Chen, Dinghao Wu, Andrew W. Appel, and Hai Fang. A provably sound tal for back-end optimization. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 208–219, San Diego, CA, May 2003.

[CWM99] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 262–275, San Antonio, TX, January 1999.

[Dav96] Rowan Davies. A temporal-logic approach to binding-time analysis. In *Proceedings, 11^{th} Annual IEEE Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, New Jersey, 27–30 July 1996. IEEE Computer Society Press.

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976. Papers from the

Fifth ACM Symposium on Operating Systems Principles, November 19–21, 1975.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[DLNS98] David Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, December 1998.

[DP90] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order.* Cambridge Mathematical Textbooks. Cambridge University Press, Cambridge, UK, 1990.

[DvH66] Jack B. Dennis and Earl C. van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, March 1966. Presented at an ACM Programming Language and Pragmatics Conference, August, 1965, San Dinas, CA.

[Eme90] E. Allen Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 995–1072. Elsevier Science Publishers, 1990.

[ES99] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the 1999 New Security Paradigms Workshop*, Caledon Hills, Ontario, Canada, September 1999.

[ES00] Úlfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *RSP: 21th IEEE Computer Society Symposium on Research in Security and Privacy*, 2000.

[ET99] David Evans and Andrew Twyman. Flexible policy-directed code safety. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Research in Security and Privacy, pages 32–45, Oakland, CA, May 1999.

[FA99] Cormac Flanagan and Martín Abadi. Types for safe locking. *Lecture Notes in Computer Science*, 1576:91–108, March 1999.

[FD02] Manuel Fahndrich and Robert DeLine. Adoption and focus: practical linear types for imperative programming. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI-02)*, volume 37, 5 of *ACM SIGPLAN Notices*, pages 13–24, New York, June 17–19 2002. ACM Press.

[Fel93] Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–81, August 1993.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, Providence, RI, 1967.

[FM98] Stephen N. Freund and John C. Mitchell. A type system for object initialization in the Java bytecode language. Technical Note CS-TN-98-62, Stanford University, Department of Computer Science, April 1998.

[FS01] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. *ACM SIGPLAN Notices*, 36(3):193–205, March 2001.

[GB97] Robert Grimm and Brian N. Bershad. Access control in extensible systems. Technical Report TR-97-11-01, University of Washington, Department of Computer Science and Engineering, November 1997.

[GB99] Robert Grimm and Brian N. Bershad. Providing policy-neutral and transparent access control in extensible systems. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, volume 1603 of *Lecture Notes in Computer Science*. Springer Verlag, 1999.

[Gen69] Gerhard Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*. North-Holland, 1969.

[Ger78] Steven M. German. Automating proofs of the absence of common runtime errors. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 105–118, Tucson, Arizona, January 1978. ACM SIGACT-SIGPLAN.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison Wesley, 1996.

[GMPS97] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.

[GMW79] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[Göd31] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatsh. Math. Phys.*, 38(1931):173–198, 1931.

[Gor88] Michael J. C. Gordon. Mechanizing programming logics in higher order logic. Technical Report UCAM-CL-TR-145, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, phone +44 1223 763500, September 1988.

[Gre97] John Greiner. *Semantics-based Parallel Cost Models and Their Use in Provably Efficient Implementations*. PhD thesis, Carnegie Mellon University, April 1997. Available as Technical Report CMU-CS-97-113.

[HCC+97] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing multiple protection domains in Java. Technical Report TR97-1660, Cornell University, Computer Science Department, December 1997.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993. Preliminary version appeared in Proc. 2nd IEEE Symposium on Logic in Computer Science, 1987, 194–204.

[HK99] Michael Hicks and Angelos D. Keromytis. A secure PLAN. In *Active Networks: First International Working Conference, Proceedings*, volume 1653 of *Lecture Notes in Computer Science*, Berlin, Germany, June 1999.

[HKM+98] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A packet language for active networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, pages 86–93, Baltimore, MD, September 1998.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[HR98] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 365–377, San Diego, CA, January 1998.

[HST+02] Nadeem A. Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code. In *Logic in Computer Science*, pages 89–100, Los Alamitos, CA, USA, July 22–25 2002. IEEE Computer Society.

[Int01] Intel. *IA-32 Intel Architecture Software Developer's Manual*. Intel Corporation, Mt. Prospect, IL, 2001.

[Kin71] J. C. King. Proving programs to be correct. *IEEE Transactions on Computers*, 20(11):1331–1336, November 1971.

[Koz98] Dexter Kozen. Efficient code certification. Technical Report TR98-1661, Cornell University, Computer Science Department, January 1998.

[Koz99] Dexter Kozen. Language-based security. Technical Report TR99-1751, Cornell University, Computer Science Department, June 1999.

[Lam71] B. W. Lampson. Protection. In *Proceedings of the 5th Princeton Symposium on Information Sciences and Systems,*. Princeton University, March 1971. Reprinted in Operating Systems Review 8,1 January 74.

[Lam80] Leslie Lamport. "Sometime" is sometimes "not never": On the temporal logic of programs. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 174–185, Las Vegas, Nevada, January 1980.

[LCC+75] R. Levin, E. Cohen, W. Corwin, Pollack F., and W. Wulf. Policy/mechanism separation in hydra. In *Proceedings of the Fifth ACM Symposium on Operating System Principles*, pages 132–140, Austin, Texas, November 1975.

[LG88] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Conference Record of the Conference on Principles of Programming Languages*, pages 47–57. ACM SIGACT and SIGPLAN, ACM Press, 1988.

[LS82] Leslie Lamport and Fred B. Schneider. The "hoare logic" of CSP, and all that. Technical Report TR82-490, Cornell University, Computer Science Department, May 1982.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison Wesley, second edition, 1999.

[MA00] Neophytos G. Michael and Andrew W. Appel. Machine instruction syntax and semantics in higher order logic. In *Proceedings of the 17th International Conference on Automated Deduction (CADE-17)*, June 2000.

[MCG+99] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, May 1999.

[MCGW98] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Types in Compilation: Second International Workshop, Proceedings*, volume 1473 of *Lecture Notes in Computer Science*, Kyoto, Japan, March 1998.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, September 1991.

[ML85] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. Technical Report 2, Scuola di Specializzazione in Logica Matematica, Dipartimento di Matematica, Università di Siena, 1985.

[ML97] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 129–142, Saint-Malo, France, October 1997.

[ML98] Andrew C. Myers and Barbara Liskov. Complete, safe information flow with decentralized labels. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 186–197, Oakland, California, May 1998.

[MP67] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In *Proceedings of Symposium in Applied Mathematics, vol 19, Mathematical Aspects of Computer Science*, pages 33–41. American Mathematical Society, 1967.

[MP90] Zohar Manna and Amir Pnueli. A hierarchy of temporal properties. In Cynthia Dwork, editor, *Proceedings of the 9th Annual ACM Symposium on Principles of Distribted Computing*, pages 377–408, Québec City, Québec, Canada, August 1990. ACM Press.

[MP91] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer Verlag, 1991.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[MWCG98] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–97, San Diego, CA, January 1998.

[MWH03] Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP-03)*, volume 38, 9 of *ACM SIGPLAN Notices*, pages 213–225, New York, August 25–29 2003. ACM Press.

[Mye99] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium* *on Principles of Programming Languages*, pages 228–241, San Antonio, TX, January 1999.

[Nec97] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, January 1997.

[Nec98] George Ciprian Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, September 1998. Available as Technical Report CMU-CS-98-154.

[NL96] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *USENIX 2nd Symposium on OS Design and Implementation*, Seattle, Washington, October 1996.

[NL98a] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 333–344, Montreal, Canada, June 1998.

[NL98b] George C. Necula and Peter Lee. Efficient representation and validation of logical proofs. In *Proceedings of the 13th Annual Symposium on Logic in Computer Science (LICS'98)*, pages 93–104, Indianapolis, Indiana, June 1998. IEEE Computer Society Press.

[NL98c] George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.

[NR01] George C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 142–154, London, UK, January 2001.

[NS03] George C. Necula and Robert R. Schneck. A sound framework for untrustred verification-condition generators. In *Proceedings, 18th Annual IEEE Symposium on Logic in Computer Science (LICS '03)*, July 2003.

[O'C99] Robert O'Callahan. A simple, comprehensive type system for Java bytecode subroutines. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 70–78, San Antonio, TX, January 1999.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Automated Deduction: 11th International Conference on Automated Deduction, Proceedings*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga Springs, NY, June 1992.

[Pfe91] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, July 1991.

[Pfe97] Frank Pfenning. Computation and deduction. Draft notes for a course on the theory of programming languages using Twelf, 1997.

[Pfe99] Frank Pfenning. Logical frameworks. In *Handbook of Automated Reasoning*, pages 1–82. Elsevier Science Publishers, 1999.

[Pfe01] F. Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*, pages 221–230, Washington - Brussels - Tokyo, June 2001. IEEE.

[Pie03a] Brigitte Pientka. Higher-order substitution tree indexing. In *Proceedings of the 19th International Conference on Logic Programming (ICLP)*, Mumbai, India, December 2003.

[Pie03b] Brigitte Pientka. *Tabled higher-order logic programming.* PhD thesis, Carnegie Mellon University, December 2003. Available as Technical Report CMU-CS-03-185.

[PS98a] Frank Pfenning and Carsten Schuermann. Twelf user's guide. Technical Report CMU-CS-98-173, Department of Computer Science, Carnegie Mellon University, November 1998.

[PS98b] Frank Pfenning and Carsten Schürmann. Algorithms for equality and unification in the presence of notational definitions. In T. Altenkirch, W. Naraschewski, and B. Reus, editors, *Types for Proofs and Programs*, pages 179–193, Kloster Irsee, Germany, March 1998. Springer-Verlag LNCS 1657.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[Ree03] Jason Reed. Extending higher-order unification to support proof irrelevance. In *Proceedings, 16th International Conference on Theorem Proving in Higher Order Logics*, Roma, Italy, September 2003.

[Rey81] John C. Reynolds. *The Craft of Programming.* Prentice Hall, 1981.

[Rey98] John C. Reynolds. *Theories of Programming Languages.* Cambridge University Press, 1998.

[SA99] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, January 1999.

[San90] David Sands. *Calculi for Time Analysis of Functional Programs.* PhD thesis, University of London, September 1990.

[Sch87] Fred B. Schneider. Decomposing properties into safety and liveness. Technical Report TR87-874, Cornell University, Computer Science Department, October 1987.

[Sch99] Fred B. Schneider. Enforceable security policies. Technical Report TR99-1759, Cornell University, Computer Science Department, July 1999.

[SD99] Aaron Stump and David L. Dill. Generating proofs from a decision procedure. In A. Pnueli and P. Traverso, editors, *Proceedings of the FLoC Workshop on Run-Time Result Verification*, Trento, Italy, July 1999.

[SFPB95] G. Sirer, M. Fiuczynski, P. Pardyak, and B. N. Bershad. Safe dynamic linking in an extensible operating system. Technical Report TR-95-11-01, University of Washington, Department of Computer Science and Engineering, November 1995.

[SG98] Abraham Silberschatz and Peter Baer Galvin. *Operating System Concepts.* Addison Wesley, fifth edition, 1998.

[Sim94] Alex K. Simpson. *The Proof Theory and Semantics of Intuitionistic Modal Logic.* PhD thesis, University of Edinburgh, 1994.

[Sta77] Richard Statman. The typed λ-calculus is not elementary recursive. In *18th Annual Symposium on Foundations of Computer Science*, pages 90–94, Providence, Rhode Island, 31 October–2 November 1977. IEEE.

[SV98] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 355–364, San Diego, CA, January 1998.

[VSI96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, December 1996.

[Wal00] David Walker. A type system for expressive security policies. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 254–267, Boston, MA, January 2000.

[WBDF97] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architecture for Java. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 116–128, Saint-Malo, France, October 1997.

[WCC+74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, June 1974.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 15 November 1994.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, NC, December 1993.

Appendix A

Glossary of Notation

A.1 Variables

Lower-case italic characters:

a^τ	$\in Par^\tau$	Parameter (always free)	Section 3.1
b	$\in TPar$	Time parameter	Section 3.1
c^τ	$\in Con^\tau$	Constant	Section 3.1
cop	$\in Val^{\texttt{cop}}$	Conditional operator	Section 4.1
e^τ		Expression	Section 3.1
ea	$\in Val^{\texttt{ea}}$	Effective address	Section 4.1
$f^{\tau_1 \times \cdots \times \tau_k \to \tau}$	$\in Fun^{\tau_1 \times \cdots \times \tau_k \to \tau}$	Function constant	Section 3.1
j	$\in \mathbb{N}$	Natural number	
jty	$\in Val^{\texttt{jty}}$	Java type	Section 5.8
k	$\in \mathbb{Z}$	Integer	
ma	$\in Val^{\texttt{ma}}$	Memory address	Section 4.1
n	$\in Val^{\texttt{wd}}$	Machine word	Section 4.1
oc		Callee Proof Outline	Section 6.6.1
oe		Evaluation Proof Outline	Section 6.6.1
ock		Checking Proof Outline	Section 6.6.1
ovc		VC Proof Outline	Section 6.6.1
$op1$	$\in Val^{\texttt{op1}}$	Unary operator	Section 4.1
$op2$	$\in Val^{\texttt{op2}}$	Binary operator	Section 4.1
$op3$	$\in Val^{\texttt{op3}}$	Ternary operator	Section 4.1
p		Proposition	Section 3.1
r	$\in Val^{\texttt{greg}}$	General-purpose register	Section 4.1
s	$\in State$	State	Section 4.3
t		Time expression	Section 3.1
v^τ	$\in Val^\tau$	Value	Section 3.2
x^τ	$\in Var^\tau$	Variable (usually bound)	Section 3.1
y		Time Variable	Section 7.3.1

Upper-case italic characters:

EA	$\subseteq Val^{\mathtt{ea}}$	Set of effective addresses	Section 7.2.2
I	$\in Val^{\mathtt{inst}}$	Instruction	Section 4.1
J		Judgment	Section 3.1
N	$\subseteq Val^{\mathtt{wd}}$	Set of machine words	Section 7.2.2
O		Reconstruction space	Section 8.1
$R^{\tau_1 \times \cdots \times \tau_k \to o}$	$\in Rel^{\tau_1 \times \cdots \times \tau_k \to o}$	Relation constant	Section 3.1

Greek characters:

α		Parameter list	Section 3.1
π^τ	$\in Seq^\tau$	Sequence	Section 3.2
θ		Trace	Section 7.2.1
ρ		Rigidity	Section 3.1
σ	$\in Exec$	Execution	Section 4.3
τ	$\in Type$	Type	Section 3.1
η	$\in TEnv$	Time environment	Section 3.2
ϕ	$\in Env$	Environment	Section 3.2
ψ		Miscellaneous function	
ω		Search Goal	Section 7.3.1
Θ		Proof Classifier	Section 7.3.1
Σ	$\subseteq Exec$	Execution set	Section 4.3
Γ		Context	Section 3.1
Φ	$\in Val^{\mathtt{prog}}$	Program	Section 4.1
Ω		Search Context	Section 7.3.1

Script characters:

$\mathcal{D}$	Proof Derivation	Chapter 7
$\mathcal{J}$	Interpretation function	Section 3.2.1
$\mathcal{S}$	Search Derivation	Chapter 7
$\mathcal{V}$	Valuation function	Section 3.2.2

A.2 Sets

Semantic domains:

Con^{τ}	$= \mathit{Fun}^{\cdot\to\tau}$	Constants of type τ
Con	$= \bigcup_{\tau\in\mathit{Type}} \mathit{Con}^{\tau}$	Constants
Env	$= \mathit{Par} \to \mathit{Seq}$	Environments
Exec	$= \mathbb{N} \to \mathit{State}$	Executions
$\mathit{Fun}^{\tau_1\times\cdots\times\tau_k\to\tau}$		Function constants from types $\tau_1,\ldots,\tau_k$ to τ
Par^{τ}		Parameters of type τ
Par	$= \bigcup_{\tau\in\mathit{Type}} \mathit{Par}^{\tau}$	Parameters
$\mathit{Rel}^{\tau_1\times\cdots\times\tau_k\to o}$		Relation constants on types $\tau_1,\ldots,\tau_k$
Reg	$= \{\mathtt{pc},\mathtt{f},\mathtt{g},\mathtt{s},\mathtt{m}\}$	Registers
Seq^{τ}	$= \mathbb{N} \to \mathit{Val}^{\tau}$	Sequences of type τ
Seq	$= \bigcup_{\tau\in\mathit{Type}} \mathit{Seq}^{\tau}$	Sequences
SReg^{τ}		Security registers of type τ
SReg	$= \bigcup_{\tau\in\mathit{Type}} \mathit{SReg}^{\tau}$	Security registers
State	$= \mathit{Reg} \to \mathit{Val}$	States
TEnv	$= \mathit{TPar} \to \mathbb{N}$	Time environments
TPar		Time parameters
Val^{τ}		Values of type τ
Val	$= \bigcup_{\tau\in\mathit{Type}} \mathit{Val}^{\tau}$	Values
Var^{τ}		Variables of type τ
Var	$= \bigcup_{\tau\in\mathit{Type}} \mathit{Var}^{\tau}$	Variables
Type		Types

Value domains for particular types:

$\mathit{Val}^{\mathtt{cop}}$	$= \{\mathit{cop}\}_{\mathit{cop}}$	Conditional operators
$\mathit{Val}^{\mathtt{ea}}$	$= \{\mathit{ea}\}_{\mathit{ea}}$	Effective address
$\mathit{Val}^{\mathtt{greg}}$	$= \{r\}_{r}$	Register tokens
$\mathit{Val}^{\mathtt{inst}}$	$= \{I\}_{I}$	Instruction
$\mathit{Val}^{\mathtt{list}\langle\tau\rangle}$	$= (\mathit{Val}^{\tau})^{*}$	Lists
$\mathit{Val}^{\mathtt{mapg}}$	$= \mathit{Val}^{\mathtt{greg}} \to \mathit{Val}^{\mathtt{wd}}$	Register maps
$\mathit{Val}^{\mathtt{mapw}}$	$= \mathit{Val}^{\mathtt{wd}} \to \mathit{Val}^{\mathtt{wd}}$	Machine-word maps
$\mathit{Val}^{\mathtt{ma}}$	$= \{\mathit{ma}\}_{\mathit{ma}}$	Memory address
$\mathit{Val}^{\mathtt{op1}}$	$= \{\mathit{op1}\}_{\mathit{op1}}$	Unary operators
$\mathit{Val}^{\mathtt{op2}}$	$= \{\mathit{op2}\}_{\mathit{op2}}$	Binary operators
$\mathit{Val}^{\mathtt{op3}}$	$= \{\mathit{op3}\}_{\mathit{op3}}$	Ternary operators
$\mathit{Val}^{\mathtt{pair}\langle\tau_1\rangle\langle\tau_2\rangle}$	$= \mathit{Val}^{\tau_1} \times \mathit{Val}^{\tau_2}$	Pairs
$\mathit{Val}^{\mathtt{prog}}$	$= \mathit{Val}^{\mathtt{wd}} \rightharpoonup \mathit{Val}^{\mathtt{inst}}$	Programs
$\mathit{Val}^{\mathtt{qstate}}$	$= \mathit{Val}^{\mathtt{state}} \times \mathit{Val}^{\mathtt{sa}}$	Extended states
$\mathit{Val}^{\mathtt{sa}}$	$= \mathit{SReg} \to \mathit{Val}$	Security-automaton states
$\mathit{Val}^{\mathtt{sreg}\langle\tau\rangle}$	$= \mathit{SReg}^{\tau}$	Security registers of type τ
$\mathit{Val}^{\mathtt{state}}$	$= \mathit{Val}^{\mathtt{wd}} \times \mathit{Val}^{\mathtt{wd}}$ $\times \mathit{Val}^{\mathtt{mapg}} \times \mathit{Val}^{\mathtt{mapw}} \times \mathit{Val}^{\mathtt{mapw}}$	State tuples
$\mathit{Val}^{\mathtt{wd}}$	$= \{n \in \mathbb{N} \mid n < 2^{32}\}$	Machine words

Appendix B

LF Representation

In this appendix, I reproduce the temporal-logic deductive system, machine model, and safety policy as they are represented in the LF Logical Framework [HHP93]. These three encodings comprise the trusted signature that is used for proof checking by the code consumer and together account for all LF code in the trusted computing base (TCB). Note that the code consumer provides a certain number of derived rules as a convenience to the code producer. I do not show these rules here in the interest of brevity—because they have explicit formal derivations, they are essentially outside the TCB.

I assume a reasonable familiarity with LF in this appendix. The concrete syntax is based on Twelf, which is fully specified in the *Twelf User's Guide* [PS98a].

The concrete notation is intended to correspond closely to the notation used in rest of this thesis. Although neither LF nor Twelf have built-in conception of modularity, I attempt to avoid "name space pollution" by prefixing most identifiers by a *module name* that collects together a set of related identifiers. Thus, the identifier `id` in module `module` is written `module/id`. Additionally, identifiers for inference rules are normally written `type|id` where `id` is the name of an inference rule for LF type `type`.

Variables are rendered into ASCII text as follows:

i	`i`	`I`
I	`i^`	`I^`
$\mathcal{I}$	`i&`	`I&`
ϕ	`phi`	`Phi`
Φ	`phi^`	`Phi^`

The second column is used when the variable is bound (explicit), and the third column is used when the variable is free (implicit). Implicit variables are a Twelf feature that do not appear in the trusted part of the signature.

Note that a constant whose name ends with a tilde (`~`) character is not used by my logic interpreter to construct a derivation during a search.

B.1 Temporal Logic

B.1.1 Abstract Syntax

B.1.1.1 Syntactic Types

```
 %{
 %  A type constructor.
}%
ty: type.  %name ty Tau tau.

 %{
 %  An expression of a particular type.
 %  Tau - the type of the expression
}%
exp: ty -> type.  %name exp E e.

 %{
 %  A parameter of a particular type.
 %  Tau - the type of the parameter
}%
par: ty -> type.  %name par A a.

 %{
 %  A constant of a particular type.
 %  Tau - the type of the constant
}%
con: ty -> type.  %name con C c.

 %{
 %  A rigidity flag.
}%
ri: type.  %name ri Rho rho.

 %{
 %  A proposition.
}%
prp: type.  %name prp P p.

 %{
 %  Asserts that a particular statement holds.
}%
jdg: type.  %name jdg J^ j^.

 %{
 %  A time expression.
}%
ti: type.  %name ti T t.

 %{
 %  A function constant with a particular arity.
 %  Tau1 ... - the types of the function's parameters
 %  Tau      - the type of the function's result
}%
% fun/0 = con
fun/1: ty -> ty -> type.              %name fun/1 F f.
fun/2: ty -> ty -> ty -> type.        %name fun/2 F f.
fun/3: ty -> ty -> ty -> ty -> type.  %name fun/3 F f.

 %{
 %  A relation constant with a particular arity.
 %  Tau1 ... - the types of the relation's parameters
}%
rel/0: type.             %name rel/0 R^ r^.
```

```
rel/1: ty -> type.        %name rel/1 R^ r^.
rel/2: ty -> ty -> type.  %name rel/2 R^ r^.
```

B.1.1.2 Parameters

```
par/`: {tau: ty} par tau -> exp tau.
```

B.1.1.3 Constants

```
con/`: {tau: ty} con tau -> exp tau.
```

B.1.1.4 Propositions

```
ri/+: ri.
ri/-: ri.

prp/and: prp -> prp -> prp.
prp/or:  prp -> prp -> prp.
prp/imp: prp -> prp -> prp.

prp/all:  {tau: ty} ri -> (exp tau -> prp) -> prp.
prp/some: {tau: ty} ri -> (exp tau -> prp) -> prp.

prp/nextT:   prp -> prp.
prp/allT:    prp -> prp.
prp/someT:   prp -> prp.
prp/untilT:  prp -> prp -> prp.
prp/unlessT: prp -> prp -> prp.

% - exports -

and = prp/and.  %infix left  113 and.
or  = prp/or.   %infix left  112 or.
imp = prp/imp.  %infix right 111 imp.

all  = prp/all.
some = prp/some.

nextT   = prp/nextT.
allT    = prp/allT.
someT   = prp/someT.
untilT  = prp/untilT.   %infix left 120 untilT.
unlessT = prp/unlessT.  %infix left 120 unlessT.
```

B.1.1.5 Times

```
ti/0:  ti.
ti/+1: ti -> ti.

% - exports -

+1t = ti/+1.  %postfix 211 +1t.
```

B.1.1.6 Functions

```
fun/app1: {tau1: ty} {tau: ty}
          fun/1 tau1 tau
       -> exp tau1 -> exp tau.
fun/app2: {tau1: ty} {tau2: ty} {tau: ty}
          fun/2 tau1 tau2 tau
       -> exp tau1 -> exp tau2 -> exp tau.
fun/app3: {tau1: ty} {tau2: ty} {tau3: ty} {tau: ty}
          fun/3 tau1 tau2 tau3 tau
       -> exp tau1 -> exp tau2 -> exp tau3 -> exp tau.
```

B.1.1.7 Relations

```
% - relations -

rel/not0: rel/0 -> rel/0.
rel/not1: {tau1: ty} rel/1 tau1 -> rel/1 tau1.
rel/not2: {tau1: ty} {tau2: ty} rel/2 tau1 tau2 -> rel/2 tau1 tau2.

rel/#true: rel/0.

rel/#false = rel/not0 rel/#true.

% - propositions -

rel/app0: rel/0 -> prp.
rel/app1: {tau1: ty} rel/1 tau1 -> exp tau1 -> prp.
rel/app2: {tau1: ty} {tau2: ty} rel/2 tau1 tau2 -> exp tau1 -> exp tau2 -> prp.

rel/true  = rel/app0 rel/#true.
rel/false = rel/app0 rel/#false.

% - exports -

true  = rel/true.
false = rel/false.
```

B.1.1.8 Equality

```
eq/#eq: {tau: ty} rel/2 tau tau.

eq/#neq = [tau: ty] rel/not2 tau tau (eq/#eq tau).

eq/eq  = [tau: ty] rel/app2 tau tau (eq/#eq tau).
eq/neq = [tau: ty] rel/app2 tau tau (eq/#neq tau).
```

B.1.1.9 Equivalence

```
eqv/eqv: prp -> prp -> prp.

% - exports -

eqv = eqv/eqv.  %infix left 110 eqv.
```

B.1.1.10 Function Properties

```
fun/inj1 = [tau1: ty] [tau: ty]
           [f: fun/1 tau1 tau]
           all tau1 ri/- ([x1: exp tau1] all tau1 ri/- ([x1': exp tau1]
               eq/eq tau
                     (fun/app1 tau1 tau f x1)
                     (fun/app1 tau1 tau f x1')
           imp eq/eq tau1 x1 x1'
           )).
fun/inj2 = [tau1: ty] [tau2: ty] [tau: ty]
           [f: fun/2 tau1 tau2 tau]
           all tau1 ri/- ([x1: exp tau1] all tau1 ri/- ([x1': exp tau1]
           all tau2 ri/- ([x2: exp tau2] all tau2 ri/- ([x2': exp tau2]
               eq/eq tau
                     (fun/app2 tau1 tau2 tau f x1 x2)
                     (fun/app2 tau1 tau2 tau f x1' x2')
           imp eq/eq tau1 x1 x1' and eq/eq tau2 x2 x2'
           )))).
fun/inj3 = [tau1: ty] [tau2: ty] [tau3: ty] [tau: ty]
           [f: fun/3 tau1 tau2 tau3 tau]
```

```
           all tau1 ri/- ([x1: exp tau1] all tau1 ri/- ([x1': exp tau1]
           all tau2 ri/- ([x2: exp tau2] all tau2 ri/- ([x2': exp tau2]
           all tau3 ri/- ([x3: exp tau3] all tau3 ri/- ([x3': exp tau3]
               eq/eq tau
                     (fun/app3 tau1 tau2 tau3 tau f x1 x2 x3)
                     (fun/app3 tau1 tau2 tau3 tau f x1' x2' x3')
           imp eq/eq tau1 x1 x1' and eq/eq tau2 x2 x2' and eq/eq tau3 x3 x3'
           )))))).

fun/assoc = [tau: ty]
            [f: fun/2 tau tau tau]
            all tau ri/- ([x1: exp tau] all tau ri/- ([x2: exp tau]
            all tau ri/- ([x3: exp tau]
            eq/eq tau
                  (fun/app2 tau tau tau f x1 (fun/app2 tau tau tau f x2 x3))
                  (fun/app2 tau tau tau f (fun/app2 tau tau tau f x1 x2) x3)
            ))).

fun/idl = [tau1: ty] [tau: ty]
          [f: fun/2 tau1 tau tau] [c1: con tau1]
          all tau ri/- ([x2: exp tau]
          eq/eq tau (fun/app2 tau1 tau tau f (con/` tau1 c1) x2) x2
          ).

fun/idr = [tau2: ty] [tau: ty]
          [f: fun/2 tau tau2 tau] [c2: con tau2]
          all tau ri/- ([x1: exp tau]
          eq/eq tau (fun/app2 tau tau2 tau f x1 (con/` tau2 c2)) x1
          ).

fun/id = [tau: ty]
         [f: fun/2 tau tau tau] [c: con tau]
         fun/idl tau tau f c and fun/idr tau tau f c.

fun/invl = [tau: ty]
           [f: fun/2 tau tau tau] [c: con tau] [f1: fun/1 tau tau]
           all tau ri/- ([x: exp tau]
           eq/eq tau
                 (fun/app2 tau tau tau f (fun/app1 tau tau f1 x) x)
                 (con/` tau c)
           ).

fun/invr = [tau: ty]
           [f: fun/2 tau tau tau] [c: con tau] [f2: fun/1 tau tau]
           all tau ri/- ([x: exp tau]
           eq/eq tau
                 (fun/app2 tau tau tau f x (fun/app1 tau tau f2 x))
                 (con/` tau c)
           ).

fun/inv = [tau: ty]
          [f: fun/2 tau tau tau] [c: con tau] [f': fun/1 tau tau]
          fun/invl tau f c f' and fun/invr tau f c f'.

fun/comm = [tau1: ty] [tau: ty]
           [f: fun/2 tau1 tau1 tau]
           all tau1 ri/- ([x1: exp tau1] all tau1 ri/- ([x2: exp tau1]
           eq/eq tau
                 (fun/app2 tau1 tau1 tau f x1 x2)
                 (fun/app2 tau1 tau1 tau f x2 x1)
           )).

fun/distl = [tau: ty]
            [f1: fun/2 tau tau tau] [f2: fun/2 tau tau tau]
```

```
            all tau ri/- ([x1: exp tau] all tau ri/- ([x2: exp tau]
            all tau ri/- ([x3: exp tau]
            eq/eq
              tau
              (fun/app2 tau tau tau f2 (fun/app2 tau tau tau f1 x1 x2) x3)
              (fun/app2 tau tau tau
                        f1
                        (fun/app2 tau tau tau f2 x1 x3)
                        (fun/app2 tau tau tau f2 x2 x3))
            ))).

fun/distr = [tau: ty]
            [f1: fun/2 tau tau tau] [f2: fun/2 tau tau tau]
            all tau ri/- ([x1: exp tau] all tau ri/- ([x2: exp tau]
            all tau ri/- ([x3: exp tau]
            eq/eq
              tau
              (fun/app2 tau tau tau f2 x1 (fun/app2 tau tau tau f1 x2 x3))
              (fun/app2 tau tau tau
                        f1
                        (fun/app2 tau tau tau f2 x1 x2)
                        (fun/app2 tau tau tau f2 x1 x3))
            ))).

fun/dist = [tau: ty]
           [f1: fun/2 tau tau tau] [f2: fun/2 tau tau tau]
           fun/distl tau f1 f2 and fun/distr tau f1 f2.

fun/monoid = [tau: ty]
             [f: fun/2 tau tau tau] [c: con tau]
             fun/assoc tau f and fun/id tau f c.

fun/group = [tau: ty]
            [f: fun/2 tau tau tau] [c: con tau] [f': fun/1 tau tau]
            fun/monoid tau f c and fun/inv tau f c f'.

fun/comm_group = [tau: ty]
                 [f: fun/2 tau tau tau] [c: con tau] [f': fun/1 tau tau]
                 fun/group tau f c f' and fun/comm tau tau f.

fun/ring = [tau: ty]
           [f1: fun/2 tau tau tau] [c1: con tau] [f1': fun/1 tau tau]
           [f2: fun/2 tau tau tau]
               fun/comm_group tau f1 c1 f1'
           and fun/dist tau f1 f2
           and fun/assoc tau f2.

fun/comm_ring = [tau: ty]
                [f1: fun/2 tau tau tau] [c1: con tau] [f1': fun/1 tau tau]
                [f2: fun/2 tau tau tau] [c2: con tau]
                    fun/ring tau f1 c1 f1' f2
                and fun/comm tau tau f2
                and fun/id tau f2 c2.

fun/idem = [tau: ty]
           [f: fun/2 tau tau tau]
           all tau ri/- ([x: exp tau]
           eq/eq tau (fun/app2 tau tau tau f x x) x
           ).

fun/absorb = [tau: ty]
             [f1: fun/2 tau tau tau] [f2: fun/2 tau tau tau]
             all tau ri/- ([x1: exp tau] all tau ri/- ([x2: exp tau]
             eq/eq
```

```
             tau
             (fun/app2 tau tau tau f1 x1 (fun/app2 tau tau tau f2 x1 x2))
             x1
           )).
```

B.1.1.11 Relation Properties

```
rel/fun2 = [tau1: ty] [tau: ty]
           [r^: rel/2 tau1 tau]
           all tau1 ri/- ([x1: exp tau1]
           all tau ri/- ([x: exp tau] all tau ri/- ([x': exp tau]
               rel/app2 tau1 tau r^ x1 x and rel/app2 tau1 tau r^ x1 x'
           imp eq/eq tau x x'
           ))).

rel/ref = [tau: ty]
          [r^: rel/2 tau tau]
          all tau ri/- ([x: exp tau]
          rel/app2 tau tau r^ x x
          ).

rel/irref = [tau: ty]
            [r^: rel/2 tau tau]
            all tau ri/- ([x: exp tau]
            rel/app2 tau tau (rel/not2 tau tau r^) x x
            ).

rel/sym = [tau: ty]
          [r^: rel/2 tau tau]
          all tau ri/- ([x1: exp tau] all tau ri/- ([x2: exp tau]
          rel/app2 tau tau r^ x1 x2 imp rel/app2 tau tau r^ x2 x1
          )).

rel/antisym = [tau: ty]
              [r^: rel/2 tau tau]
              all tau ri/- ([x1: exp tau] all tau ri/- ([x2: exp tau]
                  rel/app2 tau tau r^ x1 x2 and rel/app2 tau tau r^ x2 x1
              imp eq/eq tau x1 x2
              )).

rel/trans = [tau: ty]
            [r^: rel/2 tau tau]
            all tau ri/- ([x1: exp tau] all tau ri/- ([x2: exp tau]
            all tau ri/- ([x3: exp tau]
                rel/app2 tau tau r^ x1 x2 and rel/app2 tau tau r^ x2 x3
            imp rel/app2 tau tau r^ x1 x3
            ))).

rel/total = [tau: ty]
            [r^: rel/2 tau tau]
            all tau ri/- ([x1: exp tau] all tau ri/- ([x2: exp tau]
               eq/eq tau x1 x2
            or rel/app2 tau tau r^ x1 x2
            or rel/app2 tau tau r^ x2 x1
            )).

rel/bot = [tau: ty]
          [r^: rel/2 tau tau]
          [c: con tau]
          all tau ri/- ([x: exp tau]
          eq/eq tau (con/` tau c) x or rel/app2 tau tau r^ (con/` tau c) x
          ).

rel/top = [tau: ty]
```

```
          [r^: rel/2 tau tau]
          [c: con tau]
          all tau ri/- ([x: exp tau]
          eq/eq tau (con/` tau c) x or rel/app2 tau tau r^ x (con/` tau c)
          ).

rel/preorder = [tau: ty]
               [r^: rel/2 tau tau]
               rel/ref tau r^ and rel/trans tau r^.

rel/equiv = [tau: ty]
            [r^: rel/2 tau tau]
            rel/preorder tau r^ and rel/sym tau r^.

rel/order = [tau: ty]
            [r^: rel/2 tau tau]
            rel/preorder tau r^ and rel/antisym tau r^.

rel/tot_order = [tau: ty]
                [r^: rel/2 tau tau]
                rel/order tau r^ and rel/total tau r^.

rel/str_order = [tau: ty]
                [r^: rel/2 tau tau]
                rel/irref tau r^ and rel/trans tau r^.

rel/str_tot_order = [tau: ty]
                    [r^: rel/2 tau tau]
                    rel/str_order tau r^ and rel/total tau r^.

rel/lb2 = [tau: ty]
          [r^: rel/2 tau tau]
          [e: exp tau] [e1: exp tau] [e2: exp tau]
          rel/app2 tau tau r^ e e1 and rel/app2 tau tau r^ e e2.

rel/ub2 = [tau: ty]
          [r^: rel/2 tau tau]
          [e: exp tau] [e1: exp tau] [e2: exp tau]
          rel/app2 tau tau r^ e1 e and rel/app2 tau tau r^ e2 e.

rel/inf2 = [tau: ty]
           [r^: rel/2 tau tau]
           [e: exp tau] [e1: exp tau] [e2: exp tau]
               rel/lb2 tau r^ e e1 e2
           and all tau ri/- ([x: exp tau]
           rel/lb2 tau r^ x e1 e2 imp rel/app2 tau tau r^ x e
           ).

rel/sup2 = [tau: ty]
           [r^: rel/2 tau tau]
           [e: exp tau] [e1: exp tau] [e2: exp tau]
               rel/ub2 tau r^ e e1 e2
           and all tau ri/- ([x: exp tau]
           rel/ub2 tau r^ x e1 e2 imp rel/app2 tau tau r^ e x
           ).

rel/meet = [tau: ty]
           [r^: rel/2 tau tau] [f: fun/2 tau tau tau]
           all tau ri/- ([x1: exp tau] all tau ri/- ([x2: exp tau]
           rel/inf2 tau r^ (fun/app2 tau tau tau f x1 x2) x1 x2
           )).

rel/join = [tau: ty]
           [r^: rel/2 tau tau] [f: fun/2 tau tau tau]
```

```
          all tau ri/- ([x1: exp tau] all tau ri/- ([x2: exp tau]
          rel/sup2 tau r^ (fun/app2 tau tau tau f x1 x2) x1 x2
          )).

rel/latt = [tau: ty]
          [r^: rel/2 tau tau]
          [f1: fun/2 tau tau tau] [f2: fun/2 tau tau tau]
          rel/order tau r^ and rel/meet tau r^ f1 and rel/join tau r^ f2.

rel/dist_latt = [tau: ty]
               [r^: rel/2 tau tau]
               [f1: fun/2 tau tau tau] [f2: fun/2 tau tau tau]
               rel/latt tau r^ f1 f2 and fun/dist tau f2 f1.

rel/cmp = [tau: ty]
         [f1: fun/2 tau tau tau] [f2: fun/2 tau tau tau]
         [c0: con tau] [c1: con tau]
         [e: exp tau] [e': exp tau]
             eq/eq tau (fun/app2 tau tau tau f1 e e') (con/` tau c0)
         and eq/eq tau (fun/app2 tau tau tau f2 e e') (con/` tau c1).

rel/comp = [tau: ty]
          [f1: fun/2 tau tau tau] [f2: fun/2 tau tau tau]
          [c0: con tau] [c1: con tau]
          [f': fun/1 tau tau]
          all tau ri/- ([x: exp tau]
          rel/cmp tau f1 f2 c0 c1 x (fun/app1 tau tau f' x)
          ).

rel/bool_latt = [tau: ty]
               [r^: rel/2 tau tau]
               [f1: fun/2 tau tau tau] [f2: fun/2 tau tau tau]
               [c0: con tau] [c1: con tau]
               [f': fun/1 tau tau]
               all tau ri/- ([x: exp tau]
               rel/dist_latt tau r^ f1 f2 and rel/comp tau f1 f2 c0 c1 f'
               ).
```

B.1.1.12 Pairs

```
% - types -

pair: ty -> ty -> ty.

% - functions -

pair/#make:  {tau1: ty} {tau2: ty} fun/2 tau1 tau2 (pair tau1 tau2).
pair/#left:  {tau1: ty} {tau2: ty} fun/1 (pair tau1 tau2) tau1.
pair/#right: {tau1: ty} {tau2: ty} fun/1 (pair tau1 tau2) tau2.

pair/make  = [tau1: ty] [tau2: ty]
             fun/app2 tau1 tau2 (pair tau1 tau2) (pair/#make tau1 tau2).
pair/left  = [tau1: ty] [tau2: ty]
             fun/app1 (pair tau1 tau2) tau1 (pair/#left tau1 tau2).
pair/right = [tau1: ty] [tau2: ty]
             fun/app1 (pair tau1 tau2) tau2 (pair/#right tau1 tau2).
```

B.1.1.13 Triples

```
% - types -

trip = [tau1: ty] [tau2: ty]
       pair (pair tau1 tau2).
```

```
% - functions -

trip/#make:  {tau1: ty} {tau2: ty} {tau3: ty}
             fun/3 tau1 tau2 tau3 (trip tau1 tau2 tau3).
trip/#left:  {tau1: ty} {tau2: ty} {tau3: ty}
             fun/1 (trip tau1 tau2 tau3) tau1.
trip/#mid:   {tau1: ty} {tau2: ty} {tau3: ty}
             fun/1 (trip tau1 tau2 tau3) tau2.
trip/#right: {tau1: ty} {tau2: ty} {tau3: ty}
             fun/1 (trip tau1 tau2 tau3) tau3.

trip/make  = [tau1: ty] [tau2: ty] [tau3: ty]
             fun/app3 tau1 tau2 tau3 (trip tau1 tau2 tau3)
                      (trip/#make tau1 tau2 tau3).
trip/left  = [tau1: ty] [tau2: ty] [tau3: ty]
             fun/app1 (trip tau1 tau2 tau3) tau1 (trip/#left tau1 tau2 tau3).
trip/mid   = [tau1: ty] [tau2: ty] [tau3: ty]
             fun/app1 (trip tau1 tau2 tau3) tau2 (trip/#mid tau1 tau2 tau3).
trip/right = [tau1: ty] [tau2: ty] [tau3: ty]
             fun/app1 (trip tau1 tau2 tau3) tau3 (trip/#right tau1 tau2 tau3).
```

B.1.1.14 Lists

```
% - types -

list: ty -> ty.

% - constants -

list/empty: {tau: ty} con (list tau).

% - functions -

list/#cons: {tau: ty} fun/2 tau (list tau) (list tau).
list/#head: {tau: ty} fun/1 (list tau) tau.
list/#tail: {tau: ty} fun/1 (list tau) (list tau).

list/cons = [tau: ty] fun/app2 tau (list tau) (list tau) (list/#cons tau).
list/head = [tau: ty] fun/app1 (list tau) tau (list/#head tau).
list/tail = [tau: ty] fun/app1 (list tau) (list tau) (list/#tail tau).
```

B.1.2 Semantics

B.1.2.1 Derivation Types

```
 %{
 %  Perform a given arithmetic operation on a given pair of integers.
 %  N1 N2 -> the integers to perform the operation on
 %  N'   <-  the result of the operation
}%
integer/add: integer -> integer -> integer -> type.
integer/sub: integer -> integer -> integer -> type.
integer/mul: integer -> integer -> integer -> type.
integer/div: integer -> integer -> integer -> type.
integer/mod: integer -> integer -> integer -> type.

 %{
 %  Holds if a given pair of integers are related by a given inequality.
 %  N1 N2 -> the integers
}%
integer/eq:  integer -> integer -> type.
integer/neq: integer -> integer -> type.
integer/leq: integer -> integer -> type.
integer/gt:  integer -> integer -> type.
```

```
 %{
 %  Perform a given 32-bit arithmetic operation on a given pair of integers.
 %  N1 N2 -> the integers to perform the operation on
 %  N'   <-  the result of the operation
}%
integer32/add:  integer -> integer -> integer -> type.
integer32/sub:  integer -> integer -> integer -> type.
integer32/mul:  integer -> integer -> integer -> type.
integer32/mulu: integer -> integer -> integer -> type.
integer32/div:  integer -> integer -> integer -> type.
integer32/divu: integer -> integer -> integer -> type.
integer32/mod:  integer -> integer -> integer -> type.
integer32/modu: integer -> integer -> integer -> type.

 %{
 %  Perform a given 32-bit bitwise operation on a given pair of nonnegative
 %  integers.
 %  N1 N2 -> the integers to perform the operation on;
 %           must be nonnegative
 %  N'   <-  the result of the operation;
 %           will be nonnegative
}%
integer32/and: integer -> integer -> integer -> type.
integer32/or:  integer -> integer -> integer -> type.
integer32/xor: integer -> integer -> integer -> type.

 %{
 %  Holds if a given pair of integers are related by a given 32-bit relation.
 %  N1 N2 -> the integers
}%
integer32/eq:   integer -> integer -> type.
integer32/neq:  integer -> integer -> type.
integer32/leq:  integer -> integer -> type.
integer32/lequ: integer -> integer -> type.
integer32/gt:   integer -> integer -> type.
integer32/gtu:  integer -> integer -> type.

 %{
 %  Convert a given integer to a 32-bit signed or unsigned representation.
 %  N   -> the integer to convert
 %  N' <-  the result of the conversion
}%
integer32/unsign: integer -> integer -> type.
integer32/sign:   integer -> integer -> type.
integer32/wrap:   integer -> integer -> type.

 %{
 %  Asserts that a given pair of constants are distinct.
 %  Rules for this are untrusted.
 %  C1 C2 -> the constants to compare for inequality
}%
con/neq: {tau: ty} con tau -> con tau -> type.
```

B.1.2.2 Integers

```
integer/sub|: {n1: integer} {n2: integer} {n': integer}
              integer/add n' n2 n1
           -> integer/sub n1 n2 n'.

integer/eq|: {n: integer}
             integer/eq n n.

integer/neq|gt: {n1: integer} {n2: integer}
```

```
                integer/gt n1 n2
             -> integer/neq n1 n2.
integer/neq|lt: {n1: integer} {n2: integer}
                integer/gt n2 n1
             -> integer/neq n1 n2.

integer/leq|eq: {n: integer}
                integer/leq n n.
integer/leq|gt: {n1: integer} {n2: integer}
                integer/gt n2 n1
             -> integer/leq n1 n2.
```

B.1.2.3 32-Bit Integers

```
integer32/sub|:  {n1: integer} {n2: integer} {n': integer}
                 integer32/add n' n2 n1
              -> integer32/sub n1 n2 n'.
integer32/mul|:  {n1: integer} {n2: integer} {n1': integer} {n2': integer}
                 {n': integer} {n'': integer}
                 integer32/unsign n' n''
              -> integer/mul n1' n2' n'
              -> integer32/sign n1 n1' -> integer32/sign n2 n2'
              -> integer32/mul n1 n2 n''.
integer32/mulu|: {n1: integer} {n2: integer} {n1': integer} {n2': integer}
                 {n': integer} {n'': integer}
                 integer32/unsign n' n''
              -> integer/mul n1' n2' n'
              -> integer32/unsign n1 n1' -> integer32/unsign n2 n2'
              -> integer32/mulu n1 n2 n''.
integer32/div|:  {n1: integer} {n2: integer} {n1': integer} {n2': integer}
                 {n': integer} {n'': integer}
                 integer32/unsign n' n''
              -> integer/div n1' n2' n'
              -> integer32/sign n1 n1' -> integer32/sign n2 n2'
              -> integer32/div n1 n2 n''.
integer32/divu|: {n1: integer} {n2: integer} {n1': integer} {n2': integer}
                 {n': integer} {n'': integer}
                 integer32/unsign n' n''
              -> integer/div n1' n2' n'
              -> integer32/unsign n1 n1' -> integer32/unsign n2 n2'
              -> integer32/divu n1 n2 n''.
integer32/mod|:  {n1: integer} {n2: integer} {n1': integer} {n2': integer}
                 {n': integer} {n'': integer}
                 integer32/unsign n' n''
              -> integer/mod n1' n2' n'
              -> integer32/sign n1 n1' -> integer32/sign n2 n2'
              -> integer32/mod n1 n2 n''.
integer32/modu|: {n1: integer} {n2: integer} {n1': integer} {n2': integer}
                 {n': integer} {n'': integer}
                 integer32/unsign n' n''
              -> integer/mod n1' n2' n'
              -> integer32/unsign n1 n1' -> integer32/unsign n2 n2'
              -> integer32/modu n1 n2 n''.

integer32/or|:  {n1: integer} {n2: integer} {n1': integer} {n2': integer}
                {n': integer} {n'': integer}
                integer32/sub integer32/maxu n' n''
             -> integer32/and n1' n2' n'
             -> integer32/sub integer32/maxu n2 n2'
             -> integer32/sub integer32/maxu n1 n1'
             -> integer32/or n1 n2 n''.
integer32/xor|: {n1: integer} {n2: integer} {n1': integer} {n2': integer}
                {n': integer} {n'': integer} {n''': integer} {n'''': integer}
                {n''''': integer}
```

```
                integer32/and n''' n'''' n'''''
             -> integer32/sub integer32/maxu n'' n''''
             -> integer32/sub integer32/maxu n' n'''
             -> integer32/and n1' n2' n''
             -> integer32/and n1 n2 n'
             -> integer32/sub integer32/maxu n1 n1'
             -> integer32/sub integer32/maxu n2 n2'
             -> integer32/xor n1 n2 n'''''.

integer32/neq|gt: {n1: integer} {n2: integer}
                  integer32/gtu n1 n2
               -> integer32/neq n1 n2.
integer32/neq|lt: {n1: integer} {n2: integer}
                  integer32/gtu n2 n1
               -> integer32/neq n1 n2.

integer32/leq|eq: {n1: integer} {n2: integer}
                  integer32/eq n1 n2
               -> integer32/leq n1 n2.
integer32/leq|gt: {n1: integer} {n2: integer}
                  integer32/gt n2 n1
               -> integer32/leq n1 n2.

integer32/lequ|eq: {n1: integer} {n2: integer}
                   integer32/eq n1 n2
                -> integer32/lequ n1 n2.
integer32/lequ|gt: {n1: integer} {n2: integer}
                   integer32/gtu n2 n1
                -> integer32/lequ n1 n2.

integer32/unsign|0:     integer32/unsign 0 0.
integer32/unsign|1:     integer32/unsign 1 1.
integer32/unsign|maxu:  integer32/unsign integer32/maxu integer32/maxu.
integer32/unsign|ok:    {n: integer}
                        integer/gt n 1
                     -> integer/gt integer32/maxu n
                     -> integer32/unsign n n.
integer32/unsign|under: {n: integer} {n': integer}
                        integer/mod n integer32/numu n'
                     -> integer/gt 0 n
                     -> integer32/unsign n n'.
integer32/unsign|over:  {n: integer} {n': integer}
                        integer/mod n integer32/numu n'
                     -> integer/gt n integer32/maxu
                     -> integer32/unsign n n'.

integer32/sign|0:     integer32/sign 0 0.
integer32/sign|1:     integer32/sign 1 1.
integer32/sign|~1:    integer32/sign ~1 ~1.
integer32/sign|neg:   {n: integer}
                      integer/leq n ~2
                   -> integer/leq integer32/min n
                   -> integer32/sign n n.
integer32/sign|pos:   {n: integer}
                      integer/leq n integer32/max
                   -> integer/leq 2 n
                   -> integer32/sign n n.
integer32/sign|under: {n: integer} {n': integer} {n'': integer}
                      integer32/wrap n' n''
                   -> integer32/unsign n n'
                   -> integer/gt integer32/min n
                   -> integer32/sign n n''.
integer32/sign|over:  {n: integer} {n': integer} {n'': integer}
                      integer32/wrap n' n''
```

```
                   -> integer32/unsign n n'
                   -> integer/gt n integer32/max
                   -> integer32/sign n n''.

integer32/wrap|pos: {n: integer}
                    integer/leq n integer32/max
                 -> integer32/wrap n n.
integer32/wrap|neg: {n: integer} {n': integer}
                    integer/sub n integer32/numu n'
                 -> integer/gt n integer32/max
                 -> integer32/wrap n n'.
```

B.1.3 Inference Rules

B.1.3.1 Derivation Types

```
 %{
 %  Asserts that a given parameter can be shifted in time.
 %  Tau - the type of the parameter
 %  A   - the parameter
}%
ri/par: {tau: ty} par tau -> type.  %name ri/par D& d&.

 %{
 %  Shows that a given judgment holds.
 %  This type should only be used for proof-irrelevent arguments.
 %  J^ - the judgment
}%
pf/: jdg -> type.  %name pf/ D& d&.

% - exports -

pf = pf/.  %prefix 10 pf.
```

B.1.3.2 Judgments

```
 %{
 %  Asserts that a given proposition is local in a given parameter.
 %  Tau - the type of the parameter
 %  P   - the proposition
}%
lo/of+: {tau: ty} (exp tau -> prp) -> jdg.

 %{
 %  Asserts that a given expression has a given rigidity.
 %  Tau - the type of the expression
 %  E   - the expression
 %  Rho - the rigidity
}%
ri/ofE: {tau: ty} exp tau -> ri -> jdg.

 %{
 %  Asserts that a given proposition has a given rigidity.
 %  P   - the proposition
 %  Rho - the rigidity
}%
ri/of: prp -> ri -> jdg.

 %{
 %  Asserts that a given expression reduces (in one step) to another
 %  expression, given that all subterms of the expression are already normal.
 %  Tau - the type of the expression
 %  E   - the expression to reduce; all subterms are presumed normal
 %  E'  - the single-step reduction of E
```

```
}%
rew/step0E: {tau: ty} exp tau -> exp tau -> jdg.
rew/stepE:  {tau: ty} exp tau -> exp tau -> jdg.

 %{
 %  Asserts that a given expression reduces to a given normal form, given that
 %  all subterms of the expression are already normal.
 %  Tau - the type of the expression
 %  E   - the expression to reduce; all subterms are presumed normal
 %  E'  - the normal form of E
}%
rew/tailE: {tau: ty} exp tau -> exp tau -> jdg.

 %{
 %  Asserts that a given expression reduces to a given normal form.
 %  Tau - the type of the expression
 %  E   - the expression to reduce
 %  E'  - the normal form of E
}%
rew/normE: {tau: ty} exp tau -> exp tau -> jdg.

 %{
 %  Asserts that a given proposition reduces (in one step) to another
 %  proposition, given that all subterms of the proposition are already
 %  normal.
 %  P  - the proposition to reduce; all subterms are presumed normal
 %  P' - the single-step reduction of P
}%
rew/step0: prp -> prp -> jdg.
rew/step:  prp -> prp -> jdg.

 %{
 %  Asserts that a given proposition reduces to a given normal form, given
 %  that all subterms of the proposition are already normal.
 %  P  - the proposition to reduce; all subterms are presumed normal
 %  P' - the normal form of P
}%
rew/tail: prp -> prp -> jdg.

 %{
 %  Asserts that a given proposition reduces to a given normal form.
 %  P  - the proposition to reduce
 %  P' - the normal form of P
}%
rew/norm: prp -> prp -> jdg.

 %{
 %  Asserts that a given time is the same as or earlier than another time.
 %  T1 - the earlier time
 %  T2 - the later time
}%
ti/<=: ti -> ti -> jdg.

% - exports -

<=t = ti/<=.  %infix none 20 <=t.

 %{
 %  Asserts that a given proposition is true at a given time.
 %  P - the proposition
 %  T - the time
}%
at/@: prp -> ti -> jdg.
```

```
 %{
 %  Asserts that a given proposition is true over a given finite interval.
 %  P  - the proposition
 %  T1 - the start time
 %  T2 - the stop time
}%
at/@ov: prp -> ti -> ti -> jdg.

 %{
 %  Asserts that a given proposition is true from a given time.
 %  P - the proposition
 %  T - the time
}%
at/@fr = [p: prp] [t: ti] allT p @ t.

% - exports -

@ = at/@.  %infix none 20 @.

 %{
 %  Asserts that a given proposition is true at a given time in a restricted
 %  formal system.
 %  P - the proposition
 %  T - the time
}%
uat/@: prp -> ti -> jdg.

% - exports -

@_ = uat/@.  %infix none 20 @_.

 %{
 %  Asserts that the application of a given function constant to a given set of
 %  argument constants is a given result constant.
 %  F   -> the function to apply
 %      -> the constants to apply the function to
 %  C' <-  the result of applying F to the argument constants
}%
fun/app1#: {tau1: ty} {tau: ty}
           fun/1 tau1 tau
        -> con tau1 -> con tau
        -> jdg.
fun/app2#: {tau1: ty} {tau2: ty} {tau: ty}
           fun/2 tau1 tau2 tau
        -> con tau1 -> con tau2 -> con tau
        -> jdg.
fun/app3#: {tau1: ty} {tau2: ty} {tau3: ty} {tau: ty}
           fun/3 tau1 tau2 tau3 tau
        -> con tau1 -> con tau2 -> con tau3 -> con tau
        -> jdg.

 %{
 %  Asserts that a given relation constant holds for a given set of argument
 %  constants is a given result constant.
 %  R^ -> the relation to apply
 %     -> the constants to apply the relation to
}%
% no rel/app0#
rel/app1#: {tau1: ty}
           rel/1 tau1 -> con tau1 -> jdg.
rel/app2#: {tau1: ty} {tau2: ty}
           rel/2 tau1 tau2 -> con tau1 -> con tau2 -> jdg.
```

B.1.3.3 Locality

```
lo/of+|i_and:      {tau: ty} {p1: exp tau -> prp} {p2: exp tau -> prp}
                   pf lo/of+ tau p1 -> pf lo/of+ tau p2
                -> pf lo/of+ tau ([a: exp tau] (p1 a) and (p2 a)).
lo/of+|i_or:       {tau: ty} {p1: exp tau -> prp} {p2: exp tau -> prp}
                   pf lo/of+ tau p1 -> pf lo/of+ tau p2
                -> pf lo/of+ tau ([a: exp tau] (p1 a) or (p2 a)).
lo/of+|i_imp:      {tau: ty} {p1: exp tau -> prp} {p2: exp tau -> prp}
                   pf lo/of+ tau p1 -> pf lo/of+ tau p2
                -> pf lo/of+ tau ([a: exp tau] (p1 a) imp (p2 a)).
lo/of+|i_all:      {tau: ty} {tau': ty} {rho: ri}
                   {p: exp tau' -> exp tau -> prp}
                   ({a': exp tau'} pf lo/of+ tau (p a'))
                -> pf lo/of+
                        tau
                        ([a: exp tau] all tau' rho ([a': exp tau'] p a' a)).
lo/of+|i_some:     {tau: ty} {tau': ty} {rho: ri}
                   {p: exp tau' -> exp tau -> prp}
                   ({a': exp tau'} pf lo/of+ tau (p a'))
                -> pf lo/of+
                        tau
                        ([a: exp tau] some tau' rho ([a': exp tau'] p a' a)).
lo/of+|i_nextT:    {tau: ty} {p: prp}
                   pf lo/of+ tau ([a: exp tau] nextT p).
lo/of+|i_allT:     {tau: ty} {p: prp}
                   pf lo/of+ tau ([a: exp tau] allT p).
lo/of+|i_someT:    {tau: ty} {p: prp}
                   pf lo/of+ tau ([a: exp tau] someT p).
lo/of+|i_untilT:   {tau: ty} {p1: prp} {p2: prp}
                   pf lo/of+ tau ([a: exp tau] p1 untilT p2).
lo/of+|i_unlessT:  {tau: ty} {p1: prp} {p2: prp}
                   pf lo/of+ tau ([a: exp tau] p1 unlessT p2).
```

B.1.3.4 Rigidity

```
% - rigidity -

ri/ofE|i-:     {tau: ty} {e: exp tau}
               pf ri/ofE tau e ri/-.
ri/ofE|i_par:  {tau: ty} {a: par tau}
               ri/par tau a
            -> pf ri/ofE tau (par/` tau a) ri/+.
ri/ofE|i_con:  {tau: ty} {c: con tau}
               pf ri/ofE tau (con/` tau c) ri/+.

ri/of|i_and:      {p1: prp} {p2: prp} {rho: ri}
                  pf ri/of p1 rho -> pf ri/of p2 rho
               -> pf ri/of (p1 and p2) rho.
ri/of|i_or:       {p1: prp} {p2: prp} {rho: ri}
                  pf ri/of p1 rho -> pf ri/of p2 rho
               -> pf ri/of (p1 or p2) rho.
ri/of|i_imp:      {p1: prp} {p2: prp} {rho: ri}
                  pf ri/of p1 rho -> pf ri/of p2 rho
               -> pf ri/of (p1 imp p2) rho.
ri/of|i_all:      {tau: ty} {rho': ri} {p: exp tau -> prp} {rho: ri}
                  ({a: par tau} ri/par tau a -> pf ri/of (p (par/` tau a)) rho)
               -> pf ri/of (all tau rho' p) rho.
ri/of|i_some:     {tau: ty} {rho': ri} {p: exp tau -> prp} {rho: ri}
                  ({a: par tau} ri/par tau a -> pf ri/of (p (par/` tau a)) rho)
               -> pf ri/of (some tau rho' p) rho.
ri/of|i_nextT:    {p: prp} {rho: ri}
                  pf ri/of p rho
               -> pf ri/of (nextT p) rho.
ri/of|i_allT:     {p: prp} {rho: ri}
```

```
                pf ri/of p rho
             -> pf ri/of (allT p) rho.
ri/of|i_someT:   {p: prp} {rho: ri}
                pf ri/of p rho
             -> pf ri/of (someT p) rho.
ri/of|i_untilT:  {p1: prp} {p2: prp} {rho: ri}
                pf ri/of p1 rho -> pf ri/of p2 rho
             -> pf ri/of (p1 untilT p2) rho.
ri/of|i_unlessT: {p1: prp} {p2: prp} {rho: ri}
                pf ri/of p1 rho -> pf ri/of p2 rho
             -> pf ri/of (p1 unlessT p2) rho.

% - truth -

ri/of|e: {p: prp} {t: ti} {t': ti}
        pf ri/of p ri/+ -> pf p @ t
     -> pf p @ t'.
```

B.1.3.5 Rewriting

```
rew/step0E|i~: {tau: ty}
               {e: exp tau} {e': exp tau}
               ({t: ti} pf eq/eq tau e e' @ t)
            -> pf rew/step0E tau e e'.
rew/stepE|i~:  {tau: ty}
               {e: exp tau} {e': exp tau}
               ({t: ti} pf eq/eq tau e e' @ t)
            -> pf rew/stepE tau e e'.

rew/tailE|i_step0: {tau: ty} {e: exp tau} {e': exp tau}
                   pf rew/step0E tau e e'
                -> pf rew/tailE tau e e'.
rew/tailE|i_step:  {tau: ty} {e: exp tau} {e': exp tau} {e'': exp tau}
                   pf rew/normE tau e' e'' -> pf rew/stepE tau e e'
                -> pf rew/tailE tau e e''.

rew/normE|i_par: {tau: ty} {a: par tau}
                 pf rew/normE tau (par/` tau a) (par/` tau a).
rew/normE|i_con: {tau: ty} {c: con tau}
                 pf rew/normE tau (con/` tau c) (con/` tau c).

rew/step0|i~: {p: prp} {p': prp}
              ({t: ti} pf p eqv p' @ t)
           -> pf rew/step0 p p'.
rew/step|i~:  {p: prp} {p': prp}
              ({t: ti} pf p eqv p' @ t)
           -> pf rew/step p p'.

rew/tail|i_step0: {p: prp} {p': prp}
                  pf rew/step0 p p'
               -> pf rew/tail p p'.
rew/tail|i_step:  {p: prp} {p': prp} {p'': prp}
                  pf rew/norm p' p'' -> pf rew/step p p'
               -> pf rew/tail p p''.

rew/norm|i_and:     {p1: prp} {p2: prp} {p1': prp} {p2': prp} {p'': prp}
                    pf rew/tail (p1' and p2') p''
                 -> pf rew/norm p1 p1' -> pf rew/norm p2 p2'
                 -> pf rew/norm (p1 and p2) p''.
rew/norm|i_or:      {p1: prp} {p2: prp} {p1': prp} {p2': prp} {p'': prp}
                    pf rew/tail (p1' or p2') p''
                 -> pf rew/norm p1 p1' -> pf rew/norm p2 p2'
                 -> pf rew/norm (p1 or p2) p''.
rew/norm|i_imp:     {p1: prp} {p2: prp} {p1': prp} {p2': prp} {p'': prp}
```

```
                    pf rew/tail (p1' imp p2') p''
                 -> pf rew/norm p1 p1' -> pf rew/norm p2 p2'
                 -> pf rew/norm (p1 imp p2) p''.
rew/norm|i_all:     {tau: ty} {rho: ri}
                    {p: exp tau -> prp} {p': exp tau -> prp} {p'': prp}
                    pf rew/tail (all tau rho p') p''
                 -> ({a: par tau}
                     pf rew/norm (p (par/` tau a)) (p' (par/` tau a)))
                 -> pf rew/norm (all tau rho p) p''.
rew/norm|i_some:    {tau: ty} {rho: ri}
                    {p: exp tau -> prp} {p': exp tau -> prp} {p'': prp}
                    pf rew/tail (some tau rho p') p''
                 -> ({a: par tau}
                     pf rew/norm (p (par/` tau a)) (p' (par/` tau a)))
                 -> pf rew/norm (some tau rho p) p''.
rew/norm|i_nextT:   {p: prp} {p': prp} {p'': prp}
                    pf rew/tail (nextT p') p'' -> pf rew/norm p p'
                 -> pf rew/norm (nextT p) p''.
rew/norm|i_allT:    {p: prp} {p': prp} {p'': prp}
                    pf rew/tail (allT p') p'' -> pf rew/norm p p'
                 -> pf rew/norm (allT p) p''.
rew/norm|i_someT:   {p: prp} {p': prp} {p'': prp}
                    pf rew/tail (someT p') p'' -> pf rew/norm p p'
                 -> pf rew/norm (someT p) p''.
rew/norm|i_untilT:  {p1: prp} {p2: prp} {p1': prp} {p2': prp} {p'': prp}
                    pf rew/tail (p1' untilT p2') p''
                 -> pf rew/norm p1 p1' -> pf rew/norm p2 p2'
                 -> pf rew/norm (p1 untilT p2) p''.
rew/norm|i_unlessT: {p1: prp} {p2: prp} {p1': prp} {p2': prp} {p'': prp}
                    pf rew/tail (p1' unlessT p2') p''
                 -> pf rew/norm p1 p1' -> pf rew/norm p2 p2'
                 -> pf rew/norm (p1 unlessT p2) p''.
```

B.1.3.6 Time

```
% Peano rules
<=t|i0:     {t: ti}
            pf ti/0 <=t t.
<=t|i+1:    {t: ti}
            pf t <=t t +1t.
<=t|e+1:    {t: ti} {p': prp} {t': ti}
            pf t +1t <=t t
         -> pf p' @ t'.
<=t|e_ind2: {t0: ti} {p1: prp} {p2: prp} {p': prp} {t': ti}
             pf p1 @ t0
         -> ({t: ti} {p': prp} {t': ti}
              pf t0 <=t t -> pf p1 @ t
          -> (pf p1 @ t +1t -> pf p' @ t') -> (pf p2 @ t +1t -> pf p' @ t')
          -> pf p' @ t')
         -> (({t1: ti} pf t0 <=t t1 -> pf p1 @ t1)
          -> pf p' @ t')
         -> ({t2: ti}
              pf t0 <=t t2 -> pf at/@ov p1 t0 t2 -> pf p2 @ t2
          -> pf p' @ t')
         -> pf p' @ t'.

% partial order rules
<=t|i_ref:   {t: ti}
             pf t <=t t.
<=t|e_asym:  {t1: ti} {t2: ti} {p: prp}
             pf t1 <=t t2 -> pf t2 <=t t1 -> pf p @ t1
          -> pf p @ t2.
<=t|e_trans: {t1: ti} {t2: ti} {t3: ti}
             pf t1 <=t t2 -> pf t2 <=t t3
```

```
         -> pf t1 <=t t3.

% linearity rules
<=t|e_linp: {t0: ti} {t1: ti} {t2: ti} {p: prp} {t: ti}
            pf t1 <=t t0 -> pf t2 <=t t0
         -> (pf t2 <=t t1 -> pf p @ t) -> (pf t1 +1t <=t t2 -> pf p @ t)
         -> pf p @ t.
<=t|e_linf: {t0: ti} {t1: ti} {t2: ti} {p: prp} {t: ti}
            pf t0 <=t t1 -> pf t0 <=t t2
         -> (pf t2 <=t t1 -> pf p @ t) -> (pf t1 +1t <=t t2 -> pf p @ t)
         -> pf p @ t.
```

B.1.3.7 Propositions

```
% - truth: connectives -

and|i:  {p1: prp} {p2: prp} {t: ti}
        pf p1 @ t -> pf p2 @ t
     -> pf p1 and p2 @ t.
and|el: {p1: prp} {p2: prp} {t: ti}
        pf p1 and p2 @ t
     -> pf p1 @ t.
and|er: {p1: prp} {p2: prp} {t: ti}
        pf p1 and p2 @ t
     -> pf p2 @ t.

or|il: {p1: prp} {p2: prp} {t: ti}
       pf p1 @ t
    -> pf p1 or p2 @ t.
or|ir: {p1: prp} {p2: prp} {t: ti}
       pf p2 @ t
    -> pf p1 or p2 @ t.
or|e:  {p1: prp} {p2: prp} {p': prp} {t: ti} {t': ti}
       pf p1 or p2 @ t
    -> (pf p1 @ t -> pf p' @ t') -> (pf p2 @ t -> pf p' @ t')
    -> pf p' @ t'.

imp|i: {p1: prp} {p2: prp} {t: ti}
       (pf p1 @ t -> pf p2 @ t)
    -> pf p1 imp p2 @ t.
imp|e: {p1: prp} {p2: prp} {t: ti}
       pf p1 imp p2 @ t -> pf p1 @ t
    -> pf p2 @ t.

% - truth: quantifiers -

all|i: {tau: ty} {rho: ri} {p: exp tau -> prp} {t: ti}
       ({a: par tau} pf ri/ofE tau (par/` tau a) rho -> pf p (par/` tau a) @ t)
    -> pf all tau rho p @ t.
all|e: {tau: ty} {rho: ri} {e: exp tau} {p: exp tau -> prp} {t: ti}
       pf all tau rho p @ t -> pf ri/ofE tau e rho
    -> pf p e @ t.

some|i: {tau: ty} {rho: ri} {e: exp tau} {p: exp tau -> prp} {t: ti}
        pf ri/ofE tau e rho -> pf p e @ t
     -> pf some tau rho p @ t.
some|e: {tau: ty} {rho: ri} {p: exp tau -> prp} {p': prp} {t: ti} {t': ti}
        pf some tau rho p @ t
     -> ({a: par tau}
         pf ri/ofE tau (par/` tau a) rho -> pf p (par/` tau a) @ t
      -> pf p' @ t')
     -> pf p' @ t'.

% - truth: temporal operators -
```

```
nextT|i: {p: prp} {t: ti}
         pf p @ t +1t
      -> pf nextT p @ t.
nextT|e: {p: prp} {t: ti}
         pf nextT p @ t
      -> pf p @ t +1t.

allT|i: {p1: prp} {t: ti}
        ({t1: ti} pf t <=t t1 -> pf p1 @ t1)
     -> pf allT p1 @ t.
allT|e: {p1: prp} {t: ti} {t1: ti}
        pf allT p1 @ t -> pf t <=t t1
     -> pf p1 @ t1.

untilT|i: {p1: prp} {p2: prp} {t: ti} {t2: ti}
          pf t <=t t2 -> pf at/@ov p1 t t2 -> pf p2 @ t2
       -> pf p1 untilT p2 @ t.
untilT|e: {p1: prp} {p2: prp} {p': prp} {t: ti} {t': ti}
          pf p1 untilT p2 @ t
       -> ({t2: ti}
           pf t <=t t2 -> pf at/@ov p1 t t2 -> pf p2 @ t2 -> pf p' @ t')
       -> pf p' @ t'.

someT|:   {p: prp} {t: ti}
          pf someT p eqv true untilT p @ t.
unlessT|: {p1: prp} {p2: prp} {t: ti}
          pf p1 unlessT p2 eqv allT p1 or p1 untilT p2 @ t.

% - truth: finite intervals -

@ov|i: {p: prp} {t1: ti} {t2: ti}
       ({t: ti} pf t1 <=t t -> pf t +1t <=t t2 -> pf p @ t)
    -> pf at/@ov p t1 t2.
@ov|e: {p: prp} {t: ti} {t1: ti} {t2: ti}
       pf at/@ov p t1 t2 -> pf t1 <=t t -> pf t +1t <=t t2
    -> pf p @ t.
```

B.1.3.8 Restricted Truth

```
uat/@|i~: {p: prp} {t: ti}
          pf p @ t
       -> pf p @_ t.
uat/@|e: {p: prp} {t: ti}
         pf p @_ t
      -> pf p @ t.
```

B.1.3.9 Functions

```
% - semantics -

fun/app1#|i~: {tau1: ty} {tau: ty}
              {f: fun/1 tau1 tau}
              {c1: con tau1} {c: con tau}
              ({t: ti}
               pf eq/eq tau
                       (fun/app1 tau1 tau f (con/` tau1 c1))
                       (con/` tau c) @ t)
           -> pf fun/app1# tau1 tau f c1 c.
fun/app2#|i~: {tau1: ty} {tau2: ty} {tau: ty}
              {f: fun/2 tau1 tau2 tau}
              {c1: con tau1} {c2: con tau2} {c: con tau}
              ({t: ti}
               pf eq/eq tau
```

```
                        (fun/app2 tau1 tau2 tau
                                  f (con/‘ tau1 c1) (con/‘ tau2 c2))
                        (con/‘ tau c) @ t)
           -> pf fun/app2# tau1 tau2 tau f c1 c2 c.
fun/app3#|i~: {tau1: ty} {tau2: ty} {tau3: ty} {tau: ty}
              {f: fun/3 tau1 tau2 tau3 tau}
              {c1: con tau1} {c2: con tau2} {c3: con tau3} {c: con tau}
              ({t: ti}
               pf eq/eq tau
                        (fun/app3
                           tau1 tau2 tau3 tau
                           f (con/‘ tau1 c1) (con/‘ tau2 c2) (con/‘ tau3 c3))
                        (con/‘ tau c) @ t)
           -> pf fun/app3# tau1 tau2 tau3 tau f c1 c2 c3 c.

% - rigidity -

ri/ofE|fun/app1: {tau1: ty} {tau: ty}
                 {f: fun/1 tau1 tau} {e1: exp tau1}
                 pf ri/ofE tau1 e1 ri/+
              -> pf ri/ofE tau (fun/app1 tau1 tau f e1) ri/+.
ri/ofE|fun/app2: {tau1: ty} {tau2: ty} {tau: ty}
                 {f: fun/2 tau1 tau2 tau} {e1: exp tau1} {e2: exp tau2}
                 pf ri/ofE tau1 e1 ri/+ -> pf ri/ofE tau2 e2 ri/+
              -> pf ri/ofE tau (fun/app2 tau1 tau2 tau f e1 e2) ri/+.
ri/ofE|fun/app3: {tau1: ty} {tau2: ty} {tau3: ty} {tau: ty}
                 {f: fun/3 tau1 tau2 tau3 tau}
                 {e1: exp tau1} {e2: exp tau2} {e3: exp tau3}
                 pf ri/ofE tau1 e1 ri/+ -> pf ri/ofE tau2 e2 ri/+
              -> pf ri/ofE tau3 e3 ri/+
              -> pf ri/ofE tau (fun/app3 tau1 tau2 tau3 tau f e1 e2 e3) ri/+.

% - rewriting -

rew/step0E|fun/app1: {tau1: ty} {tau: ty}
                     {f: fun/1 tau1 tau}
                     {c1: con tau1} {c’: con tau}
                     pf fun/app1# tau1 tau f c1 c’
                  -> pf rew/step0E tau
                                  (fun/app1 tau1 tau f (con/‘ tau1 c1))
                                  (con/‘ tau c’).
rew/step0E|fun/app2: {tau1: ty} {tau2: ty} {tau: ty}
                     {f: fun/2 tau1 tau2 tau}
                     {c1: con tau1} {c2: con tau2} {c’: con tau}
                     pf fun/app2# tau1 tau2 tau f c1 c2 c’
                  -> pf rew/step0E tau
                                  (fun/app2 tau1 tau2 tau
                                            f
                                            (con/‘ tau1 c1)
                                            (con/‘ tau2 c2))
                                  (con/‘ tau c’).
rew/step0E|fun/app3: {tau1: ty} {tau2: ty} {tau3: ty} {tau: ty}
                     {f: fun/3 tau1 tau2 tau3 tau}
                     {c1: con tau1} {c2: con tau2} {c3: con tau3} {c’: con tau}
                     pf fun/app3# tau1 tau2 tau3 tau f c1 c2 c3 c’
                  -> pf rew/step0E tau
                                  (fun/app3 tau1 tau2 tau3 tau
                                            f
                                            (con/‘ tau1 c1)
                                            (con/‘ tau2 c2)
                                            (con/‘ tau3 c3))
                                  (con/‘ tau c’).

rew/normE|fun/app1: {tau: ty} {tau1: ty}
```

```
                    {f: fun/1 tau1 tau}
                    {e1: exp tau1}
                    {e1': exp tau1} {e'': exp tau}
                    pf rew/tailE tau (fun/app1 tau1 tau f e1') e''
                 -> pf rew/normE tau1 e1 e1'
                 -> pf rew/normE tau (fun/app1 tau1 tau f e1) e''.
rew/normE|fun/app2: {tau: ty} {tau1: ty} {tau2: ty}
                    {f: fun/2 tau1 tau2 tau}
                    {e1: exp tau1} {e2: exp tau2}
                    {e1': exp tau1} {e2': exp tau2} {e'': exp tau}
                    pf rew/tailE tau (fun/app2 tau1 tau2 tau f e1' e2') e''
                 -> pf rew/normE tau1 e1 e1' -> pf rew/normE tau2 e2 e2'
                 -> pf rew/normE tau (fun/app2 tau1 tau2 tau f e1 e2) e''.
rew/normE|fun/app3: {tau: ty} {tau1: ty} {tau2: ty} {tau3: ty}
                    {f: fun/3 tau1 tau2 tau3 tau}
                    {e1: exp tau1} {e2: exp tau2} {e3: exp tau3}
                    {e1': exp tau1} {e2': exp tau2} {e3': exp tau3}
                    {e'': exp tau}
                    pf rew/tailE tau
                                 (fun/app3 tau1 tau2 tau3 tau f e1' e2' e3')
                                 e''
                 -> pf rew/normE tau1 e1 e1' -> pf rew/normE tau2 e2 e2'
                 -> pf rew/normE tau3 e3 e3'
                 -> pf rew/normE tau
                                 (fun/app3 tau1 tau2 tau3 tau f e1 e2 e3)
                                 e''.
```

B.1.3.10 Relations

```
% - semantics -

rel/app1#|i~: {tau1: ty}
              {r^: rel/1 tau1} {c1: con tau1}
              ({t: ti}
               pf rel/app1 tau1 r^ (con/` tau1 c1) @ t)
           -> pf rel/app1# tau1 r^ c1.
rel/app2#|i~: {tau1: ty} {tau2: ty}
              {r^: rel/2 tau1 tau2} {c1: con tau1} {c2: con tau2}
              ({t: ti}
               pf rel/app2 tau1 tau2 r^ (con/` tau1 c1) (con/` tau2 c2) @ t)
           -> pf rel/app2# tau1 tau2 r^ c1 c2.

% these may be derivable
% no rel/app0#|not_not
rel/app1#|not_not: {tau1: ty}
                   {r^: rel/1 tau1} {c1: con tau1}
                   pf rel/app1# tau1 r^ c1
                -> pf rel/app1# tau1 (rel/not1 tau1 (rel/not1 tau1 r^)) c1.
rel/app2#|not_not: {tau1: ty} {tau2: ty}
                   {r^: rel/2 tau1 tau2} {c1: con tau1} {c2: con tau2}
                   pf rel/app2# tau1 tau2 r^ c1 c2
                -> pf rel/app2# tau1 tau2
                                (rel/not2 tau1 tau2 (rel/not2 tau1 tau2 r^))
                                c1
                                c2.

% - locality -

lo/of+|i_rel/app0: {tau: ty}
                   {r^: rel/0}
                   pf lo/of+ tau ([x: exp tau] rel/app0 r^).
lo/of+|i_rel/app1: {tau: ty} {tau1: ty}
                   {r^: rel/1 tau1} {e1: exp tau -> exp tau1}
                   pf lo/of+ tau ([x: exp tau] rel/app1 tau1 r^ (e1 x)).
```

```
lo/of+|i_rel/app2: {tau: ty} {tau1: ty} {tau2: ty}
                   {r^: rel/2 tau1 tau2}
                   {e1: exp tau -> exp tau1} {e2: exp tau -> exp tau2}
                   pf lo/of+
                        tau
                        ([x: exp tau] rel/app2 tau1 tau2 r^ (e1 x) (e2 x)).

% - rigidity -

ri/of|rel/app0: {r^: rel/0} {rho: ri}
                pf ri/of (rel/app0 r^) rho.
ri/of|rel/app1: {tau1: ty}
                {r^: rel/1 tau1} {e1: exp tau1} {rho: ri}
                pf ri/ofE tau1 e1 rho
             -> pf ri/of (rel/app1 tau1 r^ e1) rho.
ri/of|rel/app2: {tau1: ty} {tau2: ty}
                {r^: rel/2 tau1 tau2} {e1: exp tau1} {e2: exp tau2} {rho: ri}
                pf ri/ofE tau1 e1 rho -> pf ri/ofE tau2 e2 rho
             -> pf ri/of (rel/app2 tau1 tau2 r^ e1 e2) rho.

% - rewriting -

rew/step0|rel/app1_true:  {tau1: ty}
                          {r^: rel/1 tau1} {c1: con tau1}
                          pf rel/app1# tau1 r^ c1
                       -> pf rew/step0 (rel/app1 tau1 r^ (con/` tau1 c1))
                                       true.
rew/step0|rel/app1_false: {tau1: ty}
                          {r^: rel/1 tau1} {c1: con tau1}
                          pf rel/app1# tau1 (rel/not1 tau1 r^) c1
                       -> pf rew/step0 (rel/app1 tau1 r^ (con/` tau1 c1))
                                       false.
rew/step0|rel/app2_true:  {tau1: ty} {tau2: ty}
                          {r^: rel/2 tau1 tau2} {c1: con tau1} {c2: con tau2}
                          pf rel/app2# tau1 tau2 r^ c1 c2
                       -> pf rew/step0 (rel/app2 tau1 tau2
                                                 r^
                                                 (con/` tau1 c1)
                                                 (con/` tau2 c2))
                                       true.
rew/step0|rel/app2_false: {tau1: ty} {tau2: ty}
                          {r^: rel/2 tau1 tau2} {c1: con tau1} {c2: con tau2}
                          pf rel/app2# tau1 tau2 (rel/not2 tau1 tau2 r^) c1 c2
                       -> pf rew/step0 (rel/app2 tau1 tau2
                                                 r^
                                                 (con/` tau1 c1)
                                                 (con/` tau2 c2))
                                       false.

rew/norm|rel/app0: {r^: rel/0}
                   pf rew/norm (rel/app0 r^) (rel/app0 r^).
rew/norm|rel/app1: {tau1: ty}
                   {r^: rel/1 tau1} {e1: exp tau1}
                   {e1': exp tau1}
                   {p'': prp}
                   pf rew/tail (rel/app1 tau1 r^ e1') p''
                -> pf rew/normE tau1 e1 e1'
                -> pf rew/norm (rel/app1 tau1 r^ e1) p''.
rew/norm|rel/app2: {tau1: ty} {tau2: ty}
                   {r^: rel/2 tau1 tau2} {e1: exp tau1} {e2: exp tau2}
                   {e1': exp tau1} {e2': exp tau2}
                   {p'': prp}
                   pf rew/tail (rel/app2 tau1 tau2 r^ e1' e2') p''
                -> pf rew/normE tau1 e1 e1' -> pf rew/normE tau2 e2 e2'
```

```
              -> pf rew/norm (rel/app2 tau1 tau2 r^ e1 e2) p''.

% - truth -

true|i: {t: ti}
        pf true @ t.

rel/case0: {r^: rel/0} {t1: ti}
           {p: prp} {t: ti}
           (pf rel/app0 r^ @ t1 -> pf p @ t)
        -> (pf rel/app0 (rel/not0 r^) @ t1 -> pf p @ t)
        -> pf p @ t.
rel/case1: {tau1: ty}
           {r^: rel/1 tau1} {e1: exp tau1} {t1: ti}
           {p: prp} {t: ti}
           (pf rel/app1 tau1 r^ e1 @ t1 -> pf p @ t)
        -> (pf rel/app1 tau1 (rel/not1 tau1 r^) e1 @ t1 -> pf p @ t)
        -> pf p @ t.
rel/case2: {tau1: ty} {tau2: ty}
           {r^: rel/2 tau1 tau2} {e1: exp tau1} {e2: exp tau2} {t1: ti}
           {p: prp} {t: ti}
           (   pf rel/app2 tau1 tau2 r^ e1 e2 @ t1
            -> pf p @ t)
        -> (   pf rel/app2 tau1 tau2 (rel/not2 tau1 tau2 r^) e1 e2 @ t1
            -> pf p @ t)
        -> pf p @ t.

rel/contr0: {r^: rel/0} {t1: ti}
            {p: prp} {t: ti}
            pf rel/app0 r^ @ t1
         -> pf rel/app0 (rel/not0 r^) @ t1
         -> pf p @ t.
rel/contr1: {tau1: ty}
            {r^: rel/1 tau1} {e1: exp tau1} {t1: ti}
            {p: prp} {t: ti}
            pf rel/app1 tau1 r^ e1 @ t1
         -> pf rel/app1 tau1 (rel/not1 tau1 r^) e1 @ t1
         -> pf p @ t.
rel/contr2: {tau1: ty} {tau2: ty}
            {r^: rel/2 tau1 tau2} {e1: exp tau1} {e2: exp tau2} {t1: ti}
            {p: prp} {t: ti}
            pf rel/app2 tau1 tau2 r^ e1 e2 @ t1
         -> pf rel/app2 tau1 tau2 (rel/not2 tau1 tau2 r^) e1 e2 @ t1
         -> pf p @ t.
```

B.1.3.11 Equality

```
% - locality -

lo/of+|e: {tau: ty}
          {e: exp tau} {e': exp tau} {p: exp tau -> prp} {t: ti}
          pf lo/of+ tau p -> pf eq/eq tau e e' @ t -> pf p e @ t
       -> pf p e' @ t.

% - rewriting -

% this one may be derivable when the top-level structure of e is known
rew/tailE|e: {tau: ty}
             {e: exp tau} {e': exp tau} {t: ti}
             pf rew/tailE tau e e'
          -> pf eq/eq tau e e' @ t.
rew/normE|e: {tau: ty}
             {e: exp tau} {e': exp tau} {t: ti}
             pf rew/normE tau e e'
```

```
          -> pf eq/eq tau e e' @ t.

% - truth -

eq|i_ref:  {tau: ty}
           {e: exp tau} {t: ti}
           pf eq/eq tau e e @ t.
eq|i_some: {tau: ty}
           {e: exp tau} {rho: ri} {p': prp} {t': ti} {t: ti}
           ({a: par tau}
            pf ri/ofE tau (par/` tau a) rho
          -> pf eq/eq tau (par/` tau a) e @ t
          -> pf p' @ t')
         -> pf p' @ t'.
eq|e_cong: {tau: ty}
           {e: exp tau} {e': exp tau} {p: exp tau -> prp} {t: ti}
           ({t1: ti} pf eq/eq tau e e' @ t1) -> pf p e @ t
         -> pf p e' @ t.
```

B.1.3.12 Equivalence

```
% - locality -

lo/of+|i_eqv: {tau: ty} {p1: exp tau -> prp} {p2: exp tau -> prp}
              pf lo/of+ tau p1 -> pf lo/of+ tau p2
           -> pf lo/of+ tau ([a: exp tau] p1 a eqv p2 a).

% - rigidity -

ri/of|i_eqv: {p1: prp} {p2: prp} {rho: ri}
             pf ri/of p1 rho -> pf ri/of p2 rho
          -> pf ri/of (p1 eqv p2) rho.

% - rewriting -

rew/norm|i_eqv: {p1: prp} {p2: prp} {p1': prp} {p2': prp} {p'': prp}
                pf rew/tail (p1' eqv p2') p''
             -> pf rew/norm p1 p1' -> pf rew/norm p2 p2'
             -> pf rew/norm (p1 eqv p2) p''.

% these two may be derivable when the top-level structure of p is known
rew/tail|e: {p: prp} {p': prp} {t: ti}
            pf rew/tail p p'
         -> pf p eqv p' @ t.
rew/norm|e: {p: prp} {p': prp} {t: ti}
            pf rew/norm p p'
         -> pf p eqv p' @ t.

% - truth -

eqv|i: {p1: prp} {p2: prp} {t: ti}
       (pf p1 @ t -> pf p2 @ t) -> (pf p2 @ t -> pf p1 @ t)
    -> pf p1 eqv p2 @ t.

eqv|el: {p1: prp} {p2: prp} {t: ti}
        pf p1 eqv p2 @ t -> pf p1 @ t
     -> pf p2 @ t.

eqv|er: {p1: prp} {p2: prp} {t: ti}
        pf p1 eqv p2 @ t -> pf p2 @ t
     -> pf p1 @ t.
```

B.1.3.13 Pairs

```
pair/make|inj:
```

```
  {tau1: ty} {tau2: ty} {t: ti}
  pf fun/inj2 tau1 tau2 (pair tau1 tau2) (pair/#make tau1 tau2) @ t.

pair/left|make:  {tau1: ty} {tau2: ty}
                 {e1: exp tau1} {e2: exp tau2} {t: ti}
                 pf eq/eq tau1
                          (pair/left tau1 tau2 (pair/make tau1 tau2 e1 e2))
                          e1 @ t.
pair/right|make: {tau1: ty} {tau2: ty}
                 {e1: exp tau1} {e2: exp tau2} {t: ti}
                 pf eq/eq tau2
                          (pair/right tau1 tau2 (pair/make tau1 tau2 e1 e2))
                          e2 @ t.
```

B.1.3.14 Triples

```
trip/make|:
  {tau1: ty} {tau2: ty} {tau3: ty}
  {e1: exp tau1} {e2: exp tau2} {e3: exp tau3}
  {t: ti}
  pf eq/eq
       (trip tau1 tau2 tau3)
       (trip/make tau1 tau2 tau3 e1 e2 e3)
       (pair/make (pair tau1 tau2) tau3 (pair/make tau1 tau2 e1 e2) e3) @ t.
trip/left|:
  {tau1: ty} {tau2: ty} {tau3: ty}
  {e: exp (trip tau1 tau2 tau3)}
  {t: ti}
  pf eq/eq
       tau1
       (trip/left tau1 tau2 tau3 e)
       (pair/left tau1 tau2 (pair/left (pair tau1 tau2) tau3 e)) @ t.
trip/mid|:
  {tau1: ty} {tau2: ty} {tau3: ty}
  {e: exp (trip tau1 tau2 tau3)}
  {t: ti}
  pf eq/eq
       tau2
       (trip/mid tau1 tau2 tau3 e)
       (pair/right tau1 tau2 (pair/left (pair tau1 tau2) tau3 e)) @ t.
trip/right|:
  {tau1: ty} {tau2: ty} {tau3: ty}
  {e: exp (trip tau1 tau2 tau3)}
  {t: ti}
  pf eq/eq
       tau3
       (trip/right tau1 tau2 tau3 e)
       (pair/right (pair tau1 tau2) tau3 e) @ t.
```

B.1.3.15 Lists

```
% - semantics -

% no fun/app2#|list

rel/app2#|eq_list:
  {tau: ty}
  pf rel/app2# (list tau) (list tau)
               (eq/#eq (list tau)) (list/empty tau) (list/empty tau).
% no rel/app2#|neq_list

% - truth -

list/cons|inj:
```

```
  {tau: ty} {t: ti}
  pf fun/inj2 tau (list tau) (list tau) (list/#cons tau) @ t.

list/head|cons:
  {tau: ty}
  {e1: exp tau} {e2: exp (list tau)} {t: ti}
  pf eq/eq tau (list/head tau (list/cons tau e1 e2)) e1 @ t.
list/tail|cons:
  {tau: ty}
  {e1: exp tau} {e2: exp (list tau)} {t: ti}
  pf eq/eq (list tau) (list/tail tau (list/cons tau e1 e2)) e2 @ t.
```

B.2 Machine Model

B.2.1 Abstract Syntax

B.2.1.1 Machine Words

```
% - types -

wd: ty.

% - constants -

%{
   A word constant is represented by a member of its equivalence class
   modulo 2^32.  Word operations normalize results between zero and 2^31-1, so
   many pattern matchings assume that word values and constants are normalized
   within this range.  This constructor should only be applied to normalized
   values, but this is not enforced.
}%
wd/#: integer -> con wd.

wd/0  = wd/# 0.
wd/1  = wd/# 1.
wd/2  = wd/# 2.
wd/3  = wd/# 3.
wd/4  = wd/# 4.
wd/5  = wd/# 5.
wd/6  = wd/# 6.
wd/7  = wd/# 7.
wd/8  = wd/# 8.
wd/9  = wd/# 9.
wd/~1 = wd/# 4294967295.
wd/~2 = wd/# 4294967294.
wd/~3 = wd/# 4294967293.
wd/~4 = wd/# 4294967292.

wd/`  = con/` wd.
wd/`# = [n: integer] wd/` (wd/# n).

wd/`0  = wd/` wd/0.
wd/`1  = wd/` wd/1.
wd/`~1 = wd/` wd/~1.

% - functions -

wd/#zf: fun/1 wd wd.
wd/#sf: fun/1 wd wd.

wd/#neg: fun/1 wd wd.

wd/#add: fun/2 wd wd wd.
```

```
wd/#sub: fun/2 wd wd wd.
wd/#mul: fun/2 wd wd wd.

wd/#div: fun/3 wd wd wd wd.
wd/#rem: fun/3 wd wd wd wd.

wd/#and: fun/2 wd wd wd.
wd/#or:  fun/2 wd wd wd.
wd/#xor: fun/2 wd wd wd.

wd/#not: fun/1 wd wd.

wd/zf = fun/app1 wd wd wd/#zf.
wd/sf = fun/app1 wd wd wd/#sf.

wd/neg = fun/app1 wd wd wd/#neg.

wd/add = fun/app2 wd wd wd wd/#add.
wd/sub = fun/app2 wd wd wd wd/#sub.
wd/mul = fun/app2 wd wd wd wd/#mul.

wd/div = fun/app3 wd wd wd wd wd/#div.
wd/rem = fun/app3 wd wd wd wd wd/#rem.

wd/and = fun/app2 wd wd wd wd/#and.
wd/or  = fun/app2 wd wd wd wd/#or.
wd/xor = fun/app2 wd wd wd wd/#xor.

wd/not = fun/app1 wd wd wd/#not.

% - relations -

wd/#lt:  rel/2 wd wd.
wd/#ltu: rel/2 wd wd.
wd/#inc: rel/2 wd wd.

wd/#geq  = rel/not2 wd wd wd/#lt.
wd/#gequ = rel/not2 wd wd wd/#ltu.
wd/#ninc = rel/not2 wd wd wd/#inc.

wd/eq   = rel/app2 wd wd (eq/#eq wd).
wd/neq  = rel/app2 wd wd (eq/#neq wd).
wd/lt   = rel/app2 wd wd wd/#lt.
wd/geq  = rel/app2 wd wd wd/#geq.
wd/ltu  = rel/app2 wd wd wd/#ltu.
wd/gequ = rel/app2 wd wd wd/#gequ.
wd/inc  = rel/app2 wd wd wd/#inc.
wd/ninc = rel/app2 wd wd wd/#ninc.

% - exports -

0w  = wd/0.
1w  = wd/1.
~1w = wd/~1.

`0w  = wd/`0.
`1w  = wd/`1.
`~1w = wd/`~1.
```

B.2.1.2 Arithmetic Operators

```
% - types -

op/1: ty.
```

```
op/2: ty.
op/3: ty.

% - constants -

op/add:  con op/2.
op/sub:  con op/2.
op/imul: con op/2.
op/inc:  con op/1.
op/dec:  con op/1.
op/neg:  con op/1.

op/idiv: con op/3.
op/irem: con op/3.

op/and: con op/2.
op/or:  con op/2.
op/xor: con op/2.
op/not: con op/1.

op/sf2: con op/2.

op/`add  = con/` op/2 op/add.
op/`sub  = con/` op/2 op/sub.
op/`imul = con/` op/2 op/imul.
op/`inc  = con/` op/1 op/inc.
op/`dec  = con/` op/1 op/dec.
op/`neg  = con/` op/1 op/neg.

op/`idiv = con/` op/3 op/idiv.
op/`irem = con/` op/3 op/irem.

op/`and = con/` op/2 op/and.
op/`or  = con/` op/2 op/or.
op/`xor = con/` op/2 op/xor.
op/`not = con/` op/1 op/not.

op/`sf2 = con/` op/2 op/sf2.

% - functions -

op/#app1: fun/2 op/1 wd wd.
op/#app2: fun/3 op/2 wd wd wd.
op/#app3: fun/3 op/3 (pair wd wd) wd wd.

op/#of1: fun/2 op/1 wd wd.
op/#of2: fun/3 op/2 wd wd wd.
op/#of3: fun/3 op/3 (pair wd wd) wd wd.
op/#cf1: fun/2 op/1 wd wd.
op/#cf2: fun/3 op/2 wd wd wd.
op/#cf3: fun/3 op/3 (pair wd wd) wd wd.

op/#selzf: fun/1 wd wd.
op/#selsf: fun/1 wd wd.
op/#selof: fun/1 wd wd.
op/#selcf: fun/1 wd wd.

op/#updf1: fun/3 op/1 wd wd wd.
op/#updf2: fun/3 op/2 wd (pair wd wd) wd.
op/#updf3: fun/3 op/3 wd (pair (pair wd wd) wd) wd.

op/app1 = fun/app2 op/1 wd wd op/#app1.
op/app2 = fun/app3 op/2 wd wd wd op/#app2.
op/app3 = [eop: exp op/3] [e1: exp wd] [e2: exp wd]
```

```
          fun/app3 op/3 (pair wd wd) wd wd op/#app3
                   eop (pair/make wd wd e1 e2).

op/of1 = fun/app2 op/1 wd wd op/#of1.
op/of2 = fun/app3 op/2 wd wd wd op/#of2.
op/of3 = [eop: exp op/3] [e1: exp wd] [e2: exp wd]
         fun/app3 op/3 (pair wd wd) wd wd op/#of3 eop (pair/make wd wd e1 e2).
op/cf1 = fun/app2 op/1 wd wd op/#cf1.
op/cf2 = fun/app3 op/2 wd wd wd op/#cf2.
op/cf3 = [eop: exp op/3] [e1: exp wd] [e2: exp wd]
         fun/app3 op/3 (pair wd wd) wd wd op/#cf3 eop (pair/make wd wd e1 e2).

op/selzf = fun/app1 wd wd op/#selzf.
op/selsf = fun/app1 wd wd op/#selsf.
op/selof = fun/app1 wd wd op/#selof.
op/selcf = fun/app1 wd wd op/#selcf.

op/updf1 = fun/app3 op/1 wd wd wd op/#updf1.
op/updf2 = [eop: exp op/2] [ef: exp wd] [e1: exp wd] [e2: exp wd]
           fun/app3 op/2 wd (pair wd wd) wd
                    op/#updf2 eop ef (pair/make wd wd e1 e2).
op/updf3 = [eop: exp op/3] [ef: exp wd] [e1: exp wd] [e2: exp wd] [e3: exp wd]
           fun/app3 op/3 wd (pair (pair wd wd) wd) wd
                    op/#updf3 eop ef (pair/make (pair wd wd) wd
                                                (pair/make wd wd e1 e2) e3).
```

B.2.1.3 Conditional Operators

```
% - types -

cop: ty.

% - constants -

cop/z:  con cop.
cop/s:  con cop.
cop/o:  con cop.
cop/c:  con cop.
cop/na: con cop.
cop/l:  con cop.
cop/ng: con cop.

cop/`z  = con/` cop cop/z.
cop/`s  = con/` cop cop/s.
cop/`o  = con/` cop cop/o.
cop/`c  = con/` cop cop/c.
cop/`na = con/` cop cop/na.
cop/`l  = con/` cop cop/l.
cop/`ng = con/` cop cop/ng.

% - functions -

cop/#self: fun/2 cop wd wd.
cop/#not:  fun/1 cop cop.

cop/self = fun/app2 cop wd wd cop/#self.
cop/not  = fun/app1 cop cop cop/#not.

% - constants -

cop/`nz = cop/not cop/`z.
cop/`ns = cop/not cop/`s.
cop/`no = cop/not cop/`o.
cop/`nc = cop/not cop/`c.
```

```
cop/‘a  = cop/not cop/‘na.
cop/‘nl = cop/not cop/‘l.
cop/‘g  = cop/not cop/‘ng.
```

B.2.1.4 Register Tokens

```
% - types -

greg: ty.

% - constants -

greg/eax: con greg.
greg/ebx: con greg.
greg/ecx: con greg.
greg/edx: con greg.
greg/esi: con greg.
greg/edi: con greg.
greg/ebp: con greg.
greg/esp: con greg.

greg/‘ = con/‘ greg.

greg/‘eax = greg/‘ greg/eax.
greg/‘ebx = greg/‘ greg/ebx.
greg/‘ecx = greg/‘ greg/ecx.
greg/‘edx = greg/‘ greg/edx.
greg/‘esi = greg/‘ greg/esi.
greg/‘edi = greg/‘ greg/edi.
greg/‘ebp = greg/‘ greg/ebp.
greg/‘esp = greg/‘ greg/esp.

% - exports -

‘eax = greg/‘eax.
‘ebx = greg/‘ebx.
‘ecx = greg/‘ecx.
‘edx = greg/‘edx.
‘esi = greg/‘esi.
‘edi = greg/‘edi.
‘ebp = greg/‘ebp.
‘esp = greg/‘esp.
```

B.2.1.5 Register Maps

```
% - types -

mapg: ty.

% - functions -

mapg/#sel: fun/2 mapg greg wd.
mapg/#upd: fun/3 mapg greg wd mapg.

mapg/sel = fun/app2 mapg greg wd mapg/#sel.
mapg/upd = fun/app3 mapg greg wd mapg mapg/#upd.

% - relations -

mapg/eq  = rel/app2 mapg mapg (eq/#eq mapg).
mapg/neq = rel/app2 mapg mapg (eq/#neq mapg).
```

B.2.1.6 Word Maps

```
% - types -
```

```
mapw: ty.

% - functions -

mapw/#sel:  fun/2 mapw wd wd.
mapw/#upd:  fun/3 mapw wd wd mapw.
mapw/#join: fun/3 mapw wd mapw mapw.

mapw/sel  = fun/app2 mapw wd wd mapw/#sel.
mapw/upd  = fun/app3 mapw wd wd mapw mapw/#upd.
mapw/join = fun/app3 mapw wd mapw mapw mapw/#join.

% - relations -

mapw/eq  = rel/app2 mapw mapw (eq/#eq mapw).
mapw/neq = rel/app2 mapw mapw (eq/#neq mapw).
```

B.2.1.7 Registers

```
% - parameters -

reg/pc: par wd.    % EIP
reg/f:  par wd.    % EFLAGS
reg/g:  par mapg.  % EAX EBX ECX EDX ESI EDI EBP ESP
reg/s:  par mapw.
reg/m:  par mapw.

reg/`pc = par/` wd reg/pc.
reg/`f  = par/` wd reg/f.
reg/`g  = par/` mapg reg/g.
reg/`s  = par/` mapw reg/s.
reg/`m  = par/` mapw reg/m.

reg/`g_sp = mapg/sel reg/`g `esp.

% - exports -

`pc = reg/`pc.
`f  = reg/`f.
`g  = reg/`g.
`s  = reg/`s.
`m  = reg/`m.

`g_sp = reg/`g_sp.
```

B.2.1.8 States

```
% - types -

state/fsm = trip wd mapw mapw.
state     = trip wd mapg state/fsm.

% - functions -

state/make
= [epc: exp wd] [ef: exp wd] [eg: exp mapg] [es: exp mapw] [em: exp mapw]
  trip/make wd mapg state/fsm epc eg (trip/make wd mapw mapw ef es em).

state/pc = trip/left wd mapg state/fsm.
state/f  = [e: exp state]
           trip/left wd mapw mapw (trip/right wd mapg state/fsm e).
state/g  = trip/mid wd mapg state/fsm.
state/s  = [e: exp state]
```

```
          trip/mid wd mapw mapw (trip/right wd mapg state/fsm e).
state/m  = [e: exp state]
          trip/right wd mapw mapw (trip/right wd mapg state/fsm e).

state/eax = [e: exp state] mapg/sel (state/g e) `eax.
state/ebp = [e: exp state] mapg/sel (state/g e) `ebp.
state/esp = [e: exp state] mapg/sel (state/g e) `esp.

% - relations -

state/eq = eq/eq state.

% - expressions -

state/`ss = state/make `pc `f `g `s `m.

% - exports -

`ss = state/`ss.
```

B.2.1.9 Memory Addresses

```
% - types -

ma: ty.

% - functions -

ma/#d: fun/1 wd ma.
ma/#r: fun/3 greg wd ma ma.

ma/#addr: fun/2 mapg ma wd.

ma/d = fun/app1 wd ma ma/#d.
ma/r = fun/app3 greg wd ma ma ma/#r.

ma/addr = fun/app2 mapg ma wd ma/#addr.
```

B.2.1.10 Effective Addresses

```
% - types -

ea: ty.

% - functions -

ea/#i: fun/1 wd ea.
ea/#r: fun/1 greg ea.
ea/#s: fun/1 ma ea.
ea/#m: fun/1 ma ea.

ea/#addr: fun/2 state ea wd.
ea/#sel:  fun/2 state ea wd.
ea/#updg: fun/3 state ea wd mapg.
ea/#upds: fun/3 state ea wd mapw.
ea/#updm: fun/3 state ea wd mapw.

ea/i = fun/app1 wd ea ea/#i.
ea/r = fun/app1 greg ea ea/#r.
ea/s = fun/app1 ma ea ea/#s.
ea/m = fun/app1 ma ea ea/#m.

ea/addr = fun/app2 state ea wd ea/#addr.
ea/sel  = fun/app2 state ea wd ea/#sel.
```

```
ea/updg = fun/app3 state ea wd mapg ea/#updg.
ea/upds = fun/app3 state ea wd mapw ea/#upds.
ea/updm = fun/app3 state ea wd mapw ea/#updm.
```

B.2.1.11 Instructions

```
% - types -

inst: ty.

% - functions -

% byte instructions will be mov8, etc.

inst/#mov:  fun/3 wd ea ea inst.
inst/#xchg: fun/3 wd ea greg inst.
inst/#lea:  fun/3 wd ea greg inst.
inst/#push: fun/2 wd ea inst.
inst/#pop:  fun/2 wd ea inst.
inst/#op1:  fun/3 wd op/1 ea inst.
inst/#op2:  fun/3 wd op/2 (pair ea ea) inst.
inst/#op2n: fun/3 wd op/2 (pair ea ea) inst.
inst/#op3:  fun/3 wd (pair op/3 op/3) (pair ea (pair greg greg)) inst.
inst/#jmp:  fun/2 wd ea inst.
inst/#j:    fun/3 wd cop wd inst.
inst/#call: fun/2 wd ea inst.
inst/#ret:  fun/1 wd inst.
% inst/#adc
% inst/#sbb

inst/#nextpc: fun/2 state inst wd.
inst/#nextf:  fun/2 state inst wd.
inst/#nextg:  fun/2 state inst mapg.
inst/#nexts:  fun/2 state inst mapw.
inst/#nextm:  fun/2 state inst mapw.

inst/mov  = fun/app3 wd ea ea inst inst/#mov.
inst/xchg = fun/app3 wd ea greg inst inst/#xchg.
inst/lea  = fun/app3 wd ea greg inst inst/#lea.
inst/push = fun/app2 wd ea inst inst/#push.
inst/pop  = fun/app2 wd ea inst inst/#pop.
inst/op1  = fun/app3 wd op/1 ea inst inst/#op1.
inst/op2  = [eni: exp wd] [eop: exp op/2] [eea1: exp ea] [eea2: exp ea]
            fun/app3 wd op/2 (pair ea ea) inst
                     inst/#op2
                     eni
                     eop
                     (pair/make ea ea eea1 eea2).
inst/op2n = [eni: exp wd] [eop: exp op/2] [eea1: exp ea] [eea2: exp ea]
            fun/app3 wd op/2 (pair ea ea) inst
                     inst/#op2n
                     eni
                     eop
                     (pair/make ea ea eea1 eea2).
inst/op3  = [eni: exp wd] [eop1: exp op/3] [eop2: exp op/3]
            [eea: exp ea] [er1: exp greg] [er2: exp greg]
            fun/app3 wd (pair op/3 op/3) (pair ea (pair greg greg)) inst
                     inst/#op3
                     eni
                     (pair/make op/3 op/3 eop1 eop2)
                     (pair/make ea (pair greg greg)
                                eea (pair/make greg greg er1 er2)).
inst/jmp  = fun/app2 wd ea inst inst/#jmp.
inst/j    = fun/app3 wd cop wd inst inst/#j.
```

```
inst/call = fun/app2 wd ea inst inst/#call.
inst/ret  = fun/app1 wd inst inst/#ret.

inst/cmp  = [eni: exp wd] inst/op2n eni op/‘sub.
inst/nop  = [eni: exp wd] inst/mov eni (ea/r ‘edi) (ea/r ‘edi).
inst/test = [eni: exp wd] inst/op2n eni op/‘and.

inst/nextpc = fun/app2 state inst wd inst/#nextpc.
inst/nextf  = fun/app2 state inst wd inst/#nextf.
inst/nextg  = fun/app2 state inst mapg inst/#nextg.
inst/nexts  = fun/app2 state inst mapw inst/#nexts.
inst/nextm  = fun/app2 state inst mapw inst/#nextm.

inst/next
= [ess: exp state] [ei^: exp inst]
  state/make (inst/nextpc ess ei^) (inst/nextf ess ei^)
             (inst/nextg ess ei^) (inst/nexts ess ei^) (inst/nextm ess ei^).
```

B.2.1.12 Programs

```
% - types -

prog: ty.

% - constants -

prog/pm: con prog.

prog/‘pm = con/‘ prog prog/pm.

% - relations -

prog/#fetch: rel/2 (pair prog wd) inst.

prog/fetch
= [ephi^: exp prog] [en: exp wd]
  rel/app2 (pair prog wd) inst prog/#fetch (pair/make prog wd ephi^ en).

% - exports -

‘pm = prog/‘pm.
```

B.2.2 Inference Rules

B.2.2.1 Machine Words

```
% - semantics -

rel/app2#|eq_wd:  {n1: integer} {n2: integer}
                  integer32/eq n1 n2
               -> pf rel/app2# wd wd (eq/#eq wd) (wd/# n1) (wd/# n2).
rel/app2#|neq_wd: {n1: integer} {n2: integer}
                  integer32/neq n1 n2
               -> pf rel/app2# wd wd (eq/#neq wd) (wd/# n1) (wd/# n2).

fun/app1#|wd/neg: {n: integer} {n’: integer}
                  integer32/sub 0 n n’
               -> pf fun/app1# wd wd wd/#neg (wd/# n) (wd/# n’).

fun/app2#|wd/add: {n1: integer} {n2: integer} {n’: integer}
                  integer32/add n1 n2 n’
               -> pf fun/app2# wd wd wd wd/#add (wd/# n1) (wd/# n2) (wd/# n’).
fun/app2#|wd/sub: {n1: integer} {n2: integer} {n’: integer}
                  integer32/sub n1 n2 n’
```

```
               -> pf fun/app2# wd wd wd wd/#sub (wd/# n1) (wd/# n2) (wd/# n').
fun/app2#|wd/mul: {n1: integer} {n2: integer} {n': integer}
                  integer32/mul n1 n2 n'
               -> pf fun/app2# wd wd wd wd/#mul (wd/# n1) (wd/# n2) (wd/# n').

fun/app2#|wd/and: {n1: integer} {n2: integer} {n': integer}
                  integer32/and n1 n2 n'
               -> pf fun/app2# wd wd wd wd/#and (wd/# n1) (wd/# n2) (wd/# n').
fun/app2#|wd/or:  {n1: integer} {n2: integer} {n': integer}
                  integer32/or n1 n2 n'
               -> pf fun/app2# wd wd wd wd/#or (wd/# n1) (wd/# n2) (wd/# n').
fun/app2#|wd/xor: {n1: integer} {n2: integer} {n': integer}
                  integer32/xor n1 n2 n'
               -> pf fun/app2# wd wd wd wd/#xor (wd/# n1) (wd/# n2) (wd/# n').

fun/app1#|wd/not: {n: integer} {n': integer}
                  integer32/xor n 4294967295 n'
               -> pf fun/app1# wd wd wd/#not (wd/# n) (wd/# n').

rel/app2#|wd/lt:   {n1: integer} {n2: integer}
                   integer32/gt n2 n1
                -> pf rel/app2# wd wd wd/#lt (wd/# n1) (wd/# n2).
rel/app2#|wd/geq:  {n1: integer} {n2: integer}
                   integer32/leq n2 n1
                -> pf rel/app2# wd wd wd/#geq (wd/# n1) (wd/# n2).
rel/app2#|wd/ltu:  {n1: integer} {n2: integer}
                   integer32/gtu n2 n1
                -> pf rel/app2# wd wd wd/#ltu (wd/# n1) (wd/# n2).
rel/app2#|wd/gequ: {n1: integer} {n2: integer}
                   integer32/lequ n2 n1
                -> pf rel/app2# wd wd wd/#gequ (wd/# n1) (wd/# n2).

% - truth -

wd|ind:
   {p: exp wd -> prp} {e: exp wd} {t: ti}
   ({a: par wd}
    pf ri/ofE wd (par/` wd a) ri/+
 -> ({e1: exp wd}
     pf ri/ofE wd e1 ri/+ -> pf wd/ltu e1 (par/` wd a) @ t -> pf p e1 @ t)
 -> pf p (par/` wd a) @ t)
-> pf p e @ t.

wd/zf|i1: {t: ti}
          pf wd/eq (wd/zf `0w) `1w @ t.
wd/zf|i0: {e: exp wd} {t: ti}
          pf wd/neq e `0w @ t
       -> pf wd/eq (wd/zf e) `0w @ t.
wd/zf|e1: {e: exp wd} {t: ti}
          pf wd/eq (wd/zf e) `1w @ t
       -> pf wd/eq e `0w @ t.
wd/zf|e0: {e: exp wd} {t: ti}
          pf wd/eq (wd/zf e) `0w @ t
       -> pf wd/neq e `0w @ t.
wd/sf|i1: {e: exp wd} {t: ti}
          pf wd/lt e `0w @ t
       -> pf wd/eq (wd/sf e) `1w @ t.
wd/sf|i0: {e: exp wd} {t: ti}
          pf wd/geq e `0w @ t
       -> pf wd/eq (wd/sf e) `0w @ t.
wd/sf|e1: {e: exp wd} {t: ti}
          pf wd/eq (wd/sf e) `1w @ t
       -> pf wd/lt e `0w @ t.
wd/sf|e0: {e: exp wd} {t: ti}
```

```
         pf wd/eq (wd/sf e) `0w @ t
      -> pf wd/geq e `0w @ t.

wd/add|mul_comm_ring:
  {t: ti}
  pf fun/comm_ring wd wd/#add 0w wd/#neg wd/#mul 1w @ t.

wd/mul|id_0:
  {e: exp wd} {t: ti}
  pf wd/eq (wd/mul e `0w) `0w @ t.

wd/inc|bool_latt:
  {t: ti}
  pf rel/bool_latt wd wd/#inc wd/#and wd/#or wd/0 wd/~1 wd/#not @ t.

wd/sub|:
  {e1: exp wd} {e2: exp wd} {t: ti}
  pf wd/eq (wd/sub e1 e2) (wd/add e1 (wd/neg e2)) @ t.
wd/xor|:
  {e1: exp wd} {e2: exp wd} {t: ti}
  pf wd/eq (wd/xor e1 e2)
           (wd/and (wd/or e1 e2) (wd/or (wd/not e1) (wd/not e2))) @ t.

wd/lt|str_tot_order: {t: ti} pf rel/str_tot_order wd wd/#lt @ t.
wd/lt|bot:           {t: ti} pf rel/bot wd wd/#lt (wd/# 2147483648) @ t.
wd/lt|top:           {t: ti} pf rel/top wd wd/#lt (wd/# 2147483647) @ t.

wd/ltu|str_tot_order: {t: ti} pf rel/str_tot_order wd wd/#ltu @ t.
wd/ltu|bot:           {t: ti} pf rel/bot wd wd/#ltu 0w @ t.
wd/ltu|top:           {t: ti} pf rel/top wd wd/#ltu ~1w @ t.

% implied by bool_latt
% wd/inc|order: {t: ti} pf rel/order wd wd/#inc @ t.
wd/inc|bot: {t: ti} pf rel/bot wd wd/#inc 0w @ t.
wd/inc|top: {t: ti} pf rel/top wd wd/#inc ~1w @ t.
```

B.2.2.2 Arithmetic Operators

```
% - semantics -

fun/app1#|op/selzf_1: {c: con wd} {c': con wd}
                      pf rel/app2# wd wd (eq/#neq wd) c' wd/0
                   -> pf fun/app2# wd wd wd wd/#and c (wd/# 64) c'
                   -> pf fun/app1# wd wd op/#selzf c wd/1.
fun/app1#|op/selzf_0: {c: con wd}
                      pf fun/app2# wd wd wd wd/#and c (wd/# 64) wd/0
                   -> pf fun/app1# wd wd op/#selzf c wd/0.
fun/app1#|op/selsf_1: {c: con wd} {c': con wd}
                      pf rel/app2# wd wd (eq/#neq wd) c' wd/0
                   -> pf fun/app2# wd wd wd wd/#and c (wd/# 128) c'
                   -> pf fun/app1# wd wd op/#selsf c wd/1.
fun/app1#|op/selsf_0: {c: con wd}
                      pf fun/app2# wd wd wd wd/#and c (wd/# 128) wd/0
                   -> pf fun/app1# wd wd op/#selsf c wd/0.
fun/app1#|op/selof_1: {c: con wd} {c': con wd}
                      pf rel/app2# wd wd (eq/#neq wd) c' wd/0
                   -> pf fun/app2# wd wd wd wd/#and c (wd/# 2048) c'
                   -> pf fun/app1# wd wd op/#selof c wd/1.
fun/app1#|op/selof_0: {c: con wd}
                      pf fun/app2# wd wd wd wd/#and c (wd/# 2048) wd/0
                   -> pf fun/app1# wd wd op/#selof c wd/0.
fun/app1#|op/selcf_1: {c: con wd} {c': con wd}
                      pf rel/app2# wd wd (eq/#neq wd) c' wd/0
                   -> pf fun/app2# wd wd wd wd/#and c (wd/# 1) c'
```

```
                   -> pf fun/app1# wd wd op/#selcf c wd/1.
fun/app1#|op/selcf_0: {c: con wd}
                      pf fun/app2# wd wd wd wd/#and c (wd/# 1) wd/0
                   -> pf fun/app1# wd wd op/#selcf c wd/0.

% don't need updf1/updf2/updf3, since ef is never constant

rel/app2#|eq_op/1:  {c: con op/1}
                    pf rel/app2# op/1 op/1 (eq/#eq op/1) c c.
rel/app2#|neq_op/1: {c1: con op/1} {c2: con op/1}
                    con/neq op/1 c1 c2
                 -> pf rel/app2# op/1 op/1 (eq/#neq op/1) c1 c2.

rel/app2#|eq_op/2:  {c: con op/2}
                    pf rel/app2# op/2 op/2 (eq/#eq op/2) c c.
rel/app2#|neq_op/2: {c1: con op/2} {c2: con op/2}
                    con/neq op/2 c1 c2
                 -> pf rel/app2# op/2 op/2 (eq/#neq op/2) c1 c2.

rel/app2#|eq_op/3:  {c: con op/3}
                    pf rel/app2# op/3 op/3 (eq/#eq op/3) c c.
rel/app2#|neq_op/3: {c1: con op/3} {c2: con op/3}
                    con/neq op/3 c1 c2
                 -> pf rel/app2# op/3 op/3 (eq/#neq op/3) c1 c2.

% - truth -

op/app2|op/add:  {e1: exp wd} {e2: exp wd} {t: ti}
                 pf wd/eq (op/app2 op/`add e1 e2) (wd/add e1 e2) @ t.
op/app2|op/sub:  {e1: exp wd} {e2: exp wd} {t: ti}
                 pf wd/eq (op/app2 op/`sub e1 e2) (wd/sub e1 e2) @ t.
op/app2|op/imul: {e1: exp wd} {e2: exp wd} {t: ti}
                 pf wd/eq (op/app2 op/`imul e1 e2) (wd/mul e1 e2) @ t.
op/app1|op/inc:  {e: exp wd} {t: ti}
                 pf wd/eq (op/app1 op/`inc e) (wd/add e `1w) @ t.
op/app1|op/dec:  {e: exp wd} {t: ti}
                 pf wd/eq (op/app1 op/`dec e) (wd/sub e `1w) @ t.
op/app1|op/neg:  {e: exp wd} {t: ti}
                 pf wd/eq (op/app1 op/`neg e) (wd/sub `0w e) @ t.

op/app3|op/idiv: {e1: exp wd} {e2: exp wd} {e3: exp wd} {t: ti}
                 pf wd/eq (op/app3 op/`idiv e1 e2 e3) (wd/div e1 e2 e3) @ t.
op/app3|op/irem: {e1: exp wd} {e2: exp wd} {e3: exp wd} {t: ti}
                 pf wd/eq (op/app3 op/`irem e1 e2 e3) (wd/rem e1 e2 e3) @ t.

op/app2|op/and: {e1: exp wd} {e2: exp wd} {t: ti}
                pf wd/eq (op/app2 op/`and e1 e2) (wd/and e1 e2) @ t.
op/app2|op/or:  {e1: exp wd} {e2: exp wd} {t: ti}
                pf wd/eq (op/app2 op/`or e1 e2) (wd/or e1 e2) @ t.
op/app2|op/xor: {e1: exp wd} {e2: exp wd} {t: ti}
                pf wd/eq (op/app2 op/`xor e1 e2) (wd/xor e1 e2) @ t.
op/app1|op/not: {e: exp wd} {t: ti}
                pf wd/eq (op/app1 op/`not e) (wd/xor e `~1w) @ t.

op/app2|op/sf2: {e1: exp wd} {e2: exp wd} {t: ti}
                pf wd/eq (op/app2 op/`sf2 e1 e2)
                         (wd/mul (wd/sf e2) wd/`~1) @ t.

% (sf (e1 xor e2) xor 1) and sf ((e1 addw e2) xor e1)
op/of2|op/add: {e1: exp wd} {e2: exp wd} {t: ti}
               pf wd/eq (op/of2 op/`add e1 e2)
                        (wd/and (wd/xor (wd/sf (wd/xor e1 e2)) `1w)
                                (wd/sf (wd/xor (wd/add e1 e2) e1))) @ t.
% sf (e1 xor e2) and sf ((e1 subw e2) xor e1)
```

```
op/of2|op/sub: {e1: exp wd} {e2: exp wd} {t: ti}
               pf wd/eq (op/of2 op/`sub e1 e2)
                        (wd/and (wd/sf (wd/xor e1 e2))
                                (wd/sf (wd/xor (wd/sub e1 e2) e1))) @ t.
op/of1|op/inc: {e: exp wd} {t: ti}
               pf wd/eq (op/of1 op/`inc e) (op/of2 op/`add e `1w) @ t.
op/of1|op/dec: {e: exp wd} {t: ti}
               pf wd/eq (op/of1 op/`dec e) (op/of2 op/`sub e `1w) @ t.
op/of1|op/neg: {e: exp wd} {t: ti}
               pf wd/eq (op/of1 op/`neg e) (op/of2 op/`sub `0w e) @ t.

% op/of3|op/idiv is undefined
% op/of3|op/irem is undefined

op/of2|op/and: {e1: exp wd} {e2: exp wd} {t: ti}
               pf wd/eq (op/of2 op/`and e1 e2) `0w @ t.
op/of2|op/or:  {e1: exp wd} {e2: exp wd} {t: ti}
               pf wd/eq (op/of2 op/`or e1 e2) `0w @ t.
op/of2|op/xor: {e1: exp wd} {e2: exp wd} {t: ti}
               pf wd/eq (op/of2 op/`xor e1 e2) `0w @ t.
op/of1|op/not: {e: exp wd} {t: ti}
               pf wd/eq (op/of1 op/`not e) `0w @ t.

op/of2|op/sf2: {e1: exp wd} {e2: exp wd} {t: ti}
               pf wd/eq (op/of2 op/`sf2 e1 e2) `0w @ t.

op/cf2|op/add_gequ: {e1: exp wd} {e2: exp wd} {t: ti}
                    pf     wd/eq (op/cf2 op/`add e1 e2) `0w
                       eqv wd/gequ (wd/add e1 e2) e1 @ t.
op/cf2|op/add_ltu:  {e1: exp wd} {e2: exp wd} {t: ti}
                    pf     wd/eq (op/cf2 op/`add e1 e2) `1w
                       eqv wd/ltu (wd/add e1 e2) e1 @ t.
op/cf2|op/sub_gequ: {e1: exp wd} {e2: exp wd} {t: ti}
                    pf     wd/eq (op/cf2 op/`sub e1 e2) `0w
                       eqv wd/gequ e1 e2 @ t.
op/cf2|op/sub_ltu:  {e1: exp wd} {e2: exp wd} {t: ti}
                    pf     wd/eq (op/cf2 op/`sub e1 e2) `1w
                       eqv wd/ltu e1 e2 @ t.
op/cf2|op/imul:     {e1: exp wd} {e2: exp wd} {t: ti}
                    pf wd/eq (op/cf2 op/`imul e1 e2)
                             (op/of2 op/`imul e1 e2) @ t.
op/cf1|op/inc:      {e: exp wd} {t: ti}
                    pf wd/eq (op/cf1 op/`inc e) (op/cf2 op/`add e `1w) @ t.
op/cf1|op/dec:      {e: exp wd} {t: ti}
                    pf wd/eq (op/cf1 op/`dec e) (op/cf2 op/`sub e `1w) @ t.
op/cf1|op/neg:      {e: exp wd} {t: ti}
                    pf wd/eq (op/cf1 op/`neg e) (op/cf2 op/`sub `0w e) @ t.

% op/cf3|op/idiv is undefined
% op/cf3|op/irem is undefined

op/cf2|op/and: {e1: exp wd} {e2: exp wd} {t: ti}
               pf wd/eq (op/cf2 op/`and e1 e2) `0w @ t.
op/cf2|op/or:  {e1: exp wd} {e2: exp wd} {t: ti}
               pf wd/eq (op/cf2 op/`or e1 e2) `0w @ t.
op/cf2|op/xor: {e1: exp wd} {e2: exp wd} {t: ti}
               pf wd/eq (op/cf2 op/`xor e1 e2) `0w @ t.
op/cf1|op/not: {e: exp wd} {t: ti}
               pf wd/eq (op/cf1 op/`not e) `0w @ t.

op/cf2|op/sf2: {e1: exp wd} {e2: exp wd} {t: ti}
               pf wd/eq (op/cf2 op/`sf2 e1 e2) `0w @ t.

op/selzf|op/updf1_inc:
```

```
  {ef: exp wd} {e: exp wd} {t: ti}
  pf wd/eq (op/selzf (op/updf1 op/‘inc ef e)) (wd/zf (op/app1 op/‘inc e)) @ t.
op/selzf|op/updf1_dec:
  {ef: exp wd} {e: exp wd} {t: ti}
  pf wd/eq (op/selzf (op/updf1 op/‘dec ef e)) (wd/zf (op/app1 op/‘dec e)) @ t.
op/selzf|op/updf1_neg:
  {ef: exp wd} {e: exp wd} {t: ti}
  pf wd/eq (op/selzf (op/updf1 op/‘neg ef e)) (wd/zf (op/app1 op/‘neg e)) @ t.
op/selzf|op/updf1_not:
  {ef: exp wd} {e: exp wd} {t: ti}
  pf wd/eq (op/selzf (op/updf1 op/‘not ef e)) (op/selzf ef) @ t.

op/selsf|op/updf1_inc:
  {ef: exp wd} {e: exp wd} {t: ti}
  pf wd/eq (op/selsf (op/updf1 op/‘inc ef e)) (wd/sf (op/app1 op/‘inc e)) @ t.
op/selsf|op/updf1_dec:
  {ef: exp wd} {e: exp wd} {t: ti}
  pf wd/eq (op/selsf (op/updf1 op/‘dec ef e)) (wd/sf (op/app1 op/‘dec e)) @ t.
op/selsf|op/updf1_neg:
  {ef: exp wd} {e: exp wd} {t: ti}
  pf wd/eq (op/selsf (op/updf1 op/‘neg ef e)) (wd/sf (op/app1 op/‘neg e)) @ t.
op/selsf|op/updf1_not:
  {ef: exp wd} {e: exp wd} {t: ti}
  pf wd/eq (op/selsf (op/updf1 op/‘not ef e)) (op/selsf ef) @ t.

op/selof|op/updf1_inc:
  {ef: exp wd} {e: exp wd} {t: ti}
  pf wd/eq (op/selof (op/updf1 op/‘inc ef e)) (op/of1 op/‘inc e) @ t.
op/selof|op/updf1_dec:
  {ef: exp wd} {e: exp wd} {t: ti}
  pf wd/eq (op/selof (op/updf1 op/‘dec ef e)) (op/of1 op/‘dec e) @ t.
op/selof|op/updf1_neg:
  {ef: exp wd} {e: exp wd} {t: ti}
  pf wd/eq (op/selof (op/updf1 op/‘neg ef e)) (op/of1 op/‘neg e) @ t.
op/selof|op/updf1_not:
  {ef: exp wd} {e: exp wd} {t: ti}
  pf wd/eq (op/selof (op/updf1 op/‘not ef e)) (op/selof ef) @ t.

op/selcf|op/updf1_inc:
  {ef: exp wd} {e: exp wd} {t: ti}
  pf wd/eq (op/selcf (op/updf1 op/‘inc ef e)) (op/selcf ef) @ t.
op/selcf|op/updf1_dec:
  {ef: exp wd} {e: exp wd} {t: ti}
  pf wd/eq (op/selcf (op/updf1 op/‘dec ef e)) (op/selcf ef) @ t.
op/selcf|op/updf1_neg:
  {ef: exp wd} {e: exp wd} {t: ti}
  pf wd/eq (op/selcf (op/updf1 op/‘neg ef e)) (op/cf1 op/‘neg e) @ t.
op/selcf|op/updf1_not:
  {ef: exp wd} {e: exp wd} {t: ti}
  pf wd/eq (op/selcf (op/updf1 op/‘not ef e)) (op/selcf ef) @ t.

op/selzf|op/updf2:
  {eop: exp op/2} {ef: exp wd} {e1: exp wd} {e2: exp wd} {t: ti}
  pf wd/eq (op/selzf (op/updf2 eop ef e1 e2)) (wd/zf (op/app2 eop e1 e2)) @ t.
op/selsf|op/updf2:
  {eop: exp op/2} {ef: exp wd} {e1: exp wd} {e2: exp wd} {t: ti}
  pf wd/eq (op/selsf (op/updf2 eop ef e1 e2)) (wd/sf (op/app2 eop e1 e2)) @ t.
op/selof|op/updf2:
  {eop: exp op/2} {ef: exp wd} {e1: exp wd} {e2: exp wd} {t: ti}
  pf wd/eq (op/selof (op/updf2 eop ef e1 e2)) (op/of2 eop e1 e2) @ t.
op/selcf|op/updf2:
  {eop: exp op/2} {ef: exp wd} {e1: exp wd} {e2: exp wd} {t: ti}
  pf wd/eq (op/selcf (op/updf2 eop ef e1 e2)) (op/cf2 eop e1 e2) @ t.
```

```
% op/selzf|op/updf3 is undefined
% op/selsf|op/updf3 is undefined
% op/selof|op/updf3 is undefined
% op/selcf|op/updf3 is undefined

op/of1|neq0: {eop: exp op/1} {e1: exp wd} {t: ti}
             pf wd/neq (op/of1 eop e1) `0w @ t
          -> pf wd/eq (op/of1 eop e1) `1w @ t.
op/of2|neq0: {eop: exp op/2} {e1: exp wd} {e2: exp wd} {t: ti}
             pf wd/neq (op/of2 eop e1 e2) `0w @ t
          -> pf wd/eq (op/of2 eop e1 e2) `1w @ t.
op/of3|neq0: {eop: exp op/3} {e1: exp wd} {e2: exp wd} {e3: exp wd} {t: ti}
             pf wd/neq (op/of3 eop e1 e2 e3) `0w @ t
          -> pf wd/eq (op/of3 eop e1 e2 e3) `1w @ t.
op/cf1|neq0: {eop: exp op/1} {e1: exp wd} {t: ti}
             pf wd/neq (op/cf1 eop e1) `0w @ t
          -> pf wd/eq (op/cf1 eop e1) `1w @ t.
op/cf2|neq0: {eop: exp op/2} {e1: exp wd} {e2: exp wd} {t: ti}
             pf wd/neq (op/cf2 eop e1 e2) `0w @ t
          -> pf wd/eq (op/cf2 eop e1 e2) `1w @ t.
op/cf3|neq0: {eop: exp op/3} {e1: exp wd} {e2: exp wd} {e3: exp wd} {t: ti}
             pf wd/neq (op/cf3 eop e1 e2 e3) `0w @ t
          -> pf wd/eq (op/cf3 eop e1 e2 e3) `1w @ t.
```

B.2.2.3 Conditional Operators

```
% - semantics -

rel/app2#|eq_cop:  {c: con cop}
                   pf rel/app2# cop cop (eq/#eq cop) c c.
rel/app2#|neq_cop: {c1: con cop} {c2: con cop}
                   con/neq cop c1 c2
                -> pf rel/app2# cop cop (eq/#neq cop) c1 c2.

% - truth -

cop/self|neq0:
   {ecop: exp cop} {ef: exp wd} {t: ti}
   pf wd/neq (cop/self ecop ef) `0w @ t
-> pf wd/eq (cop/self ecop ef) `1w @ t.

cop/self|not:
  {ecop: exp cop} {ef: exp wd} {t: ti}
  pf wd/eq (cop/self (cop/not ecop) ef) (wd/xor (cop/self ecop ef) `1w) @ t.

cop/not|not:
  {ecop: exp cop} {t: ti}
  pf eq/eq cop (cop/not (cop/not ecop)) ecop @ t.

cop/self|z:
  {ef: exp wd} {t: ti}
  pf wd/eq (cop/self cop/`z ef) (op/selzf ef) @ t.
cop/self|s:
  {ef: exp wd} {t: ti}
  pf wd/eq (cop/self cop/`s ef) (op/selsf ef) @ t.
cop/self|o:
  {ef: exp wd} {t: ti}
  pf wd/eq (cop/self cop/`o ef) (op/selof ef) @ t.
cop/self|c:
  {ef: exp wd} {t: ti}
  pf wd/eq (cop/self cop/`c ef) (op/selcf ef) @ t.
cop/self|na:
  {ef: exp wd} {t: ti}
  pf wd/eq (cop/self cop/`na ef) (wd/or (op/selzf ef) (op/selcf ef)) @ t.
```

```
cop/self|l:
  {ef: exp wd} {t: ti}
  pf wd/eq (cop/self cop/‘l ef) (wd/xor (op/selsf ef) (op/selof ef)) @ t.
cop/self|ng:
  {ef: exp wd} {t: ti}
  pf wd/eq (cop/self cop/‘ng ef)
           (wd/or (op/selzf ef) (cop/self cop/‘l ef)) @ t.
```

B.2.2.4 Register Tokens

```
% - semantics -

rel/app2#|eq_greg:  {c: con greg}
                    pf rel/app2# greg greg (eq/#eq greg) c c.
rel/app2#|neq_greg: {c1: con greg} {c2: con greg}
                    con/neq greg c1 c2
                 -> pf rel/app2# greg greg (eq/#neq greg) c1 c2.
```

B.2.2.5 Register Maps

```
mapg/sel|mc0: {em: exp mapg} {er: exp greg} {en: exp wd} {t: ti}
              pf wd/eq (mapg/sel (mapg/upd em er en) er) en @ t.
mapg/sel|mc1: {em: exp mapg} {er: exp greg} {en: exp wd} {er’: exp greg}
              {t: ti}
              pf eq/neq greg er’ er @ t
           -> pf wd/eq (mapg/sel (mapg/upd em er en) er’)
                       (mapg/sel em er’) @ t.
mapg/sel|ext: {em: exp mapg} {em’: exp mapg} {t: ti}
              ({ar: par greg}
               pf wd/eq (mapg/sel em (par/‘ greg ar))
                        (mapg/sel em’ (par/‘ greg ar)) @ t)
           -> pf mapg/eq em em’ @ t.
```

B.2.2.6 Word Maps

```
mapw/sel|mc0:  {em: exp mapw} {en1: exp wd} {en2: exp wd} {t: ti}
               pf wd/eq (mapw/sel (mapw/upd em en1 en2) en1) en2 @ t.
mapw/sel|mc1:  {em: exp mapw} {en1: exp wd} {en2: exp wd} {en1’: exp wd}
               {t: ti}
               pf wd/neq en1’ en1 @ t
            -> pf wd/eq (mapw/sel (mapw/upd em en1 en2) en1’)
                        (mapw/sel em en1’) @ t.
mapw/sel|ltu:  {em1: exp mapw} {em2: exp mapw}
               {en: exp wd} {en’: exp wd} {t: ti}
               pf wd/ltu en’ en @ t
            -> pf wd/eq (mapw/sel (mapw/join em1 en em2) en’)
                        (mapw/sel em1 en’) @ t.
mapw/sel|gequ: {em1: exp mapw} {em2: exp mapw}
               {en: exp wd} {en’: exp wd} {t: ti}
               pf wd/gequ en’ en @ t
            -> pf wd/eq (mapw/sel (mapw/join em1 en em2) en’)
                        (mapw/sel em2 en’) @ t.
mapw/sel|ext:  {em: exp mapw} {em’: exp mapw} {t: ti}
               ({an: par wd}
                pf wd/eq (mapw/sel em (par/‘ wd an))
                         (mapw/sel em’ (par/‘ wd an)) @ t)
            -> pf mapw/eq em em’ @ t.
```

B.2.2.7 Memory Addresses

```
ma/d|inj: {t: ti} pf fun/inj1 wd ma ma/#d @ t.
ma/r|inj: {t: ti} pf fun/inj3 greg wd ma ma ma/#r @ t.
```

```
ma/addr|ma/d:
  {eg: exp mapg} {end: exp wd} {t: ti}
  pf wd/eq (ma/addr eg (ma/d end)) end @ t.
ma/addr|ma/r:
  {eg: exp mapg} {eri: exp greg} {ens: exp wd} {ema': exp ma} {t: ti}
  pf wd/eq (ma/addr eg (ma/r eri ens ema'))
           (wd/add (wd/mul (mapg/sel eg eri) ens) (ma/addr eg ema')) @ t.
```

B.2.2.8 Effective Addresses

```
ea/i|inj: {t: ti} pf fun/inj1 wd ea ea/#i @ t.
ea/r|inj: {t: ti} pf fun/inj1 greg ea ea/#r @ t.
ea/s|inj: {t: ti} pf fun/inj1 ma ea ea/#s @ t.
ea/m|inj: {t: ti} pf fun/inj1 ma ea ea/#m @ t.

% no ea/addr|ea/i
% no ea/addr|ea/r
ea/addr|ea/s:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {ema: exp ma}
  {t: ti}
  pf wd/eq (ea/addr (state/make epc ef eg es em) (ea/s ema))
           (ma/addr eg ema) @ t.
ea/addr|ea/m:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {ema: exp ma}
  {t: ti}
  pf wd/eq (ea/addr (state/make epc ef eg es em) (ea/m ema))
           (ma/addr eg ema) @ t.

ea/sel|ea/i:
  {e: exp state} {en: exp wd}
  {t: ti}
  pf wd/eq (ea/sel e (ea/i en)) en @ t.
ea/sel|ea/r:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {er: exp greg}
  {t: ti}
  pf wd/eq (ea/sel (state/make epc ef eg es em) (ea/r er))
           (mapg/sel eg er) @ t.
ea/sel|ea/s:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {ema: exp ma}
  {t: ti}
  pf wd/eq (ea/sel (state/make epc ef eg es em) (ea/s ema))
           (mapw/sel es (ma/addr eg ema)) @ t.
ea/sel|ea/m:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {ema: exp ma}
  {t: ti}
  pf wd/eq (ea/sel (state/make epc ef eg es em) (ea/m ema))
           (mapw/sel em (ma/addr eg ema)) @ t.

% no ea/updg|ea/i
ea/updg|ea/r:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {er: exp greg} {en: exp wd}
  {t: ti}
  pf mapg/eq (ea/updg (state/make epc ef eg es em) (ea/r er) en)
             (mapg/upd eg er en) @ t.
ea/updg|ea/s:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {ema: exp ma} {en: exp wd}
  {t: ti}
```

```
  pf mapg/eq (ea/updg (state/make epc ef eg es em) (ea/s ema) en) eg @ t.
ea/updg|ea/m:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {ema: exp ma} {en: exp wd}
  {t: ti}
  pf mapg/eq (ea/updg (state/make epc ef eg es em) (ea/m ema) en) eg @ t.

% no ea/upds|ea/i
ea/upds|ea/r:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {er: exp greg} {en: exp wd}
  {t: ti}
  pf mapw/eq (ea/upds (state/make epc ef eg es em) (ea/r er) en) es @ t.
ea/upds|ea/s:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {ema: exp ma} {en: exp wd}
  {t: ti}
  pf mapw/eq (ea/upds (state/make epc ef eg es em) (ea/s ema) en)
            (mapw/upd es (ma/addr eg ema) en) @ t.
ea/upds|ea/m:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {ema: exp ma} {en: exp wd}
  {t: ti}
  pf mapw/eq (ea/upds (state/make epc ef eg es em) (ea/m ema) en) es @ t.

% no ea/updm|ea/i
ea/updm|ea/r:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {er: exp greg} {en: exp wd}
  {t: ti}
  pf mapw/eq (ea/updm (state/make epc ef eg es em) (ea/r er) en) em @ t.
ea/updm|ea/s:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {ema: exp ma} {en: exp wd}
  {t: ti}
  pf mapw/eq (ea/updm (state/make epc ef eg es em) (ea/s ema) en) em @ t.
ea/updm|ea/m:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {ema: exp ma} {en: exp wd}
  {t: ti}
  pf mapw/eq (ea/updm (state/make epc ef eg es em) (ea/m ema) en)
            (mapw/upd em (ma/addr eg ema) en) @ t.
```

B.2.2.9 Instructions

```
inst/mov|inj:  {t: ti} pf fun/inj3 wd ea ea inst inst/#mov @ t.
inst/xchg|inj: {t: ti} pf fun/inj3 wd ea greg inst inst/#xchg @ t.
inst/lea|inj:  {t: ti} pf fun/inj3 wd ea greg inst inst/#lea @ t.
inst/push|inj: {t: ti} pf fun/inj2 wd ea inst inst/#push @ t.
inst/pop|inj:  {t: ti} pf fun/inj2 wd ea inst inst/#pop @ t.
inst/op1|inj:  {t: ti} pf fun/inj3 wd op/1 ea inst inst/#op1 @ t.
inst/op2|inj:  {t: ti} pf fun/inj3 wd op/2 (pair ea ea) inst inst/#op2 @ t.
inst/op2n|inj: {t: ti} pf fun/inj3 wd op/2 (pair ea ea) inst inst/#op2n @ t.
inst/op3|inj:  {t: ti}
               pf fun/inj3 wd (pair op/3 op/3) (pair ea (pair greg greg)) inst
                           inst/#op3 @ t.
inst/jmp|inj:  {t: ti} pf fun/inj2 wd ea inst inst/#jmp @ t.
inst/j|inj:    {t: ti} pf fun/inj3 wd cop wd inst inst/#j @ t.
inst/call|inj: {t: ti} pf fun/inj2 wd ea inst inst/#call @ t.
inst/ret|inj:  {t: ti} pf fun/inj1 wd inst inst/#ret @ t.

inst/nextpc|inst/mov:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eea1: exp ea} {eea2: exp ea}
```

```
  {t: ti}
  pf wd/eq (inst/nextpc (state/make epc ef eg es em) (inst/mov eni eea1 eea2))
           (wd/add epc eni) @ t.
inst/nextf|inst/mov:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eea1: exp ea} {eea2: exp ea}
  {t: ti}
  pf wd/eq (inst/nextf (state/make epc ef eg es em) (inst/mov eni eea1 eea2))
           ef @ t.
inst/nextg|inst/mov:
  {e: exp state} {eni: exp wd} {eea1: exp ea} {eea2: exp ea}
  {t: ti}
  pf mapg/eq (inst/nextg e (inst/mov eni eea1 eea2))
             (ea/updg e eea2 (ea/sel e eea1)) @ t.
inst/nexts|inst/mov:
  {e: exp state} {eni: exp wd} {eea1: exp ea} {eea2: exp ea}
  {t: ti}
  pf mapw/eq (inst/nexts e (inst/mov eni eea1 eea2))
             (ea/upds e eea2 (ea/sel e eea1)) @ t.
inst/nextm|inst/mov:
  {e: exp state} {eni: exp wd} {eea1: exp ea} {eea2: exp ea}
  {t: ti}
  pf mapw/eq (inst/nextm e (inst/mov eni eea1 eea2))
             (ea/updm e eea2 (ea/sel e eea1)) @ t.

inst/nextpc|inst/xchg:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eea: exp ea} {er: exp greg}
  {t: ti}
  pf wd/eq (inst/nextpc (state/make epc ef eg es em) (inst/xchg eni eea er))
           (wd/add epc eni) @ t.
inst/nextf|inst/xchg:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eea: exp ea} {er: exp greg}
  {t: ti}
  pf wd/eq (inst/nextf (state/make epc ef eg es em) (inst/xchg eni eea er))
           ef @ t.
inst/nextg|inst/xchg:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eea: exp ea} {er: exp greg}
  {t: ti}
  pf mapg/eq
       (inst/nextg (state/make epc ef eg es em) (inst/xchg eni eea er))
       (mapg/upd
          (ea/updg (state/make epc ef eg es em) eea (mapg/sel eg er))
          er
          (ea/sel (state/make epc ef eg es em) eea)) @ t.
inst/nexts|inst/xchg:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eea: exp ea} {er: exp greg}
  {t: ti}
  pf mapw/eq
       (inst/nexts (state/make epc ef eg es em) (inst/xchg eni eea er))
       (ea/upds (state/make epc ef eg es em) eea (mapg/sel eg er)) @ t.
inst/nextm|inst/xchg:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eea: exp ea} {er: exp greg}
  {t: ti}
  pf mapw/eq
       (inst/nextm (state/make epc ef eg es em) (inst/xchg eni eea er))
       (ea/updm (state/make epc ef eg es em) eea (mapg/sel eg er)) @ t.

inst/nextpc|inst/lea:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
```

```
  {eni: exp wd} {eea: exp ea} {er: exp greg}
  {t: ti}
  pf wd/eq (inst/nextpc (state/make epc ef eg es em) (inst/lea eni eea er))
           (wd/add epc eni) @ t.
inst/nextf|inst/lea:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eea: exp ea} {er: exp greg}
  {t: ti}
  pf wd/eq (inst/nextf (state/make epc ef eg es em) (inst/lea eni eea er))
           ef @ t.
inst/nextg|inst/lea:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eea: exp ea} {er: exp greg}
  {t: ti}
  pf mapg/eq (inst/nextg (state/make epc ef eg es em) (inst/lea eni eea er))
             (mapg/upd eg er (ea/addr (state/make epc ef eg es em) eea)) @ t.
inst/nexts|inst/lea:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eea: exp ea} {er: exp greg}
  {t: ti}
  pf mapw/eq (inst/nexts (state/make epc ef eg es em) (inst/lea eni eea er))
             es @ t.
inst/nextm|inst/lea:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eea: exp ea} {er: exp greg}
  {t: ti}
  pf mapw/eq (inst/nextm (state/make epc ef eg es em) (inst/lea eni eea er))
             em @ t.

inst/nextpc|inst/push:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eea: exp ea}
  {t: ti}
  pf wd/eq (inst/nextpc (state/make epc ef eg es em) (inst/push eni eea))
           (wd/add epc eni) @ t.
inst/nextf|inst/push:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eea: exp ea}
  {t: ti}
  pf wd/eq (inst/nextf (state/make epc ef eg es em) (inst/push eni eea))
           ef @ t.
inst/nextg|inst/push:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eea: exp ea}
  {t: ti}
  pf mapg/eq (inst/nextg (state/make epc ef eg es em) (inst/push eni eea))
             (mapg/upd eg `esp (wd/add (mapg/sel eg `esp) (wd/` wd/~4))) @ t.
inst/nexts|inst/push:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eea: exp ea}
  {t: ti}
  pf mapw/eq (inst/nexts (state/make epc ef eg es em) (inst/push eni eea))
             (mapw/upd es
                       (wd/add (mapg/sel eg `esp) (wd/` wd/~4))
                       (ea/sel (state/make epc ef eg es em) eea)) @ t.
inst/nextm|inst/push:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eea: exp ea}
  {t: ti}
  pf mapw/eq (inst/nextm (state/make epc ef eg es em) (inst/push eni eea))
             em @ t.

inst/nextpc|inst/pop:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
```

```
  {eni: exp wd} {eea: exp ea}
  {t: ti}
  pf wd/eq (inst/nextpc (state/make epc ef eg es em) (inst/pop eni eea))
           (wd/add epc eni) @ t.
inst/nextf|inst/pop:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eea: exp ea}
  {t: ti}
  pf wd/eq (inst/nextf (state/make epc ef eg es em) (inst/pop eni eea)) ef @ t.
inst/nextg|inst/pop:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eea: exp ea}
  {t: ti}
  pf mapg/eq (inst/nextg (state/make epc ef eg es em) (inst/pop eni eea))
             (mapg/upd (ea/updg (state/make epc ef eg es em)
                                eea
                                (mapw/sel es (mapg/sel eg ‘esp)))
                       ‘esp
                       (wd/add (mapg/sel eg ‘esp) (wd/‘ wd/4))) @ t.
inst/nexts|inst/pop:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eea: exp ea}
  {t: ti}
  pf mapw/eq (inst/nexts (state/make epc ef eg es em) (inst/pop eni eea))
             (ea/upds (state/make epc ef eg es em)
                      eea
                      (mapw/sel es (mapg/sel eg ‘esp))) @ t.
inst/nextm|inst/pop:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eea: exp ea}
  {t: ti}
  pf mapw/eq (inst/nextm (state/make epc ef eg es em) (inst/pop eni eea))
             (ea/updm (state/make epc ef eg es em)
                      eea
                      (mapw/sel es (mapg/sel eg ‘esp))) @ t.

inst/nextpc|inst/op1:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eop: exp op/1} {eea: exp ea}
  {t: ti}
  pf wd/eq (inst/nextpc (state/make epc ef eg es em) (inst/op1 eni eop eea))
           (wd/add epc eni) @ t.
inst/nextf|inst/op1:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eop: exp op/1} {eea: exp ea}
  {t: ti}
  pf wd/eq (inst/nextf (state/make epc ef eg es em) (inst/op1 eni eop eea))
           (op/updf1 eop ef (ea/sel (state/make epc ef eg es em) eea)) @ t.
inst/nextg|inst/op1:
  {e: exp state} {eni: exp wd} {eop: exp op/1} {eea: exp ea}
  {t: ti}
  pf mapg/eq (inst/nextg e (inst/op1 eni eop eea))
             (ea/updg e eea (op/app1 eop (ea/sel e eea))) @ t.
inst/nexts|inst/op1:
  {e: exp state} {eni: exp wd} {eop: exp op/1} {eea: exp ea}
  {t: ti}
  pf mapw/eq (inst/nexts e (inst/op1 eni eop eea))
             (ea/upds e eea (op/app1 eop (ea/sel e eea))) @ t.
inst/nextm|inst/op1:
  {e: exp state} {eni: exp wd} {eop: exp op/1} {eea: exp ea}
  {t: ti}
  pf mapw/eq (inst/nextm e (inst/op1 eni eop eea))
             (ea/updm e eea (op/app1 eop (ea/sel e eea))) @ t.
```

```
inst/nextpc|inst/op2:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eop: exp op/2} {eea1: exp ea} {eea2: exp ea}
  {t: ti}
  pf wd/eq (inst/nextpc (state/make epc ef eg es em)
                        (inst/op2 eni eop eea1 eea2))
           (wd/add epc eni) @ t.
inst/nextf|inst/op2:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eop: exp op/2} {eea1: exp ea} {eea2: exp ea}
  {t: ti}
  pf wd/eq (inst/nextf (state/make epc ef eg es em)
                       (inst/op2 eni eop eea1 eea2))
           (op/updf2 eop
                     ef
                     (ea/sel (state/make epc ef eg es em) eea2)
                     (ea/sel (state/make epc ef eg es em) eea1)) @ t.
inst/nextg|inst/op2:
  {e: exp state} {eni: exp wd} {eop: exp op/2} {eea1: exp ea} {eea2: exp ea}
  {t: ti}
  pf mapg/eq
       (inst/nextg e (inst/op2 eni eop eea1 eea2))
       (ea/updg e eea2 (op/app2 eop (ea/sel e eea2) (ea/sel e eea1))) @ t.
inst/nexts|inst/op2:
  {e: exp state} {eni: exp wd} {eop: exp op/2} {eea1: exp ea} {eea2: exp ea}
  {t: ti}
  pf mapw/eq
       (inst/nexts e (inst/op2 eni eop eea1 eea2))
       (ea/upds e eea2 (op/app2 eop (ea/sel e eea2) (ea/sel e eea1))) @ t.
inst/nextm|inst/op2:
  {e: exp state} {eni: exp wd} {eop: exp op/2} {eea1: exp ea} {eea2: exp ea}
  {t: ti}
  pf mapw/eq
       (inst/nextm e (inst/op2 eni eop eea1 eea2))
       (ea/updm e eea2 (op/app2 eop (ea/sel e eea2) (ea/sel e eea1))) @ t.

inst/nextpc|inst/op2n:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eop: exp op/2} {eea1: exp ea} {eea2: exp ea}
  {t: ti}
  pf wd/eq (inst/nextpc (state/make epc ef eg es em)
                        (inst/op2n eni eop eea1 eea2))
           (wd/add epc eni) @ t.
inst/nextf|inst/op2n:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eop: exp op/2} {eea1: exp ea} {eea2: exp ea}
  {t: ti}
  pf wd/eq (inst/nextf (state/make epc ef eg es em)
                       (inst/op2n eni eop eea1 eea2))
           (op/updf2 eop
                     ef
                     (ea/sel (state/make epc ef eg es em) eea2)
                     (ea/sel (state/make epc ef eg es em) eea1)) @ t.
inst/nextg|inst/op2n:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eop: exp op/2} {eea1: exp ea} {eea2: exp ea}
  {t: ti}
  pf mapg/eq (inst/nextg (state/make epc ef eg es em)
                         (inst/op2n eni eop eea1 eea2))
             eg @ t.
inst/nexts|inst/op2n:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eop: exp op/2} {eea1: exp ea} {eea2: exp ea}
  {t: ti}
```

```
  pf mapw/eq (inst/nexts (state/make epc ef eg es em)
                         (inst/op2n eni eop eea1 eea2))
             es @ t.
inst/nextm|inst/op2n:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eop: exp op/2} {eea1: exp ea} {eea2: exp ea}
  {t: ti}
  pf mapw/eq (inst/nextm (state/make epc ef eg es em)
                         (inst/op2n eni eop eea1 eea2))
             em @ t.

inst/nextpc|inst/op3:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eop1: exp op/3} {eop2: exp op/3}
  {eea: exp ea} {er1: exp greg} {er2: exp greg}
  {t: ti}
  pf wd/eq (inst/nextpc (state/make epc ef eg es em)
                        (inst/op3 eni eop1 eop2 eea er1 er2))
           (wd/add epc eni) @ t.
inst/nextf|inst/op3:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eop1: exp op/3} {eop2: exp op/3}
  {eea: exp ea} {er1: exp greg} {er2: exp greg}
  {t: ti}
  pf wd/eq (inst/nextf (state/make epc ef eg es em)
                       (inst/op3 eni eop1 eop2 eea er1 er2))
           (op/updf3 eop1
                     ef
                     (mapg/sel eg er1)
                     (mapg/sel eg er2)
                     (ea/sel (state/make epc ef eg es em) eea)) @ t.
inst/nextg|inst/op3:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eop1: exp op/3} {eop2: exp op/3}
  {eea: exp ea} {er1: exp greg} {er2: exp greg}
  {t: ti}
  pf mapg/eq
       (inst/nextg (state/make epc ef eg es em)
                   (inst/op3 eni eop1 eop2 eea er1 er2))
       (mapg/upd
          (mapg/upd eg
                    er1
                    (op/app3 eop1
                             (mapg/sel eg er1)
                             (mapg/sel eg er2)
                             (ea/sel (state/make epc ef eg es em) eea)))
          er2
          (op/app3 eop2
                   (mapg/sel eg er1)
                   (mapg/sel eg er2)
                   (ea/sel (state/make epc ef eg es em) eea))) @ t.
inst/nexts|inst/op3:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eop1: exp op/3} {eop2: exp op/3}
  {eea: exp ea} {er1: exp greg} {er2: exp greg}
  {t: ti}
  pf mapw/eq (inst/nexts (state/make epc ef eg es em)
                         (inst/op3 eni eop1 eop2 eea er1 er2))
             es @ t.
inst/nextm|inst/op3:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eop1: exp op/3} {eop2: exp op/3}
  {eea: exp ea} {er1: exp greg} {er2: exp greg}
  {t: ti}
```

```
  pf mapw/eq (inst/nextm (state/make epc ef eg es em)
                         (inst/op3 eni eop1 eop2 eea er1 er2))
            em @ t.

inst/nextpc|inst/jmp:
  {e: exp state} {eni: exp wd} {eea: exp ea}
  {t: ti}
  pf wd/eq (inst/nextpc e (inst/jmp eni eea)) (ea/sel e eea) @ t.
inst/nextf|inst/jmp:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eea: exp ea}
  {t: ti}
  pf wd/eq (inst/nextf (state/make epc ef eg es em) (inst/jmp eni eea)) ef @ t.
inst/nextg|inst/jmp:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eea: exp ea}
  {t: ti}
  pf mapg/eq (inst/nextg (state/make epc ef eg es em) (inst/jmp eni eea))
            eg @ t.
inst/nexts|inst/jmp:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eea: exp ea}
  {t: ti}
  pf mapw/eq (inst/nexts (state/make epc ef eg es em) (inst/jmp eni eea))
            es @ t.
inst/nextm|inst/jmp:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eea: exp ea}
  {t: ti}
  pf mapw/eq (inst/nextm (state/make epc ef eg es em) (inst/jmp eni eea))
            em @ t.

inst/nextpc|inst/j:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {ecop: exp cop} {en: exp wd}
  {t: ti}
  pf wd/eq (inst/nextpc (state/make epc ef eg es em) (inst/j eni ecop en))
          (wd/add (wd/add epc eni) (wd/mul en (cop/self ecop ef))) @ t.
inst/nextf|inst/j:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {ecop: exp cop} {en: exp wd}
  {t: ti}
  pf wd/eq (inst/nextf (state/make epc ef eg es em) (inst/j eni ecop en))
          ef @ t.
inst/nextg|inst/j:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {ecop: exp cop} {en: exp wd}
  {t: ti}
  pf mapg/eq (inst/nextg (state/make epc ef eg es em) (inst/j eni ecop en))
            eg @ t.
inst/nexts|inst/j:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {ecop: exp cop} {en: exp wd}
  {t: ti}
  pf mapw/eq (inst/nexts (state/make epc ef eg es em) (inst/j eni ecop en))
            es @ t.
inst/nextm|inst/j:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {ecop: exp cop} {en: exp wd}
  {t: ti}
  pf mapw/eq (inst/nextm (state/make epc ef eg es em) (inst/j eni ecop en))
            em @ t.

inst/nextpc|inst/call:
```

```
  {e: exp state} {eni: exp wd} {eea: exp ea}
  {t: ti}
  pf wd/eq (inst/nextpc e (inst/call eni eea)) (ea/sel e eea) @ t.
inst/nextf|inst/call:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eea: exp ea}
  {t: ti}
  pf wd/eq (inst/nextf (state/make epc ef eg es em) (inst/call eni eea))
           ef @ t.
inst/nextg|inst/call:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eea: exp ea}
  {t: ti}
  pf mapg/eq (inst/nextg (state/make epc ef eg es em) (inst/call eni eea))
             (mapg/upd eg ‘esp (wd/add (mapg/sel eg ‘esp) (wd/‘ wd/~4))) @ t.
inst/nexts|inst/call:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eea: exp ea}
  {t: ti}
  pf mapw/eq (inst/nexts (state/make epc ef eg es em) (inst/call eni eea))
             (mapw/upd es
                       (wd/add (mapg/sel eg ‘esp) (wd/‘ wd/~4))
                       (wd/add epc eni)) @ t.
inst/nextm|inst/call:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd} {eea: exp ea}
  {t: ti}
  pf mapw/eq (inst/nextm (state/make epc ef eg es em) (inst/call eni eea))
             em @ t.

inst/nextpc|inst/ret:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd}
  {t: ti}
  pf wd/eq (inst/nextpc (state/make epc ef eg es em) (inst/ret eni))
           (mapw/sel es (mapg/sel eg ‘esp)) @ t.
inst/nextf|inst/ret:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd}
  {t: ti}
  pf wd/eq (inst/nextf (state/make epc ef eg es em) (inst/ret eni)) ef @ t.
inst/nextg|inst/ret:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd}
  {t: ti}
  pf mapg/eq (inst/nextg (state/make epc ef eg es em) (inst/ret eni))
             (mapg/upd eg ‘esp (wd/add (mapg/sel eg ‘esp) (wd/‘ wd/4))) @ t.
inst/nexts|inst/ret:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd}
  {t: ti}
  pf mapw/eq (inst/nexts (state/make epc ef eg es em) (inst/ret eni)) es @ t.
inst/nextm|inst/ret:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eni: exp wd}
  {t: ti}
  pf mapw/eq (inst/nextm (state/make epc ef eg es em) (inst/ret eni)) em @ t.
```

B.2.2.10 Programs

```
% - semantics -

rel/app2#|eq_prog: pf rel/app2# prog prog (eq/#eq prog) prog/pm prog/pm.
% no rel/app2#|neq_prog
```

```
% - truth -

prog/fetch|fun: {t: ti} pf rel/fun2 (pair prog wd) inst prog/#fetch @ t.

inst/nextpc|prog/fetch:
  {t: ti}
  pf all state ri/+ ([x: exp state] all inst ri/+ ([xi^: exp inst]
         state/eq x `ss
     imp prog/fetch `pm (state/pc x) xi^
     imp nextT (wd/eq `pc (inst/nextpc x xi^))
     )) @ t.
inst/nextf|prog/fetch:
  {t: ti}
  pf all state ri/+ ([x: exp state] all inst ri/+ ([xi^: exp inst]
         state/eq x `ss
     imp prog/fetch `pm (state/pc x) xi^
     imp nextT (wd/eq `f (inst/nextf x xi^))
     )) @ t.
inst/nextg|prog/fetch:
  {t: ti}
  pf all state ri/+ ([x: exp state] all inst ri/+ ([xi^: exp inst]
         state/eq x `ss
     imp prog/fetch `pm (state/pc x) xi^
     imp nextT (mapg/eq `g (inst/nextg x xi^))
     )) @ t.
inst/nexts|prog/fetch:
  {t: ti}
  pf all state ri/+ ([x: exp state] all inst ri/+ ([xi^: exp inst]
         state/eq x `ss
     imp prog/fetch `pm (state/pc x) xi^
     imp nextT (mapw/eq `s (inst/nexts x xi^))
     )) @ t.
inst/nextm|prog/fetch:
  {t: ti}
  pf all state ri/+ ([x: exp state] all inst ri/+ ([xi^: exp inst]
         state/eq x `ss
     imp prog/fetch `pm (state/pc x) xi^
     imp nextT (mapw/eq `m (inst/nextm x xi^))
     )) @ t.
```

B.3 Security Policy

B.3.1 Abstract Syntax

B.3.1.1 Security Registers

```
% - types -

sreg: ty -> ty.

% - constants -

sreg/` = [tau: ty] con/` (sreg tau).
```

B.3.1.2 Security Automata

```
% - types -

sa: ty.

% - functions -
```

```
sa/#sel: {tau: ty} fun/2 sa (sreg tau) tau.
sa/#upd: {tau: ty} fun/3 sa (sreg tau) tau sa.

sa/sel = [tau: ty]
         fun/app2 sa (sreg tau) tau (sa/#sel tau).
sa/upd = [tau: ty]
         fun/app3 sa (sreg tau) tau sa (sa/#upd tau).

% - relations -

sa/eq  = rel/app2 sa sa (eq/#eq sa).
sa/neq = rel/app2 sa sa (eq/#neq sa).

% - expressions -

sa/q: par sa.

sa/‘q = par/‘ sa sa/q.

% - exports -

q = sa/q.

‘q = sa/‘q.
```

B.3.1.3 Extended States

```
% - types -

qstate = pair state sa.

% - functions -

qstate/#pc: fun/1 qstate wd.
qstate/#f:  fun/1 qstate wd.
qstate/#g:  fun/1 qstate mapg.
qstate/#s:  fun/1 qstate mapw.
qstate/#m:  fun/1 qstate mapw.

qstate/pc = fun/app1 qstate wd qstate/#pc.
qstate/f  = fun/app1 qstate wd qstate/#f.
qstate/g  = fun/app1 qstate mapg qstate/#g.
qstate/s  = fun/app1 qstate mapw qstate/#s.
qstate/m  = fun/app1 qstate mapw qstate/#m.

qstate/ss = pair/left state sa.
qstate/q  = pair/right state sa.

qstate/make2
= pair/make state sa.
qstate/make
= [epc: exp wd] [ef: exp wd] [eg: exp mapg] [es: exp mapw] [em: exp mapw]
  qstate/make2 (state/make epc ef eg es em).

qstate/eax = [e: exp qstate] mapg/sel (qstate/g e) ‘eax.
qstate/ebp = [e: exp qstate] mapg/sel (qstate/g e) ‘ebp.
qstate/esp = [e: exp qstate] mapg/sel (qstate/g e) ‘esp.

% - relations -

qstate/eq = eq/eq qstate.

% - expressions -
```

```
qstate/`sq = qstate/make `pc `f `g `s `m `q.

% - exports -

`sq = qstate/`sq.
```

B.3.1.4 Access Modes

```
% - types -

acc: ty.

% - constants -

acc/none: con acc.
acc/rd:   con acc.
acc/wr:   con acc.
acc/rw:   con acc.

acc/`none = con/` acc acc/none.
acc/`rd   = con/` acc acc/rd.
acc/`wr   = con/` acc acc/wr.
acc/`rw   = con/` acc acc/rw.

% - relations -

acc/#leq: rel/2 acc acc.

acc/#nleq = rel/not2 acc acc acc/#leq.

acc/eq   = rel/app2 acc acc (eq/#eq acc).
acc/neq  = rel/app2 acc acc (eq/#neq acc).
acc/leq  = rel/app2 acc acc acc/#leq.
acc/nleq = rel/app2 acc acc acc/#nleq.
```

B.3.1.5 Access Maps

```
% - types -

mapa: ty.

% - functions -

mapa/#sel: fun/3 mapa wd wd acc.

mapa/sel = fun/app3 mapa wd wd acc mapa/#sel.

% - relations -

mapa/#leq: rel/2 mapa mapa.

mapa/eq  = rel/app2 mapa mapa (eq/#eq mapa).
mapa/neq = rel/app2 mapa mapa (eq/#neq mapa).
mapa/leq = rel/app2 mapa mapa mapa/#leq.
```

B.3.1.6 Java Types

```
% - types -

jty: ty.

% - constants -

jty/bool:   con jty.
```

```
jty/char:   con jty.
jty/byte:   con jty.
jty/short:  con jty.
jty/int:    con jty.
jty/Class:  con jty.

jty/`bool  = con/` jty jty/bool.
jty/`char  = con/` jty jty/char.
jty/`byte  = con/` jty jty/byte.
jty/`short = con/` jty jty/short.
jty/`int   = con/` jty jty/int.
jty/`Class = con/` jty jty/Class.

% - functions -

jty/#array: fun/1 jty jty.
jty/#inst:  fun/1 wd jty.

jty/#size: fun/1 jty wd.

jty/array = fun/app1 jty jty jty/#array.
jty/inst  = fun/app1 wd jty jty/#inst.

jty/size = fun/app1 jty wd jty/#size.

% - expressions -

jty/len  = [ena: exp wd]
           wd/add ena (wd/`# 16).
jty/elem = [ena: exp wd] [eni: exp wd] [ety: exp jty]
           wd/add ena (wd/add (wd/mul eni (jty/size ety)) (wd/`# 20)).

% - propositions -

jty/wd    = [ety: exp jty]
            wd/eq (jty/size ety) (wd/`# 4).
jty/align = [enp: exp wd] [ety: exp jty]
            wd/eq (wd/and enp (wd/sub (jty/size ety) `1w)) `0w.
```

B.3.1.7 Java Type Assignments

```
% - types -

jta: ty.

% - relations -

jta/#leq:   rel/2 jta jta.
jta/#of:    rel/2 jta (pair wd jty).
jta/#ptr:   rel/2 (pair jta wd) jty.
jta/#field: rel/2 (trip jta wd wd) jty.

jta/leq   = rel/app2 jta jta jta/#leq.
jta/of    = [eta: exp jta] [en: exp wd] [ety: exp jty]
            rel/app2 jta (pair wd jty) jta/#of eta (pair/make wd jty en ety).
jta/ptr   = [eta: exp jta] [enp: exp wd]
            rel/app2 (pair jta wd) jty jta/#ptr (pair/make jta wd eta enp).
jta/field = [eta: exp jta] [enc: exp wd] [enf: exp wd]
            rel/app2 (trip jta wd wd) jty
                     jta/#field
                     (trip/make jta wd wd eta enc enf).

% - propositions -
```

```
jta/ptr_len
= [eta: exp jta] [enp: exp wd]
  some wd ri/- ([xna: exp wd] some jty ri/- ([xty: exp jty]
      jta/of eta xna (jty/array xty)
  and wd/neq xna `0w
  and wd/eq enp (jty/len xna)
  )).

jta/ptr_elem
= [eta: exp jta] [em: exp mapw] [enp: exp wd] [ety: exp jty]
  some wd ri/- ([xna: exp wd] some wd ri/- ([xni: exp wd]
      jta/of eta xna (jty/array ety)
  and wd/neq xna `0w
  and wd/ltu xni (mapw/sel em (jty/len xna))
  and wd/eq enp (jty/elem xna xni ety)
  )).

jta/ptr_field
= [eta: exp jta] [enp: exp wd] [ety: exp jty]
  some wd ri/- ([xno: exp wd]
  some wd ri/- ([xnc: exp wd] some wd ri/- ([xnf: exp wd]
      jta/of eta xno (jty/inst xnc)
  and wd/neq xno `0w
  and jta/field eta xnc xnf ety
  and wd/eq enp (wd/add xno xnf)
  ))).

jta/valid
= [eta: exp jta]
  all wd ri/- ([xnp: exp wd] all jty ri/- ([xty: exp jty]
      (jta/ptr_len eta xnp imp jta/ptr eta xnp jty/`int)
  and (jta/ptr_field eta xnp xty imp jta/ptr eta xnp xty)
  and (jta/ptr eta xnp xty imp jty/align xnp xty)
  )).

jta/mem
= [eta: exp jta] [em: exp mapw]
  all wd ri/- ([xnp: exp wd] all jty ri/- ([xty: exp jty]
      (jta/ptr_elem eta em xnp xty imp jta/ptr eta xnp xty)
  and (jta/ptr eta xnp xty imp jty/wd xty imp jta/of eta (mapw/sel em xnp) xty)
  )).
```

B.3.1.8 Java Type Environments

```
% - types -

jts = pair jta mapa.

% - functions -

jts/make = pair/make jta mapa.
jts/ta   = pair/left jta mapa.
jts/am   = pair/right jta mapa.

% - relations -

jts/#leq:   rel/2 jts jts.
jts/#of:    rel/2 jts (pair wd jty).
jts/#ptr:   rel/2 (pair jts wd) (pair jty acc).
jts/#field: rel/2 (trip jts wd wd) jty.

jts/#mem: rel/2 jts mapw.

jts/leq   = rel/app2 jts jts jts/#leq.
```

```
jts/of    = [ets: exp jts] [en: exp wd] [ety: exp jty]
            rel/app2 jts (pair wd jty) jts/#of ets (pair/make wd jty en ety).
jts/ptr   = [ets: exp jts] [enp: exp wd] [ety: exp jty] [ea: exp acc]
            rel/app2 (pair jts wd) (pair jty acc)
                     jts/#ptr
                     (pair/make jts wd ets enp) (pair/make jty acc ety ea).
jts/field = [ets: exp jts] [enc: exp wd] [enf: exp wd]
            rel/app2 (trip jts wd wd) jty
                     jts/#field
                     (trip/make jts wd wd ets enc enf).

jts/mem = rel/app2 jts mapw jts/#mem.

% - propositions -

jts/valid
= [ets: exp jts]
  all wd ri/- ([xnp: exp wd] all jty ri/- ([xty: exp jty]
      jta/valid (jts/ta ets)
  and (    jta/ptr_len (jts/ta ets) xnp
       imp acc/eq (mapa/sel (jts/am ets) xnp (jty/size jty/`int)) acc/`rd)
  and (    jta/ptr_field (jts/ta ets) xnp xty
       imp acc/eq (mapa/sel (jts/am ets) xnp (jty/size xty)) acc/`rw)
  )).
```

B.3.1.9 Safety

```
% - constants -

safe/page = 4096.
safe/2page = 8192.
safe/npage = 4294963200.

safe/fps:  con (sreg (list wd)).
safe/accm: con (sreg mapa).

safe/`page = wd/`# safe/page.

% - functions -

safe/#nextq: fun/2 qstate inst sa.
safe/#next:  fun/2 qstate inst qstate.

safe/nextq = fun/app2 qstate inst sa safe/#nextq.
safe/next  = fun/app2 qstate inst qstate safe/#next.

safe/sel_fps = [eq: exp sa] sa/sel (list wd) eq (sreg/` (list wd) safe/fps).
safe/upd_fps = [eq: exp sa] sa/upd (list wd) eq (sreg/` (list wd) safe/fps).
qstate/fps   = [e: exp qstate] safe/sel_fps (qstate/q e).

safe/sel_accm = [eq: exp sa] sa/sel mapa eq (sreg/` mapa safe/accm).
safe/upd_accm = [eq: exp sa] sa/upd mapa eq (sreg/` mapa safe/accm).
qstate/accm   = [e: exp qstate] safe/sel_accm (qstate/q e).

% - relations -

safe/#sp:    rel/2 qstate wd.
safe/#rd_s:  rel/2 qstate wd.
safe/#wr_s:  rel/2 (pair qstate wd) wd.
safe/#rd_m:  rel/2 qstate wd.
safe/#wr_m:  rel/2 (pair qstate wd) wd.
safe/#rd_ea: rel/2 qstate ea.
safe/#wr_ea: rel/2 (pair qstate ea) wd.
safe/#inst:  rel/2 qstate inst.
```

```
safe/sp    = rel/app2 qstate wd safe/#sp.
safe/rd_s  = rel/app2 qstate wd safe/#rd_s.
safe/wr_s  = [esq: exp qstate] [en: exp wd]
             rel/app2 (pair qstate wd) wd
                      safe/#wr_s
                      (pair/make qstate wd esq en).
safe/rd_m  = rel/app2 qstate wd safe/#rd_m.
safe/wr_m  = [esq: exp qstate] [en: exp wd]
             rel/app2 (pair qstate wd) wd
                      safe/#wr_m
                      (pair/make qstate wd esq en).
safe/rd_ea = rel/app2 qstate ea safe/#rd_ea.
safe/wr_ea = [esq: exp qstate] [eea: exp ea]
             rel/app2 (pair qstate ea) wd
                      safe/#wr_ea
                      (pair/make qstate ea esq eea).
safe/inst  = rel/app2 qstate inst safe/#inst.

% - propositions -

safe/make = [esq: exp qstate]
            some inst ri/- ([xi^: exp inst]
            prog/fetch `pm (qstate/pc esq) xi^ and safe/inst esq xi^
            ).
safe      = safe/make `sq.

safe/pre  = [ppuz: exp qstate -> exp qstate -> prp]
            [epc0: exp wd] [esp0: exp wd]
            [esq0: exp qstate] [esq: exp qstate]
            wd/eq `pc epc0 and wd/eq `g_sp esp0 and ppuz esq0 esq.
safe/post = [pquz: exp qstate -> exp qstate -> prp]
            [epc': exp qstate -> exp wd]
            [esq0: exp qstate] [esq: exp qstate]
            wd/eq `pc (epc' esq0) and pquz esq0 esq.

safe/body = [psafe: exp qstate -> prp]
            [ppre: exp qstate -> exp qstate -> prp]
            [ppost: exp qstate -> exp qstate -> prp]
            [esq0: exp qstate]
                qstate/eq `sq esq0
            imp ppre esq0 `sq
            imp psafe `sq unlessT ppost esq0 `sq.

safe/proc = [psafe: exp qstate -> prp]
            [ppre: exp qstate -> exp qstate -> prp]
            [ppost: exp qstate -> exp qstate -> prp]
            allT (
            all qstate ri/+ ([xsq0: exp qstate]
            safe/body psafe ppre ppost xsq0
            )).
```

B.3.2 Inference Rules

B.3.2.1 Security Registers

```
% - semantics -

rel/app2#|eq_sreg:
   {tau: ty}
   {c: con (sreg tau)}
   pf rel/app2# (sreg tau) (sreg tau) (eq/#eq (sreg tau)) c c.
rel/app2#|neq_sreg:
   {tau: ty}
```

```
   {c1: con (sreg tau)} {c2: con (sreg tau)}
   con/neq (sreg tau) c1 c2
-> pf rel/app2# (sreg tau) (sreg tau) (eq/#neq (sreg tau)) c1 c2.
```

B.3.2.2 Security Automata

```
sa/sel|mc0:  {tau: ty}
             {eq: exp sa} {er: exp (sreg tau)} {e: exp tau} {t: ti}
             pf eq/eq tau (sa/sel tau (sa/upd tau eq er e) er) e @ t.
sa/sel|mc1:  {tau: ty}
             {eq: exp sa} {er: exp (sreg tau)} {e: exp tau}
             {er': exp (sreg tau)}
             {t: ti}
             pf eq/neq (sreg tau) er' er @ t
          -> pf eq/eq tau
                     (sa/sel tau (sa/upd tau eq er e) er')
                     (sa/sel tau eq er') @ t.
sa/sel|ext:  {eq: exp sa} {eq': exp sa} {t: ti}
             ({tau: ty} {ar: par (sreg tau)}
              pf eq/eq tau (sa/sel tau eq (par/` (sreg tau) ar))
                           (sa/sel tau eq' (par/` (sreg tau) ar)) @ t)
          -> pf sa/eq eq eq' @ t.
```

B.3.2.3 Extended States

```
qstate/pc|: {e: exp qstate} {t: ti}
            pf wd/eq (qstate/pc e) (state/pc (qstate/ss e)) @ t.
qstate/f|:  {e: exp qstate} {t: ti}
            pf wd/eq (qstate/f e) (state/f (qstate/ss e)) @ t.
qstate/g|:  {e: exp qstate} {t: ti}
            pf mapg/eq (qstate/g e) (state/g (qstate/ss e)) @ t.
qstate/s|:  {e: exp qstate} {t: ti}
            pf mapw/eq (qstate/s e) (state/s (qstate/ss e)) @ t.
qstate/m|:  {e: exp qstate} {t: ti}
            pf mapw/eq (qstate/m e) (state/m (qstate/ss e)) @ t.
```

B.3.2.4 Access Modes

```
% - semantics -

rel/app2#|eq_acc:  {c: con acc}
                   pf rel/app2# acc acc (eq/#eq acc) c c.
rel/app2#|neq_acc: {c1: con acc} {c2: con acc}
                   con/neq acc c1 c2
                -> pf rel/app2# acc acc (eq/#neq acc) c1 c2.

rel/app2#|acc/leq_ref:  {c: con acc}
                        pf rel/app2# acc acc acc/#leq c c.
rel/app2#|acc/leq_none: {c: con acc}
                        pf rel/app2# acc acc acc/#leq acc/none c.
rel/app2#|acc/leq_rw:   {c: con acc}
                        pf rel/app2# acc acc acc/#leq c acc/rw.

rel/app2#|acc/nleq_rd_none: pf rel/app2# acc acc acc/#nleq acc/rd acc/none.
rel/app2#|acc/nleq_wr_none: pf rel/app2# acc acc acc/#nleq acc/wr acc/none.
rel/app2#|acc/nleq_rw_none: pf rel/app2# acc acc acc/#nleq acc/rw acc/none.
rel/app2#|acc/nleq_wr_rd:   pf rel/app2# acc acc acc/#nleq acc/wr acc/rd.
rel/app2#|acc/nleq_rw_rd:   pf rel/app2# acc acc acc/#nleq acc/rw acc/rd.
rel/app2#|acc/nleq_rd_wr:   pf rel/app2# acc acc acc/#nleq acc/rd acc/wr.
rel/app2#|acc/nleq_rw_wr:   pf rel/app2# acc acc acc/#nleq acc/rw acc/wr.

% - truth -

acc/leq|order: {t: ti} pf rel/order acc acc/#leq @ t.
```

```
acc/leq|bot:   {t: ti} pf rel/bot acc acc/#leq acc/none @ t.
acc/leq|top:   {t: ti} pf rel/top acc acc/#leq acc/rw @ t.
```

B.3.2.5 Access Maps

```
mapa/leq|: {em1: exp mapa} {em2: exp mapa} {t: ti}
           pf mapa/leq em1 em2
              eqv all wd ri/- ([xn1: exp wd] all wd ri/- ([xn2: exp wd]
                  acc/leq (mapa/sel em1 xn1 xn2) (mapa/sel em2 xn1 xn2)
                  )) @ t.
```

B.3.2.6 Java Types

```
% - semantics -

rel/app2#|eq_jty:  {c: con jty}
                   pf rel/app2# jty jty (eq/#eq jty) c c.
rel/app2#|neq_jty: {c1: con jty} {c2: con jty}
                   con/neq jty c1 c2
                -> pf rel/app2# jty jty (eq/#neq jty) c1 c2.

fun/app1#|jty/size_bool:
  pf fun/app1# jty wd jty/#size jty/bool (wd/# 4).
fun/app1#|jty/size_char:
  pf fun/app1# jty wd jty/#size jty/char (wd/# 4).
fun/app1#|jty/size_byte:
  pf fun/app1# jty wd jty/#size jty/byte (wd/# 4).
fun/app1#|jty/size_short:
  pf fun/app1# jty wd jty/#size jty/short (wd/# 4).
fun/app1#|jty/size_int:
  pf fun/app1# jty wd jty/#size jty/int (wd/# 4).

% - truth -

jty/array|inj: {t: ti} pf fun/inj1 jty jty jty/#array @ t.
jty/inst|inj:  {t: ti} pf fun/inj1 wd jty jty/#inst @ t.

jty/size|i_array: {ety: exp jty} {t: ti}
                  pf wd/eq (jty/size (jty/array ety)) (wd/`# 4) @ t.
jty/size|i_inst:  {enc: exp wd} {t: ti}
                  pf wd/eq (jty/size (jty/inst enc)) (wd/`# 4) @ t.
```

B.3.2.7 Java Type Assignments

```
jta/leq|order: {t: ti} pf rel/order jta jta/#leq @ t.

jta/of|i_bool:   {eta: exp jta} {en: exp wd} {t: ti}
                 pf wd/ltu en (wd/`# 2) @ t
              -> pf jta/of eta en jty/`bool @ t.
jta/of|i_char:   {eta: exp jta} {en: exp wd} {t: ti}
                 pf wd/ltu en (wd/`# 65536) @ t
              -> pf jta/of eta en jty/`char @ t.
jta/of|i_byte:   {eta: exp jta} {en: exp wd} {t: ti}
                 pf wd/geq en (wd/`# 4294967168) @ t
              -> pf wd/lt en (wd/`# 128) @ t
              -> pf jta/of eta en jty/`byte @ t.
jta/of|i_short:  {eta: exp jta} {en: exp wd} {t: ti}
                 pf wd/geq en (wd/`# 4294934528) @ t
              -> pf wd/lt en (wd/`# 32768) @ t
              -> pf jta/of eta en jty/`short @ t.
jta/of|i_int:    {eta: exp jta} {en: exp wd} {t: ti}
                 pf jta/of eta en jty/`int @ t.
jta/of|i_array0: {eta: exp jta} {ety: exp jty} {t: ti}
                 pf jta/of eta `0w (jty/array ety) @ t.
```

```
jta/of|i_inst0:  {eta: exp jta} {enc: exp wd} {t: ti}
                 pf jta/of eta `0w (jty/inst enc) @ t.

jta/ptr|fun:   {t: ti} pf rel/fun2 (pair jta wd) jty jta/#ptr @ t.
jta/field|fun: {t: ti} pf rel/fun2 (trip jta wd wd) jty jta/#field @ t.
```

B.3.2.8 Java Type Environments

```
jts/leq|:
  {ets1: exp jts} {ets2: exp jts} {t: ti}
  pf     jts/leq ets1 ets2
     eqv     jta/leq (jts/ta ets1) (jts/ta ets2)
         and mapa/leq (jts/am ets1) (jts/am ets2) @ t.

jts/of|:
  {ets: exp jts} {en: exp wd} {ety: exp jty} {t: ti}
  pf     jts/of ets en ety
     eqv     jts/valid ets
         and jta/of (jts/ta ets) en ety @ t.

jts/ptr|:
  {ets: exp jts} {enp: exp wd} {ety: exp jty} {ea: exp acc} {t: ti}
  pf     jts/ptr ets enp ety ea
     eqv     jts/valid ets
         and jta/ptr (jts/ta ets) enp ety
         and acc/eq (mapa/sel (jts/am ets) enp (jty/size ety)) ea @ t.

jts/field|:
  {ets: exp jts} {enc: exp wd} {enf: exp wd} {ety: exp jty} {t: ti}
  pf     jts/field ets enc enf ety
     eqv jta/field (jts/ta ets) enc enf ety @ t.

jts/mem|:
  {ets: exp jts} {em: exp mapw} {t: ti}
  pf jts/mem ets em
     eqv
     all wd ri/- ([xnp: exp wd] all jty ri/- ([xty: exp jty]
         jts/valid ets
     and jta/mem (jts/ta ets) em
     and (    jta/ptr_elem (jts/ta ets) em xnp xty
          imp acc/eq (mapa/sel (jts/am ets) xnp (jty/size xty)) acc/`rw)
     )) @ t.
```

B.3.2.9 Safety

```
prog/fetch|safe/nextq:
  {t: ti}
  pf all qstate ri/+ ([x: exp qstate] all inst ri/+ ([xi^: exp inst]
         qstate/eq x `sq
     imp prog/fetch `pm (qstate/pc x) xi^
     imp nextT (eq/eq sa `q (safe/nextq x xi^))
     )) @ t.

safe|esp_under:
  {t: ti}
  pf wd/gequ (wd/add `g_sp safe/`page) `g_sp @ t.

safe/nextq|inst/mov:
  {e: exp state} {eq: exp sa} {eni: exp wd} {eea1: exp ea} {eea2: exp ea}
  {t: ti}
  pf sa/eq (safe/nextq (qstate/make2 e eq) (inst/mov eni eea1 eea2)) eq @ t.
safe/nextq|inst/xchg:
  {e: exp state} {eq: exp sa} {eni: exp wd} {eea: exp ea} {er: exp greg}
  {t: ti}
```

```
  pf sa/eq (safe/nextq (qstate/make2 e eq) (inst/xchg eni eea er)) eq @ t.
safe/nextq|inst/lea:
  {e: exp state} {eq: exp sa} {eni: exp wd} {eea: exp ea} {er: exp greg}
  {t: ti}
  pf sa/eq (safe/nextq (qstate/make2 e eq) (inst/lea eni eea er)) eq @ t.
safe/nextq|inst/push:
  {e: exp state} {eq: exp sa} {eni: exp wd} {eea: exp ea}
  {t: ti}
  pf sa/eq (safe/nextq (qstate/make2 e eq) (inst/push eni eea)) eq @ t.
safe/nextq|inst/pop:
  {e: exp state} {eq: exp sa} {eni: exp wd} {eea: exp ea}
  {t: ti}
  pf sa/eq (safe/nextq (qstate/make2 e eq) (inst/pop eni eea)) eq @ t.
safe/nextq|inst/op1:
  {e: exp state} {eq: exp sa} {eni: exp wd} {eop: exp op/1} {eea: exp ea}
  {t: ti}
  pf sa/eq (safe/nextq (qstate/make2 e eq) (inst/op1 eni eop eea)) eq @ t.
safe/nextq|inst/op2:
  {e: exp state} {eq: exp sa}
  {eni: exp wd} {eop: exp op/2} {eea1: exp ea} {eea2: exp ea}
  {t: ti}
  pf sa/eq (safe/nextq (qstate/make2 e eq) (inst/op2 eni eop eea1 eea2))
         eq @ t.
safe/nextq|inst/op2n:
  {e: exp state} {eq: exp sa}
  {eni: exp wd} {eop: exp op/2} {eea1: exp ea} {eea2: exp ea}
  {t: ti}
  pf sa/eq (safe/nextq (qstate/make2 e eq) (inst/op2n eni eop eea1 eea2))
         eq @ t.
safe/nextq|inst/op3:
  {e: exp state} {eq: exp sa}
  {eni: exp wd} {eop1: exp op/3} {eop2: exp op/3}
  {eea: exp ea} {er1: exp greg} {er2: exp greg}
  {t: ti}
  pf sa/eq (safe/nextq (qstate/make2 e eq)
                       (inst/op3 eni eop1 eop2 eea er1 er2))
         eq @ t.
safe/nextq|inst/jmp:
  {e: exp state} {eq: exp sa} {eni: exp wd} {eea: exp ea}
  {t: ti}
  pf sa/eq (safe/nextq (qstate/make2 e eq) (inst/jmp eni eea)) eq @ t.
safe/nextq|inst/j:
  {e: exp state} {eq: exp sa} {eni: exp wd} {ecop: exp cop} {en: exp wd}
  {t: ti}
  pf sa/eq (safe/nextq (qstate/make2 e eq) (inst/j eni ecop en)) eq @ t.
safe/nextq|inst/call:
  {epc: exp wd} {ef: exp wd} {eg: exp mapg} {es: exp mapw} {em: exp mapw}
  {eq: exp sa}
  {eni: exp wd} {eea: exp ea}
  {t: ti}
  pf sa/eq (safe/nextq (qstate/make epc ef eg es em eq) (inst/call eni eea))
         (safe/upd_fps
            eq
            (list/cons wd
                       (wd/add (mapg/sel eg `esp) (wd/`# ~4))
                       (safe/sel_fps eq))) @ t.
safe/nextq|inst/ret:
  {e: exp state} {eq: exp sa} {eni: exp wd}
  {t: ti}
  pf sa/eq (safe/nextq (qstate/make2 e eq) (inst/ret eni))
         (safe/upd_fps eq (list/tail wd (safe/sel_fps eq))) @ t.

safe/next|:
   {esq: exp qstate} {ei^: exp inst} {t: ti}
```

```
   pf qstate/eq
        (safe/next esq ei^)
        (qstate/make2 (inst/next (pair/left state sa esq) ei^)
                      (safe/nextq esq ei^)) @ t.

safe/sp|:
   {e: exp qstate} {en: exp wd} {en': exp wd} {t: ti}
   pf wd/eq (wd/sub (list/head wd (qstate/fps e)) en) en' @ t
-> pf safe/sp e en
      eqv wd/ltu en' safe/`page and wd/eq (wd/and en' (wd/`# 3)) `0w @ t.

safe/rd_s|:
   {e: exp qstate} {en: exp wd} {en': exp wd} {t: ti}
   pf wd/eq (wd/sub en (qstate/esp e)) en' @ t
-> pf safe/rd_s e en
      eqv wd/ltu en' safe/`page and wd/eq (wd/and en' (wd/`# 3)) `0w @ t.

safe/wr_s|:
   {e: exp qstate} {en: exp wd} {en': exp wd} {en'': exp wd} {t: ti}
   pf wd/eq (wd/sub en (qstate/esp e)) en'' @ t
-> pf safe/wr_s e en en'
      eqv wd/ltu en'' safe/`page and wd/eq (wd/and en'' (wd/`# 3)) `0w @ t.

safe/rd_m|:
   {e: exp qstate} {en: exp wd} {t: ti}
   pf safe/rd_m e en
      eqv     acc/leq acc/`rd (mapa/sel (qstate/accm e) en (wd/`# 4))
          and wd/eq (wd/and en (wd/`# 3)) `0w @ t.

% don't check en' here: type of en' is only needed to show that memory is
% still well-formed
safe/wr_m|:
   {e: exp qstate} {en: exp wd} {en': exp wd} {t: ti}
   pf safe/wr_m e en en'
      eqv     acc/leq acc/`wr (mapa/sel (qstate/accm e) en (wd/`# 4))
          and wd/eq (wd/and en (wd/`# 3)) `0w @ t.

safe/rd_ea|ea/i:
  {e: exp qstate} {en: exp wd} {t: ti}
  pf safe/rd_ea e (ea/i en) @ t.
safe/rd_ea|ea/r:
  {e: exp qstate} {er: exp greg} {t: ti}
  pf safe/rd_ea e (ea/r er) @ t.
safe/rd_ea|ea/s:
  {e: exp qstate} {ema: exp ma} {t: ti}
  pf safe/rd_ea e (ea/s ema) eqv safe/rd_s e (ma/addr (qstate/g e) ema) @ t.
safe/rd_ea|ea/m:
  {e: exp qstate} {ema: exp ma} {t: ti}
  pf safe/rd_ea e (ea/m ema) eqv safe/rd_m e (ma/addr (qstate/g e) ema) @ t.

% no safe/wr_ea|ea/i
safe/wr_ea|ea/r:
   {e: exp qstate} {er: exp greg} {en: exp wd} {t: ti}
   pf eq/neq greg er `esp @ t
-> pf safe/wr_ea e (ea/r er) en @ t.
safe/wr_ea|ea/sp:
  {e: exp qstate} {en: exp wd} {t: ti}
  pf safe/wr_ea e (ea/r `esp) en eqv safe/sp e en @ t.
safe/wr_ea|ea/s:
  {e: exp qstate} {ema: exp ma} {en: exp wd} {t: ti}
  pf safe/wr_ea e (ea/s ema) en
     eqv safe/wr_s e (ma/addr (qstate/g e) ema) en @ t.
safe/wr_ea|ea/m:
  {e: exp qstate} {ema: exp ma} {en: exp wd} {t: ti}
```

```
  pf safe/wr_ea e (ea/m ema) en
     eqv safe/wr_m e (ma/addr (qstate/g e) ema) en @ t.

safe/inst|inst/mov:
  {e: exp qstate} {eni: exp wd} {eea1: exp ea} {eea2: exp ea} {t: ti}
  pf safe/inst e (inst/mov eni eea1 eea2)
     eqv     safe/rd_ea e eea1
         and safe/wr_ea e eea2 (ea/sel (qstate/ss e) eea1) @ t.
safe/inst|inst/xchg:
  {e: exp qstate} {eni: exp wd} {eea: exp ea} {er: exp greg} {t: ti}
  pf safe/inst e (inst/xchg eni eea er)
     eqv     safe/rd_ea e eea
         and safe/wr_ea e (ea/r er) (ea/sel (qstate/ss e) eea)
         and safe/wr_ea e eea (mapg/sel (qstate/g e) er) @ t.
safe/inst|inst/lea:
  {e: exp qstate} {eni: exp wd} {eea: exp ea} {er: exp greg} {t: ti}
  pf safe/inst e (inst/lea eni eea er)
     eqv safe/wr_ea e (ea/r er) (ea/addr (qstate/ss e) eea) @ t.
safe/inst|inst/push:
  {e: exp qstate} {eni: exp wd} {eea: exp ea} {t: ti}
  pf safe/inst e (inst/push eni eea) eqv safe/rd_ea e eea @ t.
safe/inst|inst/pop:
  {e: exp qstate} {eni: exp wd} {eea: exp ea} {t: ti}
  pf safe/inst e (inst/pop eni eea)
     eqv safe/wr_ea e
                  eea
                  (mapw/sel (qstate/s e) (mapg/sel (qstate/g e) ‘esp)) @ t.
safe/inst|inst/op1:
  {e: exp qstate} {eni: exp wd} {eop: exp op/1} {eea: exp ea} {t: ti}
  pf safe/inst e (inst/op1 eni eop eea)
     eqv     safe/rd_ea e eea
         and safe/wr_ea e eea (op/app1 eop (ea/sel (qstate/ss e) eea)) @ t.
safe/inst|inst/op2:
  {e: exp qstate}
  {eni: exp wd} {eop: exp op/2} {eea1: exp ea} {eea2: exp ea}
  {t: ti}
  pf safe/inst e (inst/op2 eni eop eea1 eea2)
     eqv     safe/rd_ea e eea1 and safe/rd_ea e eea2
         and safe/wr_ea e eea2 (op/app2 eop
                                     (ea/sel (qstate/ss e) eea2)
                                     (ea/sel (qstate/ss e) eea1)) @ t.
safe/inst|inst/op2n:
  {e: exp qstate}
  {eni: exp wd} {eop: exp op/2} {eea1: exp ea} {eea2: exp ea}
  {t: ti}
  pf safe/inst e (inst/op2n eni eop eea1 eea2)
     eqv safe/rd_ea e eea1 and safe/rd_ea e eea2 @ t.
safe/inst|inst/op3:
   {e: exp qstate}
   {eni: exp wd} {eop1: exp op/3} {eop2: exp op/3}
   {eea: exp ea} {er1: exp greg} {er2: exp greg}
   {en’: exp wd} {en1’: exp wd} {en2’: exp wd}
   {t: ti}
   pf wd/eq (ea/sel (qstate/ss e) eea) en’ @ t
-> pf wd/eq (mapg/sel (qstate/g e) er1) en1’ @ t
-> pf wd/eq (mapg/sel (qstate/g e) er2) en2’ @ t
-> pf safe/inst e (inst/op3 eni eop1 eop2 eea er1 er2)
      eqv     safe/rd_ea e eea
          and safe/wr_ea e (ea/r er1) (op/app3 eop1 en1’ en2’ en’)
          and safe/wr_ea e (ea/r er2) (op/app3 eop2 en1’ en2’ en’) @ t.
safe/inst|inst/jmp:
  {e: exp qstate} {eni: exp wd} {eea: exp ea} {t: ti}
  pf safe/inst e (inst/jmp eni eea) eqv safe/rd_ea e eea @ t.
safe/inst|inst/j:
```

```
  {e: exp qstate} {eni: exp wd} {ecop: exp cop} {en: exp wd} {t: ti}
  pf safe/inst e (inst/j eni ecop en) @ t.
safe/inst|inst/call:
  {e: exp qstate} {eni: exp wd} {eea: exp ea} {t: ti}
  pf safe/inst e (inst/call eni eea) eqv safe/rd_ea e eea @ t.
safe/inst|inst/ret:
  {e: exp qstate} {eni: exp wd} {t: ti}
  pf safe/inst e (inst/ret eni) @ t.
```

Appendix C

Benchmark Programs

C.1 Java Source Code

C.1.1 Alloc

```
public class Alloc
{
  public static int[] alloc()
  {
    return new int[10];
  }
}
```

C.1.2 Binary Search

```
public class BinarySearch
{
  public static int search(final int[] a, final int n)
  {
    int index = 0, count = a.length;

    while (count>0)
    {
      final int i = index+count/2;

      if (n<a[i])
      {
        count = i-index;
      }
      else if (n>a[i])
      {
        count = index+count-(i+1);
        index = i+1;
      }
      else
      {
        return i;
      }
    }

    return -1;
  }
}
```

C.1.3 Bubble Sort

```
public class BubbleSort
{
  public static void sort(int[] a)
  {
    final int count = a.length;

    if (count<2)
      return;

    for (;;)
    {
      boolean done = true;

      for (int i = 1; i<count; i++)
      {
        final int n1 = a[i-1];
        final int n2 = a[i];

        if (n1>n2)
        {
          a[i-1] = n2;
          a[i]   = n1;
          done   = false;
        }
      }

      if (done)
        break;
    }
  }
}
```

C.1.4 Checksum

```
public class Checksum
{
  public static int checksum(int a[])
  {
    int i, n = 0;

    for (i = 0; i<=a.length; i++)
      n += a[i];

    return n;
  }
}
```

C.1.5 Clone

```
public class Clone
{
  public static int[] clone(int a[])
  {
    int i = 0, count = a.length;
    int a2[] = new int[count];

    for (i = 0; i<count; i++)
      a2[i] = a[i];

    return a2;
  }
}
```

C.1.6 Dec

```
public class Dec
{
  public static int dec(int n)
  {
    int i = n;

    while (--i>0)
      ;

    return i;
  }
}
```

C.1.7 Fact

```
public class Fact
{
  public static int fact(int n)
  {
    int i, result = 1;

    for (i = 1; i<=n; i++)
      result *= i;

    return result;
  }
}
```

C.1.8 Fib

```
public class Fib
{
  public static int fib(int n)
  {
    if (n<=0)
      return 0;

    {
      int i, prev2 = 0, prev1 = 1, result = 1;

      for (i = 2; i<=n; i++)
      {
        result = prev1+prev2;
        prev2  = prev1;
        prev1  = result;
      }

      return result;
    }
  }
}
```

C.1.9 Filter

```
public class Filter
{
  public static int filter(int n)
  {
    return n>=0 ? 1 : 0;
  }
}
```

C.1.10 Heap Sort

```
public class Heap
{
  /* active size of heap is in a[0]
     elements are in a[1] ... a[a[0]]
     a[0]<a.length */

  static void heapify(final int[] a, final int index)
  {
    final int count = a[0];
    final int left  = index*2;
    final int right = index*2+1;

    int next = index;

    if (left<=count && a[left]>a[next])
      next = left;

    if (right<=count && a[right]>a[next])
      next = right;

    if (next!=index)
    {
      final int n1 = a[index];
      final int n2 = a[next];

      a[index] = n2;
      a[next]  = n1;

      heapify(a, next);
    }
  }

  static void make(final int[] a)
  {
    final int count = a[0];

    for (int i = count/2; i>=1; i--)
      heapify(a, i);
  }

  static int top(final int[] a)
  {
    return a[1];
  }

  static int pop(final int[] a)
  {
    final int count = a[0];
    final int n     = a[1];

    a[0] = count-1;
    a[1] = a[count];

    heapify(a, 1);

    return n;
  }

  static void add(final int[] a, final int n)
  {
    final int count = a[0];
```

```
    a[0] = count+1;

    int i = count+1;

    while (i>1 && a[i/2]<n)
    {
      a[i] = a[i/2];

      i /= 2;
    }

    a[i] = n;
  }

  static void sort(final int[] a)
  {
    final int count = a.length-1;

    a[0] = count;

    make(a);

    for (int i = count; i>=2; i--)
    {
      final int n1 = a[1];
      final int n2 = a[i];

      a[1] = n2;
      a[i] = n1;

      a[0]--;

      heapify(a, 1);
    }
  }
}
```

C.1.11 Huffman

```
public class Huffman
{
  /* active size of heap is in a[0]
     elements are in a[1] ... a[a[0]]
     a[0]<a.length */

  static void heapify(final int[] a, final int index, final int[] freq)
  {
    final int count = a[0];
    final int left  = index*2;
    final int right = index*2+1;

    int next = index;

    if (left<=count && freq[a[left]]<freq[a[next]])
      next = left;

    if (right<=count && freq[a[right]]<freq[a[next]])
      next = right;

    if (next!=index)
    {
      final int n1 = a[index];
      final int n2 = a[next];
```

```
    a[index] = n2;
    a[next]  = n1;

    heapify(a, next, freq);
  }
}

static void make(final int[] a, final int[] freq)
{
  final int count = a[0];

  for (int i = count/2; i>=1; i--)
    heapify(a, i, freq);
}

static int pop(final int[] a, final int[] freq)
{
  final int count = a[0];
  final int n     = a[1];

  a[0] = count-1;
  a[1] = a[count];

  heapify(a, 1, freq);

  return n;
}

static void add(final int[] a, final int n, final int[] freq)
{
  final int count = a[0];

  a[0] = count+1;

  int i = count+1;

  while (i>1 && freq[a[i/2]]<freq[n])
  {
    a[i] = a[i/2];

    i /= 2;
  }

  a[i] = n;
}

static int huffman(final int[] freq, final int count,
                   final int[] left, final int[] right)
{
  final int[] a = new int[count*2+1];

  a[0] = count;

  for (int i = 1; i<=count; i++)
    a[i] = i;

  make(a, freq);

  int next = count+1;

  for (int i = 1; i<=count-1; i++)
  {
    final int x = pop(a, freq);
    final int y = pop(a, freq);
```

```
      final int z = next++;

      left[z]  = x;
      right[z] = y;
      freq[z]  = freq[x]+freq[y];

      add(a, z, freq);
    }

    return pop(a, freq);
  }
}
```

C.1.12 Loop

```
public class Loop
{
  public static void loop()
  {
    for (;;)
      ;
  }

  public static void nop()  /* SpecialJ needs this */
  {
  }
}
```

C.1.13 Matrix

```
public class Matrix
{
  public static void id(final int[][] a)
  {
    for (int i = 0; i<a.length; i++)
      for (int j = 0; j<a[0].length; j++)
        a[i][j] = i==j ? 1 : 0;
  }

  public static boolean equal(final int[][] a1, final int[][] a2)
  {
    for (int i = 0; i<a1.length; i++)
      for (int j = 0; j<a1[0].length; j++)
        if (a1[i][j]!=a2[i][j])
          return false;

    return true;
  }

  public static void copy(final int[][] a, final int[][] b)
  {
    for (int i = 0; i<a.length; i++)
      for (int j = 0; j<a[0].length; j++)
        b[i][j] = a[i][j];
  }

  public static int min(final int[][] a)
  {
    int n = 2147483647;

    for (int i = 0; i<a.length; i++)
      for (int j = 0; j<a[0].length; j++)
        if (a[i][j]<n)
          n = a[i][j];
```

```
    return n;
  }

  public static int max(final int[][] a)
  {
    int n = -2147483648;

    for (int i = 0; i<a.length; i++)
      for (int j = 0; j<a[0].length; j++)
        if (a[i][j]>n)
          n = a[i][j];

    return n;
  }

  public static void add(final int[][] a1, final int[][] a2, final int[][] b)
  {
    for (int i = 0; i<a1.length; i++)
      for (int j = 0; j<a1[0].length; j++)
        b[i][j] = a1[i][j]+a2[i][j];
  }

  public static void sub(final int[][] a1, final int[][] a2, final int[][] b)
  {
    for (int i = 0; i<a1.length; i++)
      for (int j = 0; j<a1[0].length; j++)
        b[i][j] = a1[i][j]-a2[i][j];
  }

  public static int mul1(final int[][] a1, final int[][] a2, int i, int j)
  {
    int n = 0;

    for (int k = 0; k<a1[i].length; k++)
      n += a1[i][k]*a2[k][j];

    return n;
  }

  public static void mul(final int[][] a1, final int[][] a2, final int[][] b)
  {
    for (int i = 0; i<a1.length; i++)
      for (int j = 0; j<a2[0].length; j++)
        b[i][j] = mul1(a1, a2, i, j);
  }

  public static void transp(final int[][] a, final int[][] b)
  {
    for (int i = 0; i<a.length; i++)
      for (int j = 0; j<a[i].length; j++)
      {
        b[j][i] = a[i][j];
      }
  }
}
```

C.1.14 Matrix Multiply

```
public class MatrixMultiply
{
  public static int multiply1(final int[][] a1, final int[][] a2, int i, int j)
  {
    int n = 0;
```

```
    for (int k = 0; k<a1[i].length; k++)
      n += a1[i][k]*a2[k][j];

    return n;
  }

  public static void multiply(final int[][] a1, final int[][] a2,
                              final int[][] b)
  {
    for (int i = 0; i<a1.length; i++)
      for (int j = 0; j<a2[0].length; j++)
        b[i][j] = multiply1(a1, a2, i, j);
  }
}
```

C.1.15 Matrix Transpose

```
public class MatrixTranspose
{
  public static void transpose(final int[][] a, final int[][] b)
  {
    for (int i = 0; i<a.length; i++)
      for (int j = 0; j<a[i].length; j++)
      {
        b[j][i] = a[i][j];
      }
  }
}
```

C.1.16 Merge Sort

```
public class MergeSort
{
  private static int[] sub(final int[] a, final int index, final int count)
  {
    final int[] a1 = new int[count];

    for (int i = 0; i<count; i++)
      a1[i] = a[index+i];

    return a1;
  }

  private static void merge(final int[] a1, final int[] a2, final int[] a)
  {
    final int count1 = a1.length;
    final int count2 = a2.length;

    int i1 = 0;
    int i2 = 0;
    int i  = 0;

    while (i1<count1 && i2<count2)
    {
      final int n1 = a1[i1];
      final int n2 = a2[i2];

      if (n1<n2)
      {
        a[i++] = n1;
        i1++;
      }
      else
```

```
      {
        a[i++] = n2;
        i2++;
      }
    }

    while (i1<count1)
      a[i++] = a1[i1++];

    while (i2<count2)
      a[i++] = a2[i2++];
  }

  public static void sort(final int[] a)
  {
    final int count = a.length;

    if (count>2)
    {
      final int count1 = count/2;
      final int count2 = count-count1;

      final int[] a1 = sub(a, 0, count1);
      final int[] a2 = sub(a, count1, count2);

      sort(a1);
      sort(a2);

      merge(a1, a2, a);
    }
    else if (count>1)
    {
      final int n0 = a[0];
      final int n1 = a[1];

      if (n0>n1)
      {
        a[0] = n1;
        a[1] = n0;
      }
    }
  }
}
```

C.1.17 Min

```
public class Min
{
  public static int min(int n1, int n2)
  {
    return Math.min(n1, n2);
  }
}
```

C.1.18 NAbs

```
public class NAbs
{
  public static int nabs(int n)
  {
    return -Math.abs(n);
  }
}
```

C.1.19 Nop

```
public class Nop
{
  public static void nop()
  {
  }
}
```

C.1.20 Not

```
public class Not
{
  public static boolean not(boolean flag)
  {
    return !flag;
  }
}
```

C.1.21 N Queens

```
/*
 * This is an adaptation of Tom Murphy's code at
 * http://www.andrew.cmu.edu/~twm/queens/nqueens.c
 */
public class NQueens
{
  private static void set
    (final int[] rows, final int[] diagu, final int[] diagd,
     final int z, final int r, final int flag)
  {
    rows[r]                  = flag;
    diagu[z+r]               = flag;
    diagd[rows.length-1-r+z] = flag;
  }

  public static int[] queens(final int n)
  {
    final int[] rows  = new int[n];
    final int[] diagu = new int[n*2];
    final int[] diagd = new int[n*2];
    final int[] board = new int[n];

    int z = 0;

    for (;;)
    {
      final int r = board[z];

      if (!(rows[r]!=0 || diagu[z+r]!=0 || diagd[n-1-r+z]!=0))
      {
        set(rows, diagu, diagd, z, r, 1);
        z++;

        if (z==n)
          return board;

        board[z] = 0;
      }
      else
      {
        board[z]++;

        while (board[z]==n)
```

```
        {
          z--;
          set(rows, diagu, diagd, z, board[z], 0);
          board[z]++;
        }
      }
    }
  }
}
```

C.1.22 Packet

```
public class Packet
{
  public static boolean packet(int a[])
  {
    return a[0]==1234567;
  }
}
```

C.1.23 Quicksort

```
public class QuickSort
{
  public static void sort(int[] a, int index0, int index1)
  {
    if (index1<=index0+1)
    {
      if (index1>index0)
      {
        final int n0 = a[index0];
        final int n1 = a[index1];

        if (n0>n1)
        {
          a[index0] = n1;
          a[index1] = n0;
        }
      }

      return;
    }

    {
      final int n0 = a[index0];

      int i = index0-1;
      int j = index1+1;

      for (;;)
      {
        while (a[++i]<n0)
          ;

        while (a[--j]>n0)
          ;

        if (i>=j)
          break;

        {
          final int ni = a[i];
          final int nj = a[j];
```

```
          a[i] = nj;
          a[j] = ni;
        }
      }

      sort(a, index0, j);
      sort(a, j+1, index1);
    }
  }
}
```

C.1.24 Reverse

```
public class Reverse
{
  public static void reverse(int a[])
  {
    int i = 0, count = a.length/2, last = a.length-1;

    for (i = 0; i<count; i++)
      a[i] = a[last-i];
  }
}
```

C.1.25 Swap

```
public class Swap
{
  public static void swap(int a[])
  {
    int n = a[0];

    a[0] = a[1];
    a[1] = n;
  }
}
```