Compiling with Proofs

George Ciprian Necula

September 18, 1998

CMU–CS–98–154

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
Robert Harper
Frank Pfenning
Greg Nelson, DEC SRC

This research was sponsored in part by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software," ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050.

Keywords: mobile code, agents, proof-carrying code, safety properties, security properties, type safety, program verification, certifying compilation, theorem proving, decision procedures, proof checking, proof representation, logical frameworks, LF, LF term reconstruction, software engineering, compiler verification.

Abstract

One of the major challenges of building software systems is to ensure that the various components fit together in a well-defined manner. This problem is exacerbated by the recent advent of software components whose origin is unknown or inherently untrusted, such as mobile code or user extensions for operating-system kernels or database servers. Such extensions are useful for implementing an efficient interaction model between a client and a server because several data exchanges between them can be saved at the cost of a single code exchange.

In this dissertation, I propose to tackle such system integrity and security problems with techniques from mathematical logic and programming-language semantics. I propose a framework, called *proof-carrying code*, in which the extension provider sends along with the extension code a representation of a formal proof that the code meets certain safety and correctness requirements. Then, the code receiver can ensure the safety of executing the extension by validating the attached proof. The major advantages of proof-carrying code are that it requires a simple trusted infrastructure and that it does not impose run-time penalties for the purpose of ensuring safety.

In addition to the concept of proof-carrying code, this dissertation contributes the idea of certifying compilation. A *certifying compiler* emits, in addition to optimized target code, function specifications and loop invariants that enable a theorem-proving agent to prove non-trivial properties of the target code, such as type safety. Such a certifying compiler, along with a proof-generating theorem prover, is not only a convenient producer of proof-carrying code but also a powerful software-engineering tool. The certifier also acts as an effective referee for the correctness of each compilation, thus simplifying considerably compiler testing and maintenance.

A complete system for proof-carrying code must also contain a *proof-generating theorem prover* for the purpose of producing the attached proofs of safety. This dissertation shows how standard decision procedures can be adapted so that they can produce detailed proofs of the proved predicates and also how these proofs can be encoded compactly and checked efficiently. Just like for the certifying compiler, a proof-generating theorem prover has significant software-engineering advantages over a traditional prover. In this case, a simple proof checker can ensure the soundness of each successful proving task and indirectly assist in testing and maintenance of the theorem prover.

for Simona

who suffered with forbearance
the widowhood of a graduate student's wife
and nonetheless provided generous support

for Deanna & Sylvia

who suffered with less forebearance
but no less love

Acknowledgments

I am greatly indebted to many people for helping me complete this dissertation. First, and foremost, of course, is my friend and advisor, Peter Lee. Under his guidance, I have made significant progress in many of the skills defining a researcher. He taught me not only how to recognize and pursue promising research problems, but also that a researcher has the moral obligation to strive for perfection when communicating research results. Peter's technical contributions to this work are deep and inextricable. He has also read each part of this thesis several times, and his comments have improved it immensely on each iteration. Peter is a very gifted and dedicated researcher and teacher. I am lucky to have studied under him and I only hope that a small portion of his talents has worn off on me during our many enjoyable meetings and discussions together.

I also owe a special debt to the other members of my thesis committee. Robert Harper and Frank Pfenning taught me to be rigorous. First, their courses on programming-language semantics and logic have laid the foundations for my thesis work. Then, their comments on this dissertation have increased its technical quality significantly. Greg Nelson also has played a dual role in my research career. First, during my summer internship at DEC SRC, he and the other members of the ESC project introduced me to program verification and theorem proving, which ended up being essential ingredients of this work. Then, as an external member of my committee, Greg suggested many improvements to this dissertation. Finally, John Reynolds has studied the manuscript very carefully and his numerous comments have improved both the organization of the material and its technical quality.

In addition to my thesis committee, I had fruitful discussions of this material and have received comments and suggestions from Martin Abadi, Andrew Appel, Andrew Bernard, Mihai Budiu, Jose Carlos Brustoloni, Karl Crary, Peter Dinda, Brian Ford, Lal George, Garth Gibson, Jim Horning, Dexter Kozen, HT Kung, Jay Lepreau, Pat Lincoln, John Mitchell, Greg Morrisett, Brian Noble, Rob O'Callahan, Susan Owicki, Fred Schneider, Natarajan Shankar, Olin Shivers, Ion Stoica and David Swasey. Thanks everyone!

During my stay at Carnegie Mellon I was lucky to have many colleagues who made graduate school more enjoyable. Of them, I remember with distinct pleasure my officemates Joe Tebelskis and Jose Carlos Brustoloni and the Romanian lunch bunch—Raluca and Mihai Budiu, Marius Minea and Ion Stoica.

Although the concrete origins of this dissertation can be traced to my five years of graduate school at Carnegie Mellon University, I consider it as the culmination of many more years of education and scientific pursuit. I would like to take this opportunity to express my gratitude towards a long series of teachers and mentors that both encouraged and quenched my thirst for knowledge and served as role models in various stages of my life. My first serious encounter with science was in the field of physics, under the guidance of my high-school teacher Romulus Pop, in my country of birth, Romania. First, his charisma won me to physics. Then, his high standards and ambitions rubbed off on me and with his help and mentoring I became successful in national and international competitions. Sure, what I was doing at the time could hardly be called research, but nevertheless the lessons learned in the process have been invaluable.

I am grateful to Dorina Margineanţu who showed me a computer for the first time and who encouraged and helped my mother to buy a ZX Spectrum for me. To tip the balance in favor of computer science and away from physics was another charismatic teacher at my high school, Radu Sima. Since then, Radu has become a very close friend, a mentor and the best man at my wedding.

Among the college teachers that have played a significant role in my education are Irina Athanasiu, Nicolae Ţăpuş, Dan Suciu, Cristian Giumale and Paul Flondor, all at the Polytechnic University of Bucharest. A special place in my heart goes to Irina Athanasiu who, through parental-like advice and encouragement, ensured that I maintained high goals. She was the one to encourage me to pursue graduate studies and the one who, in cooperation with Val Breazu-Tannen, bootstraped my research career.

Last but not the least, special thanks go to my family. I thank my mother Ecaterina and my brother Edmond for their love and support. I could not have finished this research in time if my wife's parents, Florina and Mihai Manoliu, had not been helping with childcare for an entire year. Finally, I thank my wife Simona for her constant support and my daughters Deanna and Sylvia who gave me numerous moments of joy and the strength required to finish this research and to write this dissertation.

Contents

List of Figures

Chapter 1

Introduction

To provide access to its internal data and resources, a server traditionally exports a static collection of basic operations on the corresponding data, such as the set of system-call entries exported by an operating system kernel. In order to use such a *static interface*, a client must decompose its computation into a sequence of operations mapping directly to those exported by the server. A more flexible and more efficient interaction model is based on *active interfaces*. Under the active interface paradigm, the client creates and uploads a program—also referred to as an *agent* or *mobile code*—that the server installs and executes in its own environment with direct access to data and resources. In such a situation, it is convenient to refer to the client as the *code producer* and to the server as the *code receiver*.

The major advantage of the active interface paradigm is that it can reduce the amount of communication between the client and the server. To illustrate this point, consider the case when the server is a spacecraft that is gathering large amounts of data in a remote part of the Universe. The latency of the communication with the Earth can be on the order of hours, and is limited by the speed of light; the available bandwidth also seems to be limited at present to a few kilobytes per second. It seems therefore obvious that the data must be processed onboard and only select results should be sent to Earth. Furthermore, we cannot afford to fix the set of data-analysis programs for the entire duration of the trip. The natural solution to these constraints is to use an active interface, so that the Earth station can update or even replace the set of data-analysis programs that is executed executed onboard. Also, this particular example illustrates a situation when it is necessary for the agent to execute as efficiently as possible, in order to make best use of the scarce supply of power. This suggests that whatever technique we use to implement active interfaces, it must not penalize the performance of the uploaded agents over resident programs.

A more subtle advantage of the active interface model of interaction is that it allows the server to export safely a more flexible lower-level interface to the agents and indirectly to the code producers. In the traditional client-server interaction model, the server does not have advance knowledge of the agent program (residing on the client in this case) and thus does not know it uses the data and services provided. This means that a traditional server must not export secret data or give access to low-level services that could be used to subvert the system. In the active interface model, on the other hand, the code receiver can examine the agent program before running it and can ensure that even though it has access to secret data it does not leak it to untrusted parties, or that it uses properly the provided low-level services. Consider for example, an agent that prepares a tax return based on the financial data provided by the code receiver. Because the code receiver controls the agent program it can ensure that the financial data is not leaked. Similarly, a code receiver might allow an agent to temporarily disable the interrupts if, by inspecting the agent program, it can verify that the agent will keep the interrupts disabled for a short period of time.

Given the flexibility and efficiency advantages of active interfaces, we might ask ourselves why do scenarios like those described above sound more like science fiction than reality today? I think it is mostly because of that security and complexity concerns that are raised by active interfaces. The major difficulty in deploying an active interface is to ensure the security of the code-receiver system and its data in the presence of untrusted agents. This involves not only protection against malicious agents, but also protection against erroneous code that might be supplied by known and otherwise trusted code producers. The latter scenario is worth considering when the potential damages due to an error are high, such as in the spacecraft example.

Furthermore, one might be reasonably concerned that the mechanisms that must be used to enforce the security aspect of active interfaces lead to agent execution overhead and to a larger and less trustworthy receiver-side infrastructure.

My Thesis. My thesis is that ideas from logic and programming languages can and should be used to ensure the safety of executing software agents by means of static checking, without sacrificing performance and without relying on personal authority. Furthermore, this can be achieved with a small trusted infrastructure on the receiver-side. In order to minimize the complexity of the static checking, and therefore of the required infrastructure, the code receiver can rely on easily checkable producer-provided evidence attesting to the safety properties of the code. This technique is called *proof-carrying code*.

Furthermore, my thesis is that the safety evidence required for proof-carrying code can be produced automatically, for a large class of safety properties, as part of the same compilation process that generates the agent executable. This variant of compilation is called *certified compilation*. As we shall see, in addition to constituting a front-end to proof-carrying code, certified compilation provides a simple and effective method for testing that a compiler produces only code that matches a safety specification, namely by verifying the evidence that the compiler produces with the code.

In this dissertation I describe a particular instance of the above ideas, in which the evidence to the safety of the code takes the form of a formal proof that the code satisfies a safety specification. Both the proof and the specification are expressed in a mathematical logic, which in this case is an extension of first-order predicate logic.

Before I outline in more detail the techniques of proof-carrying code and certifying compilation in Section 1.2, I discuss some of the more traditional solutions for protecting code receivers from misbehaving agents. I will argue why these traditional solutions are sometimes inadequate, and show that a natural way to address their drawbacks is to allow for the static checking of safety. Then, the rest of the ideas expressed in this dissertation are merely ways to overcome the inherent complexity of static analyses and the undecidability of many interesting code properties.

1.1 Traditional Solutions to Agent Security

The techniques currently used in systems that interact through mobile code can be classified into three main categories. First, there are techniques that judge the safety of the code by the identity of the agent creator. Then, there are techniques where the agent is executed in a "sandbox", so that it cannot access critical data and resources directly. Finally, there are the techniques that rely on interpreters or receiver-side compilers to enforce safe agent behavior. These three classes of techniques, with their advantages and disadvantages, are discussed next.

1.1.1 Safety through Personal Authority

The aim of this class of techniques is not to prevent unsafe code from being executed, but to create accountability and thus a deterrent to the distribution of harmful agents. For this purpose, the agent producers are required to sign digitally the code they produce. This allows the code receiver to verify not only the identity of the producer but also that the code integrity was maintained from the code producer to the receiver. The most commonly used technology for this purpose is that of public key cryptography, as used for example in Microsoft's Authenticode [Mic96].

While this approach has its practical merits, it also has a number of disadvantages, the most important of which being that it does not prevent trusted producers from creating and uploading erroneous agents. The fact that the error can be traced to a concrete producer is of little consolation when a large amount of damage has already occurred due to the mistake.

The personal authority approach to safety penalizes unfairly the small individual agent producers in favor of larger and better-known producers. A code receiver is unlikely to grant the benefit of doubt to an agent that is received from an unknown or a lesser-known producer. While this might not be a problem in small enclaves where the interacting entities know each other, it is definitely a drawback in environments where large anonymous masses of programmers interact through active agents, as is likely to be the case for many active interfaces exported to the Internet.

Because of the above reasons, personal authority alone is not a good way to ensure safety. Instead, we must look to approaches that consider the intrinsic properties of the agent code, independently of who produced it or how it was produced. However, personal authority might be an important component of a safety policy for execution of mobile code and, in such cases, personal authority can be used in combination with one or more of the techniques discussed next.

1.1.2 Safety through Hardware-Based Address Spaces

One of the simplest methods to ensure the safe execution of untrusted code is to isolate it in an hardware-enforced address space so that all accesses to non-private data and resources can be intercepted and monitored by the code receiver infrastructure. This is also the main technique that operating system kernels use to protect themselves from misbehaving user-level applications.

The most important component of such a scheme for protection is the design of a secure system-call interface, which consists of server-provided functions that the agent code may invoke for accessing system data and resources. A proper implementation of the system-call interface must check that the invoking agent has the necessary rights to perform the requested operation and that the provided arguments are valid. Extra care is required in a multiprocessing environment to ensure the atomicity of the argument checking procedure. Usually, this is accomplished by first copying the arguments into a memory area inaccessible to user applications.

The main advantage of the hardware-based solution is that it is relatively simple, thus easy to implement and to trust. The disadvantage is the high cost associated with switching between the protection domain of the agent and the code receiver, and back, along with the cost of copying the arguments and the results between address spaces. While this might not be a problem for those agents that have a high ratio of data processing to system calls, it is likely to hurt the performance of the majority of agents.

Another, sometimes overlooked, disadvantage of enforcing safety through memory protection is that it imposes constraints on the design of the client interface. First, in order to be able to enforce what amounts to abstract data types, the server must often introduce a level of indirection between its actual data structures and the agent. For example, a typical operating system enforces the abstract type of file descriptors through the use of file handles, which are small positive integers whose identity can be verified at run time; this verification could not be easily performed if the clients were referring to files by using actual pointers to kernel data structures. Second, the system must consider the possibility of having to terminate the agent prematurely, maybe after the agent has had the chance to acquire critical resources or alter the state of the system in a significant manner. To prevent this, the server must either disallow significant side-effects as part of the agent interface, or else employ a costly transaction mechanism that is able to undo such effects. It is interesting to remark that both the data abstraction and the side-effect problems are due in fact to the exclusive use of run-time checking for implementing safety.

Furthermore, it is often necessary for the agent code to have a more intimate coupling with the code receiver than that provided by a fixed Application Programmer's Interface (API) and separate address spaces. Consider for example an agent that requires some simple but time-critical processing to be done at every interrupt. The simplest way to achieve this is to allow the agent to install an interrupt handler to be run in privileged mode, in which case the hardware-based protection mechanisms are not usable.

Finally, another disadvantage of hardware-based techniques for safety is that they require special hardware and relatively complicated operating system support, features that might not be available in "lean" environments such as smart cards or embedded systems.

1.1.3 Safety through Programming-Language Semantics

A number of approaches for enforcing the safe execution of agents proceed by selecting an agent programming language together with a semantics for it so that all valid agent programs are guaranteed to conform to the safety policy. An important characteristic of the approaches based on programming language semantics is that the complexity of the safety policies that can be enforced is strongly dependent on the expressiveness of the programming language considered. To illustrate this point, consider the so-called *type safety* approaches, where a type system is used to distinguish among all syntactically correct programs those that share certain desired safety properties. In this case, the type checker can be thought of as the decision procedure for the subset of programs that are well-behaved with respect to a safety policy.

Type-safe languages have been used for safe operating-system extensibility at least since Burroughs B-5000 [Lev84] and are still employed in research operating systems such as SPIN [BSP$^+$95]. Examples of general-purpose type-safe languages that are currently used for writing untrusted agents and system extensions are Java [GJS96], the Java Virtual Machine bytecode language [LY97] and Modula-3 [Nel91]. In these cases, type safety usually means memory safety and data abstraction. This level of safety is significant; it rivals hardware memory protection but with finer granularity and lower costs and, by careful use of abstraction, it can surpass approaches based purely on run-time checking.

Type safety can mean more than just memory protection or data abstraction if we enrich the type system. Consider, for example, a type system that includes parametric polymorphism—such as the type system of Standard ML [MTH90]—and a function `foo` with the polymorphic type "$\texttt{foo} : \forall\alpha.\texttt{vector}(\alpha) \rightarrow \alpha$". It is possible to prove within the type system that any well-typed implementation of `foo` selects and returns one of the elements of the vector given as argument.[1] In general, there are very interesting code properties that can be proved about functions typed in a polymorphic type system [ACC93, Wad89].

Unfortunately, the level of type safety provided by general-purpose languages is not enough to prevent agents from subverting the code receiver. For example, none of the type systems mentioned above can enforce resource usage bounds or revocation of capabilities. There are fundamental reasons why this is so; for languages that are expressive enough, many interesting dynamic properties can be reduced to the halting problem for a Turing Machine and are thus undecidable. Hence, such properties cannot be enforced as "type safety" for any practical type system, unless we are willing to restrict the language to one that is strictly less expressive that a Turing machine. This path is taken by approaches using *domain-specific languages* (DSL), in effect restricting the syntactically valid programs to a small subset that is of interest to a particular application domain. One example of a DSL used in system extensions is the Berkeley Packet Filter [MJ93] language for which termination is easily decidable because no looping constructs are allowed.

In general, a code receiver using a language-based technique enforces the safe semantics in two stages. First, during the *static checking* stage, the receiver performs a detailed inspection of the agent's code to ensure that it is a valid program in the selected language. This stage includes syntax checking and also type checking for typed languages. The second stage takes place while the agent is executing. In this second stage, the receiver ensures that those operations that are potentially harmful and whose safety cannot be ascertained by the static checking pass, are preceded by run-time checks for safety. I shall refer to this stage as the *dynamic checking* stage.

The purpose of the static checking phase is to catch early many of the common programming errors and sources of harmful behavior. The amount and kind of static checking that can be performed ranges from simple control-flow checking to complicated type checking. For example, in the Berkeley Packet Filter (BPF) [MJ93] architecture, the untrusted code is scanned to verify that all instructions belong to a restricted bytecode language, that all branches are forward, and that their targets are within the code boundaries. In contrast, for approaches based on type-safe languages, a full-fledged type checking pass is made over the agent's code.

After the static checking stage, the receiver must execute the agent in such a way that potentially harmful operations are guarded by run-time checks for safety, such as array bounds checking or null-pointer checking. The more complicated the static checking, the fewer run-time checks are required. In the case when static checking consists of type checking in an expressive type system, code properties such as memory safety and data abstraction are guaranteed with few or no run-time checks.

The easiest way to implement the dynamic checking stage is through *interpretation*. Basically, this means that the untrusted agent code is interpreted by a safe and trusted

[1] In most such languages, the function `foo` might also diverge.

interpreter that performs all of the required dynamic checks mandated by the safety policy. For example, in order to enforce memory safety, an interpreter can verify, before each memory access, that the agent accesses only memory areas that are permitted by the safety policy. Two examples of safe interpreters are the Berkeley Packet Filter interpreter [MJ93] for operating system extensions and the Java Virtual Machine interpreter [LY97] for mobile code.

The major drawback of the interpreter-based approach is the reduced execution speed; it is not unusual to observe an order of magnitude slowdown due to interpretation. A natural solution to the interpretation overhead problem is to replace the interpreter with a trusted just-in-time compiler that, while compiling the agent code, inserts run-time checks similar to those that an interpreter would perform. Note that I use the generic term "compiler" for such a tool even in cases when the source and target language are the same. For example, a compiler can edit machine code agents by inserting bounds checks before memory operations. This approach, called Software Fault Isolation (SFI) [WLAG93], is used to enforce memory safety in the extensible operating system VINO [SESS96].

The SFI compiler mentioned above analyzes and modifies machine code agents so as to enforce memory safety. However, it is easier to analyze agents written in high-level or intermediate-level languages, meaning that more complex safety properties can be enforced with fewer safety checks at the cost of a more complicated compiler. For example, a compiler for Java Virtual Machine bytecode produces machine code augmented with array bounds checks. The higher-level language allows a type checker to enforce statically properties like data abstraction, and it restricts the run-time checking for memory safety only to those memory operations that access arrays.

A compiler-based approach to safety leads to agents that are significantly faster than the interpreted versions. However, this advantage comes at a high price. A compiler is significantly more complex than an interpreter and this means that a code receiver must rely on the correctness of a larger and more complicated body of code. This drawback is getting more pronounced as more optimizations are incorporated in the compiler, in an attempt to maximize the performance benefit of interaction by mobile code.

1.2 Outline of the Dissertation

The discussion of traditional techniques for ensuring the safety of untrusted agents suggests that an ideal enforcement method should be based as much as possible on static checking. This way we avoid both the overhead of run-time checking and the complications associated with forcefully terminating agents. With static checking, the execution of the agent code is not even started unless it is guaranteed to be safe.

As my thesis statement indicates, I propose to use static checking for verifying the safety of untrusted code. For this to be possible in general I propose also that the agent producer creates a formal proof of safety for the agent. This enables the code receiver to verify even complex safety properties by using a small and easy-to-trust infrastructure consisting of a proof checker.

The second part of my thesis is that, for a large class of safety properties, the formal proof can be produced automatically by a certifying compiler. The particular safety properties that I have in mind are those that could also be enforced by the safe interpreter or just-in-time compiler approaches discussed in the previous section. The major difference is that the certifying compilation process takes place at the code producer's site, and the code receiver does not have to incur the cost of compilation or interpretation and it does not have to trust the compiler or the interpreter.

For a gentle introduction to the concepts of proof-carrying code and certifying compilation, I discuss in Chapter 2—at a high level—the steps that are taken by a code producer that uses certifying compilation to interact with a code receiver through proof-carrying code. As part of this process, I identify the main software components involved, and I summarize the design and implementation requirements they must meet. The actual details of the design and implementations of these components are described in subsequent chapters.

Following the overview chapter, the main body of this document is divided into three parts, dealing respectively with the PCC infrastructure residing at the code-receiving end, the tools used by the code producer and the experimental evaluation of the entire system. In each part, the interested reader can find detailed discussions of the technical barriers that must be surmounted by a successful implementation, along with a presentation of my own design and implementation. In some cases, these barriers are practical implementation issues; in other cases, they are theoretical issues whose solution is important beyond their occurrence in a proof-carrying code system.

Finally, in Chapter 9, I summarize the contributions of this dissertation and suggest some future directions for research.

Chapter 2

Overview

This chapter is an informal high-level overview of the two techniques of proof-carrying code and certifying compilation. The presentation is structured as a step-by-step description of the interaction between a server (code receiver) and a client (code producer) that use proof-carrying code produced by certifying compilation. For each interaction step I describe the software components that are involved, focusing on their functional behavior and ignoring for the time being the actual implementation details. For illustration purposes, I use a simple running example and I show fragments of the proofs and code that are transferred between the code producer and the code receiver at each step.

Although the structure of this chapter and of the running example are intended to illustrate an integrated system using both proof-carrying code and certifying compilation, it is important to realize that these techniques, when taken individually, have different strengths and characteristics. Proof-carrying code is general and powerful in the sense that it can handle complex safety properties. In contrast, certifying compilation as presented in this dissertation handles only type safety, thus only a relatively small subset of the class of safety policies enforceable with proof-carrying code. Another differentiating characteristic is that only the system components implementing proof-carrying code—also referred to as the *PCC infrastructure*—must be trusted in order to ensure that only safe agents are executed, while those implementing the certifying compiler need not be trusted. These trust relationships imply that the infrastructure must be executed on the trusted code-receiver, while the tools may be executed even on untrusted systems, such as the code producers. The certifying compiler is one particular way of producing the proof attachments required for proof-carrying code. In general, the proofs can be produced using a variety of methods, all of which are referred collectively as the *PCC tools*.

Because of the lack of detail, this chapter is necessarily informal. It is the purpose of the rest of this dissertation to provide the missing details and to formalize the methodology completely. Nevertheless, with just the information contained in this chapter the reader can proceed to read Part III devoted to the experimental evaluation of the PCC infrastructure and of the PCC tools, namely the theorem prover and the certifying compiler.

Proof-carrying code has many applications, and each such application may entail some variations on the precise details of the approach. I will have more to say about some of these variations in Section 2.2. First, I describe in Section 2.1 a canonical implementation of PCC, which is general enough that any of the variations can be seen as optimizations or special cases. This chapter ends with a summary of the benefits and costs of proof-carrying code and certifying compilation (Section 2.3) and a short question-and-answer section addressing the limitations of the design presented here along with issues that arose most frequently during the various public presentations of this material.

2.1 The Basic Proof-Carrying Code Protocol

For a more general presentation of the proof-carrying code idea, it is useful to introduce in the system a third party, the *proof producer*, in addition to the code producer and the code receiver. In practice, it often turns out that the code producer and proof producer are the same system, though in general they may be separate entities.

Figure 2.1 shows graphically the steps involved in a typical PCC session. In this figure, we have the code producer and the proof producer (the untrusted entities) on the left-hand side and the code receiver (the trusted entity) on the right-hand side. The wavy boxes represent code and data that is manipulated by the rectangular boxes, which represent PCC software components. Finally, the white boxes are trusted entities while the grey ones are untrusted.

Before a code receiver can accept PCC agents, it must establish a *safety policy*, which defines the actions that agents are allowed to perform and also the circumstances when these actions are allowed. The concrete embodiment of the safety policy in a PCC system consists, as shown in Figure 2.1, of several components that are described in Section 2.1.1.

Let us now assume that a code producer wishes to use certifying compilation to interact with the PCC-enabled code receiver. In this case, a PCC session starts with the code producer compiling the agent source code to target code annotated with loop invariants. For this purpose the code producer uses a certifying compiler, which is discussed briefly in Section 2.1.2 and in detail in Chapter 6. Then the code producer sends the annotated agent code to the code receiver, requesting its execution. The receiver first inspects the code using the VCGen component of the safety policy, and returns a verification-condition predicate whose validity is sufficient to guarantee the safety of executing the agent. This step is described in Section 2.1.3. The receiver does not attempt itself the potentially difficult task of verifying the validity of the verification condition. Instead, the verification condition is sent to the proof producer who returns a proof of it, as discussed in Section 2.1.4. If the proof passes the proof-checking process (described in Section 2.1.5), the receiver can safely install and run the agent code. In the following sections, each of the above steps are described in more detail.

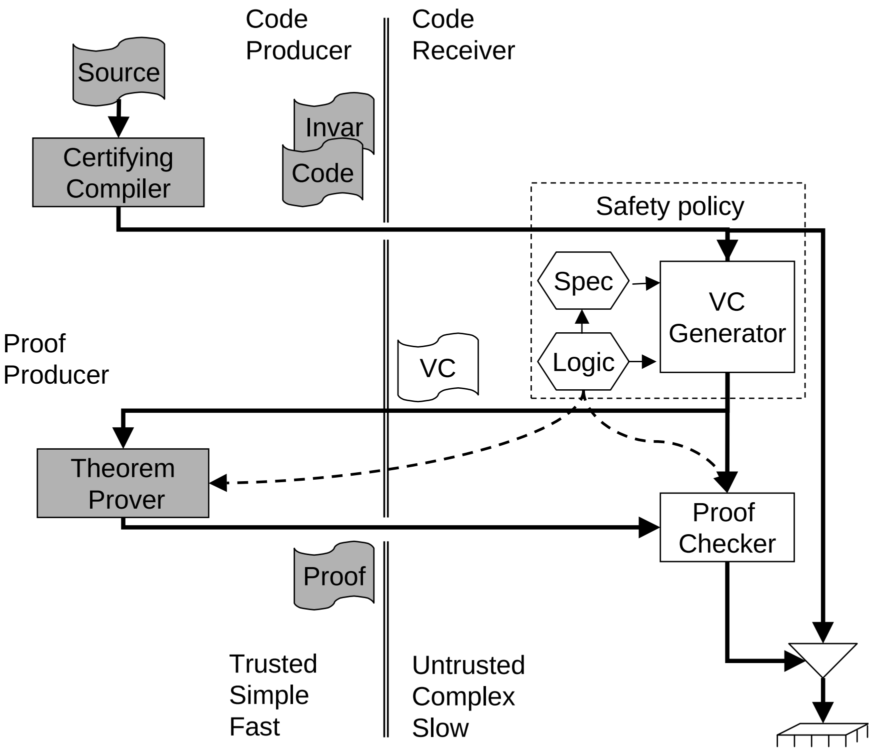


Figure 2.1: The basic proof-carrying code protocol. The wavy boxes represent data and code and the rectangular boxes represent system components. The white elements elements are trusted, while the grey elements are not trusted.

2.1.1 Preliminary Step 1: Defining the Safety Policy

The central component of any PCC implementation is the *safety policy*, which represents the set of rules that define unambiguously whether a given agent program is safe to execute. The safety policy is defined in advance by the administrator of the code-receiver system and is a trusted component of the infrastructure. The variant of proof-carrying code described in this dissertation is targeted towards safety polices that focus on the actions that the agent code is allowed to execute and in what situations each action can be executed. Informally, we can view the safety policy as a set of action preconditions. A more precise description of the covered set of policies can be found in the introduction of Chapter 3.

A safety policy is defined in the context of a given agent language. Proof-carrying code does not restrict the languages in which agents can be programmed. PCC can be adapted both to high-level languages to improve portability of agent programs, and to low-level languages, even hand-optimized machine code, to maximize the performance of the agent while minimizing the size of the trusted infrastructure. Also, a given code receiver might accept agents written in multiple programming languages, in which case the safety policy must be adapted to each language.

The safety policy, viewed as a set of action preconditions for a given agent language, is not directly usable by the code receiver. Instead, the safety policy is embodied in the PCC infrastructure as three distinct components, as follows:

- A mathematical *logic* that is able to describe the preconditions under which a given agent action is allowed. The logic is the language used in a PCC system to describe and verify the action preconditions. The same logic is also the language used to encode the code annotations, the verification conditions and the proofs. In the variant of PCC described here, I use first-order predicate logic extended with predicate symbols as required by the safety properties to be proved, although in principle any program logic can be used.

  The concrete form of the logic is a set of syntactic predicate constructors along with a set of axioms and inference rules that define provability of predicates. As shown in Figure 2.1, this representation of the logic is made available to proof producers.

- A safety policy must specify all the functions that an agent is obligated to provide and all the receiver-provided functions that an agent is allowed to invoke. This is done by means of *specifications* for all such functions. Each function specification is given as a pair of a precondition and a postcondition, expressed as predicates in the selected logic.

  A function precondition describes the state of the variables and actual arguments at the moment when the function is invoked. The precondition must be established prior to invoking the function. This enables the callee to assume that it holds without verifying it first.

  A function postcondition describes relationships between variables, actual arguments and the result values. The postcondition predicate must be established prior to returning from a function, and thus acts as a precondition for the function return action. This enables the caller to assume that it holds upon return.

- Finally, the safety policy contains a method for inspecting the agent code and for discovering the actions that an agent might perform and under which circumstances. This is accomplished by the *verification-condition generator* (VCGen), which scans the agent code and collects the set of all the actions that might be performed during execution, along with a partial description of the program state when such actions would be attempted. This information is expressed as a predicate in the logic (the *verification condition*). VCGen is designed such that the verification condition for a given agent is provable within the selected logic only when the agent code is considered safe to execute with respect to the current safety policy.

The operation of VCGen is described in Section 2.1.3 as part of the Step 3 of the PCC protocol. The other aspects and components of the safety policy are described next in the context of a simple example. For a detailed and formal discussion of the safety policy and of the relationships between its various components, see Chapter 3.

To illustrate the concepts discussed in this overview chapter, consider the case of a code receiver that wishes to enforce type safety and memory safety for agents written in a generic assembly language. For this purpose, the administrator or the code-receiver system establishes a safety policy consisting of a logic, a set of specifications and a verification-condition generator. The agent language was chosen to be a generic assembly language to avoid the need of a receiver-side compiler or interpreter.

The actions handled by the safety policy at hand are the function calls and returns along with memory reads and writes. In fact, a very large number of safety policies can be described with reference to this small class of agent actions. The safety of a function call or return is defined in this case as a set of type restrictions on the value of variables. For this example, the safety policy considers only simple types, such as, integers, booleans and array types. An array type encodes both the array-element type and also the array length. Furthermore, the code receiver wants to retain full control over the representation of booleans and thus, the safety policy discloses only that the bitwise "and" and "or" operations on boolean values produce valid boolean values.[1] The memory safety aspect of the safety policy requires that only addresses that fall within the bounds of arrays can be dereferenced and only values of the appropriate element type can be written to an array.

In order to express the precondition for function calls and returns and for memory operations, the administrator of the code receiver defines the logic whose syntax is shown at the top of Figure 2.2. The logic is an extension of first-order predicate logic with a typing predicate that is written in infix notation as "$e : \tau$", where e is an expression and τ is a type. Also, in order to express memory safety in a generic manner and without reference to arrays, the logic contains the predicate "$\texttt{saferd}(e)$" denoting that it is safe to read from the memory address denoted by the expression e.

Other extensions to first-order logic are needed to represent the effect of memory operations. The memory state is modeled in our logic explicitly using expressions. If m is an expression that denotes the current memory state then the expression $\texttt{sel}(m, a)$ denotes the 8-bit contents of the memory location whose address is denoted by a.[2] To express the new memory state after a store of the value v at address a in memory state m, we write $\texttt{upd}(m, a, v)$.

[1] This is a non-frivolous requirement on the part of the code receiver because, for example, a code receiver that is implemented using Standard ML of New Jersey reserves the least-significant bit to distinguish pointer values. On such a system boolean values are represented as the integers values 1 and 3 for false and true respectively.

[2] To simplify the presentation, I assume in this chapter that all memory operations operate at byte level. This assumption can be easily relaxed to deal with other memory-word sizes.

$$
\begin{array}{llll}
\text{Expressions:} & e & ::= & x \mid e_1 + e_2 \mid e_1 \,\&\, e_2 \mid e_1 \mid e_2 \mid \texttt{sel}(m, e) \\
\text{Memory:} & m & ::= & x \mid \texttt{upd}(m, e_1, e_2) \\
\text{Types:} & \tau & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{array}(\tau, e) \\
\text{Predicates:} & P & ::= & P_1 \wedge P_2 \mid P_1 \supset P_2 \mid \forall x.P_x \\
 & & & \mid e_1 \geq 0 \mid e : \tau \mid \texttt{saferd}(e)
\end{array}
$$

Rules:

$$
\frac{e_1 : \texttt{bool} \qquad e_2 : \texttt{bool}}{e_1 \,\&\, e_2 : \texttt{bool}} \quad \frac{e_1 : \texttt{bool} \qquad e_2 : \texttt{bool}}{e_1 \mid e_2 : \texttt{bool}} \quad \frac{}{\texttt{sizeof}(\texttt{bool}) = 1}
$$

$$
\frac{a : \texttt{array}(\tau, len) \qquad i \geq 0 \qquad i < len * \texttt{sizeof}(\tau)}{\texttt{saferd}(a + i)}
$$

$$
\frac{a : \texttt{array}(\tau, len) \qquad \texttt{sizeof}(\tau) = 1 \qquad i \geq 0 \qquad i < len}{\texttt{sel}(m, a + i) : \tau}
$$

Figure 2.2: Fragment of a logic used for type safety. Note how memory safety is enforced through array bounds checking and how booleans are specified as values of an abstract type.

In addition to the description of the syntax, the logic contains a set of inference rules that can be used by the proof producer to prove the verification condition. A fragment of the set of inference rules are shown at the bottom of Figure 2.2. Because the agents are written in a low-level language they must manipulate concrete representations of the source-level abstract types. It is by means of the logic that the code receiver discloses the necessary representation information to the producers. For example, the last two rules of Figure 2.2 say that the concrete representation of an array is the memory address of a sequence of consecutive memory locations holding values of an appropriate type. In addition, these rules establish the meaning of memory safety for our example: the only memory locations that can be safely read are those within the boundaries of an array.

While the array rules reveal the concrete representation of arrays, the first three inference rules manage to preserve an abstract view of booleans. From these rules, the agent-designer knows only that a boolean value occupies one byte and that the bitwise "and" and "or" operations on boolean values produce valid boolean values. As a consequence, a type-safe agent cannot forge boolean values; it can only transform them by using the two bit operations mentioned in the rules. If the agent code would attempt to create a boolean value using any other method, it would not be able to prove that the result is a boolean. Note that this is so even when the agent producer has prior knowledge of the actual representation of booleans. Following the same recipe, other, more complex, abstract data types can be specified in the logic so that the code receiver can enforce their abstraction boundaries. The interested reader can find in Section 6.2 a more detailed discussion of enforcing data abstraction using proof-carrying code.

With the logic component of the safety policy in place, the code receiver proceeds with identifying the functions that will act as the interface with the agent. To simplify the example, I assume that the receiver does not export any functions for the agent and that the agent's only entry point is called `main`, which is a function that given a boolean value and an array of booleans along with its length will return a boolean value. This specification is formulated in logic as shown in Figure 2.3. Note that the specification also reveals the receiver's calling convention: registers $\mathbf{r}_A$, $\mathbf{r}_B$ and $\mathbf{r}_L$ are used for actual arguments, and register $\mathbf{r}_R$ is used for the return value. Note that even though `main` is implemented by the agent, its specification is given by the code receiver, as part of the safety policy. I ignore in this dissertation the process by which a code producer and a code receiver agree on which functions must be exported by the agent and which functions are provided by the code receiver.

$$\begin{array}{lcl} Pre_{\texttt{main}} & = & \mathbf{r}_B : \texttt{bool} \ \wedge \ \mathbf{r}_A : \texttt{array}(\texttt{bool}, \mathbf{r}_L) \\ Post_{\texttt{main}} & = & \mathbf{r}_R : \texttt{bool} \end{array}$$

Figure 2.3: Sample specification for type safety

Each function specification entails certain obligations for both the code receiver and the agent, depending on how the function is defined and used, as follows:

- For functions that are exported by the agent, the receiver infrastructure must establish the precondition before invoking the function. This means, in our example, that the receiver must invoke `main` with $\mathbf{r}_B$ containing a valid boolean value and $\mathbf{r}_A$ containing the address of a properly allocated array filled with valid boolean values; the length of the array must also be passed in $\mathbf{r}_L$. Upon return, the agent must establish the post-condition, which in this case requires returning a well-formed boolean. The verification condition for the agent will contain a part verifying this.

- For functions that are exported by the receiver and used by the agent, the agent must ensure that the function precondition holds prior to invocation. This obligation will be stated in the verification condition. In return, the code receiver has the obligation to establish the postcondition prior to returning control to the agent.

- For functions that are defined by the agent for its internal use, the receiver has no obligations. The agent can define the specification for this function as it sees fit, but it must use the functions in a way that is consistent with their specification. The specification for these internal functions is part of the annotations that accompany the agent code.

Note that in the above list of obligations the code receiver can use proof-carrying code to ensure that the agent satisfies its obligations. On the other hand, the code receiver is a trusted component of the system, which means that the designer and implementor of the receiver must ensure that its interface obligations are met.

The last component of the safety policy is the verification-condition generator, which is discussed in Section 2.1.3 in the context of a concrete agent and the example safety policy introduced above. Note that the safety policy is established only once for a given service, before any agent is processed. After that, the steps 2 to 5 described below are taken by the code receiver and the code producer for each individual agent.

2.1.2 Step 2: Generating the Annotated Agent Code

The first action the code producer takes to initiate the interaction with the code receiver through PCC is to prepare the agent code as requested by the safety policy or, more specifically, by the VCGen component of the safety policy. VCGen requires that the agent code is syntactically well formed in the selected language. Also, VCGen requires that all functions defined and used internally by the agent code be annotated with a precondition and a postcondition, and also that each loop have an associated loop invariant. The loop invariants and the specifications for the internal functions are referred to as *annotations*.

From the point of view of VCGen it suffices that the annotations be well-formed predicates in the selected logic. This alone will ensure that VCGen does not reject the code, but it is not sufficient to guarantee that the code will ultimately make it through all of the steps of the PCC protocol. For this to happen, the annotations must also be correct and sufficiently strong, as described formally in Chapter 3. These requirements will become clearer once we discuss the rest of the steps in the PCC protocol. Intuitively, a loop invariant is a correct annotation if it is indeed a valid predicate every time the execution reaches the beginning of the loop. Similarly, a function specification is correct if the precondition holds every time the function is invoked, and if the postcondition holds every time the function returns. Note that the weaker the annotation the easier it is to ensure its correctness. For example, the weakest loop invariant "`true`" is evidently a correct invariant. The notion of sufficiently-strong annotations is more difficult to explain informally. A function precondition is sufficiently strong if, by assuming that it holds at the beginning of the function, one can prove that the body of the function is safe to execute. The stronger the annotation the easier it is to satisfy the sufficiency requirement. We see, thus, that annotations must be not too strong and not too weak. This is what makes the task of annotating the code a delicate one. In fact, this is just another aspect of the fact that designing a program is difficult because it must be general enough to be useful yet specific enough to be correctly and efficiently implemented.

Fortunately, for a large class of safety properties, it is possible to create the annotations automatically by using a modified compilation technique, which I call *certifying compilation.* The idea borrows from the approaches to safety based on programming-language semantics, in that a high-level language is selected for the agents and a semantics is defined such that no violations of the safety policy can occur. This semantics can be enforced through a combination of static checks (performed by the compiler) and run-time checks (inserted by the compiler in the agent code). So far, this is not different from what a traditional compiler for a safe high-level language performs. The difference is that a certifying compiler not only produces safe target code, but also emits typing information and supporting annotations for the optimizations that were performed. This information is easy to produce by the compiler and it enables an external system (the PCC code receiver in conjunction with the proof producer in this case) to verify that the result of compilation is indeed type safe. This is explained in more detail next and is formalized completely in Chapter 6.

As a proof of the certifying compilation concept I have implemented a certifying compiler, called *Touchstone*, for a type-safe subset of the C programming language. The implementation details of Touchstone are described in Chapter 6. In the rest of this section I show how Touchstone operates in the context of a simple agent, whose source is shown in Figure 2.4. Note that in the dialect of C compiled by Touchstone, arrays have both a base address and a length, which is accessed using the built-in operation "`length`". This is in contrast with the common programming practice where the length of the array is manipulated directly by the programmer.

```
bool main(bool A[], bool B) {
   int I;
   bool R = B;
   for(I = 0; I < length(A); I++)
      R = R && A[i];
   return R;
}
```

Figure 2.4: Sample source code for a type-safe agent

The function `main` computes the conjunction of all the boolean values in the array `A` and the boolean value `B` received on input. Touchstone parses the program and then translates it to an intermediate form that is then optimized. To simplify the presentation, I will use the same generic assembly language as the intermediate form and the target language.

Touchstone behaves mostly as a traditional optimizing compiler for a type-safe language. The distinguishing aspect of compilation is the generation of annotations. In this section I describe informally the generation of loop invariants for the agent of Figure 2.4. In the context of the type-safety policy, there is only one kind of loop invariants that must be produced, namely, typing loop invariants. A typing loop invariant is the conjunction of the typing predicates for all variables that are live at the beginning of a loop and are modified within the body of the loop. In our example, this includes the registers that are used in the target program to hold the values of `I` and `R`, which we call respectively $\mathbf{r}_I$ and $\mathbf{r}_R$. Thus the typing loop invariant for our example is as follows:

$$\mathbf{r}_I : \mathtt{int} \quad \wedge \quad \mathbf{r}_R : \mathtt{bool}$$

This typing loop invariant would in fact be sufficient if the compiler did not perform certain optimizations. But, in an attempt to maximize the performance of the target code, Touchstone implements several optimizations, of which the most significant are the array bounds-checking optimization, global common-subexpression elimination and loop-invariant optimizations. An interesting lesson that emerged from the Touchstone project is that only few optimizations actually complicate the certification aspect of compilation. One such interesting optimization is the array bounds-checking optimization, which is discussed next.

Array bounds-checking optimization is required in Touchstone, as opposed to a traditional C compiler, because the type-safe semantics of the source language mandates bounds checks, either at compile time or at run time. The purpose of the optimization is to perform as many of the checks statically in order to reduce their run-time overhead. To get an intuition of how Touchstone succeeds to certify even code whose bounds checks were optimized, consider again the agent of Figure 2.4. First, Touchstone translates this agent in the intermediate form shown in Figure 2.5. The register $\mathbf{r}_T$ is used to store temporarily the value of the array cell. Just as a traditional compiler, Touchstone starts by inserting code to perform bounds checking for all array operations.

$$\begin{array}{ll}
 & \mathbf{r}_I = 0 \\
 & \mathbf{r}_R = \mathbf{r}_B \\
\mathrm{L}_0: & \mathtt{INV}\ \mathbf{r}_I \geq 0 \wedge \mathbf{r}_I : \mathtt{int}\ \wedge \mathbf{r}_R : \mathtt{bool} \\
 & \mathtt{if}\ \mathbf{r}_I \geq \mathbf{r}_L\ \mathtt{goto}\ \mathrm{L}_{\mathtt{end}} \\
 & \boxed{\mathtt{if}\ \mathbf{r}_I < 0\ \mathtt{goto}\ \mathrm{L}_{\mathtt{err}}} \\
 & \boxed{\mathtt{if}\ \mathbf{r}_I \geq \mathbf{r}_L\ \mathtt{goto}\ \mathrm{L}_{\mathtt{err}}} \\
 & \mathbf{r}_T = *(\mathbf{r}_A + \mathbf{r}_I) \\
 & \mathbf{r}_R = \mathbf{r}_R\ \&\ \mathbf{r}_T \\
 & \mathtt{goto}\ \mathrm{L}_0 \\
\mathrm{L}_{\mathtt{end}}: & \mathtt{return}\ \mathbf{r}_R \\
\mathrm{L}_{\mathtt{err}}: & \boxed{\mathtt{raise\ Subscript}}
\end{array}$$

Figure 2.5: The intermediate form of the agent of Figure 2.4. The boxed fragments are introduced by the compiler to perform array bounds checking.

Note that the loop invariant shown in Figure 2.5 extends the typing loop invariant discussed before with a conjunct saying that the variable I is always non-negative. This is indeed a correct invariant because I is initialized to 0 before the loop and is incremented by 1 in every loop iteration. We will see in a moment the significance of this invariant for array bounds-checking elimination; consider now how Touchstone discovers it. Of all of the possible arithmetic invariants, Touchstone attempts to discover when an integer variable is monotonically increasing or decreasing. In our example, I is monotonically increasing because $I + 1 \geq I$. To prove simple arithmetic facts like this, Touchstone uses a decision procedure for linear arithmetic. The rest is simple; because I is a monotonically increasing variable whose initial value is 0 Touchstone emits the invariant $I \geq 0$.[3]

Returning to the intermediate code of Figure 2.5, the array bounds-checking optimization is implemented as a more general conditional optimization that tries to eliminate conditionals or to collapse several conditionals into fewer ones. In the case of our example, Touchstone successfully eliminates the boxed conditionals because their guarding boolean expressions are statically proved false; the expression "$\mathbf{r}_I < 0$" is falsified by the loop invariant and the expression "$\mathbf{r}_I \geq \mathbf{r}_L$" is falsified by the loop termination condition. Consequently, the exception-raising operation at the end of the code becomes unreachable and is eliminated as well. The resulting optimized code is shown in Figure 2.6.

$$
\begin{array}{ll}
 & \mathbf{r}_I = 0 \\
 & \mathbf{r}_R = \mathbf{r}_B \\
\mathrm{L}_0: & \texttt{INV}\ \mathbf{r}_I \geq 0 \wedge \mathbf{r}_I : \texttt{int}\ \wedge \mathbf{r}_R : \texttt{bool} \\
 & \texttt{if}\ \mathbf{r}_I \geq \mathbf{r}_L\ \texttt{goto}\ \mathrm{L}_{\texttt{end}} \\
 & \mathbf{r}_T = *(\mathbf{r}_A + \mathbf{r}_I) \\
 & \mathbf{r}_R = \mathbf{r}_R\ \texttt{\&}\ \mathbf{r}_T \\
 & \texttt{goto}\ \mathrm{L}_0 \\
\mathrm{L}_{\texttt{end}}: & \texttt{return}\ \mathbf{r}_R
\end{array}
$$

Figure 2.6: The agent code after bounds-checking optimization.

Even though Touchstone differs from other compilers in that it outputs invariants, I claim that it does not do more work than other optimizing compilers for the purpose of discovering the invariants. Take, for example, the typing invariants. The typing information is known in the compiler front end; all Touchstone has to do is to preserve it from the front end of the compiler all that way through the code generation phase. The same is true for the invariant annotations. Any optimizing compiler that removes array bounds checks from a loop body employs some static analysis to discover code properties that amount to loop invariants. On top of that, Touchstone records these invariants and emits them together with the code.

[3]The Touchstone compiler described in this dissertation does not handle correctly arithmetic overflow. While it is quite easy to arrange the logic inference rules for correct handling of arithmetic overflow (and thus to plug the safety hole in the PCC infrastructure) it is significantly more difficult to modify the certifying compiler and the theorem prover to use the correct inference rules.

Before we continue the journey of our example agent through the proof-carrying code protocol, I want to remark a couple of interesting aspects of certifying compilation. First, it is worth taking some time at this point to think of other implications of the certifying compiler design, besides its use as a PCC front end. Note that the code receiver does not have to trust the certifying compiler because it has other means to verify that the emitted code has the desired properties (type safety in this case). Note also that a correct compiler must necessarily emit code that has the desired properties. This immediately suggests that it is possible to test partially the correctness of the certifying compiler by attaching it to the PCC infrastructure and verifying the proof for each compilation result. This procedure does not verify the complete correctness of the compiler, but only that it emits code with certain properties, such as type safety. In practice, many compiler bugs will, sooner or later, result in target code that "crashes". It is exactly this class of unsafe target programs that the PCC infrastructure catches. And not that it does it without requiring the compiler tester to run the target programs with several input data sets hoping to exhibit the bug. During the development of Touchstone, numerous bugs have been reported this way. But since this is not guaranteed to reveal all bugs, it was not surprising when traditional testing discovered one more bug in the compiler. This suggests that the certifying compiler technique is useful as a compiler development strategy, independent of whether mobile code interaction by PCC is of interest or not.

The second remark is that the certifying compiler is only one of the many ways to produce proof-carrying code. From the point of view of the PCC infrastructure it does not matter how the code producer generates the annotations. The only thing that matters is that the code be annotated properly. In fact, one could write the code of Figure 2.6 by hand or by using interactive tools. This is an important aspect because it opens the proof-carrying code technology to safety properties for which there are no fully automatic ways of creating annotations. I shall have more to say about this important property of proof-carrying code in Section 2.3.

Having discussed techniques for generating the annotations, we now return to the sequence of steps in the basic PCC protocol. Once the code producer generates the annotated code, by using a certifying compiler or by any other means, it sends the code to the code receiver, requesting it to be executed. This action initiates Step 3 in the proof-carrying code protocol.

2.1.3 Step 3: Generating the Verification Condition

Upon receiving the annotated code, the code receiver performs a fast, detailed and automatic inspection of the code. This is accomplished using a program, called the verification-condition generator (VCGen), which is one component of the receiver-defined safety policy. The purpose of VCGen is twofold: to perform simple syntactic checks on the agent code, and to emit a verification-condition predicate for all agent instructions that might violate the safety policy. For a complete discussion of the verification-condition generator see Section 4.2.

The syntactic checks that VCGen performs depend on the particular safety policy. For example, a syntactic property commonly checked directly by VCGen is that all branch targets are within the code boundary and that only functions that are allowed by the code receiver are invoked. In addition, VCGen can enforce restrictions on the set of instructions or operations that might occur in the agent code.

In some cases, the safety policy designer elects to restrict the syntax of the language such that desired safety properties can be enforced syntactically. For example, VCGen could enforce termination if the safety policy disallows function calls and backward branches. This approach, while simple, could not be extended without crippling the expressiveness of the language. For this reason most safety policies do allow potentially dangerous actions but impose restrictions on their use. For example, the safety policy might admit memory read operations provided the target address lies within the boundaries of a properly allocated array. Compare this to a syntactically checkable memory-safety policy that disallows memory read operations entirely.

The specific conditions under which an action is considered safe, expressed as a predicate in the selected logic, is called the *action precondition*. In practice, most action preconditions denote properties that are difficult or even impossible to verify directly. This is why VCGen does not attempt to verify the action preconditions itself. Instead, it collects them all and combines them with control flow information and with the specification part of the safety policy to create the *verification condition* for the entire agent code.

VCGen is a constituent part of the safety policy and as such it must be designed in conjunction with the other components: the logic and the specification. The ultimate criterion that should guide this design process is to preserve the validity of the soundness theorem stated informally below:

Theorem 2.1 (Soundness of the Safety Policy—Informal Statement) *If the verification condition corresponding to an agent and a specification is provable within the logic, then the agent's execution does not violate the safety policy, or equivalently, all action preconditions are met during execution.*

There is one implementation detail of VCGen that ought to be discussed here because it imposes serious restrictions on other components of a PCC system. In order to detect the potentially hazardous instructions in the agent code and to construct meaningful verification conditions for them, VCGen must understand the semantics of the agent code in considerable detail. To simplify this task of VCGen, and consequently to simplify VCGen itself, we adopt the general design rule that whenever some information about the behavior of the agent is difficult to discover, the code producer must provide it in the form of code annotations.

However, to prevent erroneous annotations to mislead the verification process, VCGen must take special care when using them.

One important class of annotations are the *loop invariants*. An invariant is a predicate that the code producer claims to hold every time the execution reaches a given point in the code. One constraint that VCGen imposes on the code is that each loop has at least one loop invariant associated with it. Because of this constraint, VCGen can be simplified considerably by not having to perform expensive program analysis for programs with loops. The first obligation of VCGen when dealing with invariant annotations is to verify that they are indeed invariant, and thus correct. Again, VCGen does not verify the invariance itself, but emits as part of the verification condition a predicate stating that the invariant must hold at the start of the loop and another predicate that states the preservation of the invariant property for one arbitrary iteration. A second required class of annotations are the preconditions and postconditions for the functions defined and used internally by the agent. In fact, a specification is required for every function, but the external ones are provided by the safety policy.

To illustrate the operation of VCGen, consider again the agent code shown in Figure 2.6 with the specification of Figure 2.3. The verification condition for this code and specification is the closed predicate shown in Figure 2.7. Note in line 2 the occurrence of the function precondition, and in line 3 the statement that the loop invariant holds initially when $\mathbf{r}_I$ has value 0 and $\mathbf{r}_R$ is set to the initial value of $\mathbf{r}_B$. Then, in lines 4–6 we have the statement that the loop invariant is preserved through one iteration of the loop. Note in line 5 the invariant before the loop and, in line 6, the looping condition followed by the invariant at the end of the loop with the new values for $\mathbf{r}_I$ and $\mathbf{r}_R$. So far, the verification condition was concerned with verifying the invariance of the claimed invariant. In line 7 we see the first (and unique, in this example) instance of an action precondition for a potentially dangerous operation: the memory read from address $\mathbf{r}_A + \mathbf{r}_I$. Note that VCGen in isolation does not enforce a particular memory safety policy; it just marks the memory read and the target address by means of a "`saferd`" predicate. It is up to the logic to define the meaning of this predicate and hence of the memory safety. This way, the same VCGen can be used for a variety of memory safety policies just by changing the logic. Finally, the last part of the verification condition enforces the postcondition (line 8) in the event the loop actually terminates.

To initiate the next step of the PCC protocol, the code receiver sends the verification condition predicate to the proof producer and then waits for a proof to be returned. If such a proof exists within the logic, the soundness theorem for VCGen ensures that the agent is safe to run.

2.1.4 Step 4: Proving the Verification Condition

Upon receiving the verification condition, the proof producer attempts to prove it according to the logic that the administrator of the code receiver specifies as part of the safety policy.

$$
\begin{array}{rl}
1 & \forall \mathbf{r}_A . \forall \mathbf{r}_B . \forall \mathbf{r}_L . \forall m . \\
2 & \quad \mathbf{r}_B : \texttt{bool} \ \wedge \ \mathbf{r}_A : \texttt{array}(\texttt{bool}, \mathbf{r}_L) \supset \\
3 & \qquad (0 \geq 0 \ \wedge \ 0 : \texttt{int} \ \wedge \ \mathbf{r}_B : \texttt{bool}) \ \wedge \\
4 & \qquad \forall \mathbf{r}_I . \forall \mathbf{r}_R . \\
5 & \qquad\quad \mathbf{r}_I \geq 0 \ \wedge \ \mathbf{r}_I : \texttt{int} \ \wedge \ \mathbf{r}_R : \texttt{bool} \supset \\
6 & \qquad\qquad (\mathbf{r}_I < \mathbf{r}_L \supset \mathbf{r}_I + 1 : \texttt{int} \ \wedge \ \mathbf{r}_R \mathbin{\&} \texttt{sel}(m, \mathbf{r}_A + \mathbf{r}_I) : \texttt{bool} \ \ \wedge \\
7 & \qquad\qquad\qquad\qquad \texttt{saferd}(\mathbf{r}_A + \mathbf{r}_I)) \ \ \wedge \\
8 & \qquad\qquad (\mathbf{r}_I \geq \mathbf{r}_L \supset \mathbf{r}_R : \texttt{bool})
\end{array}
$$

Figure 2.7: The verification condition for the agent of Figure 2.6. The scope of universal quantification and implication operators extends to the end of the predicate or to a closing parenthesis.

Because the code receiver does not have to trust the proof producer, any system can be the proof producer; in particular the code producer can also act as a proof producer. For the most part, the proof generator is a general-purpose theorem prover for first-order predicate logic extended with special-purpose axioms, such as those presented in Figure 2.2.

$$
\dfrac{\dfrac{\overline{\triangleright\ \mathbf{r}_I \geq 0 \wedge \mathbf{r}_R : \texttt{bool}}^{\,v}}{\triangleright\ \mathbf{r}_R : \texttt{bool}} \qquad \mathcal{D}}{\triangleright\ \mathbf{r}_R \mathbin{\&} \texttt{sel}(m, \mathbf{r}_A + \mathbf{r}_I) : \texttt{bool}}
$$

$$
\mathcal{D} = \dfrac{\overline{\triangleright\ \mathbf{r}_A : \texttt{array}(\texttt{bool}, \mathbf{r}_L)}^{\,u} \quad \overline{\triangleright\ \texttt{sizeof}(\texttt{bool}) = 1} \quad \dfrac{\overline{\triangleright\ \mathbf{r}_I \geq 0 \wedge \mathbf{r}_R : \texttt{bool}}^{\,v}}{\triangleright\ \mathbf{r}_I \geq 0} \quad \overline{\triangleright\ \mathbf{r}_I < \mathbf{r}_L}^{\,w}}{\triangleright\ \texttt{sel}(m, \mathbf{r}_A + \mathbf{r}_I) : \texttt{bool}}
$$

Figure 2.8: Fragment of the proof of the verification condition of Figure 2.7. For typographic reasons, the subproof $\mathcal{D}$ is shown separately. In this proof, the following assumptions are used: "$\mathbf{r}_A : \texttt{array}(\texttt{bool}, \mathbf{r}_L)$" (from line 2, referred to as u), "$\mathbf{r}_I \geq 0 \wedge \mathbf{r}_R : \texttt{bool}$" (from line 5, referred to as v), and "$\mathbf{r}_I < \mathbf{r}_L$" (from line 6, referred to as w).

For first-order logic, many theorem-proving systems have been implemented, most of which are able to prove typical verification conditions, sometimes with the help of additional tactics. To be usable as a PCC proof producer, a theorem prover must not only be able to prove verification conditions but must be also capable of generating detailed proofs of them. Furthermore these proofs must be expressed using the axioms and inference rules specified as part of the safety policy. The major difficulty here is to make the theorem prover output the proof in any convenient format, because once we have all the proof details, it is generally easy to transform them into the format expected by the proof checker on the code receiver.

The theorem prover described in Chapter 7 of this dissertation follows the modular design first suggested by Nelson and Oppen [NO79]. The theorem prover uses several decision procedures, the most notable ones being Simplex, for deciding linear inequalities, and the congruence closure, for deciding equalities. In addition, the theorem prover can be easily customized to a particular safety policy by extending it with special-purpose axioms such as those of Figure 2.2 on page 14.

It is not possible, in general, to guarantee that a given theorem prover can prove the verification condition for an arbitrary safe agent. However, it is possible to achieve automatic proving of verification conditions when using a certifying compiler. Consider for example the output of the Touchstone certifying compiler in the absence of any optimizations. The target code is in this case a sequence of code patterns, each corresponding directly to a source level construct. Thus, the type-checking algorithm that is used at source level to ensure that the agent is type safe and memory safe can be easily modified to be able to prove the same properties for the result of the compilation. This modified type-checking algorithm would in fact be the required theorem prover. Now consider the more realistic case when the output is optimized. Even in this case, there is only a finite number of code transformation patterns that a given compiler can perform. All we need is a theorem prover that is able to discover which transformation pattern was used at each step. In fact, the theorem prover does not need to be as powerful as the compiler because a large amount of the information that the compiler discovers through complex static analyses can be communicated to the prover as part of the loop invariants.

Returning to our example verification condition of Figure 2.7, consider how the proof would proceed, given the axioms in our logic. A fragment of the proof is shown in Figure 2.8, namely the proof of the predicate "$\mathbf{r}_R \ \& \ \mathtt{sel}(m, \mathbf{r}_A + \mathbf{r}_i) : \mathtt{bool}$" that occurs on line 6 of the verification condition. When this proof is attempted, the following assumptions are available: "$\mathbf{r}_A : \mathtt{array}(\mathtt{bool}, \mathbf{r}_L)$" (from line 2, referred to as u), "$\mathbf{r}_I \geq 0 \wedge \mathbf{r}_R : \mathtt{bool}$" (from line 5, referred to as v), and "$\mathbf{r}_I < \mathbf{r}_L$" (from line 6, referred to as w). This proof fragment uses the rule of "bitwise and" and the rule for typing an array element. The proof of the memory-safety verification condition from line 7 proceeds in a similar manner.

The details of how the theorem prover discovers which rules to use and in what order are presented in Chapter 7. I only remark here that the theorem prover is a complicated system, involving complex interactions between decision procedures that are themselves complex. Therefore, it is essential that the theorem prover not be included in the trusted infrastructure or else the assurance argument for PCC is weakened considerably. In fact, the theorem prover and the system that hosts it need not be trusted because we can easily verify the resulting proof using a simple proof checker, as described in the next section. This immediately suggests a simple method to test a theorem prover: after each successful run, verify the emitted proof using the proof checker; a proof checking failure signals a soundness error in the theorem prover. My practical experience shows that this testing procedure is extremely effective at discovering subtle bugs, while having only a very small run-time cost. This procedure does not guarantee that all latent theorem prover bugs are found. But it guarantees to find them as soon as they manifest themselves.

While the proof checker can be used for testing the soundness of theorem provers, it does not help to discover completeness errors. A completeness error is when the theorem prover fails to prove a predicate that, by design of the prover and its decision procedures, should be provable. Completeness bugs are even more difficult to spot than soundness bugs, because it is difficult to distinguish them from the instances when the decision procedures themselves are incomplete. Fortunately, because we use the theorem prover in conjunction with the certifying compiler and VCGen, we can also discover many completeness errors. Recall that the certifying compiler for a type-safe language is supposed to produce only type-safe target code that, together with the loop invariants, must lead to guaranteed provable verification conditions. If the theorem prover cannot prove such a verification condition, then we are facing either a compiler bug (manifested as unsafe target code) or a prover incompleteness bug. In effect, the ensemble of the certifying compiler and VCGen provides a method for automatically producing very large predicates that are guaranteed to be provable. In this fashion, I have discovered very subtle errors both in individual decision procedures and in the interaction among them, errors which would have been extremely difficult to spot otherwise.

Because of the major software engineering advantages of designing the theorem prover to emit easily checkable proofs, and because such a design is not much more complex than the one used traditionally, my experience suggests that all theorem provers should be certifying, independent of whether they are used for proof-carrying code or not.

2.1.5 Step 5: Verifying the Proof

The last step in a PCC session is the proof validation step performed by the code receiver to verify the correctness of the proof returned by the proof producer. This phase is performed using a proof checker that verifies that each inference step in the proof is a valid instance of one of the axioms or inference rules specified as part of the logic. In addition, the proof checker verifies that the proof proves the same verification condition that was generated in Step 3 and not another predicate.

So far, we have ignored the details of the actual representation of verification conditions and proofs as they are produced and sent back and forth. We cannot ignore this aspect any longer because the algorithm used for proof checking depends intimately on the method used for representing predicates and their proofs. In this section, I describe only the basic principles that are behind proof representation and validation, and defer the complete formalism and implementation details to Chapter 5.

A good technique for representing and validating proofs must have the following desirable attributes:

- The representation of proofs and the proof checking algorithm should be logic independent so that the implementation can be reused for multiple applications of proof-carrying code. Even the details of proof checking that must necessarily depend on the particular logic must be isolated in a high-level logic description file. This property also leads to an increased level of confidence in the PCC infrastructure.

- The proof checking algorithm must be simple so that it can be trusted easily.

- Proof checking must be relatively fast so that its cost is amortized quickly.

- Proofs and predicates must be represented in a compact form in order to minimize the cost of communication between the code receiver and the proof producer.

Fortunately, most of the above desiderata can be attained by using techniques developed as part of basic type-theory research. The Edinburgh Logical Framework (also referred to as LF) has been introduced by Harper, Honsell and Plotkin [HHP93] as a metalanguage for high-level specification of logics. LF provides natural support for the management of binding operators and of hypothetical and schematic judgments through LF bound variables. This is a crucial factor for the succinct formalization of proofs.

For the purposes of this section, we can view LF as a typed λ-calculus, with variables, constants, applications and abstractions. To represent predicates in LF we first declare a set of LF constants standing for the predicate constructors. Proofs are represented similarly by using a set of constants standing for the axioms and inference rules of the logic. The ensemble of the constants denoting predicate constructors and inference rules is called an *LF signature* and constitutes the concrete encoding of the logic in LF.

A fragment of the LF signature that defines the first-order predicate logic extended with the rules of Figure 2.2, is shown in Figure 2.9. The top section of the figure contains declarations of the type constructors `exp`, `tp` and `pred` corresponding respectively to expressions, types and predicates, and of the type family `pf` indexed by predicates. In the middle section of Figure 2.9 there are the declarations of a few syntactic constructors. For example, the LF constant `true` is declared to have type `pred`, meaning that it is a nullary predicate constructor, and the constant `array` is declared as a binary type constructor whose arguments must be a type (representing the type of the elements) and an expression (representing the length) respectively.

In the bottom section of Figure 2.9 there are the declarations of a few inference rules as proof constructors. To understand these declarations, note that if P is the representation of a predicate, then "`pf` P" is the LF type of all valid proofs of P. For example, the constructor `truei` is the representation of the axiom stating the truth of the predicate "true". In the next line we have the declaration corresponding to the conjunction introduction rule, that can be used to create a proof of a conjunction from the proofs of the conjuncts. If we ignore the Π binding operators, we see that the "`andi`" proof constructor must be applied to a

`exp`	:	`Type`
`tp`	:	`Type`
`pred`	:	`Type`
`pf`	:	$\texttt{pred} \rightarrow \texttt{Type}$
`true`	:	`pred`
`and`	:	$\texttt{pred} \rightarrow \texttt{pred} \rightarrow \texttt{pred}$
`imp`	:	$\texttt{pred} \rightarrow \texttt{pred} \rightarrow \texttt{pred}$
`int`	:	`tp`
`array`	:	$\texttt{tp} \rightarrow \texttt{exp} \rightarrow \texttt{tp}$
`of`	:	$\texttt{exp} \rightarrow \texttt{tp} \rightarrow \texttt{pred}$
`saferd`	:	$\texttt{exp} \rightarrow \texttt{exp} \rightarrow \texttt{pred}$
`truei`	:	`pf true`
`andi`	:	$\Pi P\!:\!\texttt{pred}.\Pi R\!:\!\texttt{pred}.\texttt{pf}\ P \rightarrow \texttt{pf}\ R \rightarrow \texttt{pf}\ \ (\texttt{and}\ P\ R)$
`andel`	:	$\Pi P\!:\!\texttt{pred}.\Pi R\!:\!\texttt{pred}.\texttt{pf}\ \ (\texttt{and}\ P\ R) \rightarrow \texttt{pf}\ P$
`szbool`	:	$\texttt{pf}\ \ (= (\texttt{sizeof}\ \texttt{bool})\ 1)$
`rdarray`	:	$\Pi M\!:\!\texttt{exp}.\Pi A\!:\!\texttt{exp}.\Pi I\!:\!\texttt{exp}.\Pi L\!:\!\texttt{exp}.\Pi T\!:\!\texttt{tp}.$
		$\texttt{pf}\ \ (\texttt{of}\ A\ (\texttt{array}\ T\ L)) \rightarrow$
		$\texttt{pf}\ \ (= (\texttt{sizeof}\ T)\ 1) \rightarrow$
		$\texttt{pf}\ \ (>= I\ 0) \rightarrow$
		$\texttt{pf}\ \ (< I\ L) \rightarrow$
		$\texttt{pf}\ \ (\texttt{of}\ (\texttt{sel}\ M\ (\texttt{plus}\ A\ I))\ T)$

Figure 2.9: Fragment of the LF signature corresponding to the logic of Figure 2.2.

proof of some predicate P and a proof of R to obtain an LF expression that represents a valid proof of "$P \wedge R$". Similarly, the last declaration shown in Figure 2.9 encodes the last inference rule shown in Figure 2.2 on page 14.

To illustrate the LF encoding of proofs, consider the subproof $\mathcal{D}$ from Figure 2.8 on page 23. The LF encoding of this fragment is shown in Figure 2.10. If we ignore the boxed components, then the LF representation is a straightforward expression of the proof tree structure: use the "`rdarray`" inference rule with the assumption u as the first hypothesis, followed by the axiom "`szbool`", then by the conjunction-elimination-left rule applied to the assumption v, and finally by the assumption w. Furthermore, by using the declarations of Figure 2.9 and appropriate types for the assumptions (i.e., u has type "$\texttt{pf}\ \ (\texttt{of}\ A\ (\texttt{array bool}\ L)$", v has type "$\texttt{pf}\ \ (\texttt{and}\ (>= I\ 0)\ (\texttt{of}\ R\ \texttt{bool}))$", and w has type "$\texttt{pf}\ \ (< I\ L)$") then we can verify that the whole term of Figure 2.10 has type "$\texttt{pf}\ \ (\texttt{of}\ (\texttt{sel}\ M\ (\texttt{plus}\ A\ I))\ \texttt{bool})$".

$$D = \texttt{rdarray}\ \boxed{\mathrm{M}}\ \boxed{\mathrm{A}}\ \boxed{\mathrm{I}}\ \boxed{\mathrm{L}}\ \boxed{\texttt{bool}}\ u\ \texttt{szbool}\ (\texttt{andel}\ \boxed{(\texttt{>=}\ \mathrm{I}\ 0)}\ \boxed{(\texttt{of}\ \mathrm{R}\ \texttt{bool})}\ v)\ w$$

Figure 2.10: The LF representation of the proof fragment of Figure 2.8. The boxed parts are redundant as explained in the text.

So far we considered the proof representation problem in isolation from the proof validation problem. Nevertheless, it is suggested by the representation example discussed above that the LF type system can be used to check proof validity. Specifically, if D is an LF expression of type "`pf` P", for some predicate P, then D is a representation of a valid proof of P hence such a proof exists meaning that P is a valid predicate. More precisely, in order for the code receiver to be satisfied that a verification condition is provable within the logic, it must receive from the proof producer an LF expression D that can be checked to have the type "`pf` VC", where VC is the LF representation of the verification condition. Formal statements of the adequacy of LF type checking for proof checking are proved both in [HHP93] and in Chapter 5.

Although I do not show here the precise definition of the LF type system and the LF type-checking algorithm, I remark only that, because of the simplicity of the LF language, the type-checking algorithm is also simple and can be easily turned into simple and trustworthy proof checkers. Furthermore, the LF type-checking algorithm is independent of a particular signature, and therefore independent of a particular logic. The only dependency on the logic is the LF signature, which is a straightforward encoding of the axioms and inference rules. In conclusion, LF attains three of the four desirable properties listed at the beginning of this section. The only missing property is the compactness of proof representations.

LF representation of proofs are not compact because of a large amount of redundancy in the representation. There are two main forms of redundant terms that occur in the LF representation of a proof. First, there are terms that can be recovered from the proved predicate. These are called *inherited* terms. Then, there are terms that can be recovered from the context in which they occur in the proof. These are called *synthesized* terms.

To illustrate informally the redundancy of LF representation, consider the LF proof of Figure 2.10 which has type "`pf (of (sel` M `(plus` A I`)) bool)`". From this type, considering the declared type of the top-level constant "`rdarray`", we can inherit the first, second, third and fifth arguments of "`rdarray`". The fourth argument of "`rdarray`" can be synthesized from the type of "u" or "w". Then, the first argument of "`andel`" can be inherited from the type of the entire application of "`andel`" and the second argument can be synthesized from the type of "v".

It seems natural to try to avoid representing redundant subterms of proofs. In Chapter 5 I show how this can be done in a systematic manner by extending the LF framework to deal with missing subterms that can be either inherited or synthesized by using a suitably modified type-checking algorithm. The resulting framework is called implicit LF or LF_i.

$$D_i = \texttt{rdarray}\ u\ \texttt{szbool}\ (\texttt{andel}\ v)\ w$$

Figure 2.11: The LF_i representation of the proof fragment of Figure 2.8. The boxed components are redundant.

To illustrate the effect of representing proofs implicitly, I show in Figure 2.11 the LF_i version of the representation from Figure 2.10. Note that now the representation contains the entire structure of the proof tree and nothing more. In practice, the effects of implicit representation are more drastic than what is suggested in this simple example. Experimental measurements show that the size of the implicit representations of a proofs is approximately equal to the square root of the size of LF representation, so that the benefits of LF_i become larger as the proofs are larger. Furthermore, similar benefits are observed for the time required for proof checking, because the synthesized and inherited subterms do not require type checking. In a few of the larger experiments with PCC, the implicit representation can make the difference between completely impractical proofs of tens of megabytes and manageable proofs of tens of kilobytes.

By solving the proof compactness problem at the abstract level of LF, I am able to obtain a general and logic-independent solution and also to prove formally the adequacy of the LF_i type-checking algorithm for proof reconstruction and checking.

This concludes the overview of the basic proof-carrying code protocol, as shown in Figure 2.1. The rest of this dissertation is concerned both with the formal and implementation details of the components and steps described in this overview. Before that, however, it is worth considering several variations from the basic PCC protocol.

2.2 Variants of Proof-Carrying Code

Figure 2.1 on page 11 and the five-step process described in Section 2.1 present a canonical view of proof-carrying code. However, this approach to PCC is not the only one possible. By redistributing the tasks between the entities involved we can adapt PCC to special practical circumstances while maintaining the same safety guarantees.

For example, in one variant of PCC the code producer runs VCGen itself and then submits the resulting predicate to the proof producer directly. Then the code and the proof are sent together to the code receiver that runs VCGen again and verifies that the incoming proof proves the right verification condition. This arrangement is possible because there is nothing secret about VCGen and it can therefore be given to untrusted code producers to use. To retain the safety guarantees of original PCC, it is necessary that the code receiver repeats the VCGen step in order to produce a trustworthy verification condition. Because this version saves a communication step in generating the safety predicate, it is preferred over the interactive version when the latency of the verification must be minimized.

In another variant of PCC the code receiver does itself the proof generation. For this to be possible it must be the case that the verification condition be relatively easy to prove automatically without extra knowledge about the program. This variant of PCC is useful in situations when the generated proof would be too large to send over the communication channel between the proof producer and the code receiver. Even though the receiver does more work in this variant of PCC, the safety-critical infrastructure, consisting of VCGen and the proof checker, remains the same. One could be tempted to save the cost of generating, storing and verifying the proof altogether by trusting the theorem prover on the receiver side. But this savings is at the expense of greatly increasing the size and complexity of the safety-critical infrastructure, and practical experience suggests that relying on the soundness of a complex theorem prover is a dangerous game.

Yet another scheme for employing PCC is to use one of the variants above to establish the safety of the code on a firewall machine, and then forward the code to any actual receiver within the enclave, possible accompanied by a digital signature. Note that in this case it is the proof checker and not a fallible human agent who signs the code.

No matter which of these or other variants are chosen, they all share the same characteristic of requiring supporting information in addition to the code so that it is possible to rely only on a small and well-defined safety-critical infrastructure, given by a simple proof checker and VCGen.

2.3 Benefits and Costs of Proof-Carrying Code

Proof-carrying code has several key characteristics that, in combination, give it an advantage over previous approaches to safe execution of foreign untrusted code. In addition, there are costs associated with using PCC. I state these advantages and costs here up front to provide a logical completion of the overview chapter, even though the supporting data and examples are presented later in this dissertation.

For proof-carrying code several advantages can be claimed:

1. *PCC is general.* PCC can be used to enforce more than memory safety, more even than type safety. At an extreme, PCC can be used to verify any code property for which there exists a logic capable of expressing it. This includes many code properties that would otherwise be undecidable to infer from the code alone. PCC has been tested with safety properties ranging from memory and type safety to bounded resource usage.

2. *PCC receiver infrastructure is low-risk and automatic.* The proof-checking process used by the code receiver to determine agent safety is completely automatic, and can be implemented by a program that is relatively simple and easy to trust. Thus, the

safety-critical infrastructure that the code receiver must rely upon is reduced to a minimum.

3. *PCC is efficient.* In practice, the proof-checking process runs quickly. Furthermore, in contrast to previous approaches, the code receiver does not modify the code in order to insert costly run-time safety checks, nor does the receiver perform any other checking or interpretation once the proof itself has been validated and the code installed.

4. *PCC does not require trust relationships.* The code receiver does not need to trust the code producer or the proof producer. In other words, the receiver does not have to know the identity of the producer, nor does it have to know anything about the process by which the agent code was produced. All of the information needed for determining the safety of the code is included in the annotated agent code and its proof.

5. *PCC is flexible.* The proof-checker does not require that agents be programmed in a particular programming language. PCC can be used for a wide range of languages, even machine languages, after appropriate adaptation of the VCGen component. Furthermore, a code receiver can support multiple agent languages and safety policies with a minimal duplication of the infrastructure components.

6. *PCC generation can be automated in special cases.* If the safety properties can be decided statically or enforced through systematic run-time checks, a certifying compiler together with a matching theorem prover can be used on the producer side to automate the process of producing the annotations and proofs.

Once the safety policy is defined, PCC involves a two-stage interaction process. In the first stage, the code receiver inspects the agent code and replies with a challenge, a predicate that is provable only if the code is safe to execute. In the second stage, the code receiver checks the validity of the proof using a simple and fast proof checker. If the proof is found to be a valid proof of the verification condition, then the untrusted code is installed and executed.

This two-stage verification process is a key design element contributing to the advantages claimed above. In particular, this is the reason why PCC can be used to certify code properties that would be very difficult, or even impossible to infer from the code directly. Also, by staging the verification into a difficult phase (proof generation) and a simple phase (proof checking) I am able to minimize the complexity of the safety-critical infrastructure. This greatly reduces the risk that a bug in the system can lead to the failure to detect unsafe programs. In fact, I have made it a design goal of PCC that any task whose result can be more easily checked than generated should be performed by an untrusted entity (the code producer or the proof producer) and then checked by the code receiver.

In addition to the PCC benefits for untrusted code execution, the prototype PCC system that I have built demonstrates that ideas from PCC have software-engineering advantages for building more robust theorem provers and compilers. The soundness of a theorem prover does not have to be trusted; instead the prover should emit proofs that can be easily checked. Similarly, the correctness of a compiler does not have to be trusted completely; instead the compiler should annotate the code so that a simple VCGen in conjunction with a theorem prover can verify key properties of the result of each compilation. Finally, the combination of a certifying compiler and a theorem prover can be used to discover unintended incompleteness in a theorem prover. Because of these advantages, and because adapting a theorem prover or a compiler to fit in the PCC framework is not difficult, all compilers and theorem provers ought to be certifying!

The benefits of PCC and certifying compilation discussed above in this section do not come for free. The costs of using PCC for active client-server interaction are:

1. Because PCC is a cooperative process, the code producer must be involved in establishing the safety of the agent code, in contrast with other approaches where the safety enforcement is completely transparent for the code producer. This poses difficulty in deploying PCC, as the potentially more numerous code producers need to be made PCC-aware, not only the code receivers. In addition, wide deployment of PCC requires establishing a standard logic and set of axioms, or libraries of logics and axioms.

2. Proof-carrying code prescribes a precise method by which code producers must cooperate with code receivers to assist in the verification of agent properties. Less well-defined is the method by which producers can cooperate with receivers for the purpose of establishing a mutually convenient safety policy.

3. A difficult problem on the producer side is the generation of code annotations. This is even more difficult when the safety properties are complicated, or when the agent code is heavily optimized. The certifying compiler can do this automatically for certain classes of safety properties. When the certifying compiler approach fails, interactive program verification techniques must be used.

4. Finally, proving the verification conditions is a difficult task. The theorem prover that is part of the current PCC system is powerful, but not complete. It is also modular, so new decision procedures can be easily added, but in the difficult cases, the prover must be guided by the user. Theorem proving can be automated when the code and the annotations are compiler-generated, by extending the prover to handle the finite number of patterns of verification conditions that are possible. This is how automation is achieved for the Touchstone certifying compiler.

The costs enumerated above are important, but fortunately they are incurred only by the code producers. By comparing the list of benefits and the list of costs, it becomes apparent the intentional design strategy for PCC: the burden of safety should lie on the producer and not on the receiver of the code. This enables PCC to work with a very small, easy-to-trust and automatic infrastructure, so that the difficult work is done by the code producer, who is in a better position to understand the agent code or to use interactive tools for that purpose.

A final difficulty of using PCC is establishing the safety policy. However, this cost does not belong in the list above because it is a cost incurred by any method for enforcing the safety of untrusted code. What is special in the case of PCC is that the safety policy must be expressed in a concrete manner so that it can be exported to producers. This is in contrast with other approaches where the safety policy is implicit in the implementation of a large interpreter or compiler or a system-call interface. In fact, because PCC requires the safety policy to be expressed in a very precise and concise manner it becomes easier for the designer to avoid errors or even to prove formally the correctness of the safety policy.

To conclude this section I note that the relative balance of advantages and limitations of proof-carrying code demonstrate that this is a promising way to deal with the safety of untrusted code without having to pay the costs of high run-time overhead or of a complex trusted computing base. In the next section, I continue the discussion of the advantages and limitations of proof-carrying code in general and of my implementation in particular.

2.4 Frequently-Asked Questions

The purpose of this chapter is to give only a high-level overview of the techniques of proof-carrying code and certifying compilation. Thus, many details have been necessarily deferred to the main body of the dissertation. My experience with public presentations of this material shows that, at this superficial level of detail, it is quite difficult to grasp the fundamental characteristics and limitations of the proposed techniques. Matters are complicated even more by the running example that I use to make the presentation more concrete and easy to follow. This example can mislead the reader by blurring the distinction between the fundamental capabilities of proof-carrying code and certifying compilation and the limitations of a given implementation.

In this section I make an attempt to address such misunderstandings by answering some of the most common questions that arose during various presentations of this material. Unlike the main body of this overview chapter that focuses on what the proposed techniques *can do*, this section, just like some of my past audiences, focuses on what the techniques or their implementation *cannot do*, at least not yet. This section should be of interest both to users trying to understand if my thesis work solves their problems and to researchers who would like to extend this work.

Q: *What is the relationship between proof-carrying code and certifying compilation?*

A: Proof-carrying code is concerned with verifying that a fragment of code meets a safety specification. For that purpose, proof-carrying code requires code annotations and a proof of safety constructed according to precise rules. Proof-carrying code is general in the sense that it can be used with any safety property for which there exists a formalism for distinguishing safe programs from the unsafe ones. Furthermore, proof-carrying code is indifferent to how the code annotations and the proofs are generated.

Certifying compilation is one particular way of producing the code annotations and the proofs required by proof-carrying code, for a restricted class of safety properties, namely type safety. However, to compensate for the lack of generality, certifying compilation is completely automatic, provided the agents are written in the specified source language.

Q: *How does PCC manage to verify statically even undecidable code properties?*

A: It is true that for most interesting safety properties there is no decision procedure. The distinctive advantage of proof-carrying code is that it sidesteps undecidability by requiring a proof that the code has the desired property. This reduces the problem to proof checking, which is not only decidable but also relatively simple for most logics. Of course, this moves the burden of deciding whether the property holds to the code producer. This is appropriate because, while the code receiver must be prepared to handle agents produced by various means and thus safe for different reasons, a given code producer usually has specific information as to why its agents are safe. Touchstone is a notable example of such a code producer.

Q: *Is verification condition generation a required component of proof-carrying code?*

A: No, proof-carrying code can be implemented without a verification condition generator. Consider, for example, the situation when the agent code is written in a typed language, the safety property to be verified is well-typedness, the code annotations are variable declarations and the proof is a typing derivation. The PCC infrastructure in this case is a type checker. In general, any PCC infrastructure must contain a parser and a proof checker. In the case of a type checker these two functionalities are blended together. The advantage of using a verification condition generator is that it separates the two functionalities, resulting in increased portability and flexibility. The verification condition generator parses the code and performs syntax checking. All the relevant information about the code is encapsulated in a predicate from an architecture-independent logic. This predicate can then be processed with standard tools like theorem provers and proof checkers. However, there are situations when using a verification condition generator limits the enforceable safety properties, as we shall see below.

Q: *Why isn't it possible for a malicious code producer to pack a trivially-valid proof with an unsafe agent code?*

A: Because the proof is checked not only to be valid but also to correspond to the actual agent code. This is done indirectly by verifying that the proof proves the particular verification condition obtained from the agent code.

Q: *What happens if the code or the proof components of a PCC binary are modified while in* *transit from the code producer to the code receiver?*

A: There are three possible scenarios. First, if the code is modified such that its verification condition does not change, the proof and hence the code are accepted. (Section 6.5 shows that many common optimizations transform the code in exactly this manner.) Second, if the proof is unchanged but the code is modified such that its verification condition changes, the proof checker rejects the code independently of whether it is safe or not. Finally, if both the code and the proof are modified so that the new proof is a valid proof of the verification condition extracted from the new code, the PCC infrastructure accepts the code independently of whether it has the same functionality as originally programmed by the code producer. Note, however, that in the latter case the code must still be safe for it to have a proof of safety. Thus, PCC is guaranteed to enforce safety but not authenticity.

Q: *Can the certifying compiler technique be used to generate proofs of other safety properties, beyond type safety?*

A: The crucial detail that makes automatic certifying compilation possible is that the code properties that must be proved for the target programs are statically checkable for the source programs. Thus, certifying compilation can be applied to any code property for which there exists a conservative decision procedure for a suitably restricted source language. The language restrictions are almost always necessary to avoid the inherent undecidability of many interesting properties for programs written without restrictions. For example, if we want our certifying compiler to emit proofs of termination we can restrict the language to disallow general looping and recursion. We can still allow structural recursion and iteration when they can be proved statically to correspond to well-founded induction.

Q: *What are the source-level constructs of the C programming language that Touchstone does not handle?*

A: Touchstone does not support those constructs of C that can be used to generate unchecked run-time errors. These are: pointer arithmetic, the address-of operator, arbitrary casts, union types, stack-allocated arrays, and memory deallocation. None of these restrictions limits drastically the utility of the language. For example, pointer arithmetic can be replaced with array-index arithmetic, most uses of the address-of operator are unnecessary if the language supports call-by-reference, and memory deallocation can be replaced with garbage collection. In addition to the unsafe features of C, the implementation described in this dissertation does not support floating point and function pointers even though they could be supported.

Q: *Can proof-carrying code be used for programs that use dynamic allocation? What about Touchstone?*

A: Both proof-carrying code and the Touchstone compiler can handle dynamic memory allocation. The trick is to introduce an additional element to the machine state, namely the allocation state, that is changed only by the allocation function. Then, the accessibility of memory locations (i.e., the `saferd` and `safewr`) predicates are changed to depend on the allocation state.

Q: *Can proof-carrying code be used with explicit deallocation or garbage collection? What about Touchstone?*

A: Explicit deallocation can be misused to generate dangling pointers and for that reason the Touchstone compiler does not support it. However, Touchstone does not prevent the use of a conservative garbage collector. Proof-carrying code can, in principle, handle explicit deallocation, just because it is possible to formalize the notion of a safe memory deallocation operation. However, all such formalisms are difficult to use and there has been only little progress in automating the proof generation problem.

Q: *How does proof-carrying code handle the run-time stack?*

A: There are at least two ways in which the run-time stack could be handled. First, accesses to the stack could be handled as arbitrary accesses to memory, in which case the safety policy must specify the details of the dynamic allocation and deallocation of stack frames. To simplify the safety policy and to reduce the size of the proofs, the current implementation of VCGen views the stack frame as a local addition to the register file. This is possible because the program cannot create aliases to stack locations. The drawback of this approach is that it complicates the trusted infrastructure to handle a specific programming paradigm. However, in this particular case I felt that handling the stack directly would be useful to a large number of applications and would result in large reductions in the size of the proofs.

Q: *How does proof-carrying code handle arithmetic overflow? What about Touchstone?*

A: Proof-carrying code can handle arithmetic overflow correctly by carefully selecting the axioms and inference rules of arithmetic (see Section 3.3 for an example). The drawback is that many decision procedures that work well for linear arithmetic do not work anymore for the modular arithmetic involved in handling overflow. Because of this reason, my implementation of the Touchstone compiler and of the proof generator do not handle arithmetic overflow. This also means that the implementation of the compiler and theorem prover described here leaves a loophole open for malicious programs to generate unchecked memory errors.

Q: *Can proof-carrying code handle sum types?*

A: Yes, there are no difficulties in handling sum types. An example of a simple sum type is the pointer option type described in Section 6.2. Touchstone does not handle the union types of the C language but it could handle sum types as in Standard ML, where tag checking is under the control of the compiler to ensure that a value of a sum type is always used correctly.

Q: *Can proof-carrying code handle closures and objects, and more generally, pointers from* *data structures to code? What about Touchstone?*

A: The implementation of VCGen, and thus of PCC, described in this dissertation does not handle first-class functions. The reason is that at each function-call site VCGen must know what precondition and what postcondition to use. In the current implementation this is achieved by requiring the function address in all function calls to be a literal value. This limitation can be removed in those cases when all of the actual functions that might be invoked by a given call instruction share a common precondition and postcondition. This is the case, for example, in a higher-order typed language when the specifications contain only types. Another example in this category is an object-oriented language with dynamic method lookup.

Q: *Can proof-carrying code handle run-time code generation?*

A: The implementation of proof-carrying code described in this dissertation cannot handle run-time code generation. This is because VCGen must be able to perform a static inspection of the entire code that is reachable from the agent entry point. It seems that, in order to handle dynamically-generated code, the VCGen approach must be changed drastically or perhaps even abandoned.

Q: *Can proof-carrying code handle properties that are not safety properties, such as termination or information flow?*

A: Proof-carrying code could handle properties that are not safety properties because it has access to the entire agent code. However, once we have settled on using VCGen as the basis for the PCC infrastructure we are limited to handling only safety properties. This means that PCC can verify that certain run-time events defined as unsafe do not happen but it cannot enforce that certain events (e.g., termination) must happen. A limited, although practical, variant of the latter class of events can still be handled by VCGen-based proof-carrying code if we impose deadlines for various events. Thus, termination is not a safety property but termination within a given number of instructions is a safety property and can be handled by the proof-carrying code infrastructure described in this thesis.

Q: *There are processors that will malfunction when less than a specified number of machine cycles intervene between certain instructions. Is this a safety property enforceable with proof-carrying code?*

A: Yes, this is a safety property. The unsafe event that must be prevented is "instruction B is attempted after less than K cycles since instruction A was executed". Although not described in this dissertation, VCGen can be extended with instruction counting to enforce this kind of safety properties. Using such an extension, I experimented successfully with enforcing resource-usage bounds (e.g., bounded termination, bounded memory allocation, bounded bandwidth use) for agents. Such an experiment is described in [NL98c].

Q: *Can proof-carrying code handle aliasing? What about Touchstone?*

A: Yes, proof-carrying code can handle aliasing because VCGen captures all of the information required to account for the effects of aliasing. VCGen does not attempt to predict the effect of a memory write. Instead, VCGen just records the write with a symbolic address and it is up to the proof producer to discover what locations could have been affected. Aliasing is not an issue for Touchstone because it does not affect type safety. Touchstone does not care about the contents of memory locations that could be aliased. It only cares about the type of the contents.

Q: *What is the relative size of proofs, annotations and executable code?*

A: It depends on the safety property that is being verified. For type safety, the experimental results from Section 8.2 show that the proofs are about 2.5 times the size of the executable code, while the annotations are 30% of the size of the code. In a few experiments with more complex safety properties such as bounded use or resources I have observed proofs that are even 10 times larger than the code. As a rule of thumb, if the safety property is more complex the proof is longer.

This list of questions cannot be comprehensive. If you cannot find the answer to your question here, or if the answer is still too imprecise for your taste, you have just found a reason to read the rest of this dissertation and, for the ultimate detail, even the appendices.

Part I

The Proof-Carrying Code Infrastructure

In this first part, which consists of three chapters, I discuss the technique of proof-carrying code from the point of view of the system that receives the code and verifies the associated safety proof. The details of how the proofs are actually produced are the subject of Part II.

Chapter 3 gives the formal requirements of a *safety policy*, that is, the set of conditions that define when particular actions (e.g., function calls, memory reads) can be safely performed by the agent. Then, Chapter 4 describes a *verification-condition generator* that, when given a program, produces a predicate that is provable in a given logic only when the program adheres to the safety policy. This part ends with Chapter 5 that describes the *proof checker* that can be used by a code receiver to verify that a proof object is a valid proof of the verification condition.

Chapter 3

The Safety Policy

Following Schneider [Sch98], a *security policy* is a predicate on sets of agent executions. A program for an agent is a static encoding of the set of all possible executions of the agent. A program is said to satisfy a security policy if the security policy predicate holds for the set of all possible executions of the program.

An important class of security policies are the *security properties*, which are defined in [AS85] as those security policies that can be specified by means of a predicate on individual executions, or equivalently by imposing constraints on each individual execution as opposed to constraints on the set of all possible executions. There are interesting security policies that are not properties. For example, an information flow security policy prohibits correlations between the values of state components with different secrecy status, so that principals cannot infer things about a state component considered classified by observing the values of other unclassified state components. It is obvious that information flow is not a property, because the very notion of correlation involves more than one execution. If we only look at a single execution or even at at subset of the possible executions we might notice correlations that do not exist when all executions are considered.

A further subclass of security properties are the *safety properties*, which stipulate that no "bad thing" happens during the execution [Lam77]. The safety properties are characterized by the fact that they hold for an execution only if they hold for all finite prefixes of the execution. This means that if a safety property fails, the point of failure is identifiable, which makes it possible to check safety properties at run time [Sch98]. Not all security properties are safety properties. There are also *liveness properties* that stipulate that certain "good things" (e.g., termination, release of a resource) must happen [Lam77]. In fact, it can be proven that any security property can be expressed as a combination of a safety and a liveness property [AS85].

For the purposes of this dissertation, I will concentrate on security properties and furthermore on safety properties only. This subset of security policies covers many practical needs. Note that omitting liveness is not such a serious restriction as it might seem. In many practical applications, we require not only that the "good thing" happens, but that it happens quickly. By limiting the time until the "good thing" must happen (e.g., timeouts on termination or resource release) we are suddenly dealing with safety properties. However, because dealing with the notion of time is tricky and requires special machinery, the safety polices discussed in this dissertation do not involve timing properties.

Although not demonstrated in this dissertation, proof-carrying code can in fact certify more than just safety properties. It can deal even with general security properties because the code receiver has access to the code of the agent and not just to single execution traces.

More concretely, assume that an execution is represented as a sequence of state/action pairs, where the state is a mapping from registers and memory locations to values and the actions are events relevant to the safety policy (the potentially "bad things" that might happen). Each pair in the execution sequence says that an action is attempted for a given value of the state. One way to characterize a safety policy in this setup is by means of action preconditions, which are predicates on states. An execution is safe only if in each state/action pair, the action precondition holds for the state component of the pair. The distinguishing feature of this definition of safety is that it examines only the current state for the purpose of deciding whether an action is allowed. However, this does not diminish the generality of the definition, because we can arrange for the state to contain enough history information about the execution so that we can implement an arbitrary predicate on execution prefixes as a predicate on the current state only. One way to do this is by extending the register set with a history pseudo-register, whose value is the sequence of previous execution states.

The set of actions that I am going to consider in this thesis are (1) the execution of individual instructions, (2) the invocation of receiver-provided functions and (3) the termination of the agent execution. Correspondingly, this enables the safety policy to enforce various kinds of restrictions, as follows:

1. Restrictions on what instructions can be executed and in what conditions. This part of the safety policy is referred to as *instruction safety*.

2. Restrictions on what runtime functions (system calls) can be invoked, under what conditions they can be invoked and what can be assumed upon their return. This part of the safety policy is referred to as *system-call safety*.

3. Restrictions on the input/output behavior of agent entry points. This part of the safety policy is customarily referred to as *partial correctness*.

The syntax of the language (described in Section 3.1) effectively restricts the kind of instructions that an agent can execute. The other aspects of the safety policy are introduced as part of a safe interpreter for the language (described in Section 3.2). The distinguishing feature of the interpreter is that it fails whenever the safety policy is violated. Note that this interpreter serves only as a definition of the safety policy with respect to which the PCC infrastructure is proved correct. There is no need to implement such an interpreter.

3.1 SAL: A Generic Assembly Language for Safe Agents

For the code receiver to be able to execute the agent and to make an informed choice of whether the agent meets the safety policy or not, it must require that agents be written in a specific language. One of the main advantages of proof-carrying code is that it can be applied to a variety of languages, ranging from high-level languages all the way to machine languages. I am focusing in this thesis on applications involving low-level languages mainly because they require a smaller execution infrastructure at the code-receiver end and because they give the code producer more freedom in optimizing the code. On the other hand, describing proof-carrying code in the context of a concrete machine language is certainly too specific. Instead, I choose to introduce a generic RISC-like assembly language, which I call `SAL`, as the basis for the description of proof-carrying code. I designed `SAL` so that most instructions in typical machine languages(either RISC or CISC) have direct and simple translations into sequences of `SAL` instructions. `SAL` is generic in the sense that it can be extended with operators that model various features of a target machine. `SAL` is safe in the sense that there is a simple technique that enables a code receiver to ascertain that a given program meets a certain safety policy.

The proof-carrying code infrastructure, and more concretely the VCGen, is defined in this thesis as operating on `SAL` programs. There are two ways in which the infrastructure described in this dissertation can be ported to a specific language. The easy way is to implement VCGen only once for `SAL`, and to write translators from each agent language of interest to `SAL`. As an example of how this is done, I describe in Section 3.3 translators for the DEC Alpha and Intel x86 assembly languages. If the performance of the VCGen is more important than its simplicity, one can integrate the translator into VCGen, effectively obtaining a VCGen for a specific architecture.

The syntax of `SAL` instructions is shown in Figure 3.1. `SAL` is a load/store RISC architecture, with a register set composed of general purpose registers, referred to as $\mathbf{r}_i$ $(i = 1, \ldots, R)$, and a distinguished register "`ra`" used to hold the return address from the current function. The `SAL` machine also has a stack-pointer register that is manipulated using dedicated instructions. The number of general-purpose registers is not specified as it might depend on the target architecture. In the rest of this section I describe informally the purpose of each `SAL` instruction. The formal semantics of `SAL` is presented in Section 3.2.

In order to preserve the generality of `SAL`, I keep the set of arithmetic and conditional operators generic. Thus, I consider that `SAL` has one generic binary expression operator "`EOP`" and one generic unary conditional operator "`COP`" (for comparisons with zero). Note that the operands of both kinds of operators are registers. The conditional-branch instruction specifies also an immediate integer that is the relative offset of the target instruction in the case of a successful comparison. The execution continues with the next instruction if the comparison fails. The jump instruction is also relative.

Registers:	r	$::=$	$\mathbf{r}_i \mid \texttt{ra}$		$i = 1, \ldots, R$
Instructions:	I	$::=$	$r \leftarrow r'$	Move	
		$\mid$	$r \leftarrow n$	Initialize	
		$\mid$	$r \leftarrow r'\ \texttt{EOP}\ r''$	Arithmetic/Logical operations	
		$\mid$	$\texttt{jump}\ n$	Jump	
		$\mid$	$\texttt{cond}\ \texttt{COP}(r), n$	Conditional branch	
		$\mid$	$\texttt{ra} \leftarrow \texttt{pc} + n$	Compute return address	
		$\mid$	$\texttt{call}\ F$	Function call	$F \in \mathit{Func}$
		$\mid$	$\texttt{ret}$	Function return	
		$\mid$	$r \leftarrow M[r']$	Memory read	
		$\mid$	$M[r'] \leftarrow r$	Memory write	
		$\mid$	$\texttt{sp} \leftarrow \texttt{sp} + n$	Advance the stack pointer	
		$\mid$	$r \leftarrow M[\texttt{sp} + n]$	Stack read	
		$\mid$	$M[\texttt{sp} + n] \leftarrow r$	Stack write	
		$\mid$	Annot	Annotations	(defined in Section 4.2)
Numerals:	n	$\in \mathbb{Z}$			

Figure 3.1: The syntax of SAL.

The function invocation and return instructions are, just as in a typical RISC architecture, simple control instructions. The return instruction uses the contents of the ra register as its destination. The call instruction does not, as in many CISC architectures, obtain and save the return address directly. For that purpose, SAL contains a dedicated instruction that references the current value of the program counter. Note that with this definition of SAL, the destination of a call must be a literal and cannot be a computed address. This restriction simplifies considerably the PCC infrastructure but it has the drawback that it does not permit the direct implementation of higher-order languages or dynamic method lookup in object-oriented languages. We shall see in Section 4.2 that in many cases we can relax these restrictions.

The memory operations in SAL have only the register-index addressing mode. This way we are able to isolate in only one place (the generic expression operations) the various computations that might be required to implement more complicated addressing modes. In addition to the regular memory operations, SAL contains distinguished instructions for accessing the stack, in which case only the index/offset addressing mode is supported. There is also an instruction for incrementing/decrementing the stack pointer by a constant amount. This effectively prevents the allocation on the stack of data whose size is not known statically.

Finally, SAL can be extended with *annotations* whose only purpose is to communicate extra information about the program to VCGen. Semantically, the annotations have no effect on the machine state during the execution. In fact, in a practical implementation of PCC the annotations are segregated in a special segment and are not part of the code. Examples of annotations are discussed in Section 4.2 together with the definition of the VCGen.

3.2 The Operational Semantics of `SAL`

In the previous section I defined the syntax of `SAL`, and implicitly I specified the part of instruction safety that limits the kinds of instructions that an agent may execute. In this section, I specify the conditions under which each instruction can be executed (thus completing the discussion of instruction safety), as well as the system-call safety and the partial correctness. I introduce these safety aspects by means of a `SAL` interpreter that is intended to model a typical physical machine, except that it fails with an error whenever the safety policy is violated. But first, I must discuss in more detail the notion of execution state of the `SAL` interpreter.

Just as for a typical machine, the state of the execution consists of the value of the program counter, the values of the registers and the state of the memory. Let $\mathcal{U}^b$ be the universe of base values, that is, the set of possible values for a machine register. In a typical machine using two's-complement representation on W bytes, we have that $\mathcal{U}^b = \{x \in \mathbb{Z} | -2^{8W-1} \leq x \leq 2^{8W-1} - 1\}$. The state of the memory is encoded as a function from a finite set of addresses to values. To simplify the presentation I am considering that the addresses are also represented on W bytes, thus the universe of store values is $\mathcal{U}^s = \mathcal{U}^b \rightarrow \mathcal{U}^b$.

For the purposes of recording the state of the execution it is convenient to store the state of the memory as the value of a dedicated pseudo register "`mem`". Similarly, it is convenient to store the state of the stack pointer as part of the register state. Thus the evaluator records a state for the set of registers $\mathit{Regs} = \{\mathbf{r}_1, \ldots, \mathbf{r}_R, \mathtt{ra}, \mathtt{sp}, \mathtt{mem}\}$.

For technical reasons having to do with various correctness proofs, I extend the state of the `SAL` interpreter with a history $\mathcal{H}$ of register states, which contains the register states at the moment of function invocations. The length of the history sequence is equal to the depth of the function invocation chain from the initial activation of the agent program by the code receiver. Therefore, let the state Σ be a triple of values $<i, \rho, \mathcal{H}>$, where $i \in \mathcal{U}^b$ is the value of the program counter, $\rho \in \mathit{Regs} \rightarrow \mathcal{U}^b$ (except that $\rho(\mathtt{mem}) \in \mathcal{U}^s$) is the state of the registers and $\mathcal{H}$ is the call history represented as a sequence of register states.

The agent code consists of function definitions, whose bodies are sequences of `SAL` instructions. The functions implemented by the agent (whose set is denoted by Φ^A) are of two kinds. First there are the functions that are exported to the receiver (the entry points), whose set is denoted by Φ^E. Then there are the functions that the agent is using internally for the purpose of implementing the functionality of the entry functions. The latter set of functions is denoted by Φ^I. In addition to these functions, a complete execution environment also contains the code for the runtime functions (system calls) Φ^S, which are provided by the receiver and made available to the agent code. The set of external functions in the system is $\Phi^X = \Phi^S \cup \Phi^E$. Thus, the set of all functions in the system is $\Phi = (\Phi^E \cup \Phi^I) \cup \Phi^S = \Phi^A \cup \Phi^S$. It is important to distinguish between the agent-provided functions, which are untrusted, and receiver-provided functions, which are trusted to have a safe behavior and to obey their specification.

In order to preserve the generality of `SAL`, I am making only few assumptions about the layout of instructions in the agent code and in the memory while they execute. I will assume that all function bodies are loaded in disjoint areas of the code segment. Because of this assumption, there is an isomorphism between valid instruction addresses i and pairs consisting of a function name and an offset of the instruction within the function body $<F, j>$. For example, the notation $<F, 0>$ refers to the address of the first instruction of function F. Each `SAL` instruction can occupy more than one memory word, and therefore not all offsets or instruction addresses point to a valid instruction. I write $i \in Dom(F)$ to denote that i is an instruction address that marks the beginning of a valid `SAL` instruction (denoted by F_i) within the body of the function F. I assume that the size of an instruction is a function only of the instruction itself, and I write "i++" to refer to the address immediately following the instruction starting at i. Note that "i++" is not necessarily equal to "$i + 1$" even if $i + 1 \in Dom(F)$. To accommodate relative jumps and branches I use the notation "$i + n$" to denote the usual arithmetic operation on instruction addresses.

Over the course of the next few pages I describe in detail the operation of the safe interpreter shown in Figure 3.2 and at the same time I introduce notation and concepts that arise during the discussion. The operation of the interpreter in a generic state $<i, \rho, \mathcal{H}>$ consists of decoding the current instruction (shown in the first column of Figure 3.2) and to choose a resulting state (shown in the second column), possibly depending on the outcome of control flow checks shown in the third column. The interpreter as defined by the first three columns of Figure 3.2 is meant to model a physical machine. What makes this interpreter a formalization of a safety policy is the fourth column that contains the safety checks to be performed at each step. An execution for which a safety check fails is considered to violate the safety policy. Implicitly, the interpreter defines the safety policy.

For the register move, the initialization with a literal, the jump instruction and the annotations there are no safety restrictions. The instruction safety component of the safety policy for the generic expression operator "`EOP`" is described as a predicate *SafeEOP* defined on $\mathcal{U}^b \times \mathcal{U}^b$. Intuitively, the predicate $SafeEOP(v', v'')$ holds only if the safety policy allows the execution of "$r \leftarrow r'$ `EOP` r''" in a state in which the values of the registers r' and r'' are v' and v'' respectively. In a similar manner we describe the instruction safety for the conditional branches and memory operations.

The notation $\rho[r \leftarrow v]$ denotes the state obtained by setting the value of the register r to v in the current state ρ. The notation $\underline{\texttt{EOP}}$ denotes the function implemented by the machine for the expression operator with the mnemonic "`EOP`".

F_i	Σ'	Control flow test	Safety requirement
$r \leftarrow r'$	$<i{+}{+}, \rho[r \leftarrow \rho(r')], \mathcal{H}>$		
$r \leftarrow n$	$<i{+}{+}, \rho[r \leftarrow n], \mathcal{H}>$		
`Annot`	$<i{+}{+}, \rho, \mathcal{H}>$		
$r \leftarrow r'$ `EOP` r''	$<i{+}{+}, \rho[r \leftarrow \underline{\mathtt{EOP}}(\rho(r'), \rho(r''))], \mathcal{H}>$		$\mathit{SafeEOP}(\rho(r'), \rho(r''))$
`jump` n	$<n + i{+}{+}, \rho, \mathcal{H}>$		
`cond COP`$(r), n$	$<n + i{+}{+}, \rho, \mathcal{H}>$	$\underline{\mathtt{COP}}(\rho(r))$	$\mathit{SafeCOP}(\rho(r))$
`cond COP`$(r), n$	$<i{+}{+}, \rho, \mathcal{H}>$	$\neg\underline{\mathtt{COP}}(\rho(r))$	$\mathit{SafeCOP}(\rho(r))$
$r \leftarrow M[r']$	$<i{+}{+}, \rho[r \leftarrow \rho(\mathtt{mem})(\rho(r'))], \mathcal{H}>$		$\mathit{SafeRd}(\rho(\mathtt{mem}), \rho(r'))$
$M[r'] \leftarrow r$	$<i{+}{+}, \rho[\mathtt{mem} \leftarrow \rho(\mathtt{mem})[\rho(r') \leftarrow \rho(r)]], \mathcal{H}>$		$\mathit{SafeWr}(\rho(\mathtt{mem}), \rho(r'), \rho(r))$
$\mathtt{sp} \leftarrow \mathtt{sp} + n$	$<i{+}{+}, \rho[\mathtt{sp} \leftarrow \rho(\mathtt{sp}) + n], \mathcal{H}>$		$\mathit{Stack}(n + \rho(\mathtt{sp}))$
$r \leftarrow M[\mathtt{sp} + n]$	$<i{+}{+}, \rho[r \leftarrow \rho(\mathtt{mem})(\rho(\mathtt{sp}) + n)], \mathcal{H}>$		$\mathit{Stack}(n + \rho(\mathtt{sp}))$
$M[\mathtt{sp} + n] \leftarrow r$	$<i{+}{+}, \rho[\mathtt{mem} \leftarrow \rho(\mathtt{mem})[\rho(\mathtt{sp}) + n \leftarrow \rho(r)]], \mathcal{H}>$		$\mathit{Stack}(n + \rho(\mathtt{sp}))$
$\mathtt{ra} \leftarrow \mathtt{pc} + n$	$<i{+}{+}, \rho[\mathtt{ra} \leftarrow n + i{+}{+}], \mathcal{H}>$		$n + i{+}{+} \in \mathit{Dom}(F)$
`call` G	$<<G, 0>, \rho, \mathcal{H} + \rho>$	$\mathit{Stack}(\rho(\mathtt{sp}) - \mathit{Max})$	$G \in \Phi$ and $G \in \Phi^X \supset \mathit{Pre}_G(\rho)$
`call` G	$<i, \rho, \mathcal{H}>$	$\neg\mathit{Stack}(\rho(\mathtt{sp}) - \mathit{Max})$	$G \in \Phi$
`ret`	$<\rho(\mathtt{ra}), \rho, \mathcal{H}'>$		$\mathcal{H} \equiv \mathcal{H}' + \rho'$ and $F \in \Phi^X \supset \mathit{SafeRET}_F(\rho', \rho)$

Figure 3.2: The operational semantics of SAL, giving the state Σ' obtained from $<i, \rho, \mathcal{H}>$ in the case when $i \in \mathit{Dom}(F)$.

For certain instructions the resulting state depends on the outcome of control flow tests (shown in the third column of the definition) that the interpreter performs. For such instructions there are two separate lines in the definition of the interpreter, one for the case when the tests succeeds and one for the case when it fails. An obvious case of such conditional behavior is the conditional branch instruction. In this case the interpreter tests whether the current value of the tested register is in the unary relation $\underline{\mathtt{COP}}$, which is the relation that our target architecture implements for the conditional operator with the mnemonic "COP".

For a memory read operation "$r \leftarrow M[r']$, the instruction safety is described by a relation $\mathit{SafeRd}(\rho(\mathtt{mem}), \rho(r'))$. Note that this form of the relation allows the readability property of a memory location to change as the memory contents is changed. Similarly, for memory writes, the safety policy can restrict the set of values that can be written at various memory addresses in various memory states, by defining appropriately the ternary relation SafeWr. The notation $\mu[a \leftarrow v]$ denotes the memory state obtained from μ after writing v to the address a.

Normally, the runtime stack used by most programming language implementations can be modeled as a portion of the memory, without any special additions to the SAL language.

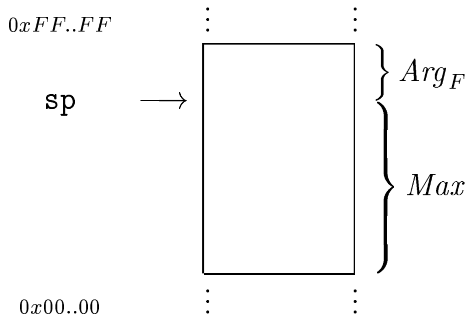


Figure 3.3: The stack frame for a `SAL` function F. At the moment of invocation the addresses $\{\mathtt{sp}, \ldots, \mathtt{sp}+Arg_F-1\}$ contain the function's arguments. The frame can be further extended towards lower addresses with up to Max local arguments.

Previous experience with PCC shows that this approach, while feasible, is unnecessarily costly. The reason is that in most cases the stack accesses follow a predictable pattern making it possible to verify their safety without using the heavyweight machinery designed for arbitrary memory operations.

To benefit from the known stack access patterns, the interpreter imposes restrictions on how `SAL` programs use the stack. First, there are two dedicated memory operations that must be used for accessing the stack. Then, the stack pointer can be changed only by a dedicated increment instruction. Finally, the interpreter imposes a stack usage discipline by restricting each function's access to the stack to a very limited area called the current invocation's stack frame, as shown in Figure 3.3. For each function F let Arg_F be the number of arguments that it expects on the stack. Then the stack frame of an invocation of F spans the addresses $\{\mathtt{sp} - Max, \ldots, \mathtt{sp} + Arg_F - 1\}$, where Max is the maximum number of stack locations that a function is allowed to use in excess of its arguments. A further requirement is that the stack grows towards lower addresses.

The safety policy is extended correspondingly with a unary relation *Stack* that holds for those addresses that have been reserved by the receiver for the runtime stack. To maximize the benefit of having preferential treatment of the stack, we need to prevent all aliasing between regular memory accesses and the stack accesses. Thus for `SAL` to be adequate for a safety policy we need that the stack aliasing condition of Property 3.1 holds. This condition is attained in practice by setting disjoint stack and data segments. Furthermore the code, data and stack segments must all be disjoint.[1]

$$\forall a \in \mathcal{U}^b.Stack(a) \supset \forall \mu \in \mathcal{U}^s.\forall v \in \mathcal{U}^b.(\neg SafeRd(\mu, a) \wedge \neg SafeWr(\mu, a, v)) \tag{3.1}$$

[1] This implies that run-time code generation is not implementable in `SAL` and therefore not verifiable with the variant of proof-carrying code described here.

Space on the stack is allocated usually upon entering a function or a local declaration block by decrementing the stack pointer. The safety condition for the stack pointer advance instruction is that the resulting value of the stack pointer is a valid stack address. This check exists mostly to simplify the formal proofs from Chapter 4. In particular, a successful outcome of this check means that there is not a danger of overflow when performing the addition.

It is possible that an agent attempts to use more stack space than was allocated by the safety policy by means of the *Stack* relation. To prevent such stack overflow, the instruction safety for all instructions involving the stack prescribes that the address involved be part of the *Stack* predicate.

The remaining instructions are the function call and return instructions, together with the return address computation instruction. The latter is the simplest, noting that safety requires the resulting address to be within the current function's body. Again, this check exists mostly for technical reasons.

There are two complications with the function call instruction. One is that this is where the system-call safety part of the safety policy is enforced. The other is that the function call is defined in Figure 3.2 to have an unexpected conditional behavior. The motivation for this behavior is mostly technical, having to do with modeling the stack overflow exception. Anticipating the difficulty of proving statically the lack of stack overflow, I designed the `SAL` interpreter so that it performs one stack-overflow check for each function invocation. The check itself verifies that the stack pointer can be decreased by *Max* without overflowing the stack. A successful outcome for this check guarantees that the entire stack frame for a function that needs less than *Max* stack slots in addition to the arguments, fits in the stack segment. For this to be true we require that the stack be allocated to a continuous area of memory:

$$a \leq a' \wedge Stack(a) \wedge Stack(a') \quad \supset \quad \forall x. a \leq x \leq a' \supset Stack(x) \tag{3.2}$$

The interesting question is what to do if the stack-overflow check fails. Technically, this situation constitutes an exception and should be treated as a non-local transfer of control to an exception handler. To simplify the formal correctness proofs, I model the exception as non-termination. The intuition is that the safety policy considers the stack overflow exception as a safe outcome of the program, for which the partial correctness aspect is not important. Because we only consider safety properties and not liveness properties, we can treat benign exceptions such as the stack overflow as infinite looping, as defined in the semantics of the call instruction in Figure 3.2.

The system-call safety and the partial correctness safety aspects of the safety policy are similar in the sense that they constrain the state of the execution at function invocation boundaries. Both of these cases can be dealt with in a uniform way by means of function preconditions, postconditions and sets of preserved registers. The *precondition* of a function

F is a predicate Pre_F on the state of the memory and registers, specifying what may be assumed when the execution of F starts. If F is a system call then the precondition must be established by the agent code prior to the call. If F is an agent entry point then the precondition can be assumed to have been established by the receiver prior to invoking the agent. In a sense, the precondition establishes the calling convention for each function.

The *postcondition* of a function F is a predicate $Post_F$ on the state of the memory and registers, denoting what may be assumed when (and if) the execution of F ends. If F is a system call then the postcondition may be assumed to have been established by the code receiver prior to resuming the execution of the agent. If F is an agent entry point then the postcondition must be established by the agent code prior to returning.

The *callee-save register set* is the set of registers CS_F that the function F must preserve. Although each function must preserve the stack pointer register, "`sp`" does not need to appear in the callee-save set because its preservation is enforced by the `SAL` interpreter. However, because the "`ra`" register is not under the complete control of the interpreter we must require that:

$$\mathtt{ra} \in CS_F \tag{3.3}$$

While performing a function return, the interpreter must verify that the postcondition holds, that all registers that are declared callee-save have been preserved and also that the contents of the stack was not changed outside the frame. It is convenient to denote these checks collectively by a relation $SafeRET_F$ between the initial and final states, as defined below:

$$SafeRET_F(\rho_0, \rho) \quad \text{iff} \quad \begin{cases} Post_F(\rho), \quad \text{and} \\ \forall r \in CS_F . \rho_0(r) = \rho(r), \quad \text{and} \\ \rho(\mathtt{sp}) = \rho_0(\mathtt{sp}), \quad \text{and} \\ \forall a . Stack(a) \wedge a \geq \rho_0(\mathtt{sp}) + Arg_F \supset \rho_0(\mathtt{mem})(a) = \rho(\mathtt{mem})(a) \end{cases} \tag{3.4}$$

where ρ_0 is the state when the current function's invocation was started and ρ is the state at the time of the return.

Note that the invocation history component of the state is changed only by the call and return instructions. In the case of a call, the current register state is added to the history. In the case of a return instruction, the interpreter first verifies that the current call history is not empty, and removes the top element from the history. Finally, note that the execution of an annotation does not change the state of the execution, except for the program counter.

The purpose of the `SAL` interpreter presented in this chapter is twofold. First, it serves as a semantics for the `SAL` language and thus indirectly as a guide for mapping concrete machine languages to `SAL`. The interpreter also constitutes the formalization of the safety policy. The interpreter is set up in such a way that it cannot make progress from a state that does not point to a valid instruction or that does not satisfy the safety requirements mandated by the safety policy. I write $\Sigma \to \Sigma'$ to say that the interpreter executes one step from the state Σ resulting in the new state Σ', and implicitly to say that there is no violation of the safety policy in state Σ. With this notation we can state formally what it means for a `SAL` program to match the safety policy, as follows:

Definition 3.5 (Safety Policy) *A function $F \in \Phi$ is* safe*—written $Safe(F)$—if for any initial register state ρ_0 and history $\mathcal{H}_0$ such that $Pre_F(\rho_0)$ and such that $Stack(a)$ holds for all addresses "a" such that $\rho_0(\mathtt{sp}) - Max \leq a \leq \rho_0(\mathtt{sp}) + Arg_F - 1$ then for any state $\Sigma = <i, \rho, \mathcal{H}>$ reachable by the* `SAL` *interpreter from the initial state $\Sigma_0 = <<F, 0>, \rho_0, \mathcal{H}_0 + \rho_0>$, we have that either:*

1. *$|\mathcal{H}| = |\mathcal{H}_0|$, in which case $i = \rho_0(\mathtt{ra})$ and $\mathcal{H} = \mathcal{H}_0$ and $SafeRET_F(\rho_0, \rho)$, or*
2. *$|\mathcal{H}| > |\mathcal{H}_0|$, in which case there exists Σ' such that $\Sigma \to \Sigma'$ (the interpreter can make progress, or equivalently there is no safety violation in Σ).*

The notation $|\mathcal{H}|$ is used to denote the size of the call history. Note that in the above safety definition the termination of an invocation is denoted by the size of the history decreasing by one. Before that happens, the history might be increased temporarily as the current function invokes other functions.

The safe interpreter is defined in terms of the generic assembly language `SAL` while, in practice, agents are expressed in concrete languages. In the next section, I describe a general strategy for adapting the safety policy to a particular assembly language and I demonstrate the strategy for the DEC Alpha and the Intel x86 assembly languages. Then, in Section 3.4, I discuss further the role of the safe interpreter defined here in the whole proof-carrying code scheme, and show how it fits with the rest of the infrastructure.

3.3 Porting Proof-Carrying Code to Concrete Architectures

In this section I describe briefly the general strategy for porting the safety policy and the PCC infrastructure to a concrete architecture. This is accomplished by designing a translator from machine instructions of the target architecture to a variant of SAL. The generic SAL language discussed in this chapter can be instantiated as needed for a particular architecture by:

1. Choosing an appropriate set of SAL registers. These are usually the target machine registers extended with SAL temporary registers.

2. Choosing instantiations of the generic expression operators "EOP" and conditional operators "COP". In some cases, the concrete operators might need to have different arities than the one presented in the generic SAL. For each added expression operator, we must also specify the mathematical functions implemented by the target machine (i.e., the "$\underline{\texttt{EOP}}$" and "$\underline{\texttt{COP}}$" functions) and the instruction safety conditions (i.e., the *SafeEOP* and *SafeCOP* relations), if any.

3. Defining the size of instructions and the function "i++".

In the rest of this chapter I will show how the outlined strategy can be applied in the concrete cases of a RISC architecture (DEC Alpha) and a CISC architecture (Intel x86).

3.3.1 Porting to the DEC Alpha Architecture

The DEC Alpha architecture [Sit92] is a load/store 64-bit RISC architecture. Because the DEC Alpha and SAL are both RISC architectures, the translation is particularly easy. This instantiation of SAL has 29 general purpose registers mapped to DEC Alpha registers. In addition to these $\mathbf{r}_{26}$ is mapped to SAL register ra and the register $\mathbf{r}_{30}$ is the stack pointer register sp and there are two more temporary registers t_1 and t_2 that are used to compute temporary results during the translation. In this 64-bit instance of SAL the base universe is $\mathcal{U}^b = \{x \in \mathbb{Z} | -2^{63} \leq x \leq 2^{63} - 1\}$.

Figure 3.4 shows the translation to SAL of a small but practical 64-bit subset of the DEC Alpha instruction set. For the added expression and conditional operators here are the mathematical functions implemented by the DEC Alpha.

$$\begin{array}{lcl}
\underline{\alpha\texttt{ADDQ}}(v_1, v_2) & = & (v_1 + v_2 + 2^{63}) \bmod 2^{64} - 2^{63} \\
\underline{\alpha\texttt{SUBQ}}(v_1, v_2) & = & (v_1 - v_2 + 2^{63}) \bmod 2^{64} - 2^{63} \\
\underline{\alpha\texttt{EQ}}(v) & = & v = 0 \\
\underline{\alpha\texttt{GE}}(v) & = & v \geq 0
\end{array}$$

DEC Alpha	$\mathtt{SAL}$
$\mathtt{lda}\ r, n(\mathtt{zero})$	$r \leftarrow n$
$\mathtt{mov}\ r, r'$	$r' \leftarrow r$
$\mathtt{addq}\ r_1, r_2, r_3$	$r_3 \leftarrow r_1\ \alpha\mathtt{ADDQ}\ r_2$
$\mathtt{subq}\ r_1, r_2, r_3$	$r_3 \leftarrow r_1\ \alpha\mathtt{SUBQ}\ r_2$
$\mathtt{jmp}\ n$	$\mathtt{jump}\ n$
$\mathtt{beq}\ r, n$	$\mathtt{cond}\ \ \alpha\mathtt{EQ}\,(r), n$
$\mathtt{bge}\ r, n$	$\mathtt{cond}\ \ \alpha\mathtt{GE}\,(r), n$
$\mathtt{jsr}\ \mathtt{ra}, F$	$\mathtt{ra}\ \leftarrow \mathtt{pc} + 1$ $\mathtt{call}\ F$
$\mathtt{jsr}\ \mathtt{zero}, (\mathtt{ra})$	$\mathtt{ret}$
$\mathtt{ldq}\ r, n(r')$	$t_1 \leftarrow n$ $t_2 \leftarrow r'\ \alpha\mathtt{ADDQ}\ t_1$ $r \leftarrow M[t_2]$
$\mathtt{stq}\ r, n(r')$	$t_1 \leftarrow n$ $t_2 \leftarrow r'\ \alpha\mathtt{ADDQ}\ t_1$ $M[t_2] \leftarrow r$
$\mathtt{lda}\ \mathtt{sp}, n(\mathtt{sp})$	$\mathtt{sp} \leftarrow \mathtt{sp} + n$
$\mathtt{ldq}\ r, n(\mathtt{sp})$	$r \leftarrow M[\mathtt{sp} + n]$
$\mathtt{stq}\ r, n(\mathtt{sp})$	$M[\mathtt{sp} + n] \leftarrow r$

Figure 3.4: The translation table for the DEC Alpha architecture.

The agent entry points must obey the standard DEC Alpha calling convention and preserve the registers $\mathbf{r}_9, \ldots, \mathbf{r}_{15}$ and the return address register. Therefore we must have the condition:

$$\forall F \in \Phi^E.\{\mathbf{r}_9, \mathbf{r}_{10}, \mathbf{r}_{11}, \mathbf{r}_{12}, \mathbf{r}_{13}, \mathbf{r}_{14}, \mathbf{r}_{15}, \mathtt{ra}\} \ \subseteq \ CS_F$$

The DEC Alpha architecture requires that all memory accesses be aligned on a 64-bit boundary. Therefore a correct safety policy must require the following alignment conditions:

$$\begin{aligned} \mathit{SafeRd}(\mu, a) &\supset a \bmod 8 = 0 \\ \mathit{SafeWr}(\mu, a, v) &\supset a \bmod 8 = 0 \\ \mathit{Stack}(a) &\supset a \bmod 8 = 0 \end{aligned}$$

3.3.2 Porting to the Intel x86 Architecture

The Intel x86 architecture [Int97] is a CISC architecture, and therefore the translator to SAL is more complex. Here are the issues that must be dealt with when translating x86 programs to SAL:

1. Many of the x86 instructions have complex definitions and thus map to sequences of SAL instructions. For example, the x86 architecture has several addressing modes that require separate SAL code for computing the operands.

2. The x86 architecture allows instructions to access fragments of a register. This register aliasing must be modeled using special expression operands.

3. The x86 architecture uses a segmented memory model with addresses specified as a pair of a segment descriptor and an offset within the segment.

4. The x86 architecture has several condition flags that are set implicitly by many arithmetic instructions. To model this behavior we need to extend SAL with arithmetic instructions having multiple results.

5. Instructions in the x86 architecture are not all of the same length and therefore the definition of the "i++" function is not trivial.

In the rest of this section I will describe sample solutions to the above issues. The focus here is on simplicity and not on efficiency. Many of the solutions that I propose here can be implemented more efficiently at the expense of some complexity in the translator or the VCGen.

The SAL register set for the x86 architecture contains the return address register and most of the standard x86 registers: EAX, EBX, ECX,EDX, ESI, EDI,EBP, DS, ES, FS and GS. In addition it contains a number of temporary registers denoted by t_i. The ESP register of the x86 architecture is not part of the SAL registers because all uses involving it are translated to instructions manipulating the SAL stack pointer. Each of the x86 condition flags is a SAL register that contains either zero or one. For simplicity I consider here only the "zero flag" (ZF), the "sign flag" (SF) and the "overflow flag" (OF), which are set to one when the result of an arithmetic operation is respectively zero, negative or has resulted in an overflow.

The calling convention of the x86 architecture requires that the segment registers and the base pointer be preserved across function calls. Thus,

$$\forall F \in \Phi^E . \{\texttt{ra}, \texttt{EBP}, \texttt{DS}, \texttt{ES}, \texttt{FS}, \texttt{GS}\} \quad \subseteq \quad CS_F$$

In the x86 architecture the memory is segmented and a memory address is expressed as a pair segment-selector/offset. The segment selector must reside in one of the segment registers. Consequently, we must change the syntax of the memory operation in SAL to allow

Intel x86	SAL
$\texttt{SUB EAX, EBX}$	$\texttt{EAX} \leftarrow \texttt{EAX x86SUB EBX}$ $\texttt{ZF} \leftarrow \texttt{EAX x86SUB.ZF EBX}$ $\texttt{SF} \leftarrow \texttt{EAX x86SUB.SF EBX}$ $\texttt{OF} \leftarrow \texttt{EAX x86SUB.OF EBX}$
$\texttt{MOV EAX}, n$	$\texttt{EAX} \leftarrow n$
$\texttt{MOV EAX, EBX}$	$\texttt{EAX} \leftarrow \texttt{EBX}$
$\texttt{MOV EAX, DS : [EBX]}$	$\texttt{EAX} \leftarrow M[\texttt{DS, EBX}]$
$\texttt{MOV DS : [EBX], EAX}$	$M[\texttt{DS, EBX}] \leftarrow \texttt{EAX}$
$\texttt{MOV EAX, SS : [ESP + n]}$	$r \leftarrow M[\texttt{SS}, \texttt{sp} + n]$
$\texttt{MOV SS : [ESP + n], EAX}$	$M[\texttt{SS}, \texttt{sp} + n] \leftarrow r$
$\texttt{JE}\ n$	$\texttt{cond x86EQ(ZF)}, n$
$\texttt{JGE}\ n$	$\texttt{cond x86GE(SF, OF)}, n$
$\texttt{JMP}\ n$	$\texttt{jump}\ n$
$\texttt{CALL}\ F$	$\texttt{ra} \leftarrow \texttt{pc} + 3$ $\texttt{sp} \leftarrow \texttt{sp} - 4$ $M[\texttt{SS}, \texttt{sp} + 0] \leftarrow \texttt{ra}$ $\texttt{call}\ F$
$\texttt{RET}\ n$	$\texttt{ra} \leftarrow M[\texttt{SS}, \texttt{sp} + 0]$ $\texttt{sp} \leftarrow \texttt{sp} + (4 + 4 * n)$ $\texttt{ret}$
$\texttt{ADD ESP}, n$	$\texttt{sp} \leftarrow \texttt{sp} + n$ $t_1 \leftarrow n$ $\texttt{ZF} \leftarrow \texttt{sp x86ADD.ZF}\ t_1$ $\texttt{SF} \leftarrow \texttt{sp x86ADD.SF}\ t_1$ $\texttt{OF} \leftarrow \texttt{sp x86ADD.OF}\ t_1$
$\texttt{PUSH EAX}$	$\texttt{sp} \leftarrow \texttt{sp} - 4$ $M[\texttt{SS}, \texttt{sp} + 0] \leftarrow \texttt{EAX}$
$\texttt{POP EAX}$	$\texttt{EAX} \leftarrow M[\texttt{SS}, \texttt{sp} + 0]$ $\texttt{sp} \leftarrow \texttt{sp} + 4$

Figure 3.5: The translation table for Intel x86.

for a composite address. Thus, $M[\texttt{DS}, \texttt{EAX}]$ refers to the memory address that is at offset EAX from the segment whose selector is in DS. Similarly, the *SafeRd*, *SafeWr* and *Stack* relations are extended with an extra segment argument. An additional restriction is that the stack addresses must be aligned on a 32-bit boundary:

$$Stack(a) \quad \supset \quad a \bmod 4 = 0$$

A fragment of the translation table for the Intel x86 architecture is shown in Figure 3.5. The translation of the subtraction function contains code to also set the flag registers that are implicitly affected by the instruction. For this purpose I introduce not only the operator `x86SUB` that denotes the main result of the subtraction but also the operators `x86SUB.ZF`, `x86SUB.SF` and `x86SUB.OF` that denote the side-effects of the subtraction on the flag registers. In the x86 architecture the conditional jump operations examine the flag registers. For the purpose of the "jump greater or equal" instruction `JGE` I had to introduce binary conditional operators.

Of particular interest in the x86 translation are the function call and the stack manipulation functions. In the case of a function call the return address is computed to point immediately after the `SAL` call instruction, and then it is saved on the top of the stack. The stack is also advanced by four bytes (the size of a stack element on x86 is 32-bits). The return function on x86 extracts its return address from the top of the stack and also pops from the stack a specified number of arguments. Adding a constant to the stack pointer is translated to `SAL` as a stack pointer advance instruction followed by instructions meant to model the potential effects of the addition on the flag registers. In most disciplined programs, the resulting values of the flags would not be used but we need to have these instructions to prevent malicious programs to make assumptions about the contents of the flags. Finally, the `PUSH` and `POP` instructions access the stack and have side-effects on the stack pointer but not on the flag registers.

In the x86 architecture each instruction has many variants, depending on how the operands are identified. Figure 3.5 shows only the simplest variant of each instruction, when the operand is a 32-bit register. The x86 architecture allows certain instructions to access 8-bit and 16-bit fragments of registers, using the names shown in Figure 3.6. In addition, most x86 instructions allow operands that reside in memory. For each such operand addressing mode, there is a standard sequence of `SAL` instructions that obtains the operand in a register t. These sequences of instructions are shown in Figure 3.7. In these definitions, I assume that the operand is a source of the computation. If it is a destination then the memory reads are changed to memory writes and the "`GET`" extraction operators are changed to "`SET`" updating operators. For example, to denote the updating of the register `AX` with

31 16	15 8	7 0	16-bit	32-bit
	AH	AL	AX	EAX
	BH	BL	BX	EBX
	CH	CL	CX	ECX
	DH	DL	DX	EDX
	BP			EBP
	SI			ESI
	DI			EDI

Figure 3.6: Register aliasing in the Intel x86 architecture.

Addressing mode	Example	SAL translation
Register	EAX	$t \leftarrow$ EAX
Register (X)	AX	$t \leftarrow$ x86GETX EAX
Register (L)	AL	$t \leftarrow$ x86GETL EAX
Register (H)	AH	$t \leftarrow$ x86GETH EAX
Immediate	n	$t \leftarrow n$
Index	DS : [EBX]	$t \leftarrow M[\mathtt{DS}, \mathtt{EBX}]$
Index + Displacement	ES : [EBX + n]	$t_1 \leftarrow n$ $t_1 \leftarrow t_1$ x86ADD EBX $t \leftarrow M[\mathtt{ES}, t_1]$
Base + Index	DS : [EBX + ESI]	$t_1 \leftarrow$ EBX x86ADD ESI $t \leftarrow M[\mathtt{DS}, t_1]$
Base + Index + Displacement	DS : [EBX + ESI + n]	$t_1 \leftarrow$ EBX x86ADD ESI $t_2 \leftarrow n$ $t_1 \leftarrow t_1$ x86ADD t_2 $t \leftarrow M[\mathtt{DS}, t_1]$

Figure 3.7: Addressing modes in the Intel x86 architecture, shown as the sequence of SAL instructions required to compute the operand in a temporary register t. These translation schemes assume that the operand is a source for the computation.

$$
\begin{array}{rcl}
\underline{\mathtt{x86ADD}}(v_1, v_2) & = & (v_1 + v_2 + 2^{31}) \bmod 2^{32} - 2^{31} \\
\underline{\mathtt{x86SUB}}(v_1, v_2) & = & (v_1 - v_2 + 2^{31}) \bmod 2^{32} - 2^{31} \\
\underline{\mathtt{x86SUB.ZF}}(v_1, v_2) & = & \mathtt{if}\ (v_1 - v_2 + 2^{31}) \bmod 2^{32} = 2^{31}\ \mathtt{then}\ 1\ \mathtt{else}\ 0 \\
\underline{\mathtt{x86SUB.SF}}(v_1, v_2) & = & \mathtt{if}\ (v_1 - v_2 + 2^{31}) \bmod 2^{32} < 2^{31}\ \mathtt{then}\ 1\ \mathtt{else}\ 0 \\
\underline{\mathtt{x86SUB.OF}}(v_1, v_2) & = & \mathtt{if}\ (v_1 - v_2 + 2^{31}) \bmod 2^{32} \neq v_1 - v_2 + 2^{31}\ \mathtt{then}\ 1\ \mathtt{else}\ 0 \\
\underline{\mathtt{x86EQ}}(f) & \text{iff} & f = 1 \\
\underline{\mathtt{x86GE}}(sf, of) & \text{iff} & sf = of \\
\underline{\mathtt{x86GETX}}(v) & = & v \bmod 2^{16} \\
\underline{\mathtt{x86GETL}}(v) & = & v \bmod 2^{8} \\
\underline{\mathtt{x86GETH}}(v) & = & (v \operatorname{div} 2^8) \bmod 2^8 \\
\underline{\mathtt{x86SETX}}(v, x) & = & (v - v \bmod 2^{16}) + (x \bmod 2^{16}) \\
\underline{\mathtt{x86SETL}}(v, x) & = & (v - v \bmod 2^{8}) + (x \bmod 2^{8}) \\
\underline{\mathtt{x86SETH}}(v, x) & = & (v - (v \operatorname{div} 2^8) \bmod 2^8) + (x \bmod 2^8) * 2^8
\end{array}
$$

Figure 3.8: The definitions of a few SAL operators introduced during the translation to SAL. In these definitions the range of the functions is $\mathcal{U}^b = \{x \in \mathbb{Z} | -2^{31} \leq x \leq 2^{31} - 1\}$ and the division and modular operations have their usual meanings for integers.

the contents of register t_1, we write "$\texttt{EAX} \leftarrow \texttt{SETX}(\texttt{EAX}, t_1)$".

To model the extraction and updating of register fragments I introduce a set of unary and binary operators. The mathematical definitions for these operators together with those of operators introduced in Figure 3.5 are shown in Figure 3.8.

3.4 Discussion

A safety policy can be viewed as consisting of a set of conditions restricting the actions that an agent might perform. Examples of actions that might be restricted are the function calls and returns and the executions of certain instructions such as the memory referencing instructions. Such a safety policy can be formalized by defining an interpreter that, as it executes the agent code, checks that the required conditions are met when a relevant action is executed. Section 3.2 defines just such an interpreter for a generic assembly language and Section 3.3 show how the definition can be adapted to deal with real assembly languages.

If we examine carefully the definition of the interpreter shown in Figure 3.2 we notice that it might not serve well its main purpose, which is to provide a realistic execution model for a generic machine language. The reason is that most of the safety checks are not practical to implement. Take for example the relation *SafeWr*; even for a finite $\mathcal{U}^b$ of cardinality $|\mathcal{U}^b|$, the cardinality of the relation could be as high as $|\mathcal{U}^b|^{2+|\mathcal{U}^b|}$. Note, however, that if memory accessibility does not change with the state of the memory, and if data that must be accessible to the agent is arranged in contiguous areas of memory, then it becomes feasible to implement the memory safety checks as simple range checks. The stack checks are always implementable this way because the stack is one contiguous area of memory. If we also ignore all checks related to the expression operators and the system-call safety and partial correctness, then it is quite feasible to implement the safe `SAL` interpreter for this very restricted safety policy. In fact, this is exactly what the technique called Software Fault Isolation (SFI) [WLAG93] does.

However, in general it is not feasible or practical to implement the safety checks. It is thus useful to use static checking to verify that the safety checks are not necessary to be performed at run time for a given program. If we succeed to verify statically that during the execution of the agent all of the safety checks succeed, then we can run the agent on a simplified `SAL` machine that does not implement the checks shown in the last column of Figure 3.2. If we examine the first three columns of the definition of the `SAL` machine we notice also that the call history component of the state can be eliminated. What remains is the description of a realistic machine similar in behavior to many physical machines.[2]

Without proof-carrying code it is quite unlikely that the code receiver has a static analysis

[2] The exception is the stack-overflow check that is required for the call instruction. However, in many physical machines this check can be performed by the hardware memory protection unit if we set *Max* to be smaller than a virtual memory page.

that is able to prove that all of the static checks are redundant. Even if it has one, it must be necessarily conservative in the sense that it fails for some perfectly safe programs. The purpose of the proofs in proof-carrying code is to allow the code producer to use arbitrarily powerful and precise static analyses to prove the safety of the code and to communicate the result of the analyses in a convincing way to the receiver. Thus the code receiver does not fix the static analysis but only the way in which its results are communicated by the code producer, as shown in the next two chapters.

Chapter 4

Enforcing Safety by Proof-Carrying Code

Proof-carrying code enables a code receiver to verify statically that the agent code satisfies a security policy without having to perform complex static analyses. Instead, it is the code producer who performs the static analysis and proves that the code is safe. This proof is then sent to the code receiver as evidence that the code is indeed safe.

A key decision in a concrete implementation of proof-carrying code is what kind of proofs are required to ascertain that a given security policy is met. This depends of course on the security policy. If we want to ensure that there is no undesired information flow, then we can probably reduce the problem to a type-checking problem [HR98, ML97, VS97] and then the proof is a typing derivation for the agent program in the appropriate type system. For the variant of proof-carrying code described in this thesis, the security policy has only a safety component and is expressed by means of a series of safety checks that a fictitious interpreter for `SAL` would perform. The proofs accompanying the code in this instance of PCC guarantee that a `SAL` translation of the agent, executed on the safe `SAL` interpreter of Section 3.2 does not fail any of the safety checks. The existence of such a proof guarantees that the agent can be executed just as safely without performing the safety checks.

For the purpose of verifying that the safety checks of the `SAL` interpreter are always satisfied for a given agent, I introduce another evaluator for `SAL` whose purpose is to execute the `SAL` program symbolically and to collect a symbolic representation of all of the safety checks that would have to be performed. The resulting collection is called the *verification condition* and this kind of interpreter is a *verification-condition generator*, or VCGen [Kin71]. The language in which the safety checks are expressed in the verification condition is that of predicate logic, so that the proofs accompanying the code are derivations in the logic. Previous experience with PCC suggests that a good starting choice for a logic is an extension of first-order predicate logic with equality and array variables. In this setup, the whole verification condition is a formula (predicate) in this logic.

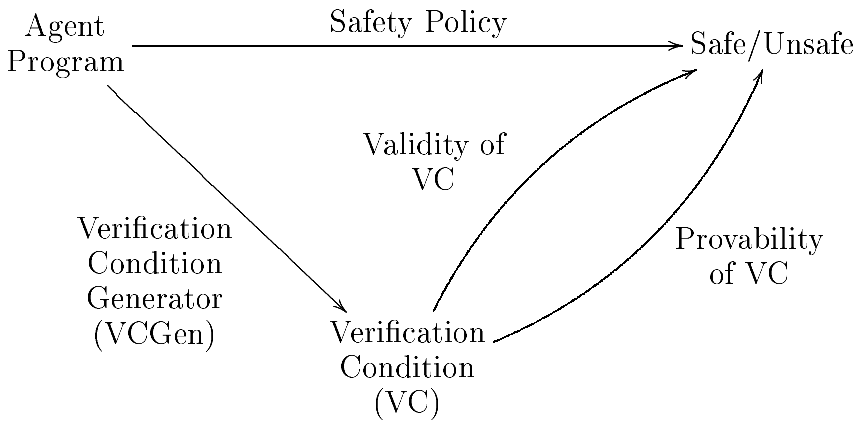


Figure 4.1: The relationship between the safety policy, the verification-condition generator, validity of verification conditions and provability of verification conditions.

The suggested setup for proof-carrying code is depicted in Figure 4.1. At the top of the picture we have the reference criterion for safety, that is, the safety policy or, more concretely, the safe `SAL` interpreter. At the bottom, we have the proof-carrying code method for enforcing safety, consisting of a verification-condition generator followed by a verification of validity of the verification condition produced. The main benefit that we get from proof-carrying code in this setup is that we can run the code safely without having to implement the safe `SAL` interpreter. This is a serious concern because, as discussed in the previous chapter, it is not always possible to implement at run time all of the checks that a safety policy might require. Furthermore, note that the safety policy defined by means of a safe interpreter can tell that a program is unsafe only after executing it and discovering a safety violation. The PCC method, on the other hand can tell statically whether a program is safe. However, the PCC method is conservative, meaning that it might reject perfectly safe programs.

In the next section I describe the syntax of the logic and the precise relationship between logical formulas and the various safety checks that the `SAL` interpreter performs. Then, in Section 4.2 I describe the verification-condition generator that extracts a logical formula (the verification condition) from the body of a function. VCGen is only interesting if it captures all of the safety checks that the `SAL` interpreter would perform. Technically, this means that the diagram of Figure 4.1 commutes through the "validity" path, which is stated formally in at the end of Section 4.2 and proved in Appendix A. Checking validity of the verification condition directly is not practical. Instead, PCC uses an indirect method for checking validity by means of provability. For this purpose I introduce in Section 4.3 a set of axioms and inference rules that can be used to prove formulas and then I show that this set of rules is sound, meaning that provability guarantees validity.

4.1 The Logic

The logic is the symbolic language that is used for expressing the safety checks in a symbolic form as part of the verification conditions, and thus is the bridge between the semantics of the agent program and the symbolic proofs of safety. The general ideas of proof-carrying code can be applied in the context of any logic that is adequate for expressing the required safety policy. I have found that first-order predicate logic with equality and array variables is sufficiently powerful for many practical safety policies. For the purposes of this dissertation, I will describe the implementation of PCC in the context of this restricted logic. However, during the presentation, and mainly within the various correctness proofs, I will point out the generic properties that are required of any PCC logic. This way, the correctness proofs serve not only as an assurance argument for the present instantiation of PCC but also as a guide for the designer who wants to extend the logic to accommodate more demanding safety policies.

4.1.1 Syntax

The logic, as a symbolic language, is defined here by means of syntactic rules for the formation of formulas and a validity function that gives meaning to the logical formulas as relations in the universe of values used for the target machine. It is through this validity function that the logical formulas that are produced by VCGen assume the same meaning as the safety checks performed by the `SAL` interpreter.

The syntactic rules for our fragment of first-order logic with equality and array variables are shown in Figure 4.2. The main syntactic components are the predicates, the base expressions and the store expressions. The base expressions are meant to denote values of the base universe $\mathcal{U}^b$, that is, values that fit in a register or in a memory word. The role of the store expressions is to denote values from the universe of stores $\mathcal{U}^s$, or equivalently the states of the memory in the target machine. Associated with the store expressions we have two constructors, one corresponding to reading from the memory and the other corresponding to writing to the memory. If E^s is an expression that denotes the current memory state then the expression "$\texttt{sel}(E^s, E^b)$" denotes the contents of the memory location whose address is denoted by E^b. To express the new memory state after storing the value denoted by E^b_v to the address E^b_a in memory state E^s, we write "$\texttt{upd}(E^s, E^b_a, E^b_v)$". The base constants ($c^b \in \mathcal{U}^b$) and the store constants ($c^s \in \mathcal{U}^s$) are required only for technical reasons.

Note that implicit in Figure 4.2 are the typing rules for base expressions, store expressions and predicates. Because of the two kinds of expressions, I introduce two versions of universal quantification, equality and disequality. However, to simplify the presentation I will drop the typing superscripts whenever there is no possibility of confusion. For example, I write "$E_1 = E_2$" to denote a comparison of a base expression with a base expression or a comparison of a store expression with a store expression.

Generic fragment:

Predicates:	P	$::=$	$\texttt{true} \mid P_1 \wedge P_2 \mid P_1 \supset P_2 \mid \forall x^b.P \mid \forall x^s.P$
			$\mid E_1^b = E_2^b \mid E_1^b \neq E_2^b \mid E_1^s = E_2^s \mid E_1^s \neq E_2^s$
Base expressions:	E^b	$::=$	$x^b \mid c^b \mid \texttt{sel}(E^s, E^b)$
Store expressions:	E^s	$::=$	$x^s \mid c^s \mid \texttt{upd}(E^s, E_1^b, E_2^b)$

Extensions:

Predicates:	P	$::=$	$\ldots \mid \texttt{safeeop}(E_1^b, E_2^b) \mid \texttt{safecop}(E^b) \mid \texttt{cop}(E^b)$
			$\mid \texttt{notcop}(E^b) \mid \texttt{saferd}(E^s, E^b) \mid \texttt{safewr}(E^s, E_1^b, E_2^b)$
Base expressions:	E^b	$::=$	$\ldots \mid \texttt{eop}(E_1^b, E_2^b) \mid \texttt{offset}(E_1^b, E_2^b)$

Figure 4.2: The syntax of the first-order predicate logic with equality and array variables. The generic fragment is extended with predicate and expression constructors to fit the safety policy and the target architecture.

In addition to the generic constructs of first-order predicate logic with array variables, a typical PCC logic contains predicate and expression constructors to denote symbolically the run-time entities of the `SAL` interpreter. For example, the predicate constructors "`safeeop`" and "`safecop`" are meant to denote the relations that define instruction safety for the "`EOP`" and "`COP`" operators respectively. The "`saferd`" and "`safewr`" constructors are meant to denote the memory safety predicates, and "`cop`" along with "`notcop`" denote the comparison operation implemented by the target machine for the conditional operator "`COP`". On the expression side, the constructor `eop` stands for the mathematical function implemented by the target machine for the operator "`EOP`". The expression constructor "$\texttt{offset}(E_1, E_2)$" is used by VCGen to denote the instruction address that is at offset E_2 in the function whose body starts at E_1.

4.1.2 The Standard Valuation Model

In this section, I define precisely what is the intended meaning of the logical expressions and predicates whose syntax was introduced in the previous section. The *standard model* $\mathcal{M}$ for the logic is a quadruple $<\mathcal{U}^b, \mathcal{U}^s, \mathcal{V}^b, \mathcal{V}^s>$, where $\mathcal{U}^b$ is the universe of base values, $\mathcal{U}^s = \mathcal{U}^b \rightarrow \mathcal{U}^b$ is a universe of store values represented as total functions from $\mathcal{U}^b$ (addresses) to $\mathcal{U}^b$, and $\mathcal{V}^b$ and $\mathcal{V}^s$ are the standard valuation functions for closed base expressions and closed store expressions respectively, defined in Figure 4.3. Recall that $\underline{\texttt{EOP}}$ is the generic mathematical function implemented by the target architecture for the generic `SAL` operator `EOP` (the one that is denoted in logic by the constructor `eop`).

Again, I shall omit the superscript on the universes and valuation functions when there

$$
\begin{array}{lcl}
\mathcal{V}^b(\mathtt{sel}(E^s, E^b)) & = & \mathcal{V}^s(E^s)(\mathcal{V}^b(E^b)) \\
\mathcal{V}^s(\mathtt{upd}(E^s, E^b_1, E^b_2)) & = & \mathcal{V}^s(E^s)[\mathcal{V}^b(E^b_1) \mapsto \mathcal{V}^b(E^b_2)] \\
\mathcal{V}^b(\mathtt{eop}(E^b_1, E^b_2)) & = & \underline{\mathtt{EOP}}(\mathcal{V}^b(E^b_1), \mathcal{V}^b(E^b_2)) \\
\mathcal{V}^b(\mathtt{offset}(E^b_1, E^b_2)) & = & \mathcal{V}^b(E^b_1) + \mathcal{V}^b(E^b_2)
\end{array}
$$

where:

$$
(f[v_1 \mapsto v_2])(a) \quad = \quad \begin{cases} v_2 & \text{if } a = v_1 \\ f(a) & \text{if } v \neq v_1 \end{cases}
$$

Figure 4.3: The definition of the standard valuation function.

$$
\begin{array}{lll}
\models \mathtt{true} & & \\
\models P_1 \wedge P_2 & \text{iff} & \models P_1 \text{ and } \models P_2 \\
\models P_1 \supset P_2 & \text{iff} & \models P_2 \text{ whenever } \models P_1 \\
\models \forall x.P & \text{iff} & \models [^v\!/\!_x]P \text{ for all } v \in \mathcal{U} \\
\models E_1 = E_2 & \text{iff} & \mathcal{V}(E_1) = \mathcal{V}(E_2) \\
\models E_1 \neq E_2 & \text{iff} & \mathcal{V}(E_1) \neq \mathcal{V}(E_2) \\
\models \mathtt{safeeop}(E_1, E_2) & \text{iff} & \mathit{SafeEOP}(\mathcal{V}(E_1), \mathcal{V}(E_2)) \\
\models \mathtt{safecop}(E) & \text{iff} & \mathit{SafeCOP}(\mathcal{V}(E)) \\
\models \mathtt{cop}(E) & \text{iff} & \underline{\mathtt{COP}}(\mathcal{V}(E)) \\
\models \mathtt{notcop}(E) & \text{iff} & \neg\underline{\mathtt{COP}}(\mathcal{V}(E)) \\
\models \mathtt{saferd}(E^s, E_1) & \text{iff} & \mathit{SafeRd}(\mathcal{V}(E^s), \mathcal{V}(E_1)) \\
\models \mathtt{safewr}(E^s, E_1, E_2) & \text{iff} & \mathit{SafeWr}(\mathcal{V}(E^s), \mathcal{V}(E_1), \mathcal{V}(E_2))
\end{array}
$$

Figure 4.4: The definition of the validity of a closed predicate in the standard model. The equality and disequality symbols are overloaded in this definition. On the left side they denote predicate constructors in logic. On the right side they denote the equality and disequality predicates on $\mathcal{U} \times \mathcal{U}$.

is no possibility of confusion. I shall use the variables u and v (appropriately superscripted when necessary) to refer to individual values in the universes. It is important to note that because of the typing rules for expressions and predicates, it is guaranteed that the valuation functions are total and that $\mathcal{V}^b(E^b) \in \mathcal{U}^b$ and that $\mathcal{V}^s(E^s) \in \mathcal{U}^s$.

Figure 4.4 contains the formal definition of validity of a predicate, in the form of a judgment $\mathcal{M} \models P$. Because, the model $\mathcal{M}$ is the same standard model throughout this thesis, I henceforth abbreviate the validity judgment as $\models P$.

4.2 The Verification-Condition Generator

The verification-condition generator (VCGen) is presented here as a symbolic evaluator for `SAL` programs. The result of the evaluation is the verification condition, which is a formula in the logic that I described in the previous section. The main components of the verification condition are the symbolic counterparts of the safety checks that are mandated by the safety policy. The verification condition also contains information about the control flow in the program so that it can express precisely, for each safety check, on which computation paths it is performed. The concept of verification-condition generation was introduced by Floyd and King [Kin71, KF72] and it appears extensively in work related to formal program verification for higher-level languages [BM81, Det96, Dij75, Dij76, GLB75, ILL73].

Before we can discuss the definition of the VCGen, I must introduce some notation and make a series of simplifying assumptions. Following the model of the `SAL` interpreter, VCGen requires that each function have a well-defined stack frame within which all stack accesses must fall. The stack grows towards lower addresses and the reserved register "`sp`" points to the word that is on top of the stack. Before invoking a function G, the caller F puts on top of the stack some or all of the callee's arguments. Assume that G expects Arg_G arguments passed on the stack. The relative location of G's stack frame on the runtime stack is defined by the value of the stack pointer at the moment of its invocation by F, a situation depicted in Figure 4.5. At this moment the words at addresses $\{\mathtt{sp}, \ldots, \mathtt{sp} + Arg_G - 1\}$ contain the Arg_G arguments of G. Assume furthermore that the stack frame of G has a total length of $Local_G$, and thus extends from "$\mathtt{sp} + Arg_G - 1$" downwards to "$\mathtt{sp} + Arg_G - Local_G$". The variable "`so`" is used to denote the offset of the current stack pointer within the current function's stack frame, so that "$\mathtt{sp} + \mathtt{so} - 1$" denotes the stack address of the beginning of the current function's frame. The following relation must hold between the number of arguments, the number of locals and the maximum extra stack that can be allocated by any invocation:

$$0 \leq Arg_F \leq Local_F \leq Max + Arg_F$$

To allow easy reference of the arguments and stack allocated variables, it is convenient to name the stack slots in the frame of G with *local pseudo-registers* $\{l_1, \ldots, l_{Local_G}\}$, so that l_i is an alias for the memory location at address $\mathtt{sp}_0 + Arg_G - i$, where $\mathtt{sp}_0$ is the value of the stack pointer on entry to the function G, sometimes called the frame pointer or the base pointer in some architectures. Note that when using this convention a given stack slot changes its alias names at the time of a function call as shown in Figure 4.5.

We must take special care so that VCGen terminates for all agent programs. Even though VCGen evaluates the program symbolically it must not attempt to evaluate the body of loops iteratively or to follow a recursive function call graph. To address these issues I use loop invariants and function specifications.

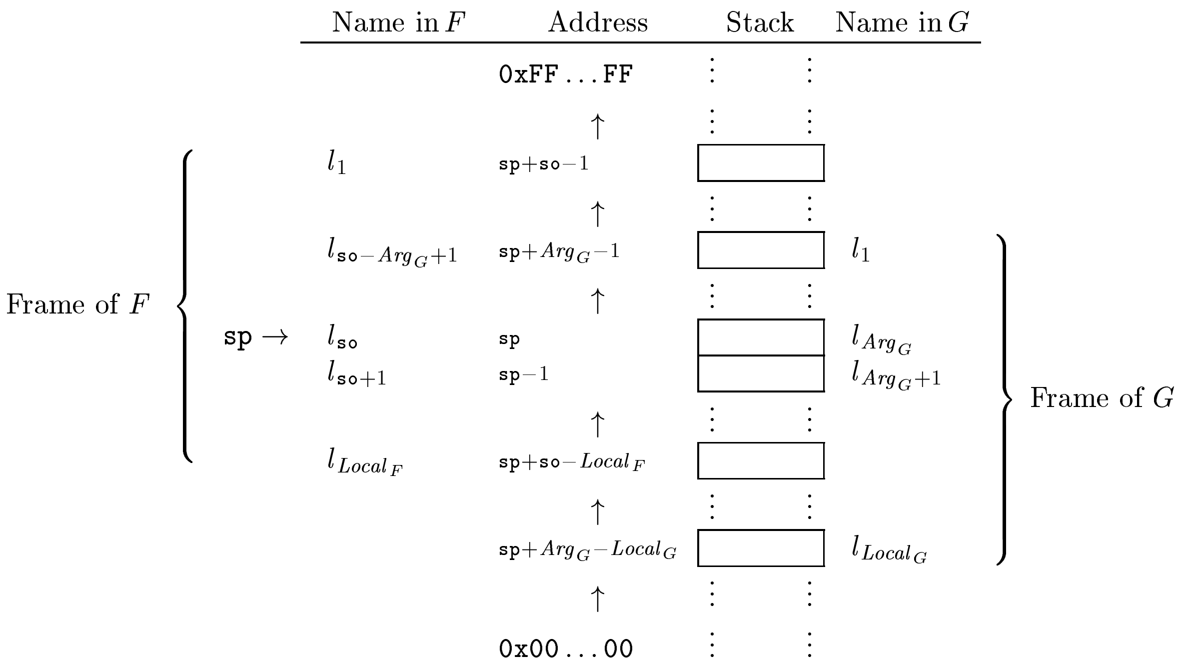


Figure 4.5: Stack management in SAL. The situation depicted occurs at the time when function F calls function G. The value of sp at the time of the call (somewhere in the stack frame of F) marks the position of the last argument of G, and implicitly defines the alignment of G's stack frame relative to F's stack frame. Note the implicit renaming of local pseudo-registers during the call.

The loop invariants are a special case of SAL annotations. They are used to state (by means of a logical formula) the state properties that can be assumed to hold during an arbitrary iteration. At run time they are ignored, but VCGen uses them as part of its operation. The loop invariants are part of the agent program and are therefore untrusted. However, there is an easy way to ensure that they are indeed invariants. If the VCGen verifies that the loop invariant holds on the first entry to the loop and also that it is preserved through one iteration of the loop, then by a simple inductive argument we know that it holds after any finite number of iterations. However, VCGen does not attempt to verify the correctness of the invariants itself. It instead extends the verification condition with logical formulas to the extent that a valid verification condition means that the loop invariants are indeed correct.

The presence of loop invariants means that VCGen needs to scan the loop body only once, when building the inductive case argument. On the other hand, VCGen must require the presence of at least one loop invariant in each loop. An easy and conservative strategy is to require that every backward branch target must be a loop invariant annotation. Referring back to Figure 3.1 on page 44, we can now define the syntax of the annotations as follows:

$$\texttt{Annot} ::= \ldots \quad | \quad \texttt{inv}\ P, n, \mathit{Mod}$$

where P is a predicate denoting the invariant property, n is a natural number that specifies the value of the stack offset at the beginning of the loop and Mod is the set of registers that the loop body might change.

In addition to loops, another possible source of non-termination of VCGen are the recursive function calls. To address this issue, and in fact to modularize the safety checking on a per-function basis, the current implementation of VCGen requires that all functions have a specification. The function specification for a function F is presented in the form of a tuple $<\texttt{Pre}_F, \texttt{Post}_F, CS_F, \mathit{Arg}_F, \mathit{Local}_F>$, where $\texttt{Pre}_F$ and $\texttt{Post}_F$ are formulas representing the precondition and postcondition respectively, CS_F is the set of registers that the function F is required to preserve (the callee-save registers), and Arg_F and Local_F are natural numbers representing the number of arguments and the maximum number of local variables allocated on the stack (including the arguments) that the function F is allowed to have. For the correct operation of VCGen, no local pseudo-registers besides the arguments can appear as free variables in the precondition or postcondition or be mentioned as callee-save registers. This condition makes sense because the values of these variables is not defined outside the function itself.

$$\{l_{\mathit{Arg}_F}, \ldots, l_{\mathit{Local}_F}\} \cap (CS_F \cup FV(\texttt{Pre}_F) \cup FV(\texttt{Post}_F)) = \emptyset \tag{4.1}$$

The specifications of the external functions, either system calls or entry points, must be provided by the code receiver. The specifications of the internal functions are the responsibility of the code producer and must accompany the agent code. Note that the precondition and postcondition relations, which are part of the safety policy definition from Section 3.2, are written using italic characters. To distinguish them from the precondition and postcondition logical formulas that are part of the function specification, I write the latter ones using typewriter characters. The correspondence between the specifications and the safety relations are made precise by the following statements:

Property 4.2 (Correctness of specifications) *For any evaluator state ρ and any function F we must have*

- $\mathit{Pre}_F(\rho)$ *iff* $\models S_{\rho,\mathit{Arg}_F}(\rho, \texttt{Pre}_F)$, *and*
- $\mathit{Post}_F(\rho)$ *iff* $\models S_{\rho,\mathit{Arg}_F}(\rho, \texttt{Post}_F)$.

where the notation $S_{\rho_0,\mathit{Arg}_F}(\rho, r)$ is a generalization of $\rho(r)$ for the case when r is allowed to range over local pseudo-registers in Regs_F, as follows:

$$S_{\rho_0,\mathit{Arg}_F}(\rho, r) \quad = \quad \begin{cases} \rho(r), & \text{if } r \in \{\mathbf{r}_1, \ldots, \mathbf{r}_R, \texttt{ra}, \texttt{mem}\} \\ \rho(\texttt{mem})(\rho_0(\texttt{sp}) + \mathit{Arg}_F - i), & \text{if } r \equiv l_i \in \mathit{Regs}_F \end{cases}$$

Because in most cases it is clear which values should be used for ρ_0 and Arg_F, I often abbreviate $S_{\rho_0,\mathit{Arg}_F}(\rho, r)$ by $S(\rho, r)$.

The verification-condition generator is defined as a symbolic evaluator that scans the SAL code for each function in the agent code and produces the verification condition. The symbolic evaluator maintains several components of state, the most important of which is a mapping σ from registers to expressions. The set of registers for a function F contains the SAL registers, a memory pseudo-register and the local pseudo-registers, as follows:

$$Regs_F = \{\mathbf{r}_1, \ldots, \mathbf{r}_R, \mathtt{ra}, \mathtt{mem}, l_1, \ldots, l_{Local_F}\}$$

The stack pointer register is not among the general purpose registers because VCGen manipulates it directly. Because the set of registers is different for each function, I use the notation σ^F to denote a symbolic register state appropriate for function F. The set of possible symbolic states for function F is $State^F = Regs_F \rightarrow E$, where E is a symbolic expression within the logic.

In addition to the symbolic state of registers, the symbolic evaluator keeps track of the loop invariants that were encountered during the execution. This enables the evaluator to distinguish between the cases when the loop invariant is seen for the first time during the evaluation or for subsequent times. This information is maintained as a sequence of pairs consisting of an instruction address where a loop invariant is encountered and the symbolic state of the execution immediately after processing the invariant. This sequence is denoted by the letter $\mathcal{L}$ and the operation of adding a new pair to the sequence is written as $\mathcal{L}+(i,\sigma)$. To say that i occurs in $\mathcal{L}$ we write $i \in Dom(\mathcal{L})$ and to denote the state mapped to i by $\mathcal{L}$ we write $\mathcal{L}(i)$. I write "$\emptyset$" to denote the empty sequence.

The symbolic evaluator is defined as a state transformation function with seven parameters, written as $SE_{F,\sigma_0^F,i_0^F}(i, \sigma^F, \mathtt{so}, \mathcal{L})$, where F is the function whose body is evaluated, σ_0^F is the symbolic state of registers at the beginning of the execution of function F, i_0^F is a variable standing for the address where the first instruction of F is loaded in memory, i is the offset of the current instruction from the beginning of the function body, σ^F is the current symbolic register state, so is a positive integer (the stack offset) denoting the difference between the top of the frame and the current value of the stack pointer, and $\mathcal{L}$ is the loop map. Of these parameters, the three that are written as subscripts do not vary during the evaluation of a given function and will be omitted most of the time to simplify the presentation. In the same spirit, the superscript F is omitted whenever it can be inferred from the context.

The definition of the symbolic evaluator function is shown in Figure 4.6 and is explained in detail in the rest of this section. But first, it is important to discuss how the symbolic evaluator function is used to produce the verification condition. The verification condition is a conjunction obtained from the results of the symbolic evaluations of all of the functions contained in the agent code. The symbolic evaluation of a function $F \in \Phi^A$ is started in a state obtained by initializing all the registers $Regs_F$ with new logical variables. A new variable i_0^F is also created to stand for the unknown address where the body of the function

F is loaded in memory. The program counter is initialized with zero (the first instruction within the function body), the stack offset variable is initialized with Arg_F (the stack pointer points to the last argument) and the loop map is initialized to empty. Before adding the result of the symbolic evaluation to the verification condition for the whole agent, all newly introduced variables are quantified, making the verification condition a closed predicate. The formal definition of this operation is shown below:

$$VC(\Phi^A) \quad = \quad \bigwedge_{F \in \Phi^A} \forall i_0^F . \forall y_1 \ldots \forall y_k . \sigma_0^F(\texttt{Pre}_F) \supset SE_{F,\sigma_0^F,i_0^F}(0, \sigma_0^F, Arg_F, \emptyset) \tag{4.3}$$

where:

$$\begin{aligned} & i_0^F, y_1, \ldots, y_k \text{ are new variables} \\ Regs_F &= \{r_1, \ldots, r_k\} \\ \sigma_0^F &= [r_1 \mapsto y_1, \ldots, r_k \mapsto y_k] \end{aligned}$$

Returning to the description of the symbolic evaluator function shown in Figure 4.6, note that as it scans the program, VCGen performs simple checks, shown in the second column of Figure 4.6. The most frequent check is that control does not fall out of a function, shown as "$i{+}{+} \in Dom(F)$" in the definition. If any of the checks performed by VCGen fail, the program is rejected right away.

The definition of the symbolic evaluator is recursive, and the symbolic evaluator terminates when it reaches the return instruction or when it reaches a loop invariant for the second time. To ensure that each loop contains at least one invariant, VCGen verifies that the target of each backward branch or jump is an invariant instruction.

In the case of a simple register move, VCGen verifies that the current instruction is not the last one in F. Then, it copies the symbolic value of the source register into the target register and continues the symbolic evaluation with the following instruction.

In the case of a binary operation, the verification condition is extended with the additional predicate $\texttt{safeeop}(\sigma(r'), \sigma(r''))$, the symbolic value of the target register is set to $\texttt{eop}(\sigma(r'), \sigma(r''))$, and the symbolic execution continues with the following instruction. The intuition behind the extended verification condition is that a valid verification condition must ensure that the actual operands of `EOP` fall in the safe domain of the operator. This intuition relies on the known meaning within logic of the constructors "`eop`" and "`safeeop`", as shown in Figures 4.3 and 4.4.

In the case of a jump instruction the symbolic evaluator verifies that the target instruction is within the current function's body. Furthermore, if this is a backward branch the evaluator verifies that the target is an invariant annotation. Then, the symbolic evaluator continues with the target instruction.

The case of a conditional branch is similar to that of a jump instruction except that the safety predicate for the conditional operator is appended to the verification condition just as for expression operators. Also, because the execution is symbolic, the evaluator cannot,

F_i	Check	Verification condition
$r \leftarrow r'$	$i{+}{+} \in \mathit{Dom}(F)$	$\mathit{SE}(i{+}{+}, \sigma[r \mapsto \sigma(r')], \texttt{so}, \mathcal{L})$
$r \leftarrow n$	$i{+}{+} \in \mathit{Dom}(F)$	$\mathit{SE}(i{+}{+}, \sigma[r \mapsto n], \texttt{so}, \mathcal{L})$
$r \leftarrow r'$ `EOP` r''	$i{+}{+} \in \mathit{Dom}(F)$	$\texttt{safeeop}(\sigma(r'), \sigma(r''))$ $\wedge\ \mathit{SE}(i{+}{+}, \sigma[r \mapsto \texttt{eop}(\sigma(r'), \sigma(r''))], \texttt{so}, \mathcal{L})$
`jump` n	$n + i{+}{+} \in \mathit{Dom}(F)$ $n < 0 \supset F_{n+i{+}{+}} = \texttt{inv}\ \ldots$	$\mathit{SE}(n + i{+}{+}, \sigma, \texttt{so}, \mathcal{L})$
`cond` $\texttt{COP}(r), n$	$i{+}{+} \in \mathit{Dom}(F)$ $n + i{+}{+} \in \mathit{Dom}(F)$ $n < 0 \supset F_{n+i{+}{+}} = \texttt{inv}\ \ldots$	$\texttt{safecop}(\sigma(r))$ $\wedge\ \texttt{cop}(\sigma(r)) \supset \mathit{SE}(n + i{+}{+}, \sigma, \texttt{so}, \mathcal{L})$ $\wedge\ \texttt{notcop}(\sigma(r)) \supset \mathit{SE}(i{+}{+}, \sigma, \texttt{so}, \mathcal{L})$
$\texttt{ra} \leftarrow \texttt{pc} + n$	$n + i{+}{+} \in \mathit{Dom}(F)$ $i{+}{+} \in \mathit{Dom}(F)$	$\mathit{SE}(i{+}{+}, \sigma[\texttt{ra} \mapsto \texttt{offset}(i_0, n + i{+}{+})], \texttt{so}, \mathcal{L})$
`call` G	$G \in \Phi$ $\sigma(\texttt{ra}) \equiv \texttt{offset}(i_0, i{+}{+})$ $i{+}{+} \in \mathit{Dom}(F)$ $\texttt{so} \geq \mathit{Arg}_G$	$\sigma_1^G(\texttt{Pre}_G)$ $\wedge\ \forall y_1 \ldots y_k . z_{\texttt{so}+1} \ldots z_{\mathit{Local}_F} .$ $\sigma_2^G(\texttt{Post}_G) \supset \mathit{SE}(i{+}{+}, \sigma_2^F, \texttt{so}, \mathcal{L})$ **where:** $\sigma_1^G = \texttt{CopyIn}^G(\sigma^F, \mathit{Arg}_G, \texttt{so})$ $\{r_1, \ldots, r_k\} = \mathit{Regs}_G - \mathit{CS}_G$ $y_1, \ldots, y_k, z_{\texttt{so}+1}, \ldots, z_{\mathit{Local}_F}$ are new variables $\sigma_2^G = \sigma_1^G[r_1 \mapsto y_1, \ldots r_k \mapsto y_k]$ $\sigma_2^F = \texttt{CopyOut}^F(\sigma_2^G, \sigma^F, \mathit{Arg}_G,$ $\texttt{so}, \{z_{\texttt{so}+1}, \ldots, z_{\mathit{Local}_F}\})$
`ret`	$\texttt{so} = \mathit{Arg}_F$	$\sigma(\texttt{Post}_F) \wedge \texttt{CheckEq}(\sigma, \sigma_0, \mathit{CS}_F)$
$r \leftarrow M[r']$	$i{+}{+} \in \mathit{Dom}(F)$	$\texttt{saferd}(\sigma(\texttt{mem}), \sigma(r'))$ $\wedge\ \mathit{SE}(i{+}{+}, \sigma[r \mapsto \texttt{sel}(\sigma(\texttt{mem}), \sigma(r'))], \texttt{so}, \mathcal{L})$
$M[r'] \leftarrow r$	$i{+}{+} \in \mathit{Dom}(F)$	$\texttt{safewr}(\sigma(\texttt{mem}), \sigma(r'), \sigma(r))$ $\wedge\ \mathit{SE}(i{+}{+}, \sigma[\texttt{mem} \mapsto \texttt{upd}(\sigma(\texttt{mem}), \sigma(r'), \sigma(r))], \texttt{so}, \mathcal{L})$
$\texttt{sp} \leftarrow \texttt{sp} + n$	$1 \leq \texttt{so} - n \leq \mathit{Local}_F$ $i{+}{+} \in \mathit{Dom}(F)$	$\mathit{SE}(i{+}{+}, \sigma, \texttt{so} - n, \mathcal{L})$
$r \leftarrow M[\texttt{sp} + n]$	$1 \leq \texttt{so} - n \leq \mathit{Local}_F$ $i{+}{+} \in \mathit{Dom}(F)$	$\mathit{SE}(i{+}{+}, \sigma[r \mapsto \sigma(l_{\texttt{so}-n})], \texttt{so}, \mathcal{L})$
$M[\texttt{sp} + n] \leftarrow r$	$1 \leq \texttt{so} - n \leq \mathit{Local}_F$ $i{+}{+} \in \mathit{Dom}(F)$	$\mathit{SE}(i{+}{+}, \sigma[l_{\texttt{so}-n} \mapsto \sigma(r)], \texttt{so}, \mathcal{L})$
`inv` $P, n, \{r_1, \ldots r_k\}$	$i \notin \mathit{Dom}(\mathcal{L})$ $n = \texttt{so}$ $i{+}{+} \in \mathit{Dom}(F)$	$\sigma(P)$ $\wedge\ \forall y_1 \ldots y_k . \sigma'(P) \supset \mathit{SE}(i{+}{+}, \sigma', \texttt{so}, \mathcal{L} + (i, \sigma'))$ **where:** $y_1, \ldots, y_k$ are new variables $\sigma' = \sigma[r_1 \mapsto y_1, \ldots r_k \mapsto y_k]$
`inv` P, n, Mod	$i \in \mathit{Dom}(\mathcal{L})$ $n = \texttt{so}$ $i{+}{+} \in \mathit{Dom}(F)$	$\sigma(P)$ $\wedge\ \texttt{CheckEq}(\sigma, \mathcal{L}(i), \mathit{Regs}_F - \mathit{Mod})$

Figure 4.6: The definition of the Verification Condition Generator for `SAL`, in the form of a symbolic evaluation function $\mathit{SE}_{F,\sigma_0,i_0}(i, \sigma, \texttt{so}, \mathcal{L})$ defined by cases on the instruction F_i.

in general, discover which branch must be taken. Therefore, it considers both branches (reflected in the definition by the two recursive calls). To provide more information to the proof producer as to the precise conditions under which a branch is taken, VCGen builds two implications whose left sides are predicates that are assumed to hold for each branch. Again these predicates rely on the meaning of the constructors "`cop`" and "`notcop`".

In the case of a return address computation, the symbolic state of the return address register is changed to "$\texttt{offset}(i_0, n + i{+}{+})$", where i_0 is the variable that was reserved to stand for the address of the first instruction in the body of F, and "$n + i{+}{+}$" is the offset from the start of F of the target instruction. The expression constructor `offset` is just the symbolic counterpart of addition of addresses.

The most complicated case is that of the function call instruction, mostly because of the register renaming that must take place twice, once when going into the callee and once when returning. The checks that the symbolic evaluator performs in this case ensure (1) that G is a function known in the system and thus with a known specification, (2) that the symbolic value of the return address register points to the instruction immediately following the call, (3) that this is not the last instruction in F so that when execution resumes it does not fall out of the body of F, and (4) that there are enough arguments on the stack within the current stack frame. The latter check ensures that VCGen does not have to keep track of the contents of more than one stack frame at a time.

If all of these checks succeed, the verification condition is extended with the callee's precondition $\texttt{Pre}_G$, to ensure that it holds on function entry. The symbolic evaluator abstracts the execution of G and continues the evaluation with the following instruction, while assuming that the postcondition $\texttt{Post}_G$ holds on return. What is tricky about the case of a call are the symbolic state manipulations. There are two reasons for these manipulations. First, the function G might change some of the registers and this has to be modeled by setting these registers to new variables that are quantified. This ensures that nothing can be assumed about their values. Second, the local pseudo-register renaming is modeled as a copy-in/copy-out. This is done with the helper functions $\texttt{CopyIn}^G$ and $\texttt{CopyOut}^F$, whose definition is shown in Figure 4.7 and explained below with reference to the stack layout of Figure 4.5.

The "$\texttt{CopyIn}^G(\sigma^F, Arg_G, \texttt{so})$" function creates a symbolic state appropriate for G from the symbolic state σ^F of F at the moment when it invokes G. At this moment the stack offset is "`so`" and there are Arg_G arguments on the top of the stack. $\texttt{CopyIn}^G$ copies the arguments from σ^F (where it refers to them using the local pseudo-registers of F) to the new state, where they would be referred using the local pseudo-registers $l_1, \ldots, l_{Arg_G}$ of G. The other local pseudo-registers of G are initialized with an arbitrary value (zero in this case). However, because of Property 4.1, these registers cannot appear in the precondition of G nor in the callee-save set CS_G and thus, it does not matter what value is chosen for them. Finally, the `SAL` registers are copied directly to the new state. It is the state obtained by the application of `CopyIn` that is used to instantiate the precondition $\texttt{Pre}_G$ because the

$$\texttt{CopyIn}^G(\sigma^F, Arg_G, \texttt{so})(r) = \begin{cases} \sigma^F(l_{\texttt{so}-Arg_G+i}) & \text{if } r \in \{l_i | 1 \leq i \leq Arg_G\} \\ 0 & \text{if } r \in \{l_i | Arg_G < i \leq Local_G\} \\ \sigma^F(r) & \text{if } r \in \{\mathbf{r}_1, \dots, \mathbf{r}_R, \texttt{ra}, \texttt{mem}\} \end{cases}$$

$$\begin{aligned}\texttt{CopyOut}^F(\sigma_2^G, \sigma_F, Arg_G, \texttt{so},& \\ \{z_{\texttt{so}+1}, \dots, z_{Local_F}\})(r)&\end{aligned} = \begin{cases} \sigma^F(r) & \text{if } r \in \{l_i | 1 \leq i < \texttt{so} - Arg_G + 1\} \\ \sigma_2^G(l_{i-\texttt{so}+Arg_G}) & \text{if } r \in \{l_i | \texttt{so} - Arg_G + 1 \leq i \leq \texttt{so}\} \\ z_i & \text{if } r \in \{l_i | \texttt{so} < i \leq Local_F\} \\ \sigma_2^G(r) & \text{if } r \in \{\mathbf{r}_1, \dots, \mathbf{r}_R, \texttt{ra}, \texttt{mem}\} \end{cases}$$

$$\texttt{CheckEq}(\sigma, \sigma', CS) = \bigwedge_{r \in CS} \sigma(r) = \sigma'(r)$$

Figure 4.7: Additional helper definitions for the symbolic evaluator of Figure 4.6.

precondition is written from the point of view of G.

The next step in the processing of the call to G is to generate new variables for each of the registers that are not declared as saved by G and thus potentially modified during the call. These variables stand for the new unknown values of these registers upon return from G. The resulting state σ_2^G is used to instantiate the postcondition $\texttt{Post}_G$, again because it is written from the point of view of G. But the evaluation resumes in the context of F and thus the state σ_2^G must be changed to fit the new names of stack slots. This is accomplished through a call to $\texttt{CopyOut}^F$.

The "$\texttt{CopyOut}^F(\sigma_2^G, \sigma^F, Arg_G, \texttt{so}, \{z_{\texttt{so}+1}, \dots, z_{Local_F}\})$" function, defined in Figure 4.7, creates a symbolic state appropriate for the caller F given the state from the point of view of the callee G and the state of F previous to the call. This state transformation preserves all `SAL` registers and those local pseudo-registers of σ^F that are located on the stack above (at higher addresses) the frame of G. Then, the local registers of F that were set to contain the arguments of G are copied back as G might have changed them. Finally, the local registers of F that are located on the stack below the arguments of G, in an area that G might have used, are initialized with the new variables z_i to model the fact that their contents is unknown. This completes the discussion of the "`call`" instruction.

Note that VCGen as described here does not support computer-function call or function pointers and thus, it limits drastically the implementation of higher-order languages and object-oriented languages with dynamic method lookup. This restriction can be relaxed by noting that VCGen does not actually need to know the exact function that is called but just its specification. Thus, we can still allow function pointers when all of the functions that could be invoked by a particular computed-function call share the specifications. In this case we also need to be able to declare a function-pointer variable to have a given specification.

Next is the case of the function return. In this case the postcondition $\texttt{Post}_F$ is asserted in the current state and a series of equality predicates are generated to verify that all registers that F is supposed to preserve are indeed preserved. This set of equalities is generated using the function `CheckEq` that is passed the current state and the symbolic state of registers on function entry σ_0.

In the case of a memory read or a memory write the appropriate safety predicate is added to the verification condition, the state is changed appropriately and the execution continues with the next instruction. In the case of a memory read the target register is changed, while in the case of a memory write the memory pseudo-register is changed.

In the case of a stack register advance instruction, the stack offset component of the state is decremented but only if the new value has a legal value within the frame of the current function F. The instructions for reading from and writing to the stack are modeled as register moves between the target register and a local pseudo-register.

Finally, there is the case of invariant annotations. First, the symbolic evaluator verifies that the stack offset is in the position specified by the annotation. Then, the operation of the evaluator depends on whether the annotation is seen for the first time ($i \notin Dom(\mathcal{L})$) or it has been seen already. If the invariant is seen for the first time, the invariant predicate is instantiated according to the current state and is added to the verification condition. The intuition is that a valid verification condition must guarantee that the invariant holds on entry to the loop. Then, the registers that the loop is allowed to modify are initialized with new variables and the execution is resumed using the new state. Note also that the loop map is extended with the state at the beginning of the execution of the loop body. The purpose of this operation is to be able to detect that a loop invariant was seen and to verify that only the registers that are declared as modified were modified. This is shown in the last line of the definition.

This concludes the definition of the verification-condition generator. VCGen is quite complex and an error in its implementation could lead to security holes that can be exploited by malicious code producers to subvert the safety checks. While verifying formally the correctness of a particular implementation of VCGen is beyond the scope of this work, it is important to prove that at least the algorithm is correct. The verification-condition generator is correct if it produces a valid verification condition (for the definition of validity of Section 4.1.2) only for an agent that satisfies the safety policy (for the definition of the safety policy from Section 3.2). Because the agent functions might invoke system calls, we also need to assume that the system calls are safe. This is stated formally as Theorem 4.4, using the notion of safety as defined in Definition 3.5.

Theorem 4.4 (Soundness of VCGen) *If all system calls are safe, i.e., $Safe(\Phi^S)$, and if the agent's verification condition is valid, i.e., $\models VC(\Phi^A)$, then all functions in the system are safe, i.e., $Safe(\Phi)$.*

This soundness condition is the keystone of the whole proof-carrying code infrastructure because it bridges the semantics of the agent code with the safety policy and with the logic. The proof of Theorem 4.4 is rather technical and in order to keep the main body of the thesis more accessible I give the formal proof in Appendix A. The proof should be of great interest to the reader desiring a precise understanding of the details of the verification-condition generator. The proof should also be of interest to those intending to extend the symbolic evaluator with more features.

Informally, the proof of the soundness theorem is by induction on the length of the execution. At each step (in any reachable state) we show that the execution is either immediately after the end of the invocation or else it can make further progress. During the induction we must collect and propagate the information that the execution was initiated in a state satisfying the precondition and that the verification condition is valid. This is customarily done by means of an induction hypothesis. In our case the induction hypothesis for an execution state $<\!<\!F, i\!>, \rho, \mathcal{H}_0 + \rho_0\!>$ is that there exists a related state of the symbolic evaluator at the same point in function F. Let this state be $SE_{F,\sigma_0,i_0}(i, \sigma, \mathtt{so}, \mathcal{L})$. The key property of the run-time and symbolic states is that they are related, in a complicated sense that is stated formally later in this section. We need to prove that this relation is an invariant of the execution, or in other words that we can simulate the execution of the interpreter with a related symbolic evaluator.

4.3 The Axiomatization of the Logic

I have shown in the previous section that instead of implementing the safe `SAL` interpreter of Section 3.2, we can instead check the code statically by first running the verification-condition generator and then verifying the validity of the resulting verification condition. The major gain of this alternative strategy is that the whole checking for safety is done statically, without having to pay run-time penalties. However the potentially prohibitive cost of verifying the relations *SafeRd*, *SafeWr* and others still exists. In fact, it is exacerbated by the universal quantifications that require the checks to be done for a large number of cases. This is not surprising given that we want to verify statically that *all* possible executions are safe.

While it might seem that we have made the problem even worse through the alternative strategy for checking safety, this is not so because there are other ways to verify the validity of predicates without having to compute the valuations explicitly. The purpose of this section is to introduce a framework of symbolic computation on formulas so that the validity of formulas can be verified symbolically. This framework consists of a set of axioms and inference rules that allow us to prove formulas from other formulas believed to be valid.

An axiomatic system for a logic is a set of derivation rules that can be used to derive the validity of a formula from other formulas that are assumed to be valid. A predicate P is said to have a derivation (written $\triangleright P$) if it can be derived using the rules shown in a natural deduction style in Figure 4.8. Although the rules given here are within the intuitionistic fragment of the logic, this is not strictly necessary. It can be seen from the valuation model for the logic presented in Section 4.1.2 that a classical axiomatization is also allowed.

Before we start discussing the individual rules of Figure 4.8, let us note that we have the obligation to prove that derivability within the logic preserves validity. This is stated below as the *soundness* property of the axiomatic system.

Theorem 4.5 (Soundness) *For any closed predicate P, if $\triangleright P$ then $\models P$.*

PROOF: Each derivation $\triangleright P$ is a finite sequence of uses of axioms and inference rules. The proof of the theorem follows by induction on the structure of the derivation $\triangleright P$ once we prove that each inference rule in the logic is valid. This is stated below as Lemma 4.6. $\square$

Lemma 4.6 (Soundness of an inference rule) *An inference rule with conclusion C and hypotheses H_i $(i = 1..n)$ and with parameters a_j $(j = 1..m)$ is valid if and only if for all $u_j \in \mathcal{U}$, whenever $\models [^{u_1}/_{a_1}, \ldots, ^{u_m}/_{a_m}]H_i$ $(i = 1..n)$ we have $\models [^{u_1}/_{a_1}, \ldots, ^{u_m}/_{a_m}]C$.*

The soundness lemma must be proved for every inference rule in the logic, as it is introduced. To simplify the notation in such proofs we shall use the notation τ to stand for the arbitrary substitution "$[^{u_1}/_{a_1}, \ldots, ^{u_m}/_{a_m}]$" and let $\tau(E)$ be the result of that substitution applied to the expression E.

Returning to the deductive rules of Figure 4.8, they can be classified into four main categories. Firstly, there are the rules of first-order logic, customarily defined as introduction and elimination rules for all of the logical connectives. The introduction rule for implication is hypothetical. In order to prove $P_1 \supset P_2$, we assume that we have a proof of P_1 and from that we derive a proof of P_2. In this case the assumption is named u and there is a side condition requiring u to be used only locally, for the purpose of proving P_2. Similarly, the introduction rule for universal quantification is parametric, in the sense that the a is a fresh parameter that can be used only locally. For these rules there is no need to prove the soundness lemma because that is just the standard argument of soundness for natural deduction.

The second class of rules refers to equality. Equality is defined using the identity and congruence rules. From these rules we can derive the rules of symmetry and transitivity. The rules of case analysis (**case**) and contradiction (**contr**) express the properties of disequality without requiring general purpose negation and disjunction. The proofs of soundness for these inference rule is also omitted because they are standard.

The third class of deductive rules are the rules referring to the array variables in our logic (memory variables). The first rule says that the contents of a memory location that was updated with the value E_2^b is equal to E_2^b. The second rule says that updating a memory

First order logic:

$$\frac{}{\rhd\ \mathtt{true}}\,\mathbf{truei} \qquad \frac{\rhd P_1 \qquad \rhd P_2}{\rhd\ P_1 \wedge P_2}\,\mathbf{andi} \qquad \frac{\rhd P_1 \wedge P_2}{\rhd\ P_1}\,\mathbf{andel} \qquad \frac{\rhd P_1 \wedge P_2}{\rhd\ P_2}\,\mathbf{ander}$$

$$\frac{\begin{array}{c}\overline{\rhd\ P_1}^{\,u}\\ \vdots u\\ \rhd P_2\end{array}}{\rhd\ P_1 \supset P_2}\,\mathbf{impi}^u \qquad \frac{\rhd P_1 \supset P_2 \qquad \rhd P_1}{\rhd\ P_2}\,\mathbf{impe} \qquad \frac{\rhd [a/x]P}{\rhd\ \forall x.P}\,\mathbf{alli}^a \qquad \frac{\rhd \forall x.P}{\rhd\ [E/x]P}\,\mathbf{alle}$$

Equality:

$$\frac{}{\rhd\ E = E}\,\mathbf{eqid} \qquad \frac{\rhd E_1 = E_2 \qquad \rhd [E_1/x]P}{\rhd\ [E_2/x]P}\,\mathbf{congr}$$

$$\frac{\begin{array}{c}\overline{\rhd\ E_1^b = E_2^b}^{\,u}\\ \vdots u\\ \rhd P\end{array} \qquad \begin{array}{c}\overline{\rhd\ E_1^b \neq E_2^b}^{\,v}\\ \vdots v\\ \rhd P\end{array}}{\rhd\ P}\,\mathbf{case}^{u,v} \qquad \frac{\rhd E_1^b = E_2^b \qquad \rhd E_1^b \neq E_2^b}{\rhd\ P}\,\mathbf{contr}$$

Array variables:

$$\frac{}{\rhd\ \mathtt{sel}(\mathtt{upd}(E^s, E_1^b, E_2^b), E_1^b) = E_2^b}\,\mathbf{mc0} \qquad \frac{\rhd E_1^b \neq E_3^b}{\rhd\ \mathtt{sel}(\mathtt{upd}(E^s, E_1^b, E_2^b), E_3^b) = \mathtt{sel}(E^s, E_3^b)}\,\mathbf{mc1}$$

Figure 4.8: The set of deductive rules for the first-order predicate logic with equality and array variables.

location does not change the contents of other memory locations. These rules are most often used in combination with the case analysis rule. These rules are named the McCarthy rules and they have been first introduced in [MP67]. As a representative case for the proof of soundness for these rules, consider the rule **mc1**.

Soundness of the rule mc1. Let $m = \mathcal{V}^s(\tau(E^s))$, $a = \mathcal{V}^b(\tau(E_1^b))$, $v = \mathcal{V}^b(\tau(E_2^b))$ and $a' = \mathcal{V}^b(\tau(E_3^b))$. From the hypothesis we get that $a \neq a'$. By the definition of $\mathcal{V}$ we get that $\mathcal{V}^b(\mathtt{sel}(\mathtt{upd}(\tau(E^s), \tau(E_1^b), \tau(E_2^b)), \tau(E_3^b))) = (m[a \mapsto v])a' = m(a') = \mathcal{V}^b(\mathtt{sel}(\tau(E^s), \tau(E_3^b)))$, hence the conclusion is valid.

A fourth class of rules are often present in PCC logics for the purpose of defining properties of the various custom expression and predicate constructors defined as extensions of the base logic. In the current version of the logic, I do not show any special rules for the constructors related to the safety checks and rely instead on the congruence rule to derive

$$\frac{}{\alpha\texttt{ADDQ}(E_1, 0) = E_1}\alpha\textbf{add0} \qquad \frac{E_1 \geq 0 \quad E_2 \geq 0 \quad \alpha\texttt{GE}(\alpha\texttt{SUBQ}(E_1, E_2), 0)}{E_1 \geq E_2}\alpha\textbf{geq0}$$

$$\frac{\texttt{x86GE}(\texttt{x86SUB.SF}(E_1, E_2), \texttt{x86SUB.OF}(E_1, E_2))}{E_1 \geq E_2}\textbf{x86geq}$$

Figure 4.9: Deductive rules for custom operators introduced when porting the infrastructure to the DEC Alpha and Intel x86.

predicates involving them. For the operators introduced while porting the infrastructure to concrete target architecture it is often useful to specify at least partially the meaning of operators. This can be done through deductive rules, as shown in Figure 4.9. The rule α**add0** says that adding zero is idempotent on the DEC Alpha. The rule α**geq0** says that if the αGE comparison with zero of the result of subtracting two positive values using 'αSUBQ is true, then the two values are in the $\geq$ relation. The surprising part of this rule is that the conclusion is not necessarily true if the two values are not positive. In fact, the conclusion is true only if the operation "$\alpha\texttt{SUBQ}(E_1, E_2)$" does not overflow and a sufficient condition for this is that both E_1 and E_2 be non-negative. On the Intel x86 architecture, subtraction followed by comparison can be used to achieve the expected result even for non-positive input values, as stated in the **x86geq** rule.

As usual, we must give proofs of soundness for the newly introduced rules. I show here only the non-trivial proofs for the α**geq0** and **x86geq** rules.

Soundness of the αgeq0 rule. Let $v_1 = \mathcal{V}(\tau(E_1))$ and $v_2 = \mathcal{V}(\tau(E_2))$. From the first two hypotheses we infer that $v_1 \geq 0$ and $v_2 \geq 0$. Because both v_1 and v_2 are in $\mathcal{U}$ and we obtain the inequalities $0 \leq v_1 \leq 2^{63}-1$ and $-2^{63}+1 \leq -v_2 \leq 0$. By adding these inequalities followed by adding 2^{63} to the result we obtain that $1 \leq v_1 - v_2 + 2^{63} \leq 2^{64} - 1$ and therefore, from the definition of SUBQ, that $\mathcal{V}(\alpha\texttt{SUBQ}(\tau(E_1), \tau(E_2))) = v_1 - v_2$. From here, by using the last hypothesis, it is easy to verify that $v_1 \geq v_2$. Note that there is nothing magic about v_1 and v_2 being positive, just that this is a sufficient condition to ensure that the subtraction does not overflow.

Soundness of the x86geq rule. Let $v_1 = \mathcal{V}(\tau(E_1))$ and $v_2 = \mathcal{V}(\tau(E_2))$ and $\mathit{of} = \underline{\texttt{x86SUB.OF}}(v_1, v_2)$ and $\mathit{sf} = \underline{\texttt{x86SUB.SF}}(v_1, v_2)$. From the hypothesis and the valuation of x86GE we know that $\mathit{sf} = \mathit{of}$. Finally, let $x = v_1 - v_2 + 2^{31}$. Because both v_1 and v_2 are in $\mathcal{U}^b$ we have that $-2^{31} + 1 \leq x \leq 3 * 2^{31} - 1$. We need to show that $x \geq 2^{31}$. We assume the contrary and we try to derive a contradiction. There are two cases:

- If $-2^{31}+1 \leq x < 0$ then $x \bmod 2^{32} = x+2^{32}$. Therefore $\mathit{of} = 1$ and hence $\mathit{sf} = 1$ which means that $x + 2^{32} < 2^{31}$, which in turn contradicts our assumption that $x \geq -2^{31}+1$.

- If $0 \leq x < 2^{31}$ then $x \bmod 2^{32} = x$ and $\mathit{of} = 0$, hence zf is also zero meaning that

$x \geq 2^{31}$ which contradicts again our assumption.

4.4 Discussion

In Chapter 3 I have described and formalized a generic safety policy that is enforceable with the described implementation of proof-carrying code. The essence of the formalization is a the definition of safety (Definition 3.5). Then in Section 4.2 I define a verification-condition generator that scans the agent code and produces a formula in the logic introduced in Section 4.1. Theorem 4.4 states that the given algorithm for verification-condition generation is sound with respect to the definition of safety, in the sense that it produces only valid verification conditions for safe agents. Then, in Section 4.3 I introduce a symbolic framework for deciding the validity of formulas, by means of a set of derivation rules. Theorem 4.5 states that the presented set of derivation rules is guaranteed to derive only valid formulas. Now we can put all of these results together and state formally that the presented algorithm for verification-condition generation along with the axiomatization of the logic is a sound method to verify the safety of agents. This is stated below as the correctness theorem for proof-carrying code.

Theorem 4.7 (Soundness of proof-carrying code) *If all of the system calls are safe, i.e., $Safe(\Phi^S)$, and if the verification condition for the agent functions is provable within the logic, i.e., $\rhd\ VC(\Phi^A)$, then all functions in the system are safe.*

PROOF: Because of the soundness of the axiomatization (Theorem 4.5) we have that $\models VC(\Phi^A)$. Then, we use the soundness of VCGen (Theorem 4.4) and we obtain the desired conclusion.

$\square$

In general, a concrete PCC logic is an extension of the logic that I discussed in this chapter. Typical extensions do not change the set of logical connectives but introduce new expression and predicate constructors. In such cases the proof of soundness of the new logic is a strict extension of the proof shown in this chapter. If a more substantial change to the logic is attempted then the safety policy designer has the obligation to redo the soundness proof from scratch. Hopefully, much of the technical development from this chapter can be reused even in that case.

To help logic designers extend the logic I summarize the general strategy that must be followed:

1. Extend the syntax with new expression or predicate constructors.

2. Define the valuation functions for the newly introduced constructors. If the added constructors are meant to denote machine operations, the valuation functions must correspond to the behavior of the physical machine. The valuation function must be total.

3. Add axioms and inference rules involving the new constructors. This step is not mandatory but, without it, many predicates that contain the new constructors will not be provable.

4. For each newly added derivation rule, extend the proof of the soundness lemma, basically proving that each derivation rule is sound with respect to the standard model.

Chapter 5

Proof Engineering

In Chapter 4 I have shown that verification-condition generation constructs a valid verification condition for an agent program only if the agent meets the safety policy. I have then shown that instead of verifying the validity of the verification condition directly it is enough to find a derivation (i.e., a proof) of the formula using a given system of axioms and inference rules. These are the technical facts that proof-carrying code relies upon. In addition, proof-carrying code exploits the fact that proving a verification condition is more difficult in general than verifying the validity of a proof. This motivates the PCC requirement that the code producer generates the proofs so that the code receiver has only to check their validity. For this to work properly in practice we need a framework for encoding proofs of logical formulas so that they are relatively compact and easy to check. We need a framework and not just one proof checker for a given logic because the set of axioms and inference rules are likely to change frequently, as we adapt PCC to different architectures or safety policies. We would like that such adaptation require few changes to the system, ideally limited to a high-level description of the inference rules in a format resembling their mathematical formulation. Based on these desiderata I summarize below the necessary properties of a successful candidate for a framework for encoding and checking proofs.

- The framework must be able to encode judgments and derivations from a wide variety of logics, including first-order and higher-order logics.

- The implementation of the proof checker must be parameterized by a high-level description of the logic. This allows a unique implementation of the proof checker to be configured easily for all of the logics that can be encoded in the framework.

- The proof checker should perform a directed, one-pass inspection of the proof object, without having to perform proof search. This leads to a simple implementation of the proof checker that is easy to trust and install in extensible systems.

- The proof representation must be compact in order to minimize the resources needed for transmitting, storing and checking proofs.

The above desiderata are important not only for proof-carrying code but for any application where proofs are represented and manipulated explicitly. One such application is a proof-generating theorem prover. A theorem prover that generates an explicit proof object for each successfully proved predicate enables a distrustful user to verify the validity of the proved theorem by checking the proof object. This effectively eliminates the need to trust the soundness of the theorem prover at the relatively small expense of having to trust a much simpler proof checker. The generated proofs and the proof checker are also of great software engineering benefit as they can lead to the timely discovery of soundness bugs that are introduced during development or maintenance of the theorem prover.

The first impulse when designing efficient proof representation and validation algorithms is to specialize them to a given logic or a class of related logics. For example, we might define the representation and validation algorithms by cases, with one case for each inference rule in the logic. This approach has the major disadvantage that a new representation and validation algorithm has to be designed and implemented for each logic. To make matters worse, such proof checking algorithms are rather large for realistic logics. We would prefer instead to use general algorithms that are parameterized by a high-level description of the particular logic of interest.

Instead of such specialized representation and validation algorithms, I choose the Edinburgh Logical Framework (LF) introduced by Harper, Honsell and Plotkin [HHP93] as the starting point in the quest for efficient proof manipulation algorithms, because it scores very high on the first three of the four desirable properties listed above. Edinburgh LF is a very simple variant of λ-calculus with the property that if a predicate is represented as an LF type then a valid proof of that predicate must necessarily be an LF expression of that type. Thus, the simple logic-independent LF type-checking algorithm can be used for checking proofs.

However, LF is not a perfect choice for proof-carrying code because the representation of proofs is unnecessarily large due to a high degree of redundancy. To address this issue, I have extended LF to handle proof representations with missing subterms, and I have also extended the LF type checking algorithm to synthesize the missing parts. In order to keep the resulting proof reconstruction algorithm simple, and in particular to avoid having to search while reconstructing, I impose certain syntactic restrictions on which proof subterms can be missing. Because of these restrictions, it might be the case sometimes that small amounts of redundant information cannot be eliminated from the representation of proofs. However, my practical experience shows that this is the case only for the representation of level-two proofs (proofs of theorems *about* deductive systems, such as soundness or completeness), which are anyway beyond the scope of the proof-carrying code variant described in this dissertation. Experiments with PCC and with a proof-generating theorem prover for first-order logic show that by using the resulting logical framework, which I call *implicit LF* or LF_i, we obtain reductions of more than two orders of magnitude in the size of the proofs and also in the time required for proof checking. Furthermore, these factors become larger for larger proofs.

The rest of this chapter is organized as follows. Section 5.1 describes the encoding of predicates and proofs in the Edinburgh Logical Framework and demonstrates, for a simple example, the effects of redundancy in the representation of proofs. Then, Section 5.2 discusses the LF_i type system as an extension of LF. Because the LF_i type system, unlike the plain LF type system, does not suggest a simple deterministic type-checking algorithm, I show in Section 5.3 an LF_i reconstruction algorithm that is proved (in Appendix B) to be adequate for type checking LF_i proof representations. Then, Section 5.4 describes one algorithm that can be used to convert proofs represented in LF to the more compact LF_i representations. Concluding this chapter is Section 5.5 that shows how to represent efficiently proofs of linear arithmetic in LF_i and Section 5.7 that compares LF_i with other related techniques for representing proofs. The experimental data validating the LF_i algorithms described here is discussed in Section 8.3. An abbreviated form of this chapter appeared before in [NL98b].

5.1 The Edinburgh Logical Framework

The Edinburgh Logical Framework (also referred to as LF) was introduced by Harper, Honsell and Plotkin [HHP93] as a metalanguage for high-level specification of logics. LF provides natural support for the management of binding operators, hypothetical and schematic judgments. For example it captures the convention that expressions that differ only in the names of bound variables are considered identical. Similarly, it allows direct expression of contexts and variable lookup as they arise in a hypothetical and parametric judgment. Consider for example the usual formulation of the implication introduction rule in first-order logic. This rule is hypothetical because the proof of the right-hand side of the implication can use the assumption that the left-hand side holds. However, there is a side condition requiring that this assumption not be used elsewhere in the proof. As we shall see below, LF can represent this side condition in a natural way by representing the assumption as a local variable bound in the proof of the implication. The fact that these techniques are supported directly by the logical framework is a crucial factor for the succinct formalization of proofs.

The LF representation of a logic consists of two stages. The first stage is the representation of the abstract syntax of the logic under consideration. For example, I will show how to represent expressions and predicates of first-order predicate logic in LF. The second stage is the representation of the axiomatization of the logic, and implicitly of the proofs.

The LF type theory is a language with entities of three levels: objects, types and kinds. Types are used to qualify objects and similarly, kinds are used to qualify types. The abstract

$$
\begin{array}{llcl}
\text{Kinds} & K & ::= & \texttt{Type} \mid \Pi x\!:\!A.K \\
\text{Types} & A & ::= & a \mid A\,M \mid \Pi x\!:\!A_1.A_2 \\
\text{Objects} & M & ::= & x \mid c \mid M_1 M_2 \mid \lambda x\!:\!A.M
\end{array}
$$

Figure 5.1: The syntax of Edinburgh LF.

syntax of these entities is shown in Figure 5.1, where $\Pi x\!:\!A.B$ is a dependent function type with x bound in B. In the special case when x does not occur in B, the more familiar notation $A \to B$ can be used. Also, `Type` is the base kind, a is a type constant and c is an object constant.

The encoding of a logic in LF consists of an *LF signature* Σ that contains declarations for a set of LF type and expression constants corresponding to the syntactic formula constructors and to the proof rules. For a more concrete discussion, I describe in this section the representation of the first-order predicate logic with equality and array variables, whose syntax is shown in Figure 4.2 on page 64 and whose axiomatic system was defined in Figure 4.8.

The syntax of the logic is described by the LF signature shown in Figure 5.2. This signature defines an LF type constant for each kind of syntactic entity within the logic. Then, there is an LF constant declaration for each syntactic constructor. The LF type of such a constant describes arity of the constructor, the types of the arguments and the type of the constructed value. The only interesting case that is not straightforward is that of the universal quantification, when a bound LF variable is used to represent the variable bound by the quantification. This higher-order representation trick ensures that predicates that are equivalent up to renaming of bound logical variables are represented by terms that are equivalent up to renaming of local LF variables. Furthermore, now we can use substitution in LF to represent substitution in the object logic.

The LF representation function $\ulcorner \cdot \urcorner$ is defined inductively on the structure of expressions, types and predicates. For example, we have the following two representations:

$$
\begin{array}{rcl}
\ulcorner P \supset (P \wedge P) \urcorner & = & \texttt{imp}\, \ulcorner P \urcorner\, (\texttt{and}\, \ulcorner P \urcorner \ulcorner P \urcorner) \\
\ulcorner \forall x^b.\texttt{safecop}\; x \urcorner & = & \texttt{allb}\, (\lambda x : \texttt{exp}.\texttt{safecop}\; x)
\end{array}
$$

The strategy for representing proofs in LF is to define a type family "`pf`" indexed by predicates. Then, we represent the proof of "P" as an LF expression having type "`pf` P". This representation strategy is called "judgments as types and derivations as objects" and was first used in the work of Harper, Honsell and Plotkin [HHP93]. Note that the dependent types of LF allows us to encode not only that an expression encodes a proof but also whose proof it is.

One can view the axioms and inference rules as proof constructors. This justifies representing the axioms and inference rules similarly to the syntactic constructors, by means of LF constants. The LF signature that constitutes the representation of the axiomatic system

$$
\begin{array}{lcl}
\texttt{exp} & : & \texttt{Type} \\
\texttt{mem} & : & \texttt{Type} \\
\texttt{pred} & : & \texttt{Type} \\
\\
\texttt{sel} & : & \texttt{mem} \to \texttt{exp} \to \texttt{exp} \\
\texttt{upd} & : & \texttt{mem} \to \texttt{exp} \to \texttt{exp} \to \texttt{mem} \\
\texttt{eop} & : & \texttt{exp} \to \texttt{exp} \to \texttt{exp} \\
\texttt{offset} & : & \texttt{exp} \to \texttt{exp} \to \texttt{exp}
\end{array}
$$

(a)

$$
\begin{array}{lcl}
\texttt{true} & : & \texttt{pred} \\
\texttt{and} & : & \texttt{pred} \to \texttt{pred} \to \texttt{pred} \\
\texttt{impl} & : & \texttt{pred} \to \texttt{pred} \to \texttt{pred} \\
\texttt{allb} & : & (\texttt{exp} \to \texttt{pred}) \to \texttt{pred} \\
\texttt{alls} & : & (\texttt{mem} \to \texttt{pred}) \to \texttt{pred} \\
\texttt{eqb} & : & \texttt{exp} \to \texttt{exp} \to \texttt{pred} \\
\texttt{neqb} & : & \texttt{exp} \to \texttt{exp} \to \texttt{pred} \\
\texttt{eqs} & : & \texttt{mem} \to \texttt{mem} \to \texttt{pred} \\
\texttt{neqs} & : & \texttt{mem} \to \texttt{mem} \to \texttt{pred} \\
\texttt{safeeop} & : & \texttt{exp} \to \texttt{exp} \to \texttt{pred} \\
\texttt{safecop} & : & \texttt{exp} \to \texttt{pred} \\
\texttt{cop} & : & \texttt{exp} \to \texttt{pred} \\
\texttt{notcop} & : & \texttt{exp} \to \texttt{pred} \\
\texttt{saferd} & : & \texttt{mem} \to \texttt{exp} \to \texttt{pred} \\
\texttt{safewr} & : & \texttt{mem} \to \texttt{exp} \to \texttt{exp} \to \texttt{pred}
\end{array}
$$

(b)

Figure 5.2: The LF signature Σ that constitutes the representation of the syntax first-order predicate logic with equality and array variables. Both expression (a) and predicate constructors (b) are shown.

presented in Figure 4.8 on page 77 is shown in Figure 5.3. Note how the dependent types of LF can define precisely the meaning of each rule. For example, the declaration of the constant "`andi`" says that, in order to construct the proof of a conjunction of two predicates, one can apply the constant "`andi`" to four arguments, the first two being the two conjuncts and the other two being the proofs of the conjuncts, respectively.

The LF representation function $\ulcorner \cdot \urcorner$ is extended to derivations and is defined recursively on the derivation, as shown in the following examples:

$$
\ulcorner \dfrac{\begin{array}{cc} \mathcal{D}_1 & \mathcal{D}_2 \\ \rhd P_1 & \rhd P_2 \end{array}}{\rhd\ P_1 \wedge P_2} \urcorner = \texttt{andi}\ \ulcorner P_1 \urcorner \ulcorner P_2 \urcorner \ulcorner \mathcal{D}_1 \urcorner \ulcorner \mathcal{D}_2 \urcorner
$$

$$
\begin{array}{lcl}
\texttt{pf} & : & \texttt{pred} \rightarrow \texttt{Type} \\
\\
\texttt{truei} & : & \texttt{pf true} \\
\texttt{andi} & : & \Pi p\!:\!\texttt{pred}.\Pi r\!:\!\texttt{pred}.\texttt{pf}\ p \rightarrow \texttt{pf}\ r \rightarrow \texttt{pf}\ (\texttt{and}\ p\ \ r) \\
\texttt{andel} & : & \Pi p\!:\!\texttt{pred}.\Pi r\!:\!\texttt{pred}.\texttt{pf}\ (\texttt{and}\ p\ \ r) \rightarrow \texttt{pf}\ p \\
\texttt{ander} & : & \Pi p\!:\!\texttt{pred}.\Pi r\!:\!\texttt{pred}.\texttt{pf}\ (\texttt{and}\ p\ \ r) \rightarrow \texttt{pf}\ r \\
\texttt{impi} & : & \Pi p\!:\!\texttt{pred}.\Pi r\!:\!\texttt{pred}.(\texttt{pf}\ p \rightarrow \texttt{pf}\ r) \rightarrow \texttt{pf}\ (\texttt{impl}\ p\ \ r) \\
\texttt{impe} & : & \Pi p\!:\!\texttt{pred}.\Pi r\!:\!\texttt{pred}.\texttt{pf}\ (\texttt{impl}\ p\ \ r) \rightarrow \texttt{pf}\ p \rightarrow \texttt{pf}\ r \\
\texttt{alli} & : & \Pi p\!:\!\texttt{exp} \rightarrow \texttt{pred}.(\Pi v\!:\!\texttt{exp}.\texttt{pf}\ (p\ v)) \rightarrow \texttt{pf}\ (\texttt{ball}\ \ p) \\
\texttt{alle} & : & \Pi p\!:\!\texttt{exp} \rightarrow \texttt{pred}.\Pi e\!:\!\texttt{exp}.\texttt{pf}\ (\texttt{ball}\ \ p) \rightarrow \texttt{pf}\ (p\ e) \\
\texttt{eqid} & : & \Pi e\!:\!\texttt{exp}.\texttt{eqb}\ e\ e \\
\texttt{congr} & : & \Pi e_1\!:\!\texttt{exp}.\Pi e_2\!:\!\texttt{exp}.\Pi p\!:\!\texttt{exp} \rightarrow \texttt{pred}.\texttt{pf}\ (\texttt{eqb}\ e_1\ e_2) \rightarrow \texttt{pf}\ (p\ e_1) \rightarrow \texttt{pf}\ (p\ e_2) \\
\texttt{case} & : & \Pi e_1\!:\!\texttt{exp}.\Pi e_2\!:\!\texttt{exp}.\Pi p\!:\!\texttt{pred}. \\
 & & \qquad (\texttt{pf}\ (\texttt{eqb}\ e_1\ e_2) \rightarrow \texttt{pf}\ p) \rightarrow (\texttt{pf}\ (\texttt{neqb}\ e_1\ e_2) \rightarrow \texttt{pf}\ p) \rightarrow \texttt{pf}\ p \\
\texttt{contr} & : & \Pi e_1\!:\!\texttt{exp}.\Pi e_2\!:\!\texttt{exp}.\Pi p\!:\!\texttt{pred}.\texttt{pf}\ (\texttt{eqb}\ e_1\ e_2) \rightarrow \texttt{pf}\ (\texttt{neqb}\ e_1\ e_2) \rightarrow \texttt{pf}\ p \\
\texttt{mc0} & : & \Pi e_1\!:\!\texttt{mem}.\Pi e_2\!:\!\texttt{exp}.\Pi e_3\!:\!\texttt{exp}.\texttt{pf}\ (\texttt{eqb}\ (\texttt{sel}\ (\texttt{upd}\ e_1\ e_2\ e_3)\ e_2)\ e_3) \\
\texttt{mc1} & : & \Pi e_1\!:\!\texttt{mem}.\Pi e_2\!:\!\texttt{exp}.\Pi e_3\!:\!\texttt{exp}.\Pi e_4\!:\!\texttt{exp}. \\
 & & \qquad \texttt{pf}\ (\texttt{neqb}\ e_2\ e_4) \rightarrow \texttt{pf}\ (\texttt{eqb}\ (\texttt{sel}\ (\texttt{upd}\ e_1\ e_2\ e_3)\ e_4)\ (\texttt{sel}\ e_1\ e_4))
\end{array}
$$

Figure 5.3: The LF signature Σ that constitutes the representation of the axiomatization of first-order predicate logic with equality and array variables.

$$
\ulcorner \dfrac{\begin{array}{c}\overline{\triangleright\ P_1}^{\,u} \\ \vdots\ \mathcal{D}^u \\ \triangleright\ P_2\end{array}}{\triangleright\ P_1 \supset P_2}u \urcorner = \texttt{impi}\ \ulcorner P_1 \urcorner \ulcorner P_2 \urcorner\ (\lambda u\!:\!\texttt{pf}\ \ulcorner P_1 \urcorner . \ulcorner \mathcal{D}^u \urcorner)
$$

To conclude the presentation of the LF representation, consider the proof of the simple predicate "$P \supset (P \wedge P)$". The LF representation of this proof is shown in Figure 5.4.

5.1.1 The LF Type System

The main advantage of using LF for proof representation is that proof validity can be checked by a simple type-checking algorithm. That is, to check that the LF object M is the representation of a valid proof of the predicate P we use the LF typing rules (to be presented below) to verify that M has type $\texttt{pf}\ \ulcorner P \urcorner$ in the context of the signature Σ declaring the valid proof rules.

$$M \;=\; \texttt{impi} \ulcorner P \urcorner (\texttt{and} \ulcorner P \urcorner \ulcorner P \urcorner) \\ (\lambda u{:}\texttt{pf} \ulcorner P \urcorner.\texttt{andi} \ulcorner P \urcorner \ulcorner P \urcorner\, u\, u)$$

Figure 5.4: The LF representation of the proof by implication introduction followed by conjunction introduction of the predicate $P \supset (P \wedge P)$.

Type checking in the LF type system is defined by means of four judgments shown and described below:

$\Gamma \vdash^{LF} M : A$	M is a valid object of type A
$\Gamma \vdash^{LF} A : K$	A is a valid type of kind K
$A \equiv_\beta B$	A is β-equivalent to B
$M \equiv_\beta N$	M is β-equivalent to N

where Γ is a typing context assigning types to LF variables.

The derivation rules for the LF typing judgments are shown in Figure 5.5. For the β-equivalence judgments I omit the rules that define it to be an equivalence and a congruence.

Types :

$$\frac{\Sigma(a) = K}{\Gamma \vdash^{LF} a : K} \quad \frac{\Gamma \vdash^{LF} A : \Pi x{:}B.K \qquad \Gamma \vdash^{LF} M : B}{\Gamma \vdash^{LF} A\, M : [^{M}/_{x}]K} \quad \frac{\Gamma \vdash^{LF} A : \texttt{Type} \qquad \Gamma, x : A \vdash^{LF} B : \texttt{Type}}{\Gamma \vdash^{LF} \Pi x{:}A.B : \texttt{Type}}$$

Objects :

$$\frac{\Sigma(c) = A}{\Gamma \vdash^{LF} c : A} \quad \frac{\Gamma(x) = A}{\Gamma \vdash^{LF} x : A} \quad \frac{\Gamma, x : A \vdash^{LF} M : B \qquad \Gamma \vdash^{LF} A : \texttt{Type}}{\Gamma \vdash^{LF} \lambda x{:}A.M : \Pi x{:}A.B}$$

$$\frac{\Gamma \vdash^{LF} M : \Pi x{:}A.B \qquad \Gamma \vdash^{LF} N : A}{\Gamma \vdash^{LF} M N : [^{N}/_{x}]B} \quad \frac{\Gamma \vdash^{LF} M : A \qquad A \equiv_\beta B}{\Gamma \vdash^{LF} M : B}$$

Equivalence :

$$\frac{}{(\lambda x{:}A.M)N \equiv_\beta [^{N}/_{x}]M}$$

Figure 5.5: Type checking in the LF type discipline

As an example of how LF type checking is used to perform proof checking, consider the proof representation M shown in Figure 5.4. It is easy to verify, given the LF typing rules and the declaration of the constants involved, that this proof has the LF type "`pf (imp` $\ulcorner P \urcorner$ `(and` $\ulcorner P \urcorner \ulcorner P \urcorner$`))`". The adequacy of LF type checking for proof checking in the logic under consideration is stated formally in the Theorems 5.1 and 5.2 below. These theorems are proved in [HHP93]. From their proofs it is evident that they continue to hold if the logic is extended with new expression and predicate constructors or even with higher-order constructs.

Theorem 5.1 (Adequacy of Syntax Representation.)

1. *If E^b is a closed expression, then* $\cdot \vdash^{LF} \ulcorner E^b \urcorner : \mathtt{exp}$
2. *If E^s is a closed expression, then* $\cdot \vdash^{LF} \ulcorner E^s \urcorner : \mathtt{mem}$
3. *If P is a closed expression, then* $\cdot \vdash^{LF} \ulcorner P \urcorner : \mathtt{pred}$
4. *If M is a closed LF object such that* $\cdot \vdash^{LF} M : \mathtt{exp}$*, then there exists an expression E^b such that* $\ulcorner E^b \urcorner \equiv_{\beta\eta} M$.
5. *If M is a closed LF object such that* $\cdot \vdash^{LF} M : \mathtt{mem}$*, then there exists an expression E^s such that* $\ulcorner E^s \urcorner \equiv_{\beta\eta} M$.
6. *If M is a closed LF object such that* $\cdot \vdash^{LF} M : \mathtt{pred}$*, then there exists a predicate P such that* $\ulcorner P \urcorner \equiv_{\beta\eta} M$.

Theorem 5.2 (Adequacy of Derivation Representation.)

1. *If $\mathcal{D} :: \triangleright \mathcal{P}$ is a derivation of P then* $\cdot \vdash^{LF} \ulcorner \mathcal{D} \urcorner : \mathtt{pf}\ \ulcorner P \urcorner$.
2. *If M is a closed LF object such that* $\cdot \vdash^{LF} M : \mathtt{pf}\ \ulcorner P \urcorner$*, then there exists a derivation $\mathcal{D} :: \triangleright \mathcal{P}$ of P such that* $\ulcorner \mathcal{D} \urcorner \equiv_{\beta\eta} M$.

Note that Theorem 5.2(2) says that if the code producer can exhibit an LF object having the type "`pf` $\ulcorner VC(\Phi^A) \urcorner$" then we know that there is a derivation of the verification condition within the logic, which in turn means that the verification condition is valid and the agent code satisfies the safety policy.

Owing to the simplicity of LF and of the LF type system, the implementation of the type checker is simple and easy to trust. Furthermore, because all of the dependencies on the particular object logic are segregated in the signature, the implementation of the type checker can be reused directly for proof checking in various first-order or higher-order logics. The only logic-dependent component of the proof checker is the signature, which is usually easy to verify by visual inspection.

Unfortunately, the above-mentioned advantages of LF representation of proofs come at a high price. The typical LF representation of a proof is large, due to a significant amount of redundancy. This fact can already be seen in the proof representation shown in Figure 5.4, where there are six copies of P as opposed to only three in the predicate to be proved. The effect of redundancy observed in practice increases non-linearly with the size of the proofs. Consider for example, the representation of the proof of the n-way conjunction $P \wedge \ldots \wedge P$. Depending on how balanced is the binary tree representing this predicate, the number of copies of P in the proof representation ranges from an expected value of $n \log n$ to a worst case value of n^2. The redundancy in the representation is not only a space problem but also leads to inefficient proof checking, because all of the redundant copies have to be type checked and then checked for equivalence with instances of P from the predicate to be proved. The focus of the remainder of this chapter is to modify the LF representation to reduce the redundancy of the representation.

5.2 The Implicit LF Representation

The LF representation and type-checking algorithm presented in the previous section are adequate for the representation and validation of proofs. However the proof representations are unnecessarily large. The size of proofs, in general, is an important factor in any application that manipulates proofs explicitly, but the redundancy of representation in particular, has important consequences for the efficiency of proof checking. Consider the typical situation when the code receiver desires to check that an untrusted safety proof (for example, the one from Figure 5.4) proves a certain predicate. In such a situation every subterm of the proof representation must be type checked. This means that each of the six occurrences of the term "P" must be type checked separately. Moreover, following each of the type-checking operations the term must be compared with the instance of itself contained in the predicate to be proven, to ensure that every subderivation proves the desired predicate and not another one. Therefore the redundancy in the representation increases the amount of required checks and therefore can lead to inefficient proof validation.

The solution to the redundancy problem is to eliminate the redundant subterms from the proof. In most cases we can eliminate all copies of a given subterm from the proof and rely instead on the copy that exists within the predicate to be proved, which I am going to assume is trusted to be well formed. But now the code receiver will be receiving proofs with missing subterms. One possible strategy is for the code receiver to reconstruct the original form of the proof and then to use the simple LF type checking algorithm to validate it. But this would not save proof-checking time and would require significantly more working memory than the size of the incoming LF_i proof. Instead, I propose to modify the LF type-checking algorithm to reconstruct the missing subterms while it performs type checking. One major advantage of this strategy is that terms that are reconstructed based on copies from the verification condition do not need to be type checked themselves.

Before we plunge into the formal details of the type reconstruction I describe its operation on a simple example. For that purpose, consider a simple extension of LF syntax with a new term, called a placeholder, written as $*$ and used to mark a missing LF subterm. Consider now the proof of the predicate $P \supset (P \wedge P)$ of Figure 5.4. If we replace all copies of "P" with placeholders we get the following implicit proof:

$$\mathtt{impi}\ *_1\ *_2\ (\lambda u{:}*_3.\mathtt{andi}\ *_4\ *_5\ u\ u) \tag{5.3}$$

This implicit proof captures the essence of the proof and nothing more. The subterms marked with placeholders can be recovered while verifying that the term has type "$\mathtt{pf}\ (\mathtt{impl}\ \ulcorner P \urcorner\ (\mathtt{and}\ \ulcorner P \urcorner \ulcorner P \urcorner))$", as described below.

Reconstruction starts by recognizing the top-level constructor `impi`. The type of the entire term, "$\mathtt{pf}\ (\mathtt{impl}\ \ulcorner P \urcorner\ (\mathtt{and}\ \ulcorner P \urcorner \ulcorner P \urcorner))$", is "matched" against the result type of the `impi` constant, as given by the signature Σ. The result of this matching is an instantiation for placeholders 1 and 2 and a residual type-checking constraint for the explicit argument of `impi`, as follows:

$$\begin{array}{rlcl} & *_1 & \equiv & \ulcorner P \urcorner \\ & *_2 & \equiv & \mathtt{and}\ \ulcorner P \urcorner \ulcorner P \urcorner \\ \vdash & (\lambda u{:}*_3.\mathtt{andi}\ *_4\ *_5\ u\ u) & : & \mathtt{pf}\ \ulcorner P \urcorner \to \mathtt{pf}\ (\mathtt{and}\ \ulcorner P \urcorner \ulcorner P \urcorner) \end{array}$$

Reconstruction continues with the remaining type-checking constraint. From its type we can recover the value of placeholder 3 and a typing constraint for the body:

$$\begin{array}{rlcl} & *_3 & \equiv & \mathtt{pf}\ \ulcorner P \urcorner \\ u : \mathtt{pf}\ \ulcorner P \urcorner \vdash & \mathtt{andi}\ *_4\ *_5\ u\ u & : & \mathtt{pf}\ (\mathtt{and}\ \ulcorner P \urcorner \ulcorner P \urcorner) \end{array}$$

Now `andi` is the top-level constant and by matching its result type as declared in the signature with the goal type of the constraint we get the instantiation for placeholders 4 and 5 and two residual typing constraints:

$$\begin{array}{rlcl} & *_4 & \equiv & \ulcorner P \urcorner \\ & *_5 & \equiv & \ulcorner P \urcorner \\ u : \mathtt{pf}\ \ulcorner P \urcorner \vdash & u & : & \mathtt{pf}\ \ulcorner P \urcorner \\ u : \mathtt{pf}\ \ulcorner P \urcorner \vdash & u & : & \mathtt{pf}\ \ulcorner P \urcorner \end{array}$$

The remaining two constraints are solved by the variable typing rule, and this concludes the reconstruction and checking of the entire proof. We reconstructed the full representation of the proof by instantiating all placeholders with well-typed LF objects. We know that these instantiations are well-typed because they are ultimately extracted from the original constraint type, which is assumed to contain only well-typed subterms.

The formalization of the reconstruction algorithm described informally above is in two stages. First, I show a variant of the LF type system, called implicit LF or LF_i, that extends

LF with placeholders. This type system has the property that all well-typed LF_i terms can be reconstructed to well-typed LF terms. However, unlike the original LF type system, the LF_i type system is not amenable to a direct implementation of deterministic type checking. Instead, I describe a separate reconstruction algorithm in Section 5.3 that I prove to be adequate for type checking LF_i terms.

Just as in the previous example, I extend the syntax of LF objects with placeholders, written as $*$. An object M is fully reconstructed, or fully explicit, when it is placeholder free. I write $\mathrm{PF}(M)$ to denote this property. Then I extend this notation to type environments and I write $\mathrm{PF}(\Gamma)$ to denote that all types assigned in Γ to variables are placeholder-free. I also introduce the implicitly typed abstraction, written $\lambda x.M$.

The LF_i typing rules are an extension of the LF typing rules with two new typing rules for dealing with implicit abstraction and placeholders, and one new β-equivalence rule dealing with implicit abstraction. These additions are shown in Figure 5.6. The LF_i typing judgment is written $\Gamma \overset{i}{\vdash} M : A$.

Objects :

$$\frac{\Sigma(c) = A}{\Gamma \overset{i}{\vdash} c : A} \qquad \frac{\Gamma(x) = A}{\Gamma \overset{i}{\vdash} x : A} \qquad \frac{\Gamma \overset{i}{\vdash} M : A \qquad A \equiv_\beta B \qquad \mathrm{PF}(A)}{\Gamma \overset{i}{\vdash} M : B}$$

$$\frac{\Gamma, x : A \overset{i}{\vdash} M : B}{\Gamma \overset{i}{\vdash} \lambda x.M : \Pi x{:}A.B} \qquad \frac{\Gamma, x : A \overset{i}{\vdash} M : B}{\Gamma \overset{i}{\vdash} \lambda x{:}A.M : \Pi x{:}A.B}$$

$$\frac{\Gamma \overset{i}{\vdash} M : \Pi x{:}A.B \qquad \Gamma \overset{i}{\vdash} N : A \qquad \mathrm{PF}(A)}{\Gamma \overset{i}{\vdash} M\ N : [{}^{N}\!/_{x}]B} \qquad \frac{\Gamma \overset{i}{\vdash} M : \Pi x{:}A.B \qquad \Gamma \overset{i}{\vdash} N : A \qquad \mathrm{PF}(A)}{\Gamma \overset{i}{\vdash} M\ * : [{}^{N}\!/_{x}]B}$$

Equivalence :

$$\frac{}{(\lambda x{:}A.M)N \equiv_\beta [{}^{N}\!/_{x}]M} \qquad \frac{}{(\lambda x.M)N \equiv_\beta [{}^{N}\!/_{x}]M}$$

Figure 5.6: The LF_i type-system as an extension of LF.

Note that according to the LF_i type system placeholders cannot occur on a function position, but only as arguments in an application. This restriction allows us to simplify the reconstruction algorithm by avoiding higher-order unification. Note also that several LF_i rules require that the types involved do not contain placeholders. This restriction simplifies greatly the proofs of soundness and does not seem to diminish the effectiveness of the LF_i representation.

A quick analysis of the LF_i type-system reveals that it is not very useful for type checking or type inference. The main reason is that type checking an application involves "guessing" appropriate A and N. The type A can sometimes be recovered from the type of the application head, but the term N in an application to a placeholder cannot be found easily in general. This is not a problem for us because we are going to use the LF$_i$ type-system only as a step in proving the correctness of the type-reconstruction algorithm presented in the next section, and not as the basis for an implementation of a type-checking algorithm.

The only property of interest of the LF$_i$ type system is that once we have a typing derivation we can reconstruct the object involved and a corresponding LF typing derivation for it. To make this more precise we introduce the notation $M \nearrow M'$ to denote that M' is a fully-reconstructed version of the implicit object M (i.e., $\mathrm{PF}(M')$). This means that M' can be obtained from M by replacing all of its placeholders with fully-explicit LF objects and similarly all the implicit abstractions with explicit abstractions. Note that the reconstruction relation is not a function as there might be several reconstructions of a given implicit object or type.

Theorem 5.4 Soundness of LF$_i$ typing *If* $\Gamma \overset{i}{\vdash} M : A$ *and* $\mathrm{PF}(\Gamma)$, $\mathrm{PF}(A)$, *then there exists* M' *such that* $M \nearrow M'$ *and* $\Gamma \overset{LF}{\vdash} M' : A$.

Theorem 5.4 is restated and proved as Theorem B.2 on page 243 in Appendix B.

5.3 An Algorithm for LF$_i$ Type Reconstruction

The LF$_i$ type-system presented in the previous section has the benefit that it allows implicit LF terms. However this type-system does not immediately suggest a type-checking algorithm, for reasons explained before. I show in this section an algorithm that can be used to type check LF$_i$ terms, or more precisely to reconstruct and type check LF$_i$ terms.

Notation

In addition to the placeholder constants introduced in the previous section I introduce a new brand of variables. These unification variables play a similar role to that of placeholders in that they stand for missing terms. I shall use the letters x and y to denote traditional LF variables and the letter u to denote a unification variable. Also, I use Δ to denote a type environment containing only type assignments for unification variables. The letter Γ will denote type environments containing both kinds of variables. In the special case when an LF object M does not contain unification variables I write $\mathrm{UVF}(M)$. Note that unification variables always occur free. I extend this notation to $\mathrm{UVF}(\Gamma(FV(M)))$ to mean that the types associated by Γ to the free variables of M do not contain unification variables.

The main operation on unification variables is substitution with LF terms. I define the syntactic class of substitutions Ψ as follows:

$$\Psi \quad ::= \quad \cdot \quad | \quad u \mapsto M \quad \text{Substitutions}$$

The notation $\Psi(M)$ denotes the term obtained by performing the substitution Ψ on M. I write $Dom(\Psi)$ to refer to the set of unification variables on which Ψ is defined. I write PF(Ψ) to mean that all terms substituted by Ψ for unification variables are placeholder free. The notation $\Psi \circ \Psi'$ denotes the composition of substitutions Ψ and Ψ'. The resulting composition has the domain $Dom(\Psi') \cup Dom(\Psi)$ and maps u to $\Psi(\Psi'(u))$. Finally, I write $\Psi|_S$ to denote the substitution obtained from Ψ by restricting it to the set of unification variables S.

One of the key operations performed by the reconstruction algorithm is to compute substitutions through unification of terms or types. For a more precise presentation of the algorithm I use two flavors of unification, shown below in the case of unifying objects. The atomic flavor of unification is only defined for applications, while the regular flavor is defined for all kinds of objects. The same notation is used for expressing unification of types.

$$\begin{array}{ll} M \approx_a M' \Rightarrow \Psi & \text{Atomic Unification} \\ M \approx M' \Rightarrow \Psi & \text{Unification} \end{array}$$

The last syntactic construct that I introduce is a list of type reconstruction constraints defined as follows:

$$C \quad ::= \quad \cdot \quad | \quad C, M : A \quad | \quad C, A \approx_a B$$

The reconstruction algorithm is described by the two unification judgments introduced above and three additional mutually recursive judgments shown below:

$$\begin{array}{ll} \Gamma \overset{r}{\vdash} M : A \Rightarrow \Psi & \text{Main reconstruction judgment} \\ \Gamma \overset{r}{\vdash} M \Rightarrow (\Delta \,;\, C \,;\, B) & \text{Collect constraints} \\ \Gamma \overset{r}{\vdash} C \Rightarrow \Psi & \text{Solve constraints} \end{array}$$

In the rest of this section I show the definition of the five judgments introduced above, as a collection of inference rules. These inference rules can be implemented in a straightforward manner to produce a type reconstructor for LF$_i$.

Collecting Type-checking Constraints : $\Gamma \overset{r}{\vdash} M \Rightarrow (\Delta \,;\, C \,;\, B)$

This judgment is used for atomic objects, which are constants or variables applied to zero or more canonical objects. A canonical object is a sequence of abstractions with an atomic body, or equivalently a term in β-normal form. The atomic object M is scanned to find the application head, whose type is read from the signature Σ if it is a constant, or from the variable type environment Γ if it is a variable. Then all arguments are collected as typing constraints in the constraint list C. For each placeholder argument in M, a unification variable is introduced. These unification variables are collected in Δ along with their types. The resulting type B is the type of the whole application and might contain unification variables from Δ. Similarly, the types in the constraint list C might contain unification variables from Δ. Note that no placeholders can occur in types because of the side condition on the rule pertaining to the application to a placeholder.

$$\frac{}{\Gamma \overset{r}{\vdash} c \Rightarrow (\cdot\,;\,\cdot\,;\,\Sigma(c))} \qquad \frac{}{\Gamma \overset{r}{\vdash} x \Rightarrow (\cdot\,;\,\cdot\,;\,\Gamma(x))}$$

$$\frac{\Gamma \overset{r}{\vdash} M \Rightarrow (\Delta\,;\,C\,;\,\Pi x\!:\!A.B) \qquad \begin{array}{r} x \in FV(B) \supset (\mathrm{PF}(N) \textit{ and } \mathrm{UVF}(A) \\ \textit{and } \mathrm{UVF}(\Gamma(FV(N)))) \end{array}}{\Gamma \overset{r}{\vdash} M\ N \Rightarrow (\Delta\,;\,C, N : A\,;\,[N/x]B)}$$

$$\frac{\Gamma \overset{r}{\vdash} M \Rightarrow (\Delta\,;\,C\,;\,\Pi x\!:\!A.B)}{\Gamma \overset{r}{\vdash} M \ast \Rightarrow (\Delta, u : A\,;\,C\,;\,[u/x]B)} \qquad u \text{ is a new unification variable}$$

The restriction in the explicit-application rule ensures that the resulting type $[N/x]B$ does not contain placeholders, provided that B does not have placeholders itself. This restriction simplifies the proof of correctness of the reconstruction and is also required in order to match the LF_i typing judgments, which are defined only on types without placeholders.

Next, I describe how the list of constraints is solved. The main reason I separate the tasks of collecting constraints and solving them is to allow an implementation to choose an arbitrary order of solving the constraints. Previous experience with implementations of the reconstruction algorithm shows that having flexibility in the solving order greatly increases the effectiveness of the reconstruction algorithm, with the benefit that proofs with more missing components can be reconstructed.

Solving Residual Constraints: $\Gamma \overset{r}{\vdash} C \Rightarrow \Psi$

This judgment, whose rules are shown below, defines the process of solving all of the type-checking constraints in a list C, in an arbitrary order. To accommodate arbitrary solving orders I introduce the constraint reordering rule. There are two kinds of constraints, typing constraints for which the main reconstruction judgment is invoked, and unification constraints that are solved by unification.

$$\frac{}{\Gamma \overset{r}{\vdash} \cdot \Rightarrow \cdot} \qquad \frac{\Gamma \overset{r}{\vdash} C_1, C_2, C_3 \Rightarrow \Psi}{\Gamma \overset{r}{\vdash} C_2, C_1, C_3 \Rightarrow \Psi}$$

$$\frac{\Gamma \overset{r}{\vdash} M : A \Rightarrow \Psi \qquad \Psi(\Gamma) \overset{r}{\vdash} \Psi(C) \Rightarrow \Psi'}{\Gamma \overset{r}{\vdash} C, M : A \Rightarrow \Psi' \circ \Psi} \quad \frac{A \approx_a B \Rightarrow \Psi \qquad \Psi(\Gamma) \overset{r}{\vdash} \Psi(C) \Rightarrow \Psi'}{\Gamma \overset{r}{\vdash} C, A \approx_a B \Rightarrow \Psi' \circ \Psi}$$

Type Reconstruction for Objects: $\Gamma \overset{r}{\vdash} M : A \Rightarrow \Psi$

This is the base judgment of the type-reconstruction algorithm. The term M can contain variables (but no unification variables) that are declared in Γ and constants that are declared in Σ. M can also contain placeholders and implicit abstractions. However, the type environment Γ and the type A cannot contain placeholders. This is again required in order to relate the type reconstruction judgment to the LF$_i$ typing judgment and has the beneficial effect of simplifying the proof of correctness.

Recall that a canonical LF object is either an abstraction whose body is itself canonical or an atomic object. The type-checking judgment deals directly with abstractions and invokes the constraint collecting and solving judgments for atomic objects.

$$\frac{\Gamma, x : A \overset{r}{\vdash} M : B \Rightarrow \cdot}{\Gamma \overset{r}{\vdash} \lambda x.M : \Pi x{:}A.B \Rightarrow \cdot}$$

Note that in the case of the implicit abstraction the argument type is recovered from the goal type, which must itself be a functional type. Note also that while reconstruction is allowed for the abstraction body the substitution returned must be empty, meaning that subterms from the body of the abstraction cannot be used for reconstruction outside the abstraction. This restriction does not harm in any way the usefulness of the reconstruction algorithm for level-1 proofs (proof of formulas) but simplifies significantly the correctness arguments by eliminating the concern that the returned substitution might contain the bound variable x.

In the case of a constant, a variable or an application the typing judgment first collects type reconstruction constraints that are then solved in an arbitrary order.

$$\frac{\Gamma \overset{r}{\vdash} M \Rightarrow (\Delta \,;\, C \,;\, B) \qquad \Gamma, \Delta \overset{r}{\vdash} C, A \approx_a B \Rightarrow \Psi \qquad Dom(\Delta) \subseteq Dom(\Psi)}{\Gamma \overset{r}{\vdash} M : A \Rightarrow \Psi\big|_{Dom(\Gamma)}}$$

In the previous rule it is required that the substitution returned be defined on all unification variables introduced by the current collecting operation. This means that the entire list of constraints can be discarded at this point. In this respect the presented type-reconstruction algorithm is simpler, and potentially less powerful, than constraint-based algorithms that allow unsolved reconstruction constraints to persist beyond the place where they were introduced. However, this restriction does not seem to limit the power of the algorithm for reconstructing implicit representations of level-1 proofs and it does eliminate the need for the machinery for managing persistent constraints.

Note that the reconstruction algorithm does not check explicitly that the returned substitution is well-typed. This is the reason why the reconstruction algorithm runs faster on an implicit representation of a proof than the LF type checking algorithm on the fully-explicit form of the proof. The returned substitution is guaranteed to be well typed by the design of the unification judgment, described next.

Unification: $M \approx_a M' \Rightarrow \Psi$ and $M \approx M' \Rightarrow \Psi$

The purpose of unification is to check the equivalence of two objects or two types. The result of the unification is a substitution of terms for unification variables. Both terms being checked might contain implicit abstractions and might not be in canonical form, but they cannot contain placeholders.

The main limitation of the unification judgment is that it does not try to unify terms in which the unification variable is at the head of an application. In such a case, the resulting substitution would not be uniquely defined and thus it is preferable to avoid it if we can. In order to express this restriction precisely I use two flavors of unification: atomic and normal. Only the atomic unification can be used for the head of an application. Consequently, a unification variable cannot be instantiated by an atomic unification. The unification judgments are presented below.

Atomic Unification

$$\frac{}{c \approx_a c \Rightarrow \cdot} \qquad \frac{}{x \approx_a x \Rightarrow \cdot}$$

$$\frac{M \approx_a M' \Rightarrow \Psi \qquad \Psi(N) \approx \Psi(N') \Rightarrow \Psi'}{M\ N \approx_a M'\ N' \Rightarrow \Psi' \circ \Psi} \qquad \frac{[{}^{N_n}\!/\!{}_{x_n}] \ldots [{}^{N_1}\!/\!{}_{x_1}]M \approx_a M' \Rightarrow \Psi}{(\lambda x_1. \ldots \lambda x_n.M)\ N_1 \ldots N_n \approx_a M' \Rightarrow \Psi}$$

Normal Unification

$$\frac{M \approx M' \Rightarrow \cdot}{\lambda x.M \approx \lambda x.M' \Rightarrow \cdot} \qquad \frac{M \approx_a M' \Rightarrow \Psi}{M \approx M' \Rightarrow \Psi}$$

$$\frac{u \notin FV(M)}{u \approx M \Rightarrow u \mapsto M}$$

The side condition from the instantiation rule is required for correctness. This check is sometimes called the "occurs-check". Checking this condition requires scanning the object M that is used to replace the placeholder denoted by the unification variable u. We would very much want to avoid this check. Recall that we had two arguments in favor of the implicit representation: smaller representation size and faster checking based on the fact that the reconstructed objects, such as M here, do not require checking. As expressed by the unification judgment, the reconstructed fragments do not require checking but they require scanning. However, in most cases the occurs check is not necessary and can be eliminated. Such an optimization is discussed in Section 5.6.

Why does type reconstruction work?

The five reconstruction judgments described in the previous sections can be used directly as a type reconstruction algorithm. To summarize, the process of reconstructing and type checking an object M proceeds as follows:

- If M is an abstraction then use the abstraction rule and continue with the reconstruction of the body.
- Otherwise, M must be a variable or a constant applied to zero or more arguments. Use the collecting judgment to scan the arguments and replace the implicit ones with unification variables. Also, collect the explicit arguments in a list of type reconstruction constraints. Add to the list of constraints the unification of the required type for the application and the type computed based on the type of the application head and the arguments.
- Solve the list of constraints in a convenient order. As a result, return an instantiation for some unification variables.
- Verify that all local placeholders have an instantiation.

The derivation rules for the reconstruction judgments are a faithful description of an effective algorithm for LF_i reconstruction. This algorithm constitutes a sound method for validating proofs because any successful execution implies that there exist an LF_i typing derivation for the input proof representation, which in turn (by Theorem 5.4) means that there exists a fully explicit and well-typed LF term of the same type, which in turn (by Theorem 5.2(2)) means that the verification condition is derivable in the logic. The formal statement of correctness of the reconstruction algorithm is stated below.

Theorem 5.5 (Correctness of proof reconstruction) *If M is an LF_i object such that* $\mathrm{UVF}(M)$ *and* $\cdot \overset{r}{\vdash} M : \mathtt{pf}\ \ulcorner P \urcorner \Rightarrow \cdot$ *then* $\cdot \overset{i}{\vdash} M : \mathtt{pf}\ \ulcorner P \urcorner$.

Unfortunately, the proof of Theorem 5.5 is nowhere as simple as its statement and therefore I devote the entire Appendix B to it.

5.4 An Algorithm for LF_i Representation

The reconstruction judgment presented in Section 5.3 does not accept all implicit forms of a well-typed LF object. It is therefore useful to define an algorithm that can be used to erase from an LF representation of a proof as many redundant subterms as possible, while ensuring that the resulting implicit object will be accepted by the reconstruction. In a proof-carrying code environment, such a representation algorithm is applied to the proof immediately after it is produced by the theorem prover, in order to reduce the burden on the communication network.

There are many possible algorithms for erasing redundant subterms. Of these, I discuss in this section only one, called *bimodal representation*, that I determined experimentally to be a good compromise between complexity and effectiveness. Then, at the end of the section I briefly discuss other representation algorithms.

Any LF_i representation algorithm must depend on the reconstruction algorithm because it must be able to predict when a given subterm can be reconstructed and when not. Only terms that can be reconstructed can be omitted from the representation. One relevant detail of the reconstruction algorithm is the order in which the constraints are solved. We have seen that the correctness of the algorithm does not depend on the order, but the ability to reconstruct a given subterm does. As the representation algorithm must be able to predict the behavior of reconstruction we must either fix the solving order or else, we must allow the representation algorithm to embed in the representation the order in which the constraints must be solved for each particular application. I choose the simpler solution for the algorithm presented here and I require that the reconstruction algorithm solves constraints in the reverse order in which they were introduced, meaning that the unification constraint is solved first followed by the typing constraints. Experimental evidence suggests that this is a good solving order.[1]

Before discussing the precise details of the representation algorithm it is useful to see how it operates on a simple example. Consider that the representation task at hand is to remove redundancy from the LF object

$$\texttt{add0}\ E_1\ E_2\ M \tag{5.6}$$

where the declaration of the constant add0 is:

$$\texttt{add0} : \Pi e_1 \mathbin{:} \texttt{exp}.\Pi e_2 \mathbin{:} \texttt{exp}.\texttt{pf}\ (=\ e_1\ 0) \rightarrow \texttt{pf}\ (=\ (+\ e_1\ e_2)\ e_2) \tag{5.7}$$

(The add0 constant might represent, for example, the idempotency of adding zero. However, the representation algorithms presented in this section, just like the reconstruction

[1] Fixing the solving order does not necessarily mean that the results of representation must be suboptimal. We can still analyze each constant in turn and decide, for that particular constant, which is the best constraint solving order. Then we can usually reorder the arguments of the constant so the fixed solving order matches the optimal one.

algorithm of Section 5.3 are independent of the significance of signatures. Thus, it is not relevant here that we are reconstructing proofs in a given logic where a given derivation rule is represented by the constant "`add0`".)

The representation algorithm works by considering the behavior of reconstruction when faced with the task of reconstructing an implicit representation of the LF object 5.6 with a given type A, as follows:

$$\cdot \overset{r}{\vdash} \texttt{add0}\ E_1'\ E_2'\ M' : A \Rightarrow \Psi \tag{5.8}$$

In the above equation E_1' is an implicit representation of E_1, possible even $*$, and E_2' and M' are implicit representations of E_2 and M respectively.

From 5.8, the reconstruction algorithm generates the following constraints, ordered in the order in which they are solved:

$$A \approx_a \texttt{pf}\ (=\ (+\ E_1'\ E_2')\ E_2') \Rightarrow \Psi_1 \tag{5.9}$$
$$\cdot \overset{r}{\vdash} \Psi_1(M') : \texttt{pf}\ (=\ (\Psi_1(E_1'))\ 0) \Rightarrow \Psi_2 \tag{5.10}$$
$$\cdot \overset{r}{\vdash} (\Psi_2 \circ \Psi_1)(E_2') : \texttt{exp} \Rightarrow \Psi_3 \tag{5.11}$$
$$\cdot \overset{r}{\vdash} (\Psi_3 \circ \Psi_2 \circ \Psi_1)(E_1') : \texttt{exp} \Rightarrow \Psi_4 \tag{5.12}$$

The most optimistic scenario, from the point of view of reconstruction, is when A is fully explicit and hence E_2' and E_1' need not be specified at all because they can be recovered from 5.9. The case when the type of the application is fully explicit is called the *checking* case. In this case $\Psi_1(E_1')$ and $\Psi_1(E_2')$ are also well typed and fully explicit, thus the constraints 5.11 and 5.12 are satisfied automatically.[2] Furthermore, the constraint 5.10 will also be in checking mode. Thus, in the checking mode the representation of 5.6 can be as small as "`add0` $*$ $*$ M'", where M' is the representation in checking mode of M.

The most pessimistic scenario is when A does not contain any structural information (e.g., A = `pf` $*$), and thus no information about E_1 and E_2 can be recovered from 5.9. To make it even more difficult on the reconstruction, we also require that the type A be reconstructed in addition to the term. This case is called the *inference* mode. Even though reconstruction is not able to infer the structure of E_1 and E_2 just from the constraint 5.9 it does not mean that E_1 and E_2 must appear explicitly in 5.6. Instead we try to see if they can be recovered from the other constraints. In our example, E_1 could be recovered from 5.10 if that constraint is done in inference mode. In that case the constraint 5.12 is automatically satisfied. And because in inference mode we must fully reconstruct the type also, E_2 must be fully explicit. Thus the representation of 5.6 in inference mode is "`add0` $*$ E_2 M'", where M' is the representation in inference mode of M.

In addition to the two extreme cases discussed above, we could consider the situations when the type A is only partially implicit and can be used to recover partially some of the

[2]There are not even collected, because E_1' and E_2' are implicit arguments.

arguments. We could also consider the variant of inference when it is not mandatory that the type be fully recovered after reconstruction. In this section we consider only the two extreme cases, hence the name of *bimodal* representation. Then, in Section 5.7 I briefly discuss the potential benefits of more complex representation algorithms.

The major advantage of the bimodal representation algorithm is that a large part of the work can be done statically, while the signature is loaded. More precisely, for each constant we can compute, based on the constant's declared type, two *representation recipes*, one for checking and one for inference. A representation recipe for an n-ary constant c is a sequence of n representation directives, with one directive corresponding to an argument of c. Each representation directive specifies whether the corresponding argument can be left implicit or it must be at least partially explicit. Also, in the latter case the directive says in which mode should the explicit argument be represented. Hence, the bimodal representation algorithm uses three representation directives, $*$ to denote an argument that can be omitted, and e_c and e_i to denote explicit arguments for which the checking and inference recipes, respectively, must be used recursively. The two recipes corresponding to the constant c are denoted by by $R^c(c)$ (the "checking" recipe) and $R^i(c)$ (the "inference recipe"). To refer to both the checking and inference modes at the same time I shall use the letter m to stand for either c or i.

There are two components of the representation algorithm. First is the algorithm that computes the representation recipes for each constant based on the declared type for the constant. Then, there is the algorithm that scans a fully-explicit LF object and, depending on the representation recipes, decides which subterms can be omitted. The latter part is the simplest and I describe it first.

The main representation function is described in Figure 5.7 by the judgments $M \stackrel{m}{\to} M'$ (compute M', which is the representation of M in mode m, with $m \in \{c, i\}$) and the auxiliary judgment $M \stackrel{m}{\Rightarrow} M' + R$ (compute M', the representation of the application head M in mode m and compute also the recipe R to be used for the arguments of M). There are two cases for the main judgment. In the case of an abstraction, the type of the bound variable is left implicit and the body is represented recursively. In all other cases, the auxiliary judgment is invoked.

The auxiliary representation judgment first scans the application until it reaches its head. The head can be a constant, whose representation is the constant itself and the recipe for the remaining arguments is the constant's recipe, or it can be a variable, in which case the representation is the variable itself and the recipe for the remaining arguments is a sequence of "e_m" directives as long as the arity of the variable.[3] (m is either c or i depending on whether we are computing the checking or the inference recipe.) On return from an auxiliary judgment the representation of the application's argument is done according to the

[3]This means that the bimodal representation algorithm does not attempt to do a good job on applications whose head is a variable.

$$\frac{M \overset{m}{\to} M'}{\lambda x\!:\!A.M \overset{m}{\to} \lambda x\!:\!*.M'} \qquad \frac{M \overset{m}{\Rightarrow} M' + \cdot}{M \overset{m}{\to} M'} \qquad \frac{}{c \overset{m}{\Rightarrow} c + R^m(c)} \qquad \frac{}{x \overset{m}{\Rightarrow} x + e_m \ldots e_m}$$

$$\frac{M_1 \overset{m}{\Rightarrow} M_1' + *R}{M_1\ M_2 \overset{m}{\Rightarrow} M_1'\ * + R} \qquad \frac{M_1 \overset{m}{\Rightarrow} M_1' + e_{m'}\,R \qquad M_2 \overset{m'}{\to} M_2'}{M_1\ M_2 \overset{m}{\Rightarrow} M_1'\ M_2' + R}$$

Figure 5.7: The bimodal representation algorithm.

first directive.

The core of the representation algorithm is the computation of the representation recipes for each constant based on its declared type. In order to describe precisely the computation of the representation recipes, I define first the function $\mathit{Unif}(A)$ which computes the set of free variables of A that are guaranteed to be instantiated during a successful unification with another fully-explicit type B. The definition of Unif follows that of the unification. The complement of $\mathit{Unif}(A)$ is denoted by $\mathit{NUnif}(A)$, which normally consists of those variables that occur in functor position in A. This notation is extended to objects and the rules for computing the unifiable sets are described below:

$$\mathit{Unif}(A) = \begin{cases} \emptyset, & \text{if } A = a \\ \mathit{Unif}(A') \cup (\mathit{Unif}(M) - \mathit{NUnif}(A')), & \text{if } A = A'\ M \\ \mathit{Unif}(A') \cup (\mathit{Unif}(A'') - \{x\}), & \text{if } A = \Pi x\!:\!A'.A'' \end{cases}$$

$$\mathit{Unif}(M) = \begin{cases} \emptyset, & \text{if } M = c \\ \{x\}, & \text{if } M = x \\ \emptyset, & \text{if } M = \lambda x\!:\!A.M' \\ \mathit{Unif}(M') \cup (\mathit{Unif}(M'') - \mathit{NUnif}(M')), & \text{if } M = M'\ M'' \text{ and the head of } M' \text{ is a constant} \\ \emptyset, & \text{if } M = M'\ M'' \text{ and the head of } M' \text{ is } \textit{not} \text{ a constant} \end{cases}$$

$$\mathit{NUnif}(A) = \mathit{FV}(A) - \mathit{Unif}(A)$$
$$\mathit{NUnif}(M) = \mathit{FV}(M) - \mathit{Unif}(M)$$

Consider an application of the constant c whose type is $\Sigma(c) = \Pi x_1\!:\!A_1.\ldots\Pi x_n\!:\!A_n.A_{n+1}$ such that A_{n+1} is not a type abstraction. Due to the right-to-left order of solving constraints, the first solving operation unifies A_{n+1} with a given type B. For this unification to succeed it is necessary that the arguments corresponding to variables in $\mathit{NUnif}(A_{n+1})$ are explicit. If we are in a checking situation (i.e., B does not contain unification variables) then unification finds instantiations for the arguments corresponding to variables in $\mathit{Unif}(A_{n+1})$. If we are

k	1	2	3	4
A_k	`exp`	`exp`	`pf` $(= x_1\ 0)$	`pf` $(= (+\ x_1\ x_2)\ x_2)$
$\mathit{Unif}(A_k)$	$\emptyset$	$\emptyset$	$\{x_1\}$	$\{x_1, x_2\}$
$\mathit{NUnif}(A_k)$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
I_k^c	$\{x_1, x_2\}$	$\{x_1, x_2\}$	$\{x_1, x_2\}$	$\{x_1, x_2\}$
E_k^c	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
I_k^i	$\{x_1\}$	$\{x_1\}$	$\{x_1\}$	$\emptyset$
E_k^i	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
r_k^c	$*$	$*$	e_c	
r_k^i	$*$	e_c	e_i	

Figure 5.8: The computation of the recipe directives in the case of the constant `add0` with the declared type $\Pi x_1 : \mathtt{exp}.\Pi x_2 : \mathtt{exp}.\mathtt{pf}\ (= x_1\ 0) \rightarrow \mathtt{pf}\ (= (+\ x_1\ x_2)\ x_2)$. The values in the table are computed from right to left.

in an inference situation then we assume conservatively that unification does not find any instantiation. Next, the typing constraints for the arguments of c are processed, in reverse order. Let I_k^m, with $m \in \{c, i\}$ and $k \in 1, \ldots, n+1$, be the set of the variables among $\{x_1, \ldots, x_n\}$ that are guaranteed to be instantiated after processing the constraints corresponding to $A_k, \ldots, A_{n+1}$ (the last constraint is a unification constraint and the others are typing constraints). Similarly, let E_k^m be the set of those variables that are required to be explicit so that unification does not fail when processing the constraints corresponding to $A_k, \ldots, A_{n+1}$. These sets are computed starting with $k = n+1$ as follows:

$$\begin{array}{lcl} I_{n+1}^c & = & \mathit{Unif}(A_{n+1}) \\ I_{n+1}^i & = & \emptyset \\ E_{n+1}^m & = & \mathit{NUnif}(A_{n+1}) \\ I_k^m & = & \left\{ \begin{array}{ll} I_{k+1}^m & \text{if } x_k \in I_{k+1}^m \\ I_{k+1}^m \cup (\mathit{Unif}(A_k) - E_{k+1}^m) & \text{otherwise} \end{array} \right. \\ E_k^m & = & \left\{ \begin{array}{ll} E_{k+1}^m & \text{if } x_k \in I_{k+1}^m \\ E_{k+1}^m \cup (\mathit{NUnif}(A_k) - I_{k+1}^m) & \text{otherwise} \end{array} \right. \end{array} \tag{5.13}$$

With these definitions we can define the representation recipe as $R^m(c) = r_1^m \ldots r_n^m$, where the recipe directives are defined as follows:

$$r_k^m = \left\{ \begin{array}{ll} * & \text{if } x_k \in I_{k+1}^m \\ e_c & \text{if } FV(A_k) \subseteq I_{k+1}^m \cup E_1^m \\ e_i & \text{otherwise} \end{array} \right.$$

These definitions say that an argument corresponding to x_k can be implicit if it can be inferred from the constraints corresponding to $A_{k+1}, \ldots A_{n+1}$, and if x_k does not appear in

Constant	R^c	R^i
`and`	$e_c\ e_c$	$e_c\ e_c$
`andi`	$*\ *\ e_c\ e_c$	$*\ *\ e_i\ e_i$
`andel`	$*\ *\ e_i$	$*\ *\ e_i$
`impi`	$*\ *\ e_c$	$e_c\ e_c\ e_i$
`alle`	$e_c\ e_c\ e_c$	$e_c\ e_c\ e_c$
`add0`	$*\ *\ e_c$	$*\ e_c\ e_i$
`mc1`	$*\ *\ *\ *\ e_c$	$e_c\ *\ e_c\ *\ e_i$

Figure 5.9: A fragment of the representation recipes for the signature shown in Figure 5.3.

a functor position in any of the $A_{k+1}, \ldots, A_{n+1}$. (This is because $I_k^m \cap E_k^m = \emptyset$, for all k.) In this case, the constraint is not even collected and therefore $I_k^m = I_{k+1}^m$ and $E_k^m = E_{k+1}^m$. Otherwise, the argument must be explicit and is represented recursively in checking mode only if all of the variables occurring in its type are either required to be explicit (E_1^m) or can be inferred from the constraints solved before (I_{k+1}^m).

Consider for example the operation of the recipe computation algorithm in the case of the "`add0`" constant declared as in 5.7. First, we compute the unifiable and non-unifiable sets for $A_1, \ldots, A_4$, as shown in Figure 5.8. Then, we start computing the I_k^m and E_k^m sets for $k = 4, \ldots 1$. Finally, we can compute the checking and inference representation directives. Figure 5.9 shows more examples of representation recipes for constants used in the representation of first-order logic with equality and array variables. The case of the constant "`alle`" is one where there are non-unifiable variables. In this particular case, the representation algorithm is not particularly effective.

5.5 Representing Arithmetic Proofs

The preceding sections argue formally and informally that the Edinburgh Logical Framework and especially the LF$_i$ variant is an excellent choice for representing and checking derivations in a variety of logics of interest to proof-carrying code. This is true as long as the logic of interest is finitely axiomatizable. Only then we can build an LF signature containing a declaration for each derivation rule.

Unfortunately there is at least one instance when it would be convenient to encode a logic whose most convenient axiomatization is not finite. This is the case with various forms of arithmetic. Take for example the commutative group $(\mathbb{Z}, +)$ of addition over integers with the usual commutativity, associativity, identity and inversability axioms. Equality in this group can be axiomatized with a finite number of derivation rules only if we encode integers using some sort of unary or binary notation. This is inconvenient because it leads to complex proofs even for simple facts about addition of integer literals. It would be a lot more convenient to be able to encode the literals directly. But then we need at least as many axioms as there are literals so that we can define them.

In this section I present a simple extension to LF whose purpose is to encode and check equality proofs in the commutative group $(\mathbb{Z}, +)$. Once this is in place, a small number of axioms are required to handle more complex and more realistic fragments of arithmetic, such as integer linear programming and modular arithmetic. Both examples are considered in Chapter 7 in the context of decision procedures that emit LF proof representations.

The extension consists mainly of extending the LF set of objects to include integer literals that behave as constants but are not required to be declared explicitly in the signature. All integer literals are assumed to have type "`exp`". I also add the declarations or the addition, negation and equality between integers, as shown in Figure 5.10.

$$
\begin{array}{lcl}
\mathtt{exp} & : & \mathtt{Type} \\
\mathtt{pred} & : & \mathtt{Type} \\
\\
\mathtt{add} & : & \mathtt{exp} \to \mathtt{exp} \to \mathtt{exp} \\
\mathtt{neg} & : & \mathtt{exp} \to \mathtt{exp} \\
\mathtt{aeq} & : & \mathtt{exp} \to \mathtt{exp} \to \mathtt{pred} \\
\\
\mathtt{arith} & : & \Pi x_1 : \mathtt{exp}. \Pi x_2 : \mathtt{exp}. \mathtt{pf}\ (\mathtt{aeq}\ x_1\ x_2)
\end{array}
$$

Figure 5.10: The encoding of the additive group of integers in LF.

The special constant "`arith`" is introduced to encode proofs of arithmetic equality. The constant "`arith`" is unusual in the sense that it constructs a proof of equality without any assumption except that the constituents of the equality are well-typed expressions. It seems that this signature is not adequate for representing the additive group equality because we would be able to construct a representation of the equality proof for just about any pair of expressions. To fix the problem, I extend the LF type checking algorithm so that an occurrence of "`arith`" is not just type checked but also analyzed using the algorithm $\mathcal{A}rith$, described in Figure 5.11, to ensure that it is indeed a valid proof.

The intuition behind the $\mathcal{A}rith$ function is that for each expression E involving only variables $Vars = \{x_1, \ldots, x_n\}$, integer literals, addition and unary negation, there exist integer constants $c, a_1, \ldots, a_n$ such that the value of E for a mapping $\rho \in Vars \to \mathbb{Z}$ is "$c + \sum_{i=1}^{n} a_i \rho(x_i)$". In other words, such an expression can be equivalently written as a linear combination of its free variables. The function $\mathcal{A}rith$ computes the coefficients for an arbitrary expression. The function $\mathcal{A}rith$ is defined to take an integer factor and a set of intermediary coefficients $\mathcal{C} \in \mathbb{Z} \times (Vars \to \mathbb{Z})$ in addition to the expression E. The set of coefficients consists of a constant c and an integer coefficient for each variable in $Vars$. The initial value of the factor is 1 and of the coefficients is $0_{Vars} = (0, \lambda x \in Vars.0)$.

$$\mathcal{A}rith(E, f, \mathcal{C}) = \begin{cases} (c + f \cdot n, C), & \text{if } E \equiv n \quad \text{and } \mathcal{C} \equiv (c, C) \\ (c, C[x \mapsto C(x) + f]), & \text{if } E \equiv x \quad \text{and } \mathcal{C} \equiv (c, C) \\ \mathcal{A}rith(E', f, \mathcal{A}rith(E'', f, \mathcal{C})), & \text{if } E \equiv \texttt{add}\ E'\ E'' \\ \mathcal{A}rith(E', -f, \mathcal{C}), & \text{if } E \equiv \texttt{neg}\ E' \end{cases}$$

Figure 5.11: The algorithm $\mathcal{A}rith$ for checking equalities in the additive group $(\mathbb{Z}, +)$.

To complete the extension of LF, I introduce a special case of the rule used in LF_i to type check an application, for the situation when the head of the application is "`arith`", as follows:

$$\frac{\begin{array}{ll} \Gamma \stackrel{r}{\vdash} \texttt{arith}\ M_1\ M_2 \Rightarrow (\Delta\ ;\ C\ ;\ B) & \\ \Gamma, \Delta \stackrel{r}{\vdash} C, A \approx_a B \Rightarrow \Psi & \Psi\big|_{Dom(\Gamma)}(A) = \texttt{pf}\ (\texttt{aeq}\ E_1\ E_2) \\ Dom(\Delta) \subseteq Dom(\Psi) & \mathcal{A}rith(\texttt{add}\ E_1\ (\texttt{neg}\ E_2), 1, 0_{Dom(\Gamma)}) = 0_{Dom(\Gamma)} \end{array}}{\Gamma \stackrel{r}{\vdash} \texttt{arith}\ M_1\ M_2 : A \Rightarrow \Psi\big|_{Dom(\Gamma)}}$$

Note that the conclusion and the left side of the hypothesis are copied directly from the original rule. What is new is the right side of the hypothesis, which ensures that the coefficients of the expression $E_1 - E_2$ as computed by the $\mathcal{A}rith$ function are all zero. Now, we are able to extend the proof of adequacy of representation and of reconstruction even for objects using the constants "`arith`". For this we need the following theorem.

Theorem 5.14 *If E_1 and E_2 are expressions with variables within Vars and such that $\mathcal{A}rith(\texttt{add}\ \ulcorner E_1 \urcorner\ (\texttt{neg}\ \ulcorner E_2 \urcorner), 1, 0_{Vars}) = 0_{Vars}$, then in any valuation $\rho \in Vars \rightarrow \mathbb{Z}$ we have that $\mathcal{V}(\rho(E_1)) = \mathcal{V}(\rho(E_2))$.*

PROOF: The proof is by means of a more general lemma that allows for arbitrary values of the factor, the expression and the intermediate set of coefficients. The lemma states that the function $\mathcal{A}rith$ computes a correct set of coefficients, when it succeeds. Then we use that fact that $\mathcal{V}(E_1 + (-E_2)) = 0 \iff \mathcal{V}(E_1) = \mathcal{V}(E_2)$.

□

Thus, by extending the LF reconstruction algorithm with the simple arithmetic decision procedure of Figure 5.11 we are able to encode many arithmetic proofs in a very concise manner. The savings are due to encoding the integer literals explicitly and by avoiding all trivial but lengthy proofs of equality in the additive group of integers. With this extension we can now encode in LF other logics without finite axiomatizations such as integer linear programming, shown in Section 7.3.2.

5.6 The Implementation of LF$_i$

In the previous sections I describe the LF$_i$ reconstruction algorithm, used to validate implicit representation of proofs, and one LF$_i$ representation algorithm, used to compact the proofs by erasing most of the redundant subterms. Even though these algorithms are described abstractly in terms of inference rules they can be both implemented by a straightforward transcription of the rules into code. However, several implementation details were left unspecified to provide more freedom to the implementor. My experience with several implementations shows that some of these details can have a significant impact on the performance of the algorithms. In this section I discuss some of the implementation choices that I discovered to be beneficial.

One such detail is the implementation of substitution. For example, the straightforward implementation of the substitution $[{}^{M}\!/\!{}_{x}]N$ would introduce as many copies of M as there are free occurrences of x in N, leading to excessive memory usage [Wad71]. A better strategy, also employed in the Twelf implementation of LF [PSC], is to use *explicit substitutions* [ACCL91] so that the substitution is performed lazily when the resulting term is later examined. Thus, each LF type or object is represented as a pair of a regular LF type or object with free variables and a substitution of these free variables to pairs of terms and substitution, as follows:

$$
\begin{array}{lcll}
\text{LF objects} & & ::= & (M, \Omega) \\
\text{LF types} & & ::= & (A, \Omega) \\
\text{Explicit Substitutions} & \Omega & ::= & \cdot \;\mid\; \Omega, x \mapsto (M, \Omega')
\end{array}
$$

The invariant property of an explicit substitution representation is that it is always closed, i.e., for each pair (M, Ω) we have $FV(M) \subseteq Dom(\Omega)$. To maintain this property and still be able to represent the terms with free occurrences of variables that arise during reconstruction we make the convention that whenever a binding is removed from a term, a fresh constant is created and substituted for the bound variable, as in the following modified rule for reconstruction of abstractions:

$$
\frac{\Gamma, x : (A, \Omega_A) \overset{r}{\vdash} (M,\ \Omega_M, x \mapsto (c_x, \cdot)) : (B,\ \Omega_A, x \mapsto (c_x, \cdot))}{\Gamma \overset{r}{\vdash} (\lambda x.M, \Omega_M) : (\Pi x\!:\!A.B, \Omega_A)} \quad c_x \text{ is a new constant} \tag{5.15}
$$

The substitution is now very cheap because all it does is to extend the explicit substitution of the subject term, as in the following rule for β-reduction:

$$
(\lambda x.N, \Omega_N)(M, \Omega_M) \quad \xrightarrow{\beta} \quad (N,\ \Omega_N, x \mapsto (M, \Omega_M))
$$

The true cost of substitution is paid when an object is traversed and a variable is reached. In this case, the traversal must continue with the explicit substitution of that variable, as suggested by the following identity:

$$(x, \Omega) \equiv (M, \Omega') \qquad \text{if } x \mapsto (M, \Omega') \text{ appears in } \Omega$$

Storing explicit substitutions requires less memory than the substituted term because the space required to record the substitution is usually less than the space required to store several copies of the substituted term. In addition, the explicit-substitution representation saves processing time because the terms involved in the substitution do not have to be rewritten to avoid variable name clashes. However, the time required to scan the resulting term is not changed by using explicit substitutions.

In addition to substitution, another potentially costly operation is the composition of substitutions during unification. The purpose of composition is to ensure that once an instantiation is found for a unification variable, that instantiation is applied to *all* occurrences of the unification variable. This effect can be achieved at a low cost if all the explicit substitutions share the mapping corresponding to the same unification variable. Then, when unification finds an instantiation for a unification variable, it can update the shared cell that stores the explicit substitution for that variable with the result that the new instantiation is seen simultaneously by all relevant explicit substitutions. The required sharing can be established by making sure that in rule 5.15 the two occurrences of the cell $(c_x, \cdot)$ are shared.

Another implementation choice that we must face is the representation of variables. Given that we are using explicit substitutions, it is natural to represent the bound variables using *deBruijn indices* [DeB72]. In this notation, each occurrence of a bound variable is represented by a positive integer that counts how many lambda abstractions there are in the abstract syntax tree between this occurrence and the binding abstraction. Thus, $\lambda x.\lambda y.a\ x\ y$ is represented as $\lambda\lambda\, a\ 1\ 0$. The advantage of using deBruijn indices is that the explicit substitutions can be implemented as linked lists of pairs. The explicit substitution of a variable can be found by skipping a number of list cells equal to the deBruijn index of the variable occurrence.

Throughout this dissertation, I ignore the details of the external representation of LF$_i$ terms. This is because any reasonable way to linearize an LF term in an architecture-independent manner is acceptable. My experience suggests that it is a good idea to encode applications as a head followed by a number of arguments and abstractions as a number of bindings followed by a body. This encoding is called functor/arguments in [MP93]. An object in the external representation has a tag denoting whether it is one of the following:

- a variable, in which case the rest of the object is the deBruijn index of the variable, or
- a constant, in which case the rest of the object is an index into a global table of constants (the current signature), or
- an application, in which case the rest of the object contains the number of arguments, then the head followed by the encoding of the arguments, or

- an abstraction, in which case the rest of the object encodes the number of bindings, followed by the encoding of the types of each binding and the body.

In my implementation, the basic unit of representation is a 16-bit word. In this case, the tag word can also contain the deBruijn index for a variable, the index of a constant or the length of an application or abstraction. The integer literals are represented as constants having indices that are outside the maximum size signature.

To close the implementation section, I describe a couple of opportunities for optimizing the reconstruction algorithm. One of the most promising opportunities for optimization is the elimination of the occurs-check in the unification rule. As shown clearly in the correctness proofs of LF_i reconstruction (shown in Appendix B), it is necessary to verify that the variable to be unified does not occur free in the target of the instantiation. However, in many cases this can be guaranteed without checking.

For this purpose, assume that we partition the set of unification variables into those that are guaranteed not to occur twice in any unification goal and the rest. I refer to the former set of variables and the *linear* variables. Then, we know that the occurs check does not need to be performed when unifying a linear variable.

The question is how to discover which variables are linear. A solution is suggested by considering the place where unification variables are created, that is, the constraint collection rule shown again below:

$$\frac{\Gamma \overset{r}{\vdash} M \Rightarrow (\Delta \,;\, C \,;\, \Pi x\!:\!A.B)}{\Gamma \overset{r}{\vdash} M * \Rightarrow (\Delta, u : A \,;\, C \,;\, [^{u}\!/\!_{x}]B)} \qquad u \text{ is a new unification variable}$$

where B is a dependent type of the form $B = \Pi x_1 : A_1. \ldots \Pi x_n : A_n.A_{n+1}$. If x occurs at most once in each of the A_i, then we know that u does not occur more than once in any constraint, and thus u is a good candidate for a linear unification variable. Note, that the linearity of u in this case can be precomputed statically in the frequent case when the head of M is a constant and thus B is a portion of a type declared in the signature.

Note that the linearity property of a unification variable is preserved by a substitution of another linear variable, but it might not be preserved if the substituted variable is not linear and occurs multiple times. To account for this case, the implementation of unification disables the occurs-check optimization while traversing an explicit substitution for a non-linear variable. The experiments presented in Section 8.3 show that this simple optimization reduces the time required for proof checking by a factor of 2 or 3, or in some cases even as high as 8.

Another possible optimization is to reduce the memory usage required for reconstruction. A close inspection of the reconstruction rules reveals that, in addition to the memory required to store the original LF object and type, during reconstruction we must allocate space to hold the explicit substitution list cells and the constraint list data structure. The constraint data structure can be allocated on the stack because it can be discarded immediately after the current reconstruction goal is finished. We can also observe that if during the type reconstruction of a given subterm there were no instantiations, then it is safe to deallocate the entire memory that was allocated during type reconstruction for this particular subterm. To take advantage of this common allocation pattern my implementation of PCC uses a private heap from which it allocates sequentially and deallocates entire chunks at a time in a stack-like manner. This way, both the allocation and deallocation are quite inexpensive.

There is however the case when there are instantiations during the reconstruction of a subterm, and in such cases the implementation refrains to deallocate at all. To detect such cases on a per-subterm basis, the implementation maintains a counter of the number of uninstantiated distinct unification variables present. This counter is initialized to zero at top level, it is incremented when a unification variable is introduced and is decremented when the unification finds an instantiation. Note that a given unification variable is only instantiated once. Then, it is safe to deallocate all the memory that was allocated during the reconstruction of a subterm if the number of uninstantiated unification variables did not change. The experiments shown in the Section 8.3 show that this optimization can reduce the memory usage by a factor of over 20 in some cases. However, this optimization might not be as interesting as it might seem because the memory usage is very small anyway.

5.7 Discussion

The problem of redundancy in the representation of proofs has been addressed before for the purpose of simplifying the user interface of theorem provers and proof assistants. Miller suggests in [Mil87] an extreme approach where the proof object records only the substitutions for the quantifiers, relying on the decidability of the tautology of the resulting matrix. This leads to very compact proof representations at the expense of an NP-complete tautology checking problem. Furthermore, in the presence of interpreted function symbols and arithmetic (as is always the case in the PCC proofs) the tautology checking problem can easily become undecidable.

The argument synthesis and term reconstruction algorithms implemented in the LEGO system [Pol90] and in the Coq [DFH+93] proof assistant are less effective than my algorithm, in the sense that fewer proof subterms can be omitted from the proof representation, and therefore more redundancy has to be tolerated. This is because they address the more difficult problem of representing a proof so that the predicate that it proves can be recovered from it. This is as if the bimodal representation algorithm is reduced to the less effective inference mode only. For example, an application of the constant "`eqid`" of type "$\Pi e : \texttt{exp}.\texttt{pf}\ (= e\ e)$" must always be explicit even though e could be sometimes recovered from the context. The algorithm presented in this chapter is able to synthesize more proof subterms by using information both from the context and from the predicate that the proof is supposed to prove.

The implementation of Elf [Pfe94], a logic programming language based on LF, contains a reconstruction algorithm that is similar to the one presented here in the sense that missing application arguments can be recovered both from the type of the application (inherited arguments) and from the other arguments of the same application (synthesized arguments). In fact, the Elf reconstruction algorithm is more powerful than the one discussed here because it does not impose any restriction on which terms can be missing from the proof. Elf goes as far as allowing the entire proof to be implicit, in which case it searches for a proof itself. To achieve this flexibility, Elf type reconstruction uses an algorithm based on constraint solving [Pfe91a, Pfe91b], where the constraints can survive the local context where they were introduced. Sometimes, there might be constraints that remain unsolved, in which case Elf declare failure to reconstruct the proof. The proof checking algorithm described here can be characterized as a special and more efficient case of the Elf's reconstruction algorithm, where enough of the proof structure is present so that (1) there is not need for search, (2) higher-order unification is reduced to a simple extension of first-order unification that respects bound variables, and (3) all constraints that are generated have the simple rigid-rigid or flex-rigid form that can be solved eagerly and locally.

The design of the language L_λ [Mil91] also relies on syntactic restrictions for the purpose of eliminating the need for higher-order unification during type checking. However, the restrictions of L_λ are too strict for our purposes because they prevent the free use of higher-order abstract syntax in the representation of predicates and proofs, requiring instead costly [MP92] explicit implementations of substitution. In LF_i we can still make use of all of the representation techniques of the full LF language, and thus gain leverage from substitution in the meta-language, because the restrictions are imposed only on which subterms might be missing from the representation.

Similar ideas to those presented in this chapter appear in [PT98] in the context of a simple, yet effective type-inference algorithm for a language with subtyping and impredicative polymorphism. The similarity lies in the fact that both algorithms rely on a combination of local constraint solving and bidirectional checking. However neither result subsumes the other because of the different characteristics of the underlying languages: LF has dependent types and the language considered in [PT98] has subtyping.

Finally, a distinguishing feature of the LF_i framework from the related work discussed above is the representation algorithm, which take a regular LF representation of a proof and eliminate as many subterms from it as possible, while keeping the resulting proof within the reconstruction capability of the proof checker. The experimental data supports the claim that the LF_i representation for first-order logic proofs has substantial practical benefits.

The bimodal representation algorithm presented in Section 5.4 has the advantage that most of the work can be done statically while loading the signature. Then, once the representation recipes have been computed, the actual representation algorithm is just a copy of the full LF representation with certain subterms dropped. However, the bimodal algorithm considers only two possible contexts for each application. One direction for improving the effectiveness of the representation is to consider each LF application in the actual context where it occurs. To experiment with this case I implemented a *global* representation algorithm that mimics the reconstruction algorithm to detect which subterms can be omitted. For each application the global algorithm starts assuming optimistically that all arguments can be dropped. Then it runs the reconstruction algorithm that might fail when checking whether all constraints have been checked (i.e, the check $Dom(\Delta) \subseteq Dom(\Psi)$ in the rule of type reconstruction for application might fail). Such a failure points to a given set of arguments that must be explicit. The global reconstruction algorithm makes these arguments explicit are tries again until it succeeds. In a sense, the global representation algorithm is optimal given a constraint solving order. However, the improvement in the size of proofs over the bimodal representation is so minimal, that it is not worth to pay the expense of the trial-and-error global representation algorithm. This also says that the bimodal representation achieves near-optimal results, at least for the fragments of first-order logic used in the experiments.

An even more effective reconstruction algorithm would improve on the global representation algorithm and also search for the best order of solving the constraints for each application separately. I did not experiment with this feature partly because it would require special notation to encode in the representation the order in which the constraints should be solved, and also because the visual inspection of the results of the bimodal algorithm suggests that there is not much left to be gained from more complex algorithms. If the order of constraint solving is important for a given constant then we can usually change the order of arguments in the type of each constant declaration so that it matches the optimal order. This is possible because, in general, it is not important in which order we mention the hypotheses of a derivation rule.

The dependency on the context of the bimodal representation algorithm is a disadvantage if we want to use it in a system that must generate LF terms incrementally, such as a theorem prover that builds an LF representation of the proof gradually from smaller components. The bimodal algorithm cannot be used to compact a component until it is known in which context the component is used. On the other had, it is wasteful to make the theorem prover emit fully-explicit representation of proofs only to pass them to the bimodal representation algorithm. What we need in this case is a context independent representation algorithm.

Fortunately, such an algorithm can be obtained with a simple modification of the bimodal algorithm, by making the restriction that only the "e_c" and "$*$" representation directives be used. This can be achieved by changing the rule for computing E^m_{n+1} in Equation 5.13 to:

$$E^m_{n+1} = \{x_1, \ldots, x_n\} - I^m_{n+1}$$

This will ensure that for all k we have that $FV(A_k) \subseteq I^c_{k+1} \cup E^c_{k+1}$, which in turn guarantees that the directive "e_i" will not be used. Thus we obtain a context-independent representation algorithm with just the checking mode. Each subterm of an object can be represented independently of the context and that makes it possible to have simple representation algorithms for use in programs that must generate LF terms. Note that the bimodal representation algorithms do not require that the input LF object be fully-explicit. If the object is missing only subterms that the representation algorithm was going to omit anyway, then the representation algorithm still works. This makes it possible to implement the theorem prover so that it emits proof representations using the less-effective context-independent representation and then run the bimodal representation algorithm to compress the proof further before sending it to the proof checker.

With this, I conclude Part I dedicated to the proof-carrying code infrastructure. I have shown, in Chapter 3, how a safety policy can be defined as a fictitious interpreter that verifies each action performed by the agent. Then, in Chapter 4, I have shown the proof-carrying code alternative to enforcing the safety policy, by means of a verification-condition generator that computes a predicate that is valid only if the fictitious interpreter would never fail one of the run-time checks. This justifies the PCC infrastructure to accept as evidence of safety, a valid proof of the verification condition predicate. Finally, in this last chapter, I describe a framework that can be used to encode verification conditions and their proofs such that they are relatively small and a simple logic-independent proof checker can validate them. In Part II I show a couple of tools that can be used on the code producer end to generate the annotations and the proofs that are required to satisfy a PCC code receiver.

Part II

Proof-Carrying Code Tools

In this second part of the dissertation I describe a pair of tools that a code producer can use to accomplish its share of the attributions in a PCC system. In Chapter 6 I describe a special class of safety policies based on type systems. For such type-based safety policies it is possible to generate the required loop invariants in a completely automatic manner. I describe the design of the Touchstone certifying compiler that, as it compiles and optimizes programs written in a type-safe subset of the C programming language, generates the loop invariants required by VCGen.

Then in Chapter 7 I describe a theorem prover for first-order logic that is both sufficiently powerful to prove all the verification conditions arising from the output of the Touchstone compiler and also capable of generating the proofs of such verification conditions in the LF_i format required by the PCC proof checker described in Chapter 5.

Chapter 6

The Touchstone Certifying Compiler

Part I of this dissertation describes in detail the infrastructure of proof-carrying code, consisting of a safety policy, implemented by means of a verification-condition generator and code specifications, along with a proof checker. These components are shown with interrupted lines in Figure 6.1. The other components, shown with solid lines, are part of the code producer.

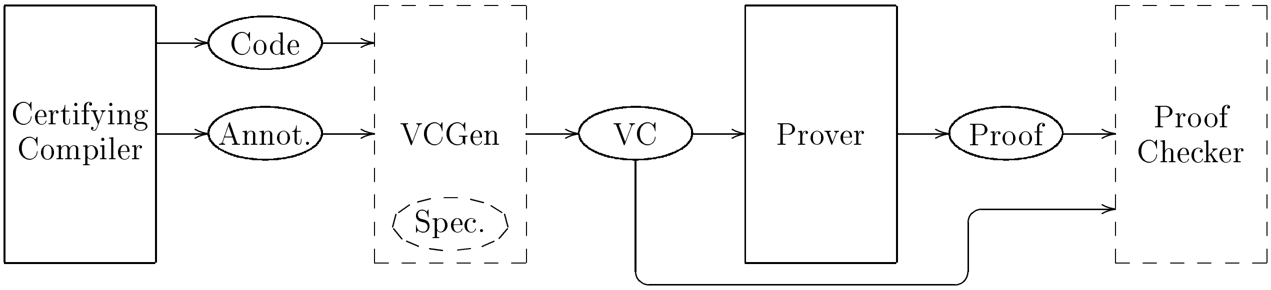


Figure 6.1: The interaction between the PCC tools (shown with continuous lines) and the PCC infrastructure (shown with interrupted lines).

I have shown that a successfully checked proof of the verification condition guarantees that the verification condition is valid, which in turn guarantees that the code adheres to the safety policy. The PCC infrastructure is simple, easy-to-trust and automatic. But this is only because all of the difficult tasks have been delegated to the code and proof producers. The first difficult task, besides writing code that is indeed safe, is to generate the code annotations consisting of loop invariants for all loops and of function specifications for all local functions. The other difficult task is to prove the verification condition produced by the verification-condition generator. This chapter describes one method for generating the annotated code, by means of a special compiler. Then, in Chapter 7, I describe a theorem prover capable of generating proofs of the verification conditions.

Of the two difficult tasks facing a code producer, the most important is the generation of the code annotations. This chapter starts with a general discussion of loop invariants, focussing on why it is so difficult in practice to find appropriate invariants. The discussion then hints that by restricting the safety properties to be checked to type safety, the problem of generating loop invariants is simplified considerably. Then, in Section 6.2, I discuss in detail what is type safety and, in particular, what is a type-based safety policy for proof-carrying code along with a series of examples showing how common safety policies can be expressed based on types. The actual description of the Touchstone certifying compiler starts in Section 6.3 with a discussion of the subset of the C programming language compiled by Touchstone. Then, Section 6.4 revisits the issue of automatic generation of loop invariants, this time in the concrete case of type safety for the Touchstone subset of C. The main body of this chapter is Section 6.5 where I discuss in turn several common optimizations and show how each of them must be adapted for use in a certifying compiler. Finally, the chapter ends with a discussion (in Section 6.6) of the software engineering benefits of the certifying compiler approach.

Note that this chapter, unlike the entire Part I of this dissertation, is targeted towards a rather narrow application of proof-carrying code, in which the safety properties to be checked is limited to type safety. Thus the numerous restrictions that are imposed in this chapter for the purpose of achieving automation of proof generation are not to be taken as necessary limitations of the underlying proof-carrying code infrastructure.

6.1 The Basics of Loop Invariants

The first of the two challenges facing a code producer is the generation of the code annotations. The second is the generation of proof. As we shall see, any systematic method for discovering annotations that are both valid and sufficient for proving safety must go through the same reasoning steps that a theorem prover would prove the verification conditions. This suggests that the task of generating annotations is more difficult than the theorem proving task. To illustrate this point let us discuss the problems of discovering the loop invariant and of proving the verification condition for the generic program fragment shown below:

$$
\begin{array}{ll}
 & i := e_0 \\
\text{Loop:} & \underline{\textbf{if}}\ C\ \underline{\textbf{then}}\ \underline{\textbf{return}}\ \ i \\
 & \ldots \\
 & i := e \\
 & \underline{\textbf{go to}}\ \text{Loop}
\end{array}
$$

This program fragment contains a single loop with the variable i initialized to the expression e_0 and updated with the value of the expression e. The loop termination condition is a predicate C. Let us assume that the precondition for this fragment is a predicate `Pre` and the postcondition another predicate `Post` and that the body of the loop (shown as ellipses in line 3) has a safety predicate P.

There are two annotations that the code producer is responsible for in this example, namely the loop invariant `Inv` and the set of modified registers, both for the loop starting in line 2. The latter is always trivial to compute. In this case it is the singleton set $\{i\}$. Before we even try to infer a loop invariant it is useful to compute the verification condition using the symbolic evaluation algorithm shown in Figure 4.6 on page 71. The result is as follows:

$$\begin{array}{rcl} VC & = & \texttt{Pre} \supset [{}^{e_0}\!/_i]\texttt{Inv} \wedge \\ & & \qquad (\forall i.\texttt{Inv} \supset ((C \supset \texttt{Post}) \wedge \\ & & \qquad\qquad (\neg C \supset (P \wedge [{}^{e}\!/_i]\texttt{Inv})))) \end{array}$$

The important points to notice are that the precondition is assumed for the whole predicate, and the invariant is assumed for both the loop body and the code following the loop. Notice also that the modified variable i is quantified so that no assumption can be made about its value in the body of the loop, except as specified by `Inv`. This invariant condition can be broken into four separate predicates, as shown below:

$$\texttt{Pre} \wedge \texttt{Inv} \wedge C \supset \texttt{Post} \tag{6.1}$$
$$\texttt{Pre} \wedge \texttt{Inv} \wedge \neg C \supset P \tag{6.2}$$
$$\texttt{Pre} \supset [{}^{e_0}\!/_i]\texttt{Inv} \tag{6.3}$$
$$\texttt{Pre} \wedge \texttt{Inv} \wedge \neg C \supset [{}^{e}\!/_i]\texttt{Inv} \tag{6.4}$$

The predicates 6.1 and 6.2 require the invariant to be sufficiently strong so that it implies both the safety of the loop body and the postcondition, and also the safety of the code following the loop if such code existed. These are the *strength conditions* on invariants and can be summarized by the following predicate:

$$\texttt{Inv} \supset \neg\texttt{Pre} \vee ((C \vee \texttt{Post}) \wedge (\neg C \vee P)) \tag{6.5}$$

The invariants must be not only strong, but also correct, as defined by the conditions 6.3 and 6.4. A loop invariant is correct if it holds every time the execution reaches the beginning of the loop. The predicate 6.3 requires that the invariant holds the first time the execution reaches the loop body and the predicate 6.4 requires that the invariant be preserved in one arbitrary iteration through the loop. If these predicates are both valid, then by a simple inductive argument we can see that the invariant must hold after an arbitrary number of iterations. Unlike the strength conditions, the correctness predicates cannot be reduced easily to a simple condition such as 6.5 above.

Note that the strength conditions and the correctness conditions are somewhat antagonistic because the invariant occurs on opposite sides of the implication. An illustration of this fact is that the invariant `true` satisfies the correctness conditions but not the strength conditions, while the predicate `false` has the opposite behavior. This means that the "right" invariant must be chosen carefully to be not too strong and not too weak. The problem of discovering the right invariant is undecidable in general. However, in many practical instances acceptable invariants can be discovered by iterative methods, such as abstract interpretation [CC77], starting from the invariant of 6.5 and strengthening it until it satisfies the correctness conditions.

It is important to realize that any algorithm for inferring loop invariants must posses the means to prove implicitly the conditions 6.1 to 6.4. This means that we should be able to extract from such an algorithm a proof generator capable of proving the resulting verification conditions. This suggests that it is more difficult and more important to discover the right loop invariants than it is to prove the verification conditions they determine. In this chapter I will show how to infer the loop invariants automatically for a restricted class of safety policies. Then, in the next chapter, I will show how we can derive a proof-generating decision procedure for the verification conditions.

I have argued informally that it is impossible to give a general algorithm for discovering the loop invariants. Therefore we must first impose sufficient restrictions on the form of the predicates `Pre`, `Post` and P, and thus on the safety policy, to simplify the invariant inference problem. There are two simple techniques that we can use. One is to insert run-time checks in the code and another is to restrict the predicates so that they can be verified in a syntax-directed manner, so that the substitutions occurring in the invariant correctness conditions do not pose any problems.

Let us consider the run-time checking method first. For this purpose, assume that the predicates P and `Post` are such that they can be implemented in the programming language used for the agent code, by means of two conditionals T^P and $T^{\texttt{Post}}$. In fact, these conditionals do not even need to be as strong as P and `Post` respectively, but just strong enough to make the following predicates provable:

$$\texttt{Pre} \wedge C \wedge T^{\texttt{Post}} \supset \texttt{Post} \tag{6.6}$$

$$\texttt{Pre} \wedge \neg C \wedge T^{P} \supset P \tag{6.7}$$

Then, we can rewrite our program with two additional conditionals to prevent the return or the execution of the loop body in the situations when the postcondition or the safety predicate of the body do not hold. One variant for the resulting code is shown below:

```
        i := e0
Loop:   if C then go to Done
        if ¬T^P then go to Idle
        ...
        i := e
        go to Loop
Done:   if T^Post then return i
Idle:   go to Idle
```

The added conditionals in lines 3 and 7 redirect the unsafe execution to an idling loop. This is acceptable because, as explained in detail on page 49, non-termination is considered safe behavior. In an actual implementation, instead of entering an idling loop the program would raise an exception that is then handled by the runtime system.

The effect of the added run-time checks can be explained formally on the modified verification condition, shown below:

$$\begin{array}{l}
\texttt{Pre} \supset [^{e_0}\!/_i]\texttt{Inv} \wedge \\
\quad \forall i.\texttt{Inv} \supset (C \supset ((T^{\texttt{Post}} \supset \texttt{Post}) \wedge \\
\qquad\qquad (\neg T^{\texttt{Post}} \supset (\texttt{Inv}_{\texttt{idle}} \wedge (\texttt{Inv}_{\texttt{idle}} \supset \texttt{Inv}_{\texttt{idle}})))) \wedge \\
\qquad \neg C \supset ((\neg T^{P} \supset (\texttt{Inv}_{\texttt{idle}} \wedge (\texttt{Inv}_{\texttt{idle}} \supset \texttt{Inv}_{\texttt{idle}}))) \\
\qquad\qquad (T^{P} \supset (P \wedge [^{e}\!/_i]\texttt{Inv}))))
\end{array}$$

Due to the conditions 6.6 and 6.7 the verification condition is provable even with the loop invariants `Inv` and $\texttt{Inv}_{\texttt{idle}}$ set to `true`. This example shows that by using run-time checking we can simplify dramatically both the task of inferring loop invariants and that of proving the verification conditions. If we use run-time checking, then the proof of the verification condition essentially certifies that the checking is done properly. In our case this means that the implications 6.6 and 6.7 are provable. But excessive use of run-time checking detracts from the advantages of proof-carrying code because of the run-time overhead and because, as discussed in Section 6.2, many code properties can be checked more conveniently statically.

The second technique that we can use to simplify the task of inferring loop invariants is to restrict the predicates of the safety policy so that the correctness conditions 6.3 and 6.4 can be reduced in a manner similar to the strength conditions. The complications are due to the substitutions "$[^{e_0}\!/_i]\texttt{Inv}$" and "$[^{e}\!/_i]\texttt{Inv}$". We notice that such substitutions arise from assignments in the agent program and thus, one idea is to restrict the assignments so that substitutions like these can be simplified. One possible restriction is suggested by typed high-level languages, where an assignment is allowed only if the type of the variable on the left-hand side agrees with that of the expression on the right-hand side. If we further restrict all of the predicates `Pre`, `Post` and P to be conjunctions of typing declarations for variables, then it becomes easier to infer the loop invariants. For example, `Inv` in our example can simply be the type declaration for the variable i, written for example as "$i : \tau$". Because of the typing rule for assignment, we know that "$[^{e_0}\!/_i]\texttt{Inv}$" and "$[^{e}\!/_i]\texttt{Inv}$" are both valid predicates whenever the precondition is valid.

In the next section I introduce a type system adequate for low level languages and I discuss in detail the implications of using type-based safety policies. Then, in Section 6.3 I show a type system for a high-level language derived from the C programming language. Then, the rest of this chapter is concerned with bridging the gap between the high-level language and its type system and the low-level type-based safety policy by means of a compiler that generates assembly language annotated with typing loop invariants.

Base types	τ^b	$::=$	`int` $\mid$ `bool` $\mid$ $\sigma*$ $\mid$ $\sigma?$ $\mid$ $\sigma[l]$	$(l > 0)$
Structured types	σ	$::=$	τ^b $\mid$ $\sigma_1 \times \sigma_2$ $\mid$ $\sigma[]$ $\mid$ $\mu t.\sigma$ $\mid$ t	
Memory types	τ^s	$::=$	`Mem`	

Figure 6.2: A type system for type-based safety policies.

6.2 Type-Based Safety Policies

In this section I describe a distinguished class of proof-carrying code safety policies that use type systems to classify the values stored in registers and in memory locations. There are several advantages to using type-based safety policies. First, they are extremely flexible and easy to configure. Once a type system is selected, many interesting safety policies can be easily obtained by simply declaring the types of values stored in certain registers and memory location. Even more safety polices can be obtained by varying the type system itself. Another major advantage of type-based safety policies is that, for many interesting type systems, the loop invariants can be generated automatically by a certifying compiler and the verification conditions can be proved using a simple theorem prover.

Before I proceed with a detailed discussion of type-based safety polices, it is useful to recall a few terms and notations introduced in Chapter 3. PCC safety policies are described as a series of safety predicates on the state of the execution of a fictitious interpreter for a generic assembly language called `SAL`. For the purposes of this section, I will consider that the `SAL` machine words span W bytes, and thus the universe of base values $\mathcal{U}^b$ consists of those integer values that can be represented in two's-complement notation on $8 * W$ bits. I also consider that addresses are represented on W bytes and each memory access references a memory word consisting of W consecutive bytes. With these assumptions a memory value m is a function from positive multiples of W to $\mathcal{U}^b$. Let this set of functions be $\mathcal{U}^s$. The superscripts of $\mathcal{U}^b$ and $\mathcal{U}^s$ are dropped when it is clear form the context which universe is meant.

Type-based safety policies are described in terms of a classification of register values by types in a given type system. To illustrate the concept, consider the types shown in Figure 6.2. The base types τ^b are used to classify the values stored in machine registers or in individual memory locations. The structured types σ classify the contents of sequences of memory locations. There is only one memory type, used to distinguish the memory states whose contents are well-typed, in a sense to be defined later in this section. The next few paragraphs describe informally the relationship between the various types and source-level types such as pointers and arrays.

The simplest base types are `int` and `bool`. A value is classified using one of these types if it is a valid integer value or, respectively, a valid boolean value. The class of "$\sigma*$" contains those values that are valid memory addresses pointing to a sequence of memory locations storing a structure of type σ. This type is also called a pointer type and is the assembly-language counterpart of source-level pointer types. To accommodate languages like Java and Modula-3, where source-level pointer values include a distinguished value called `null`, I introduce the pointer-option type, written as "$\sigma?$". Informally, this class contains all values of "$\sigma*$" and also the value representing `null`, which I consider to be the numeral zero. Before continuing the discussion of the other base types it is useful to examine the structured types.

The structured types are used to classify the contents of sequences of memory words. A special case is when the sequence contains only one memory word. In this case the contents of the sequence is a base type τ^b. Several consecutive memory words can be classified with the tuple type "$\sigma_1 \times \sigma_2$". For example, the tuple type "$\tau_1^b \times (\tau_2^b \times \tau_3^b)$" is used to classify a sequence of three memory words storing, in sequence, a value of type τ_1^b, then one of τ_2^b followed by a value of type τ_3^b. The array structured type "$\sigma[]$" denotes two memory words, of which the last one stores the address of an array whose elements are of the structured type σ and whose length is stored in the first memory word.

The recursive type "$\mu t.\sigma$" is just like σ but is allows recursive references to the entire type in σ. The bound type variable t marks such recursive references. The usual observations about bound variables, such as α-equivalence and substitution avoiding variable capture, apply to type variables introduced by recursive types. To illustrate the use of recursive types, consider the common representation of lists of boolean values as either `null` or as a pointer to a pair containing a boolean and a list of booleans. This type can be written in our type system as "$\mu t.((\texttt{bool} \times t)?)$".

Recursive types are restricted so that type variables occur only within a pointer, pointer option or array modifier. This means, among other things, that the body of a recursive type cannot be a type variable. This restriction can be specified precisely by defining the size of a structured type as the number of memory locations that it requires. The size of a type σ is written as "$|\sigma|$" and is defined as follows:

$$|\tau| = W \qquad |\sigma_1 \times \sigma_2| = |\sigma_1| + |\sigma_2| \qquad |\sigma[]| = 2W \qquad |\mu t.\sigma| = |\sigma|$$

Note that the size-of operator is not defined for type variables. Thus if a type variable occurred outside the scope of any pointer or array modifier the size-of operator would not be defined and the type itself would be invalid.

Returning to base types, the class of "$\sigma[l]$" contains those values that are valid memory addresses marking the beginning of a sequence of "l" memory locations storing vectors of structures of type σ. The constraint on the length parameter is that it must be positive and l must be a multiple of $|\sigma|$.

This informal description of the meaning of types is formalized by means of a representation function $\mathcal{R}$ that associates with each closed base type the subset of values in $\mathcal{U}$ that have that type. Not any such association is a valid representation, but only those that

a) $\mathcal{R}(\texttt{int}) = \mathcal{U}^b$

b) $\mathcal{R}(\texttt{bool}) = \{0, 1\}$

c) $\mathcal{R}(\sigma?) = \{0\} \cup \mathcal{R}(\sigma*)$

d) $0 \notin \mathcal{R}(\sigma*)$

e) If $v \in \mathcal{R}(\sigma*)$ then $v \bmod W = 0$ and $v + |\sigma| - W \in \mathcal{U}^b$

f) If $v \in \mathcal{R}(\tau*)$ and $v \in \mathcal{R}(\tau'*)$ then $\tau \equiv \tau'$

g) $\mathcal{R}((\mu t.\sigma)*) = \mathcal{R}(([^{\mu t.\sigma}/_t]\sigma)*)$

h) If $v \in \mathcal{R}(\sigma[l])$ then $v + l - W \in \mathcal{U}^b$ and for all i such that $0 \leq i < l$ and $i \bmod |\sigma| = 0$ we have that $v + i \in \mathcal{R}(\sigma*)$

i) If $v \in \mathcal{R}((\sigma_1 \times \sigma_2)*)$ then $v \in \mathcal{R}(\sigma_1*)$ and $v + |\sigma_1| \in \mathcal{R}(\sigma_2*)$

j) If $v \in \mathcal{R}(\sigma[]*)$ then $v \notin \mathcal{R}(\tau*)$ for all τ

k) $m \in \mathcal{R}(\texttt{Mem})$ if and only if

   (1) for all $v \in \mathcal{R}(\tau*)$ we have that $v \in \mathit{Dom}(m)$ and $m(v) \in \mathcal{R}(\tau)$, and

   (2) for all $v \in \mathcal{R}(\sigma[]*)$ we have that $v \in \mathit{Dom}(m)$ and $m(v) > 0$ and $(m(v)) \bmod |\sigma| = 0$ and $v + W \in \mathcal{R}(\sigma[m(v)]*)$.

Figure 6.3: The validity conditions for type representations.

satisfy the conditions shown in Figure 6.3. To denote that $\mathcal{R}$ is a valid representation we write $\mathit{Valid}(\mathcal{R})$.

In any valid representation $\mathcal{R}$ all of the values in the base universe are valid integers and a boolean value is either the numeral zero or one. A pointer option value is either zero (the representation of `null`) or else a valid pointer value. Condition 6.3(d) ensures that zero is never among valid pointer values. Condition 6.3(e) ensures that pointer values are multiples of the word size and are not too close to the end of the memory range. Condition 6.3(f) ensures that a given value cannot be a valid pointer to two different base types. Note that this condition is only required for base types and not for structured types. Condition 6.3(h) says that a value of array type points to a contiguous sequence of values of the element type. Condition 6.3(i) says that the components of structured types are laid out in consecutive memory locations. Open arrays are represented in memory as a length field and an base address field. Condition 6.3(j) says that values pointing to open arrays cannot be valid pointers to base types, to ensure that the length field cannot be written except when writing both fields.

Note that I take extreme care not to involve the contents of the memory in the conditions discussed so far. Condition 6.3(k) restricts the contents of memory by using two subconditions. Condition 6.3(k.1) says that any well-typed memory value must be defined at all addresses that are pointers to base types, and furthermore, the contents of the corresponding locations must be well-typed. Condition 6.3(k.2) specifies the representation of open arrays.

Note also that the representation function is defined precisely for some types, such as `int` and `bool`, and only partially for others, such as `int`$*$. For example, there is not a rule saying that "$m(v) \in \mathcal{R}(\texttt{int}) \supset v \in \mathcal{R}(\texttt{int}*)$". Such a rule would not be sound because it would allow the writing of any integer value in any accessible memory location. However, the converse of this rule is sound.

Chapter 3 defines a general PCC safety policy as having three components: instruction safety (i.e., restrictions on what instructions can be executed and in what conditions), system-call safety (i.e., restrictions on what receiver-provided functions can be called and in what conditions), and partial correctness (i.e., restrictions on the input/output behavior of agent functions specified as preconditions and postconditions). The instruction safety component is described as a series of safety predicates *SafeRd*, *SafeWr*, *SafeEOP*, and *SafeCOP*. The system-call and the partial correctness components of safety are described using a pair of precondition and postcondition predicates for each function.

According to Chapter 3, the predicate $\mathit{SafeRd}(m, a)$ must hold only if it is safe to read the memory word at address a in memory state m. Similarly, the predicate $\mathit{SafeWr}(m, a, v)$ must hold only if it is safe to write the value v at address a in memory state m. A typed-based safety policy defines a memory address to be accessible for read or write if and only if it has a pointer type, as follows:

$$\begin{array}{rll} \mathit{SafeRd}(m, a) & \text{iff} & \exists \sigma \text{ such that } a \in \mathcal{R}(\sigma *) \\ \mathit{SafeWr}(m, a, v) & \text{iff} & \exists \sigma \text{ such that } a \in \mathcal{R}(\sigma *) \end{array}$$

The definition of the *SafeWr* might be surprising at first because it does not restrict the type of the value being written. If, instead, we define $\mathit{SafeWr}(m, a, v)$ such that $v \in \mathcal{R}(\tau)$, whenever $a \in \mathcal{R}(\tau *)$, then it would be impossible to update safely locations containing structured types, such as "$\sigma[]$", where there is a dependency between the values of the components. To reconcile the liberal definition of *SafeWr* with type safety, each function postcondition must require that "$m \in \mathcal{R}(\texttt{Mem})$", where m is the value of the memory at the end of the function invocation. Informally, this constrains the memory values to be well-typed only at function boundaries and not after each instruction.

All of the type-based safety policies defined in this section include memory safety. This fact is stated formally below and it is proved easily from Condition 6.3(k) of validity of representations.

Theorem 6.8 (Soundness of type-based memory safety) *For any valid representation $\mathcal{R}$, and $v \in \mathcal{R}(\sigma*)$ then, in any well-typed memory state $m \in \mathcal{R}(\texttt{Mem})$ we have that v is an accessible memory address, i.e., $v \in Dom(m)$.*

It is obvious at this point that the safety predicates depend on a particular representation function $\mathcal{R}$. Furthermore, for our type system there are many valid representation functions and thus multiple safety policies. This property can be turned into a major advantage for the safety policy designer. By requiring that the agent code behaves safely with respect to *all* valid representation functions, the safety policy can effectively hide aspects of the concrete representation of types and expose just the abstract properties guaranteed to be true for all valid representations. Section 6.2.2 shows examples of type-based safety policy where abstraction is used.

Recall from Chapter 3 that an agent function must be proved safe for all input values of the memory state and registers that satisfy the precondition. This suggests a simple trick to enforce safety for all possible representation functions. We add a pseudo-register `repr` to the set of `SAL` registers and we consider that the current representation function is passed as an argument to the agent. The agent code cannot access this register directly and, in the absence of allocation, the representation function does not change during a given invocation of the agent. In the presence of allocation, a trusted allocation function will expand the representation $\mathcal{R}$ with the newly allocated value.

Let the set of `SAL` registers be $Regs = \{\mathbf{r}_1, \ldots, \mathbf{r}_R, \texttt{mem}, \texttt{repr}\}$, where R is the number of machine registers and `mem` is a special pseudo-register to hold the values of memory. The register state of the `SAL` interpreter is a function $\rho \in Regs \to \mathcal{U}$, such that $\rho(\mathbf{r}_i) \in \mathcal{U}^b$ and $\rho(\texttt{mem}) \in \mathcal{U}^s$. The state of the `repr` register is a representation function.

All of the preconditions and postconditions require that the representation be valid, written as $Valid(\rho(\texttt{repr}))$, and that the memory be well typed in the current representation, written as $\rho(\texttt{mem}) \in \rho(\texttt{repr})(\texttt{Mem})$. Additionally, they might require that the contents of various registers have certain types in the current representation.

As explained in Chapter 4, the safety predicates that are part of the safety policy are not used directly in the proof-carrying code infrastructure. They are only useful to define formally the safety policy and to prove the soundness of the proof-carrying code technique. Instead, proof-carrying code encodes the safety predicate as symbolic expressions in a logic and the uses a set of axioms and inference rules to verify symbolically that the safety predicates hold. Next section describes the syntax, the meaning and the axiomatization of the logic. Then, in Section 6.2.2 I discuss several examples of type-based safety policies.

6.2.1 A Logic for Type Safety

In order to express symbolically the predicates involved in a type-based safety policy I extend the fragment of first-order predicate logic shown in Figure 4.2 on page 64 with a

Extensions:

$$
\begin{array}{llcl}
\text{Predicates:} & P & ::= & \ldots \;|\; E : E^t \;|\; \mathtt{sizeOf}(E^t, E^b) \\
\text{Type expressions:} & E^t & ::= & \mathtt{int} \;|\; \mathtt{bool} \;|\; \mathtt{Mem} \;|\; \mathtt{ptr}(E^t) \;|\; \mathtt{ptropt}(E^t) \\
 & & | & \mathtt{array}(E^t, E^b) \;|\; \mathtt{openarray}(E^t) \;|\; \mathtt{pair}(E^t_1, E^t_2) \\
 & & | & \mathtt{mu}\ (\lambda t.E^t)
\end{array}
$$

Figure 6.4: The extensions required to the first-order logic shown in Figure 4.2 to handle types and type safety.

set of type expressions and a pair of additional predicates, as shown in Figure 6.4. The predicate "$E : E^t$" is the symbolic notation for "$E \in \mathcal{R}(E^t)$", for a given representation $\mathcal{R}$. The predicate "$\mathtt{sizeOf}(E^t, E^b)$" is valid when "$|E^t| = E^b$". The type expressions mimic the language of types defined in Figure 6.2. Note that the recursive type uses higher-order representation techniques to handle the bound type variable.

Following the model of Section 4.1.2, I set up a valuation function $\mathcal{V}^t$ mapping type expressions to types, as follows:

$$
\begin{array}{rlrl}
\mathcal{V}^t(\mathtt{int}) &= \mathtt{int} & \mathcal{V}^t(\mathtt{bool}) &= \mathtt{bool} \\
\mathcal{V}^t(\mathtt{ptr}(E^t)) &= (\mathcal{V}^t(E^t))* & \mathcal{V}^t(\mathtt{ptropt}(E^t)) &= (\mathcal{V}^t(E^t))? \\
\mathcal{V}^t(\mathtt{array}(E^t, E^b)) &= \mathcal{V}^t(E^t)[\mathcal{V}^b(E^b)] & \mathcal{V}^t(\mathtt{openarray}(E^t)) &= \mathcal{V}^t(E^t)[] \\
\mathcal{V}^t(\mathtt{pair}(E^t_1, E^t_2)) &= \mathcal{V}^t(E^t_1) \times \mathcal{V}^t(E^t_2) & \mathcal{V}^t(\mathtt{mu}\ (\lambda t.E^t)) &= \mu t.(\mathcal{V}^t(E^t)) \\
\mathcal{V}^t(t) & & &= t
\end{array}
$$

I also extend the validity relation for predicates as follows:

$$\models E : E^t \quad \text{iff} \quad \mathcal{V}(E) \in \mathcal{R}(\mathcal{V}^t(E^t))$$

Note that the validity of a typing predicate is with respect to a representation function $\mathcal{R}$. Technically, the typing predicate should also refer to the representation function. To simplify the notation I do not change the typing predicate and I assume that in any particular use of the logic, the representation function is an arbitrary valid representation.

As discussed in Chapter 4, the validity of verification conditions involving types and typing predicates is not verified directly, but instead it is derived using a set of axioms and inference rules. These rules consist of the axiomatization of first-order logic with equality shown in Figure 4.8 on page 77 along with the type-specific rules shown in Figure 6.5. In this latter set of rules, I use the constant `W` to stand for the machine-word size.

The set of inference rules shown in Figure 6.5 is split into four parts. In the first part there are the rules concerning pointer types. The rules `read` and `write` say that all addresses that

Pointer Types:

$$\frac{A : \mathtt{ptr}(T)}{\mathtt{saferd}(M, A)}\textbf{read} \qquad \frac{A : \mathtt{ptr}(T)}{\mathtt{safewr}(M, A, E)}\textbf{write} \qquad \frac{A : \mathtt{ptropt}(T) \qquad A \neq 0}{A : \mathtt{ptr}(T)}\textbf{ptropt}$$

$$\frac{M : \mathtt{Mem} \qquad A : \mathtt{ptr}(T) \qquad \mathtt{sizeOf}(T, \mathtt{W})}{\mathtt{sel}(M, A) : T}\textbf{sel} \qquad \frac{M : \mathtt{Mem} \qquad A : \mathtt{ptr}(\mathtt{openarray}(T))}{\mathtt{sel}(M, A + \mathtt{W}) : \mathtt{array}(T, \mathtt{sel}(M, A))}\textbf{seloa}$$

$$\frac{A : \mathtt{ptr}(\mathtt{pair}(T_1, T_2))}{A : \mathtt{ptr}(T_1)}\textbf{pairl} \qquad \frac{A : \mathtt{ptr}(\mathtt{pair}(T_1, T_2)) \qquad \mathtt{sizeOf}(T_1, S_1)}{A + S_1 : \mathtt{ptr}(T_2)}\textbf{pairr}$$

$$\frac{A : \mathtt{array}(T, L) \qquad \mathtt{sizeOf}(T, S) \qquad I \geq 0 \qquad I < L \qquad I \bmod S = 0}{A + I : \mathtt{ptr}(T)}\textbf{array}$$

Other Types:

$$\frac{}{E : \mathtt{int}}\textbf{inti} \qquad \frac{}{0 : \mathtt{bool}}\textbf{true} \qquad \frac{}{1 : \mathtt{bool}}\textbf{false} \qquad \frac{E_1 : \mathtt{bool} \qquad E_2 : \mathtt{bool}}{E_1 \,\&\, E_2 : \mathtt{bool}}\textbf{boolop}$$

$$\frac{}{\mathtt{mu}\ (\lambda t.T) = [{}^{\mathtt{mu}\ T}\!/_{t}]T}\textbf{mu} \qquad \frac{E' : T \qquad E = E'}{E : T}\textbf{ofcongr}$$

$$\frac{}{0 : \mathtt{ptropt}(T)}\textbf{null} \qquad \frac{A : \mathtt{ptr}(T)}{A : \mathtt{ptropt}(T)}\textbf{ptropti}$$

Memory updates:

$$\frac{M : \mathtt{Mem} \qquad A : \mathtt{ptr}(T) \qquad \mathtt{sizeOf}(T, \mathtt{W}) \qquad E : T}{\mathtt{upd}(M, A, E) : \mathtt{Mem}}\textbf{upd}$$

$$\frac{M : \mathtt{Mem} \qquad A : \mathtt{ptr}(\mathtt{openarray}(T)) \qquad E : \mathtt{array}(T, L)}{\mathtt{upd}(\mathtt{upd}(M, A + \mathtt{W}, E), A, L) : \mathtt{Mem}}\textbf{updoa}$$

Auxiliary:

$$\frac{}{\mathtt{sizeOf}(\mathtt{int}, \mathtt{W})}\textbf{szint} \qquad \frac{}{\mathtt{sizeOf}(\mathtt{bool}, \mathtt{W})}\textbf{szbool} \qquad \frac{}{\mathtt{sizeOf}(\mathtt{ptr}(T), \mathtt{W})}\textbf{szptr}$$

$$\frac{}{\mathtt{sizeOf}(\mathtt{array}(T, L), \mathtt{W})}\textbf{szarr} \qquad \frac{}{\mathtt{sizeOf}(\mathtt{ptropt}(T), \mathtt{W})}\textbf{szptropt}$$

$$\frac{\mathtt{sizeOf}(T_1, S_1) \qquad \mathtt{sizeOf}(T_2, S_2)}{\mathtt{sizeOf}(\mathtt{pair}(T_1, T_2), S_1 + S_2)}\textbf{szpair} \qquad \frac{}{\mathtt{sizeOf}(\mathtt{openarray}(T), 2 * \mathtt{W})}\textbf{szoarr}$$

Figure 6.5: The type-safety axioms.

have a pointer type (not necessarily pointer to base types) are safe to use in read and write operations. The `ptropt` rule says that non-null pointer options are valid pointers. Then there are two rules for inferring the types of values read from memory. The rule `sel` corresponds to Condition 6.3(k.1) and is used for pointer to base types while the rule `seloa` corresponds to Condition 6.3(k.2) and is used for pointers to open arrays. Note that Condition 6.3(j) and the hypothesis "$\texttt{sizeOf}(T, \texttt{W})$" ensures that the two rules do not apply both at the same time. The next two rules correspond to Condition 6.3(i). The last rule in this section allows breaking of an array into a series of pointers, as specified by Condition 6.3(h).

In the next section of Figure 6.5 there are the rules concerning the base types. The rule `boolop` shows how the type of assembly-language operators can be specified. The rule `mu` corresponds directly to Condition 6.3(g) and the last two rules in this section correspond to Condition 6.3(c).

The third section provides rules for proving that various memory update operations preserve the well-typedness of the memory as defined by Condition 6.3(k). Finally, the last section of Figure 6.5 specifies the rules for computing the size of the representation of a type.

Just as in Section 4.3 we have the obligation to verify the soundness of the axiomatization. This can be done by verifying for each rule that, given arbitrary valid representation and valuation functions, if the hypotheses of the rule are valid then the conclusion is also valid. In most cases, this is relatively easy to prove by using the validity conditions shown in Figure 6.3. Two more delicate cases are for the `upd` and `updoa` rules. As an illustration of how these proofs are done I sketch below the soundness proof for the rule `upd`.

Theorem 6.9 (Soundness of rule `upd` from Figure 6.5) *If $Valid(\mathcal{R})$ and if $m \in \mathcal{U}^s$, and a and e are such that $m \in \mathcal{R}(\texttt{Mem})$, $a \in \mathcal{R}(\tau *)$, and $e \in \mathcal{R}(\tau)$ then $m[a \mapsto e] \in \mathcal{R}(\texttt{Mem})$.*

PROOF: We have to prove that $m[a \mapsto e] = m'$ satisfies Condition 6.3(k). We do each part of the condition in turn:

1. Pick $v \in \mathcal{R}(\tau' *)$. We must prove that $m'(v) \in \mathcal{R}(\tau')$. If $v = a$ then, from Condition 6.3(f), we have that $\tau \equiv \tau'$ and hence $m'(v) = e \in \mathcal{R}(\tau')$. If $v \neq a$ then $m'(v) = m(v)$ and the desired conclusion follows from $m \in \mathcal{R}(\texttt{Mem})$.

2. Pick $v \in \mathcal{R}(\sigma[]*)$. We must prove that $m'(v) > 0$ and that $v + \texttt{W} \in \mathcal{R}(\sigma[m'(v)]*)$. From Condition 6.3(j) we infer that $v \neq a$ and hence that $m'(v) = m(v)$. Now the required conclusions follow from $m \in \mathcal{R}(\texttt{Mem})$.

□

Theorem 6.9 and similar theorems for the other rules of the axiomatization establish that it is sound to verify the validity of verification conditions by exhibiting a derivation of it using the inference rules given in Figure 6.5. In the next section, I describe a few examples of safety policies that can be described easily using the type system introduced in this section.

6.2.2 Examples of Type-Based Safety Policies

It is instructive to discuss several type-based safety policies based on the type system introduced above. Keep in mind that a safety policy is determined by the current representation that in turn is restricted by preconditions. In all of the examples considered here one of the restrictions is that the current representation is valid. Recall also that the agent must behave safely for any valid representation that matches the precondition, and in particular for the most restrictive representation that matches the preconditions. A representation $\mathcal{R}_0$ is more restrictive than another representation $\mathcal{R}$ if $\mathcal{R}_0(\tau) \subseteq \mathcal{R}(\tau)$ for all closed types τ. The most restrictive valid representation has the following properties:

$$\mathcal{R}_0(\sigma *) = \emptyset \quad \mathcal{R}_0(\sigma[l]) = \emptyset \quad \mathcal{R}_0(\texttt{Mem}) = \mathcal{U}^s$$

It can be seen that there is a unique valid representation with these properties.

No-memory-access policy. This safety policy disallows any memory operation outside the run-time stack area. To simplify the presentation, assume that the only agent entry point is the function `entry` that expects a boolean value in register $\mathbf{r}_1$. The "no-memory-access" safety policy can be established with the following precondition:

$$Pre_{\texttt{entry}}(\rho) \text{ if and only if } \begin{cases} \mathit{Valid}(\rho(\texttt{repr})) \\ \rho(\texttt{mem}) \in \rho(\texttt{repr})(\texttt{Mem}) \\ \rho(\mathbf{r}_1) \in \rho(\texttt{repr})(\texttt{bool}) \end{cases}$$

This precondition can be expressed symbolically in logic as "$\mathbf{r}_1 : \texttt{bool} \wedge \texttt{mem} : \texttt{Mem}$". Note that the fact that the representation must be is valid is left implicit. Any valid representation matches this precondition. Because the agent function `entry` must behave safely for all valid representations it must be also safe for $\mathcal{R}_0$ as defined above. But in this case the safety predicates *SafeRd* and *SafeWr* are empty and therefore, the only way the execution of `entry` can be safe is if it does not attempt to reference the memory.

Sandboxing. This safety policy allows read and write memory accesses as long as they are within a given memory range, say starting at address a (a non-zero multiple of `W`) and having length l (a strictly positive multiple of `W`) such that "$a + l - W \in \mathcal{U}^b$". To achieve this effect, the function `entry` is being passed in register $\mathbf{r}_1$ an array of integers, as specified by the precondition:

$$Pre_{\texttt{entry}}(\rho) \text{ if and only if } \begin{cases} \mathit{Valid}(\rho(\texttt{repr})) & \text{and} \\ \rho(\texttt{mem}) \in \rho(\texttt{repr})(\texttt{Mem}) & \\ \rho(\mathbf{r}_1) \in \rho(\texttt{repr})(\texttt{int}[l]) & \text{and} \\ \rho(\mathbf{r}_2) = l & \end{cases}$$

This precondition can be expressed symbolically in logic as "$\mathbf{r}_1 : \texttt{array}(\texttt{int}, \mathbf{r}_2) \wedge \mathbf{r}_2 > 0 \wedge \texttt{mem} : \texttt{Mem}$". The representation function $\mathcal{R}_0$, although valid, does not satisfy the above precondition. One particular representation that satisfies the precondition is $\mathcal{R}_1$ defined as follows:

$$\mathcal{R}_1(\sigma*) = \begin{cases} \{\, x \mid a \leq x \leq a + l - \texttt{W} \text{ and } x \bmod \texttt{W} = 0 \,\} & \text{if } \sigma \equiv \texttt{int} \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathcal{R}_1(\sigma[x]) = \begin{cases} \{a\} & \text{if } \sigma \equiv \texttt{int} \text{ and } x \equiv l \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathcal{R}_1(\texttt{Mem}) = \{\, m \in \mathcal{U}^s \mid \mathcal{R}_1(\texttt{int}*) \subseteq \mathit{Dom}(m) \,\}$$

If the representation is $\mathcal{R}_1$ then the precondition is satisfied by any register state such that $\rho(\texttt{repr}) = \mathcal{R}_1$ and $\rho(\mathbf{r}_1) = a$ and $\rho(\mathbf{r}_2) = l$. With this representation we have indeed that all accessible memory locations x are those in $\mathcal{R}_1(\texttt{int}*)$, which is exactly as we intended.

Abstract types. My last example of a type-based safety policy exploits to an even larger extent the capability of abstracting representation details by considering that the representation is passed as an argument and by requiring that the execution proceed safely for all valid representations. To demonstrate the power of abstraction consider a safety policy that requires each agent function to pass along a given token to all runtime functions that it invokes. This token is generated by the code receiver and passed to the agent when it starts executing. The safety policy will ensure by means of abstraction that the agent code passes the exact token that it has received from the receiver and not a forged one.

For this purpose, the safety policy designer extends the type system and the type expressions in the logic with an abstract base type "`token`", without changing the definition of the validity of representations. This effectively hides from the agent all information about the concrete representation of a token. The precondition for the `entry` function is:

$$Pre_{\texttt{entry}}(\rho) \text{ if and only if } \begin{cases} \mathit{Valid}(\rho(\texttt{repr})) & \text{and} \\ \rho(\texttt{mem}) \in \rho(\texttt{repr})(\texttt{Mem}) & \\ \rho(\mathbf{r}_1) \in \rho(\texttt{repr})(\texttt{token}) & \end{cases}$$

This precondition can be expressed symbolically in logic as "$\mathbf{r}_1 : \texttt{token} \wedge \texttt{mem} : \texttt{Mem}$". The agent code can invoke the runtime function `runtime` with the same token that it received in input. The precondition of the `runtime` function is:

$$
Pre_{\texttt{runtime}}(\rho) \text{ if and only if } \begin{cases} \mathit{Valid}(\rho(\texttt{repr})) & \text{and} \\ \rho(\texttt{mem}) \in \rho(\texttt{repr})(\texttt{Mem}) \\ \rho(\mathbf{r}_1) \in \rho(\texttt{repr})(\texttt{token}) \end{cases}
$$

One of the most restrictive representations satisfying the above preconditions can be obtained by assuming that the representation of tokens is a singleton set containing a value v. The corresponding representation is $\mathcal{R}_2$ defined as follows:

$$
\mathcal{R}_2(\sigma*) = \emptyset \quad \mathcal{R}_2(\sigma[l]) = \emptyset \quad \mathcal{R}_2(\texttt{token}) = \{v\} \quad \mathcal{R}_0(\texttt{Mem}) = \mathcal{U}^s
$$

If ρ_0 is the register state at the start of the agent execution such that $\rho_0(\texttt{repr}) = \mathcal{R}_2$ and $\rho_0(\mathbf{r}_1) = v$, then assume that the agent eventually tries to invoke the `runtime` function with a state of registers ρ satisfying the precondition of `runtime`. We know that the agent function cannot change the value of the `repr` register. Thus, $\rho(\texttt{repr}) = \mathcal{R}_2$, which implies that $\rho(\mathbf{r}_1) = v$ because v is the only member of $\rho(\texttt{repr})(\texttt{token})$. This shows how type safety can enforce abstraction.

If we wanted instead to avoid types and use only run-time checking to enforce the same safety policy, the concrete representation of `token` must be restricted so that a run-time check can identify whether a given value is a member of the abstract type. For example, an implementation of `runtime` that does not use type safety can use instead costly cryptographic functions to detect with high probability if the given token is an unforged member of the abstract type.

The type system described in this section is appropriate for low-level languages but not for high-level languages. In the next section I describe how a the type system of a type-safe subset of the C programming language can be translated to typing predicates defined in this section.

6.3 The Safe-C Source Language

In the first part of this chapter I have suggested that by using safety policies that are inspired from high-level type systems we can simplify considerably the task of generating loop invariants. Also, in Section 6.2 I have shown the details of a type-based safety policy adequate for low-level languages. In the rest of this chapter I will present a typed high-level language and a method for bridging the abstraction gap between the high-level type system and the low-level type-based safety policy by means of a certifying compiler called Touchstone.

The high-level language considered here is a type-safe subset of the C programming language. The notable features of C that are missing are memory deallocation and the address-of operator (&). These features can be misused to generate unchecked run-time errors. Also, to simplify the implementation I have not implemented source language features that are type-safe such as floating-point operations and function pointers. The language compiled by Touchstone has certain features not present in the standard C language such as exceptions, a `length` operator for array expressions and built-in array bounds checking and null-pointer checking. The built-in run-time checks are needed to ensure the type safety of the compiled code. The purpose of the `length` operator is to enforce read-only access to the length component of an array and exceptions are necessary for a clean implementation of the run-time checks. The core of the Touchstone language is shown in Figure 6.6.

The main goal of the language selection process was to keep it as close as possible to the C programming language and at the same time to ensure that no execution can lead to unchecked run-time errors. Checked run-time errors, such as array-bounds violations, are allowed and a flexible exception mechanism is provided to handle them.

A Touchstone program consists of a series of top-level declarations. These can be function declarations, global variable declarations and structured type declarations. Note that global variables must always have an initializer. The declarations simply associate types to identifiers. Although not shown here the compiler supports more complex forms of declarations allowing initializers and multiple declarations for a single type. At the level of types, the notable omissions are the function and union types. Also, to simplify the parser type abbreviations are not allowed except for global structured types.

At the level of expressions a few language constructs require further explanation. The construct "$\texttt{new}\,(\mathit{Typ}, \mathit{Exp}_1, \mathit{Exp}_2)$" allocates on the heap an array consisting of Exp_1 elements of type Typ, initialized to the value of Exp_2. If Typ is a structured type, then an initializer for each field is required. The predefined exception `Subscript` is thrown as part of a failed array bounds check and the `Nil` exception is thrown as part of a failed null-pointer check. There is no source-level construct for throwing these exceptions, although they can be handled by using a `try`...`catch` construct. The compiler also supports the derived form `try`...`finally` where the statement after the `finally` keyword is guaranteed to be executed before the normal or exceptional termination of the execution of the construct.

In addition to the core constructs for l-values, the compiler also supports several derived forms. For example, "$*\mathit{Lval}$" is a synonym for "$\mathit{Lval}[0]$" and "$\mathit{Lval} \to \mathit{Id}$" is a synonym for "$(\mathit{Lval}[0]).\mathit{Id}$".

For brevity I will not show the formal type checking and compilation rules for the selected subset of the C programming language. Instead I enumerate below the few aspects of compilation that are different from a usual complier for the C programming language.

Let us start with the compilation of variable and field declarations. No code is generated in this case. Instead new intermediate-language variables, henceforth named temporaries, are created to hold the values of the source-level variables. The number of temporaries needed depends on the type of the expression as follows. For an expression of type `int` or

$$
\begin{array}{lcl}
\mathit{Top} & ::= & \mathit{Typ}\ \mathit{Id}(\mathit{Decl}_1, \ldots, \mathit{Decl}_n)\ \{\mathit{Decls}\ \mathit{Stm}_1; \ldots; \mathit{Stm}_n\} \\
 & & \mid\ \mathit{Typ}\ \mathit{Id} = \mathit{Exp} \\
 & & \mid\ \texttt{typedef struct}\ \mathit{Id}\{\mathit{Decl}_1; \ldots; \mathit{Decl}_n\} \\
\mathit{Decl} & ::= & \mathit{Typ}\ \mathit{Id} \\
\mathit{Typ} & ::= & \texttt{int} \mid \texttt{bool} \mid \mathit{Typ}[] \mid *\mathit{Typ} \mid \texttt{struct}\ \mathit{Id} \\
\mathit{Exp} & ::= & \mathit{Lval} \mid n \mid \texttt{true} \mid \texttt{false} \mid \mathit{Exp}_1\ \mathit{Binop}\ \mathit{Exp}_2 \mid \mathit{Id}(\mathit{Exp}_1, \ldots, \mathit{Exp}_n) \\
 & & \mid\ \texttt{new}(\mathit{Typ}, \mathit{Exp}_1, \mathit{Exp}_2) \mid \mathit{Exp}.\texttt{length} \\
\mathit{Binop} & ::= & + \mid - \mid * \mid == \mid \neq \mid \geq \mid > \mid \leq \mid < \mid \& \mid \text{“}|\text{”} \mid \&\& \mid \text{“}||\text{”} \\
\mathit{Lval} & ::= & \mathit{Id} \mid \mathit{Lval}[\mathit{Exp}] \mid \mathit{Lval}.\mathit{Id} \\
\mathit{Stm} & ::= & \{\mathit{Decls}\ \mathit{Stm}_1; \ldots; \mathit{Stm}_n\} \mid \texttt{return}\ \mathit{Exp} \mid \mathit{Lval} = \mathit{Exp} \\
 & & \mid\ \mathit{Id}(\mathit{Exp}_1, \ldots, \mathit{Exp}_n) \mid \texttt{if}(\mathit{Exp})\ \mathit{Stm}_1\ \texttt{else}\ \mathit{Stm}_2 \mid \texttt{while}\ (\mathit{Exp})\ \mathit{Stm} \\
 & & \mid\ \texttt{for}\ (\mathit{Stm}_1; \mathit{Exp}; \mathit{Stm}_2)\ \mathit{Stm}_3 \mid \texttt{break} \mid \texttt{continue} \\
 & & \mid\ \texttt{try}\ \mathit{Stm}_1\ \texttt{catch}\ \mathit{Exn}\ \mathit{Stm}_2 \mid \texttt{try}\ \mathit{Stm}_1\ \texttt{finally}\ \mathit{Stm}_2 \\
\mathit{Exn} & ::= & \texttt{Subscript} \mid \texttt{Nil}
\end{array}
$$

Figure 6.6: The abstract syntax of the Touchstone subset of C.

`bool` or $*T$ for some T, only one temporary is required. For an expression of type $T[]$ two temporaries are used, one to hold the length of the array and one the base address. Finally, for a structured type the number of temporaries is the sum of the number of temporaries required to hold each of the fields. Thus, for example, a declaration “`struct foo` x”, where the structured type `foo` is declared as “`typedef struct foo`$\{T_1\ a; T_2\ b\}$”, is equivalent to the pair of declarations “$T_1\ x_a; T_2\ x_b$”. The process is repeated recursively until the types involved in the declaration are base types. The compilation of declarations is discussed in more detail in the next section.

The compilation of expressions results in code that computes the value of the expression in one or more temporaries. The return type of a function is not restricted to be a base type, and thus functions can have multiple return values. This is in fact required in order to compensate for the absence of the “address-of” operator of the C programming language that is sometimes used to return multiple values from a function. The compilation of an expression “E.`length`” consists of the code to compute E in two temporaries after which the temporary holding the length is moved to the temporary selected to hold the value of the whole expression. The result of the `new` expression is an array and thus there are two temporaries required to represent it. The result of the compilation in this case consists of a call to an allocation function provided by the runtime followed by an initialization loop.

The compilation of an *Lval* is done in two stages. In the first stage Touchstone computes a set of temporaries or memory addresses as follows. For a variable, this is the set of temporaries associated with the variable when its declaration was compiled. If *Lval* has a structured type then the form *Lval.Id* can be used, in which case the set of temporaries or memory addresses computed for *Lval* is restricted to those used to hold the field called *Id*. Finally, if the *Lval* has an array or pointer type then the form *Lval*[*Exp*] can be used. In this case the value of the expression *Exp* is multiplied by a compile-time constant representing the size of the base type of the array. The result is added to the variable or memory address representing *Lval* and the resulting memory address is returned. If the type of *Lval*[*Exp*] is not a base type then several consecutive memory addresses are returned.

The second stage of compilation of *Lval* depends on whether the *Lval* is used as an expression or on the left of an assignment. In the former case, the memory addresses computed by the first phase are read and the results put in the set of temporaries selected to hold the value of the expression. In the latter case, the expression to be assigned is compiled and the resulting temporaries are written to the set of addresses computed for *Lval* in the first phase. In either case, the set of memory operations is preceded by two bounds checks, one verifying that the first array index in the set is greater or equal to zero and the other verifying the the last index is less than the length of the array. The failure case is set to throw the `Subscript` exception using a mechanism discussed below. If the memory addresses are obtained from a pointer and not from an array then one run-time check is inserted to verify the the pointer value is non-zero. Otherwise, the exception `Nil` is raised.

Finally, the compilation of statements results in code. During compilation three labels are maintained for the destination of a `break` statement, a `continue` statement and the current exception handler. The first two destinations are set to the beginning of the innermost enclosing loop while the last destination is set to the innermost enclosing `catch` statement. The compiler rejects programs with `break` or `continue` statements occurring outside the body of all loops and it wraps the body of all functions with a `try`...`catch` statement that aborts the execution (by calling a special run-time function) if an unhandled exception is thrown. This simplifies the compiler considerably but it means that exceptions must be handled in the functions that throw them.

With this superficial discussion of the Touchstone compilation process we are prepared to consider now the issue of automatic loop-invariant generation. We discuss first the simplest and most general form of loop invariants and then we consider the additional complications raised by various optimizations and code generation techniques.

6.4 Automatic Generation of Loop Invariants

The discussion from the beginning of this chapter suggests that, if we limit the function specifications and safety predicates to typing predicates, then the loop invariants are just predicates equivalent to type declarations for the modified variables of a loop. It is obviously easy to detect at the level of the source language presented in the previous section what variables are modified inside a loop and furthermore what source-level types they have. On

$$
\begin{array}{lcll}
\mathcal{C}^D(\texttt{int}\ v) & = & \{v_1 : \texttt{int}\} & \\
\mathcal{C}^D(\texttt{bool}\ v) & = & \{v_1 : \texttt{bool}\} & \\
\mathcal{C}^D(T[]\ v) & = & \{v_1 : \texttt{int}, v_2 : \sigma[v_1]\} & \text{where } \mathcal{C}^T(T) = \sigma \\
\mathcal{C}^D(*T\ v) & = & \{v : \sigma?\} & \text{where } \mathcal{C}^T(T) = \sigma \\
\mathcal{C}^D(\texttt{struct}\ id\ v) & = & \bigcup_{i=1..n} \mathcal{C}^D(T_i\ v_i) & \text{where } \texttt{typedef struct}\ id\{T_1\ id_1, \ldots, T_n\ id_n\} \\
 & & & \\
\mathcal{C}^T(\texttt{int}) & = & \texttt{int} & \\
\mathcal{C}^T(\texttt{bool}) & = & \texttt{bool} & \\
\mathcal{C}^T(T[]) & = & \mathcal{C}^T(T)[] & \\
\mathcal{C}^T(*T) & = & \mathcal{C}^T(T)? & \\
\mathcal{C}^T(\texttt{struct}\ id) & = & \mu t.(\sigma_1 \times \cdots \times \sigma_n) & \text{where } \texttt{typedef struct}\ id\{T_1\ id_1, \ldots, T_n\ id_n\} \\
 & & & \text{and } \mathcal{C}^T([{}^{t}\!/\!_{\texttt{struct}\ id}]T_i) = \sigma_i
\end{array}
$$

Figure 6.7: The compilation rules for types are variable declarations.

the other hand, Section 6.2 describes an extension of first-order predicate logic consisting of typing predicates in a type system similar but not identical to that of the high-level language.

We start this section with a discussion of how high-level types are mapped to low-level types. For this purpose I define formally a compilation procedure that was discussed informally in the previous section, namely the compilation of declarations and of types. The former is defined as a function $\mathcal{C}^D(T\ v)$ that compiles the declaration of the source-level variable v to a list of declarations for temporaries. Each element of the returned list is of the form "$v_i : \tau_i^b$", where v_i is a fresh temporary and τ_i^b is a base type as defined precisely in Section 6.2. The compilation of types is defined as a function $\mathcal{C}^T$ that given a source-level type returns a corresponding structured type σ also defined precisely in Section 6.2. These compilation functions are defined in Figure 6.7.

Notice that source-level pointer types are compiled to pointer-option types in the low-level language. Notice also that structured types are compiled to recursive types. The resulting recursive types are valid because the C language restricts the definition of recursive types in a manner similar the restrictions imposed by the type-based safety policy. To simplify the presentation in the rest of this chapter I am going to consider that a single temporary is generated when compiling the declaration of a variable i, and furthermore I am going to use the same name i for that temporary. This will allow us to say that the source-level declaration $T\ i$ is compiled as the predicate "$i : \tau$" in the intermediate language. This notation can be easily generalized to the case when multiple temporaries correspond to a source-level variable by replacing the typing predicate with conjunctions of such typing predicates.

Let us reconsider the simple program shown on page 116. Recall that the strength and correctness conditions were as shown below, where `Pre` is the precondition predicate, `Post` is the postcondition predicate, P is the safety predicate for the body of the loop, `Inv` is the loop invariant and C is the predicate that denotes the loop termination condition. The subscript i marks the predicates that can depend on i.

$$\begin{array}{rcl} \texttt{Pre} \wedge \texttt{Inv} \wedge C & \supset & \texttt{Post} \\ \texttt{Pre} \wedge \texttt{Inv} \wedge \neg C & \supset & P \\ \texttt{Pre} & \supset & [^{e_0}/_i]\texttt{Inv} \\ \texttt{Pre} \wedge \texttt{Inv} \wedge \neg C & \supset & [^{e}/_i]\texttt{Inv} \end{array}$$

Now let us further consider that this code fragment is the body of a function `foo` with the prototype "T `foo`$(T_1\ x_1, \ldots, T_n\ x_n)$" and which returns the value of the variable i immediately following the loop. This function must have declared the local variable i using a declaration of the form "$T\ i$" or otherwise it would not be possible to return the value of i from the function. From here we obtain the following forms for the predicate `Pre`, `Post` and P, where τ_k is the low-level counterpart of the high-level type T_k and τ is the low-level counterpart of T.

$$\begin{array}{rcl} \texttt{Pre} & = & x_1 : \tau_1 \wedge \ldots \wedge x_n : \tau_n \\ \texttt{Post} & = & i : \tau \\ P & = & i : \tau \wedge x'_1 : \tau'_1 \wedge \ldots x'_m : \tau'_m \end{array}$$

The form of the predicate P is such as above because we are restricting the safety policy to be based only on types. Thus, if i is mentioned in P it must be as part of a predicate "$i : \tau$" or otherwise the source program would not have passed the type checking phase. Furthermore, each "$x'_k : \tau'_k$" must be one of the typing predicates that constitute the precondition. And finally, the well-typedness of the source program implies that the following predicates are provable:

$$\begin{array}{rcl} x_1 : \tau_1 \wedge \ldots \wedge x_n : \tau_n & \supset & e_0 : \tau \\ i : \tau \wedge x_1 : \tau_1 \wedge \ldots \wedge x_n : \tau_n & \supset & e : \tau \end{array}$$

From these facts it is easy to see that the invariant $\texttt{Inv} = i : \tau$ satisfies both the strength and the correctness conditions and it therefore a good invariant. This suggests that it is sufficient for an invariant to be the conjunction of the typing predicates for the modified variables of the loop. Both the set of modified variables and their types are known to the compiler which means that it is very easy for Touchstone to emit the necessary loop invariants. Furthermore, it is easy to see on the above example that proving the resulting verification conditions is also going to be easy. This is in fact true even when Touchstone performs simple code transformations. But there are also optimizations that require more complicated loop invariants. The interaction between the code transformations and the necessary loop invariants is explored in the next section.

6.5 Touchstone Optimizations and the Invariants

In the absence of optimizations the loop invariants that must be generated by Touchstone are limited to typing predicates for the modified variables of the loop. Furthermore, this remains true for many common optimizations and code transformations. In this section, I examine in turn several code transformations that a compiler might want to perform while optimizing or during code generation. For each optimization I will discuss whether any additional invariants must be generated and I will describe techniques that can be used to infer such additional invariants. This section is intended as a guide for the certifying-compiler writer interested in finding in a systematic way the loop invariants that must be emitted for each optimization.

In the rest of this section I will use the notation "...P..." to refer to a sequence of instructions whose verification condition is the predicate P. This predicate might depend on all variables that are live at the entrance of the code sequence. For example, the sequence "$i = e; r = i;$ `return`" can be abbreviated using the notation "...$[{}^{e}\!/_{r}]$`Post`..." where `Post` is the postcondition of the current function written in terms of the dedicated variable r that must hold the return value. Each code transformation discussed in this section is analyzed from the point of view of the modifications it generates in the verification condition of a generic fragment of code.

In this section we will need a series of properties of verification conditions, stated below as Property 6.10 and proved using the definition of the verification conditions from Section 4.2.

Property 6.10

1. *The verification condition for the code "$i = e; ...P...$" is $[{}^{e}\!/_{i}]P$.*
2. *If we replace the expression e for all occurrences of the variable i up to and including the first assignment of i in the sequence of instructions "...P...", then the verification condition for the resulting sequence of instructions is "$[{}^{e}\!/_{i}]P$". Hence the resulting sequence of instructions can be abbreviated as "...$[{}^{e}\!/_{i}]P$...".*

The prevailing argument used throughout this section is that if the verification condition of the transformed code is identical to that of the original code, then no additional loop invariants and also no modifications to the theorem prover are necessary. If, however, the verification condition does change then we explore first if it remains logically equivalent to the original one, in which case no additional invariants are necessary although the theorem prover might require changes to be able to prove the equivalence. The third and more difficult case is when the resulting verification condition is not even equivalent to the original one. In this case, we explore what additional invariants are required to preserve the equivalence and to reduce the problem to the second case mentioned above.

Let us now consider several code transformations, starting first with those that fall within the case of unchanged verification conditions. Then, starting with Section 6.5.7, I will discuss a couple of optimizations where additional work is required in the area of loop invariants.

6.5.1 Dead-Code Elimination

Dead-code elimination is a code transformation that removes from the program those instructions that are statically not reachable. A common situation is that shown on the left side of the table below, where the code on line 3 is not reachable.

Before		After	
Code	VC	Code	VC
1 $i = e_1$ 2 <u>**go to**</u> L 3 $i = e_2$ 4 L: ...P...	$[^{e_1}/_{i}]P$	$i = e_1$ L: ...P...	$[^{e_1}/_{i}]P$

One obvious possibility for optimizing this program fragment is to eliminate the dead code altogether, obtaining the code shown on the right side of the table. The table also shows the verification condition of the code before and after the optimization. We notice that the verification condition did not change during the optimization. This means that if the prover was able to prove it before the optimization, it will be able to prove it after the optimization as well, without any additional loop invariants or changes to the theorem prover.

The verification-condition generator is able to "see" beyond syntactic features of the code such as manifestly unreachable code. Because of this property, the invariance of the verification condition can serve as an effective criterion for the correctness of many code transformations.

Dead-code elimination is an important optimization not only for the purpose of eliminating code that was manifestly unreachable in the source program, but also for code that becomes unreachable during optimizations. In fact, we shall revisit dead-code elimination when discussing redundant-conditional removal in Section 6.5.8. This optimization removes conditionals that can be proved statically to always take one of the branches. The removal of the conditional instruction exposes the untaken branch as dead code.

6.5.2 Common-Subexpression Elimination

Common-subexpression elimination (CSE) is a program transformation whereby repeated computation of the same expression is avoided by reusing the results of the first computation.

We ignore here the static analysis that detects which subexpressions can be eliminated. A simple but representative case of CSE is shown on the left of the table below, where the static analysis has determined that e_1 is a common subexpression. Note that the expression e_1 is common in this example only if the variable i does not occur in e_1. In this case we can eliminate the second computation of e_1 as shown at the right of the following table.

Before		After	
Code	VC	Code	VC
1 $i = e_1$ 2 $j = e_1$ 3 $...P...$	$[^{e_1}/_{i}, ^{e_1}/_{j}]P$	$i = e_1$ $j = i$ $...P...$	$[^{e_1}/_{i}, ^{e_1}/_{j}]P$

If i does not occur in the expression e_1 then the verification condition of the code before the optimization is "$[^{e_1}/_{i}]([^{e_1}/_{j}]P) = [^{e_1}/_{i}, ^{e_1}/_{j}]P$". The verification condition after the optimization is "$[^{e_1}/_{i}][^{i}/_{j}]P = [^{e_1}/_{i}, ^{e_1}/_{j}]P$". Again the verification condition is preserved through the optimization, so CSE is another optimization that does not require anything special from a certification point of view.

Another optimization is suggested by the code resulting from CSE in our example. The copy propagation optimization discussed next attempts to eliminate the assignment instruction "$j = i$".

6.5.3 Copy Propagation

Consider the code resulting from the common-subexpression elimination example discussed in the previous section. Copy propagation is a code transformation where an assignment of a variable i to another variable j is eliminated and all code following the assignment up to the next reassignment of j is changed such that all references to j are replaced with references to i. This optimization is shown in the following table:

Before		After	
Code	VC	Code	VC
1 $i = e_1$ 2 $j = i$ 3 $...P...$	$[^{e_1}/_{i}, ^{e_1}/_{j}]P$	$i = e_1$ $...[^{i}/_{j}]P...$	$[^{e_1}/_{i}, ^{e_1}/_{j}]P$

Here the assignment from line 2 can be eliminated and all occurrences of j in the code that follows are replaced with i. The verification condition of the resulting code is $[^{e_1}/_{i}]([^{i}/_{j}]P) = [^{e_1}/_{i}, ^{e_1}/_{j}]P$. This means that copy propagation is yet another optimization that does not require special treatment.

6.5.4 Instruction Scheduling

Instruction scheduling is a code transformation that reorders unrelated instructions in an attempt to maximize the utilization of a processor's pipeline. Let us ignore now how the compiler decides which instructions to reorder because this depends on the characteristics of the target architecture. Instead we assume that the compiler has determined that it is beneficial for the lines 1 and 2 in the program shown below to be reordered. This operation is valid only if there is no dependency between the two instructions, which in this case means that $i \notin FV(e_2)$ and that $j \notin FV(e_1)$.

Before		After	
Code	VC	Code	VC
1 $i = e_1$ 2 $j = e_2$ 3 $...P...$	$[{}^{e_1}/_{i}, {}^{e_2}/_{j}]P$	$j = e_2$ $i = e_1$ $...P...$	$[{}^{e_1}/_{i}, {}^{e_2}/_{j}]P$

The verification condition of the code after the transformation is $[{}^{e_2}/_{j}]([{}^{e_1}/_{i}]P)$, which is equal to $[{}^{e_1}/_{i}, {}^{e_2}/_{j}]P$ if there is no dependency between the instructions. This means that instruction scheduling is another code transformation that preserves the verification conditions.

6.5.5 Register Allocation

Register allocation is a code transformation that is required during the code generation phase to map temporary names to machine registers. The difficulty lies in the large number of temporaries that the intermediate-language form of the program uses and the relatively small number of physical machine registers available. Many of the temporaries are introduced by the compiler to simplify the implementation of expression evaluation and calling conventions. In such instances, most of the temporaries are live for a short number of instructions and can therefore share the same machine registers with other similar temporaries. In the most fortunate case, the maximum number of live registers at any given instruction is less than the number of available machine registers. In this case register allocation is simply a renaming of temporaries with register names.

Let us consider this simple case first, as shown on the left of the table below:

Before		After	
Code	VC	Code	VC
1 $i = e$ 2 $j = e'$ 3 $...P...$	$\left[{}^{[{}^{e}/_{i}]e'}/_{j}\right] P$	$\mathbf{r}_j = [{}^{\mathbf{r}_i}/_{i}]e$ $\mathbf{r}_j = [{}^{\mathbf{r}_j}/_{i}]e'$ $...[{}^{\mathbf{r}_j}/_{j}]P...$	$[{}^{\mathbf{r}_i}/_{i}](\left[{}^{[{}^{e}/_{i}]e'}/_{j}\right] P)$

Let us further assume that the temporary i occurs in both of the expressions e and e' and that i does not occur in the code following line 2. In this case we say that i is a dead register after line 2. (This also means that i does not occur in the predicate P.) I further assume that an aggressive register allocator allocates the machine register $\mathbf{r}_i$ to hold the variable i before line 1, register $\mathbf{r}_j$ to hold the variable i from line 1 to line 2 and the same register $\mathbf{r}_j$ to hold the value of j in the code following line 2. Note that two different registers have been allocated to i for the two disjoint live ranges.

The verification condition of the code after the renaming is $[{}^{[{}^{[{}^{\mathbf{r}_i}/_{i}]e}/_{i}]e'}/_{j}]P$. Because i does not occur in P, this predicate is equal to the one shown on the right side of the table above. Now it seems that we have finally encountered a code transformation that changes the verification condition. However, recall that the final step in verification-condition generation quantifies over all of the free variables in the verification condition. This means that the resulting verification condition is still identical with the original one, up to renaming of bound variables, which in turn means that register allocation does not change the final form of the verification condition.

Most register allocators try to assign the same machine registers to two non-interfering temporaries (i.e., two temporaries that are never both live at the same instruction) that are assigned to each other. In this case the resulting program will contain assignments of the form "$\mathbf{r}_i = \mathbf{r}_i$" that can be *coalesced* as a simple case of copy propagation.

Sometimes, many practical programs have more temporaries than there are machine registers. In this case the register allocator must spill some temporaries to the stack frame. Basically, the register allocator "borrows" from the stack frame enough locations to make up the difference between the number of machine registers and the maximum number of temporaries that are live at a given moment. Then, it constructs a mapping from temporaries to register names or spill slots. The renaming operation is more involved in this case as each occurrence of the temporary on the left of an assignment must be changed to a write to the spill slot, while the occurrences on the right must be changed to a memory read. Fortunately, the verification-condition generator described in Section 4.2 was designed with spilling in mind and it considers the memory accesses to the stack frame as accesses to pseudo-registers. The overall effect is that the verification condition is still unchanged even in the presence of spilling.

6.5.6 Loop-Invariant Hoisting

Loop-invariant hoisting is a code transformation that eliminates the repeated computation of a part of a loop body that is not modified in the loop body. The loop invariant is computed once before the loop starts and the resulted value is used in the loop body. Consider for example the fragment of code shown on the left in the table below. Consider furthermore that the only modified variable in the loop is i and that i does not occur in e_1. Let I be the loop invariant. The notation "$...P \wedge I...$" denotes a code fragment that performs some operations whose safety predicate is P and then loops back to the beginning of the loop, hence the presence of the loop invariant I.

Before		After	
Code	VC	Code	VC
L_1: <u>**if**</u> C <u>**go to**</u> L_2 $i = e_1$ $...P \wedge I...$ L_2: $...P'...$	$I \wedge$ $\forall i. I \supset ((C \supset P') \wedge$ $(\neg C \supset [^{e_1}/_i]I) \wedge$ $(\neg C \supset [^{e_1}/_i]P))$	$j = e_1$ L_1: <u>**if**</u> C <u>**go to**</u> L_2 $i = j$ $...P \wedge I...$ L_2: $...P'...$	$I \wedge$ $\forall i. I \supset ((C \supset P') \wedge$ $(\neg C \supset [^{e_1}/_i]I) \wedge$ $(\neg C \supset [^{e_1}/_i]P)$

The compiler notices that the expression e_1 is a loop invariant and decides to hoist it outside the loop. For this purpose it introduces a fresh temporary j to hold the value of e_1 for the entire duration of the loop. The resulting code is shown at the right of the table. The verification condition of the transformed loop (without the initialization of j) is just like that of the code before the loop but with the expression e_1 replaced with j. Because j is a fresh variable that does not occur in I, C or in P, the effect of the initialization of j on the verification condition is to make it identical with the one before the code transformation. Thus, loop invariant hoisting does not affect the verification condition and it does not require special treatment for certification purposes.

6.5.7 Induction-Variable Elimination

Basic induction-variable elimination is another loop invariant optimization whose purpose is to replace expensive multiplicative operations in the body of the loop with faster additive operations. Consider for example that a variable i is incremented in the body of the loop by the loop-invariant expression e_1. Then the value of a computation $i * e_2$, where e_2 is also a loop invariant, increases by $e_1 * e_2$ every time around the loop. This can be exploited to replace the multiplicative computation by an increment by $e_1 * e_2$. This situation is shown in the code fragment below.

Before		After	
Code	VC	Code	VC
$i = e_0$ L_1: **if** C **go to** L_2 $j = i * e_2$ $i = i + e_1$ $\ldots P \wedge I \ldots$ L_2: $\ldots P' \ldots$	$[{}^{e_0}/_{i}]I \wedge$ $\forall i.I \supset ((C \supset P') \wedge$ $(\neg C \supset I_1) \wedge$ $(\neg C \supset P_1))$ `where :` $I_1 = [{}^{i+e_1}/_{i}, {}^{i*e_2}/_{j}]I$ $P_1 = [{}^{i+e_1}/_{i}, {}^{i*e_2}/_{j}]P$	$i = e_0$ $k = e_1 * e_2$ $j = e_0 * e_2 - k$ L_1: **if** C **go to** L_2 $j = j + k$ $i = i + e_1$ $\ldots P \wedge I' \ldots$ L_2: $\ldots P' \ldots$	$[{}^{e_0}/_{i},$ ${}^{e_0 * e_2 - e_1 * e_2}/_{j}]I' \wedge$ $\forall i.\forall j.I' \supset ((C \supset P') \wedge$ $(\neg C \supset I'_1) \wedge$ $(\neg C \supset P'_1))$ `where :` $I'_1 = [{}^{i+e_1}/_{i},$ ${}^{j+e_1*e_2}/_{j}]I'$ $P'_1 = [{}^{i+e_1}/_{i},$ ${}^{j+e_1*e_2}/_{j}]P$

Anticipating the need to modify the loop invariant in this case, the verification condition of the resulting code uses I' to denote the new loop invariant for the transformed code. The verification condition also makes the simplifying assumption that the variable j is not live at the beginning of the original loop and hence that it does not occur in P'. Also, in the transformed code k is a fresh temporary and hence does not need to occur in the invariant I' and it certainly does not occur in the predicates P or P'.

Let us start by assuming that we do not change the loop invariant, hence that $I' = I$. In this case j does not occur in I' and therefore $I'_1 = I_1$. The only significant difference is that P_1 is replaced by P'_1. Consider for example the case when $P = (i - e_1) * e_2 = j$. After replacements we get that $P_1 = (i + e_1 - e_1) * e_2 = i * e_2$, which is certainly provable, while $P'_1 = (i + e_1 - e_1) * e_2 = j + e_1 * e_2$, which is not provable because there are no assumptions about j (j does not occur neither in I nor in C.)

Here we have encountered an optimization that changes the code in such a way that the new verification condition is not provable because of assumptions that are too weak. The remedy in such a case is to strengthen the loop invariant by adding the missing assumption. Returning to our example, the difference between the predicates P_1 and P'_1 is that the expression "$j + e_1 * e_2$" occurs in P'_1 where "$i * e_2$" occurs in P_1. Thus, we attempt to modify the loop invariant as follows:

$$I' \quad = \quad I \wedge j + e_1 * e_2 = i * e_2$$

If we replace this invariant in the new verification condition we reduce it to the four subgoals shown on the right-hand side of the table below. On the left side of the table we have the same four subgoals in the original verification condition. Assuming that the original subgoals are provable we can verify that the new ones are also provable. The only one that might pose some difficulty to the theorem prover is the fourth one. In this case the theorem prover must be able to apply the rule of congruence starting from the equality $j + e_1 * e_2 = i * e_2$.

Before	After
$[^{e_0}/_{i}]I$	$[^{e_0}/_{i}]I \wedge (e_0 * e_2 - e_1 * e_2) + e_1 * e_2 = e_0 * e_2$
$I \supset C \supset P'$	$(I \wedge j + e_1 * e_2 = i * e_2) \supset C \supset P'$
$I \supset \neg C \supset [^{i+e_1}/_{i}]I$	$(I \wedge j + e_1 * e_2 = i * e_2) \supset \neg C \supset$ $[^{i+e_1}/_{i}]I \wedge (j + e_1 * e_2) + e_1 * e_2 = (i + e_1) * e_2$
$I \supset \neg C \supset [^{i+e_1}/_{i}, ^{i*e_2}/_{j}]P$	$(I \wedge j + e_1 * e_2 = i * e_2) \supset \neg C \supset [^{i+e_1}/_{i}, ^{j+e_1*e_2}/_{j}]P$

In conclusion, induction-variable elimination is an optimization that does require additional loop invariants and possibly an increased capability of the theorem prover to apply the congruence rule. However, the form of the required additional invariants is very simple and it is an easy task to modify the implementation of induction-variable elimination to emit them.

6.5.8 Redundant-Conditional Elimination

Redundant-conditional elimination is an optimization that removes those conditional branches whose outcome is statically predictable. This code transformation removes both the redundant conditional branch instruction, thus improving code speed, and also the unreachable body of the conditional, thus improving code size. As an example of redundant-conditional elimination consider the code shown below, where we assume that the compiler is able to infer that if all of C_k hold (for $k = 1..n$) then C also holds. Thus, because of the positioning of the conditional C after all of C_k have been tested and found true, the conditional C will always succeed. The compiler removes the conditional instruction and its "false" branch, shown here as the code "...P''...". Instead it places the loop invariant annotations with the predicate C and an empty set of modified variables.[1] The resulting code is shown on the right side of the table.

[1]The invariant annotation is only necessary to simplify the task of the theorem prover. Touchstone does not actually insert one.

Before		After	
Code	VC	Code	VC
$\underline{\mathbf{if}}\ \neg C_1\ \underline{\mathbf{go\ to}}\ \mathrm{L}_2$ $\underline{\mathbf{if}}\ \neg C_2\ \underline{\mathbf{go\ to}}\ \mathrm{L}_2$ ... $\underline{\mathbf{if}}\ \neg C_n\ \underline{\mathbf{go\ to}}\ \mathrm{L}_2$ $\underline{\mathbf{if}}\ C\ \underline{\mathbf{go\ to}}\ \mathrm{L}_1$ $...P''...$ L_1: $...P'...$ L_2: $...P...$	$\neg C_1 \supset P \wedge$ $C_1 \supset (\neg C_2 \supset P) \wedge$ $(C_2 \supset ...$ $C_n \supset (C \supset P') \wedge$ $(\neg C \supset P'')$	$\underline{\mathbf{if}}\ \neg C_1\ \underline{\mathbf{go\ to}}\ \mathrm{L}_2$ $\underline{\mathbf{if}}\ \neg C_2\ \underline{\mathbf{go\ to}}\ \mathrm{L}_2$ ... $\underline{\mathbf{if}}\ \neg C_n\ \underline{\mathbf{go\ to}}\ \mathrm{L}_2$ $\underline{\mathbf{inv}}\ C, \{\}$ $...P'...$ L_2: $...P...$	$\neg C_1 \supset P \wedge$ $C_1 \supset (\neg C_2 \supset P) \wedge$ $(C_2 \supset ...$ $C_n \supset C \wedge$ $C \supset P')$

We notice that this time the verification condition is altered in a significant way. However, when attempting to prove the new verification conditions all of the subgoals are also subgoals in the proof of the original verification condition, except for the predicate:

$$C_1 \supset (C_2 \supset \cdots \supset (C_n \supset C)$$

Notice that this predicate expresses the fact that the conditional C is redundant. It was this predicate that the compiler had to prove in order to figure out that the conditional C can be removed. We need, therefore, that the theorem prover be at least as powerful as the compiler in terms of proving such predicates.

Let us now return to the invariant instruction that was introduced in place of the conditional. If this annotation was missing then the resulting verification condition would be as follows:

$$\begin{array}{l} \neg C_1 \supset P \wedge \\ C_1 \supset (\neg C_2 \supset P) \wedge \\ \qquad (C_2 \supset ... \\ \qquad\qquad C_n \supset P') \end{array}$$

This is still a valid precondition if the original one is, but proving it requires a little bit more power on the part of the theorem prover. Basically, the task is to prove the following predicate:

$$C_1 \supset (C_2 \supset \cdots \supset (C_n \supset P') \tag{6.11}$$

We know that the theorem prover is able to prove the following predicates:

$$\begin{array}{l} C_1 \supset (C_2 \supset \cdots \supset (C_n \supset (C \supset P') \\ C_1 \supset (C_2 \supset \cdots \supset (C_n \supset C) \end{array}$$

The first of the above predicates occurs in the original verification condition, which the theorem prover is able to prove, and the second one is just the redundancy condition. A complete theorem prover would have no problem handling the predicate 6.11 directly, without any mention of the subgoal C. But in many practical instances the presence of the subgoal guides the theorem prover towards a successful proof. Practical experience with theorem provers suggests that many of the proving tasks fail either because the theorem is not provable or because the prover is exploring too many non-productive subgoals. In the latter situation, the prover can be helped by user-provided intermediate subgoals. The invariant annotations provide a convenient way for the code producer to specify subgoals for the theorem prover.

Because the theorem prover used with the Touchstone compiler is powerful enough to prove predicates like 6.11 directly, no invariant annotations are emitted for the eliminated conditionals.

The conditional elimination optimization presented in this section detects redundant conditionals based only on the conditionals already existing in the code. The optimization can be improved if the compiler employs static program analyses to discover properties of the program that are not as obvious as the user-provided conditionals. In the next section, I describe the implementation in Touchstone of array bounds-checking elimination first as a special case of conditional elimination and then in the situation when a static analysis is used to improve the optimization.

6.5.9 Array Bounds-Checking Elimination

Array bounds-checking elimination is a special case of conditional elimination. It targets the conditionals introduced by the compiler to check that array accesses fall within the bounds of the array. In a language like that compiled by Touchstone the compiler has the obligation to insert bounds-checks or otherwise the code would not be safe. However, many of the inserted checks can be eliminated by proving statically that the array index is within the bounds of the array. If we want Touchstone to generate code of comparable performance with that of plain C compilers it is crucial that we do a good job at discovering and eliminating the redundant checks. I discuss in this section the approach taken in Touchstone to eliminate array bounds checks. First, I show a simple-minded optimization that is a special case of conditional elimination discussed previously. Then, I show a static analysis that improves the optimization. In either case I will focus on the implications on the required loop invariants and theorem proving techniques for certification purposes.

Let us consider the simple example of a function that adds all elements of an array of integers whose indices are a multiple of a given stride parameter. The source code for such a function is shown on the left side of Figure 6.8, while a stylized form of the target code is shown on the right-hand side of the same figure. Consistent with the type representation techniques discussed in Section 6.4, the source-level array argument is represented using the

```
int add(int [] a, int stride) {
  int s = 0;
  if(stride ≤ 0) return 0;
  for(i = 0; i < a.length; i = i + stride)
    s = s + a[i];
  return s;
}
```

```
1  add:  /* r1 : array(int, r2) ∧ r3 : int */
2         s = 0
3         if r3 ≤ 0 go to L2
4         i = 0
5  L1:    inv i : int ∧ s : int
6         if i ≥ r2 go to L2
7         if i < 0 go to Err
8         if i ≥ r2 go to Err
9         t = M[r1 + 4 * i]
10        s = s + t
11        i = i + r3
12        go to L1
13 L2:    r0 = s                      /* r0 : int */
14        return
15 Err:   go to Err
```

Figure 6.8: Compilation example with array bounds checking

registers $\mathbf{r}_1$ to hold the base address and $\mathbf{r}_2$ to hold the length. The register $\mathbf{r}_3$ holds the integer stride parameter. Assume also that the calling convention specifies that the return value be put in the register $\mathbf{r}_0$.

Note that the loop invariant for the loop starting on line 5 consists of typing declarations for only those temporaries that are modified in the loop and live at the beginning of the loop. This invariant is sufficient because of the array bounds checks from lines 7 and 8. To see this recall that VCGen emits a `saferd` predicate for each memory read operation. Thus the fragment of the verification condition emitted for the lines 7–9 is $i \geq 0 \supset i < \mathbf{r}_2 \supset \texttt{saferd}(\mathbf{r}_1 + 4 * i)$. Furthermore, this predicate fragment is in the scope of the precondition that contains the assumption $\mathbf{r}_1 : \texttt{array}(\texttt{int}, \mathbf{r}_2)$. This fragment of the verification condition is provable because of the `read` proof rule introduced in Section 6.2.1 and reproduced in a simplified form below:

$$\frac{a : \texttt{array}(\texttt{int}, l) \qquad i \geq 0 \qquad i < l}{\texttt{saferd}(a + 4 * i)}\,\textbf{read}$$

Notice that with the above rule and the run-time checks, the safety predicate for an array access inside the array a is equivalent to "$a : \texttt{array}(T, L)$". This is very important because it means that even in the presence of arrays the safety predicates consist only of typing declarations and thus the simplified typing loop invariants are sufficient.

So far we have not attempted to remove the bounds checks. We could attempt to eliminate them using the redundant-conditional elimination optimization discussed in Section 6.5.8.

$D^{C'} = D^C$
<u>**if**</u> $(x, x) \notin Dom(D^C)$ <u>**then**</u>
  <u>**foreach**</u>$(u, u) \in Dom(D^C)$ <u>**do**</u>
    $D^{C'}(x, u) = \infty; D^{C'}(u, x) = \infty$
  $D^{C'}(x, x) = 0$
<u>**if**</u> $(y, y) \notin Dom(D^C)$ <u>**then**</u>
  <u>**foreach**</u>$(u, u) \in Dom(D^C)$ <u>**do**</u>
    $D^{C'}(y, u) = \infty; D^{C'}(u, y) = \infty$
  $D^{C'}(y, y) = 0$
<u>**if**</u> $D^{C'}(x, y) \leq c$ <u>**then**</u> <u>**raise**</u> `True` <u>**else**</u> $D^{C'}(x, y) = c$
<u>**if**</u> $D^{C'}(y, x) < -c$ <u>**then**</u> <u>**raise**</u> `False`
<u>**foreach**</u>$(u, v) \in Dom(D^{C'})$ <u>**do**</u>
  $D^{C'}(u, v) = \mathtt{min}(D^{C'}(u, v), D^{C'}(u, x) + c + D^{C'}(y, v))$
<u>**return**</u> $D^{C'}$

Figure 6.9: The loop-residue decision procedure for linear arithmetic shown as a function to compute $D^{C'}$ from D^C where C' is obtained from the set of constraints C by adding the constraint "$x - y + c \geq 0$".

Before we proceed, it is useful to show how Touchstone infers that the validity of one predicate follows from the validity of several enclosing conditionals.

Touchstone uses a simple decision procedure for inequalities of the form "$x - y + c \geq 0$", where c is an integer numeral and x and y are variables. These variables can in fact stand for arbitrary expressions but the decision procedure will not be able to benefit from their internal structure. A decision procedure for this fragment of arithmetic was first given by Pratt [Pra77] in terms of finding negative-weight cycles in directed graphs. Let C be a collection of such inequalities and let D^C be a function defined on pair of variable names such that $D^c(x, y)$ is the maximal value of the expression "$y - x$" for all points in $\mathbb{Z}^n$ that satisfy all the equalities in C. If the value of "$y - x$" is not bounded then we use the special symbol ∞ as the value of $D^C(x, y)$. We further consider that ∞ behaves like positive infinity when added and compared with finite values.

The Touchstone implementation of the loop-residue decision procedure computes the value of $D^C(x, y)$, for a set of linear constraints C and for all variables x, y occurring in the constraints. Then we can easily verify whether the set of constraints C entails the validity or the falsity of an arbitrary "$x - y + c \geq 0$" as follows:

$$\begin{array}{llll} \text{If} & D^C(x, y) \leq c & \text{then} & C \supset x - y + c \geq 0 \\ \text{If} & D^C(y, x) < -c & \text{then} & C \supset x - y + c < 0 \end{array}$$

The proof of these facts is simple given the definition of D^C. Let us now focus on how

Touchstone computes D^C.

If there are no constraints, the function $D^{\emptyset}$ is undefined at all variable names. The function in Figure 6.9 can be used to compute $D^{C'}$ from D^C, where the set of constraints C' is obtained from C by adding the new constraint "$x-y+c \geq 0$". This function either returns the new value of $D^{C'}$ or it raises one of the exceptions `True` or `False` when the additional constraint is provably true or false respectively given the original set of constraints.

Returning to the example of Figure 6.8, let us consider the operation of the loop residue decision procedure given the sequence of conditionals "$\mathbf{r}_3 > 0$", "$i < \mathbf{r}_2$", and "$i \geq 0$" leading to the execution of line 8. Each of these constraints is first put in the form "$x - y + c \geq 0$". For this purpose we sometimes need to use a dummy variable `zero` whose value is fixed to zero. The third column in the table below shows the transformed constraints. The last column shows the significant values of D^C. The values not shown are those of the form $D^C(u, u)$, which are all zero, and the ∞ values. In this simple example, the only non-trivial inference that the loop-residue decision procedure makes is shown on the very last line of the table. Because $D^{C_3}(\mathbf{r}_2, i) = -1$ and $D^{C_3}(i, \texttt{zero}) = 0$ the decision procedure sets $D^{C_3}(\mathbf{r}_2, \texttt{zero}) = -1$. Informally, this means that the decision procedure has inferred that $\mathbf{r}_2 > 0$ when the execution reaches line 8.

i	Conditional	Constraint	D^C
1	$\mathbf{r}_3 > 0$	$\mathbf{r}_3 - \texttt{zero} + (-1) \geq 0$	$D^{C_1}(\mathbf{r}_3, \texttt{zero}) = -1$
2	$i < \mathbf{r}_2$	$\mathbf{r}_2 - i + (-1) \geq 0$	$D^{C_2}(\mathbf{r}_3, \texttt{zero}) = -1$ $D^{C_2}(\mathbf{r}_2, i) = -1$
3	$i \geq 0$	$i - \texttt{zero} + 0 \geq 0$	$D^{C_3}(\mathbf{r}_3, \texttt{zero}) = -1$ $D^{C_3}(\mathbf{r}_2, i) = -1$ $D^{C_3}(i, \texttt{zero}) = 0$ $D^{C_3}(\mathbf{r}_2, \texttt{zero}) = -1$

When Touchstone encounters the condition of line 8 it first checks whether its truth or its falsity is implied by the current function D^{C_3}. Thus it takes the conditional and transforms it to the canonical form "$i - \mathbf{r}_2 + 0 \geq 0$". Then, it checks whether "$D^{C_3}(i, \mathbf{r}_2) \leq 0$" or if "$D^{C_3}(\mathbf{r}_2, i) < 0$". The latter check succeeds and Touchstone infers that the check of line 8 is falsified by the enclosing conditionals. Thus, Touchstone removes it and lets the execution fall through.

So far Touchstone applied only the redundant-conditional elimination which, as we saw in Section 6.5.8 does not necessitate additional invariants or changes in the theorem prover. In fact, we shall see in Chapter 7 that the theorem prover uses a more complex and more powerful decision procedure for arithmetic and can therefore infer for itself everything that the loop-residue decision procedure can.

After redundant-conditional elimination there is still the bounds check of line 7 that is redundant because the value of the temporary i starts at zero and is always increasing by a positive amount on each iteration. (The increment $\mathbf{r}_3$ is guaranteed positive by the conditional of line 3.) This conditional could be proved redundant if we add to the loop invariant of line 6 the conjunction "$i \geq 0$". In general, the effectiveness of many code optimizations can be improved if the compiler uses a static analysis first to infer non-obvious program properties.

I have observed that a very simple loop invariant analysis is able to supply the additional loop invariants that are required to eliminate most of the array-bounds checks. The analysis attempts to discover those variables that are modified in a loop body in such a way that they are always increasing or always decreasing. Such variables are called *monotone variables*. The compiler collects, for each modified variable i and for each path through the loop body, the expression e that is assigned to i. Then, the compiler attempts to discover using simple rules of arithmetic whether e can be written as "$i + e'$". If not, then i is not a monotone variable. On the other hand, if such an e' exists then Touchstone attempts to discover whether it is greater or equal to zero or less or equal to zero using the loop residue decision procedure. Note that this is attempted with all of the conditionals in scope, even those inside the loop body. If all of the expressions assigned to i on all paths through the loop are of the form $i + e'$, with e' greater or equal to zero the variable i is declared a monotonically increasing variable. If the symbolic value of i in at the beginning of the loop is e_0, then the predicate $i \geq e_0$ is added to the loop invariant. Using this procedure Touchstone discovers that i is a monotone variable and adds the predicate $i \geq 0$ to the loop invariant. This step is performed before the loop body is scanned and therefore the conditional of line 7 will also be found redundant and eliminated.

Let us now consider whether additional loop invariants must be emitted as part of the code. Assume for now that the loop invariant of line 5 is I. Then, the interesting part of the generated verification condition, for the code without array bounds checking, is as follows:

$$\begin{array}{l} \mathbf{r}_1 : \mathtt{array}(\mathtt{int}, \mathbf{r}_3) \supset \mathbf{r}_3 \geq 0 \supset ([{}^0\!/\!_i]I \wedge \\ \qquad\qquad\qquad\qquad \forall i. I \supset (i < \mathbf{r}_2 \supset (\mathtt{saferd}(\mathbf{r}_1 + 4 * i) \\ \qquad\qquad\qquad\qquad\qquad\qquad [{}^{i + \mathbf{r}_3}\!/\!_i]I))) \end{array}$$

We see that the only way we can prove the `saferd` predicate by using the `read` rule shown before is if the invariant I contains the conjunct $i \geq 0$. This is in fact exactly the invariant that Touchstone used in removing the array bounds checks. Note that the last line in the verification condition verifies that the invariant is a correct invariant. When proving this part of the predicate the theorem prover must reconstruct the same reasoning that led Touchstone to infer that i is in fact greater or equal to zero. The advantage for the theorem prover is that, unlike the compiler, it does not have to consider all possible candidates for monotone variables.

This concludes the discussion of optimizations in a certifying compiler. The list of possible optimizations is much longer than the number of optimizations I could possibly discuss in this dissertation. Thus, this section should be taken as a recipe for analyzing code transformation and for determining what additional loop invariants, if any, must be emitted by the certifying compiler.

6.6 Discussion

One of the goals of the Touchstone project was to demonstrate that a certifying compiler can also generate code of quality comparable with good non-certifying compilers such as GNU gcc [Fou93] and Digital `cc`. Touchstone starts at a disadvantage in this performance competition because the semantics of the safe subset of C mandates array bounds checks and null-pointer checks for all memory operations. Touchstone attempts to minimize the performance cost of these run-time checks by using the redundant-conditional optimizations described in Sections 6.5.8 and 6.5.9. These optimizations, in conjunction with others such as global register allocation and caching of global variables in registers allow Touchstone to generate code whose performance is within 10% of that generated by optimizing C compilers. A detailed description of the experimental results that I collected for the Touchstone compiler can be found in Section 8.2.

But of course, performance is not everything. The code emitted by Touchstone is not only safe-by-design but also provably safe using a theorem prover such as that described in the next chapter. Thus the optimized code generated by Touchstone can be installed as untrusted extensions in a variety of environments where type safety is a requirement. One particular application of the Touchstone compiler could be to generate type-safe native-code extensions to typed languages such as Java and Standard ML.

The Touchstone project was initiated to explore automatic front-ends to a proof-carrying code system. However, the usefulness of the certifying compiler concept extends beyond the environments where the safety of untrusted code is the key issue. Probably the main benefit of a certifying compiler over a traditional compiler is that the certification stage, consisting of verification-condition generation and theorem proving, acts as an effective referee for the correctness of each compilation, thus simplifying compiler testing and development.

In sections 6.5.1 through 6.5.9 I have shown that each optimization in Touchstone has a precise effect on the verification condition. I have also shown that, assuming that the verification condition is provable before the optimization, it should remain provable after the optimization. This is in fact a simple correctness criterion for all compiler optimizations. For many optimizations, no additional loop invariants are required. For some optimizations the compiler designer must put some effort into designing appropriate extensions to the loop invariants to guarantee the provability of the verification conditions. But, in my experience, the benefits outweigh the cost of this extra effort.

Testing a certifying compiler is easier and more effective than testing a traditional compiler. As in the traditional case, the compiler writer must still write and compile test cases.

But while in the traditional case the compiled output must be executed on various data sets hoping to trip over a compiler bug, in the certifying compiler case VCGen and then the theorem prover inspect the code and prove that *for all possible input values* the code is type safe. Furthermore, if the theorem prover fails to prove type safety it points to a precise place in the code along with the typing predicate that it cannot prove. This is in contrast to traditional debugging of compilers where the actual compilation bug must often be scavenged from the corpse of a failed execution. For example, bugs in the register allocator or in the instruction scheduler are notoriously difficult to find because they lead to subtle errors in the output program that tend to surface as sporadic program failures, usually many instructions past the actual erroneous instruction. Furthermore, the low-level nature of the output and the fact that such errors are most likely to occur in large programs, makes the visual inspection of the output quite tedious.

It is true that the Touchstone certifying compiler guarantees only the detection of compilation errors that break type safety. This means that those errors that preserve the type safety of the code are not detected. For example, the certifier would not detect if a compiler generates code that always returns the integer zero for any integer source-level function. Nevertheless, my practical experience with compiler debugging suggests that most compilation errors are such that, sooner or later, they lead to the generation of code that is not memory safe and thus not type safe. It is exactly this kind of error that the certifier detects. During the development of Touchstone I have encountered only one error that was not signaled by the certifier, as opposed to a large number of errors that were caught.

The question of compiler correctness is as old as the first compiler implementations. In a paper published in 1963, John McCarthy refers to this problem as *"one of the most interesting and useful goals for the mathematical science of computation"* [McC63]. However, despite a large body of work in the area [Dyb85, GRW95, Moo89, Mor73, ORW95, TWW79, You89], we still lack the technology to prove automatically the correctness of an optimizing compiler. Even manual proofs are rare, and they tend to verify only the algorithms rather than the implementations. Plus, the correctness proofs need to be redone after even the slightest modification or improvement to the compiler.

Proving compiler correctness is just a means towards the actual goal of ensuring that only correct output is ever produced by the compiler. The certifying compiler is a potentially more practical approach to the same goal. Instead of verifying the compiler once and for all, we check aspects of the correctness of every individual compilation. This will not ensure that the compiler is bug-free, but it will signal most incorrect compiler outputs as soon as they are produced. To reduce the complexity of the checking process, Touchstone does not try to check full equivalence of the source and target programs, but instead it verifies only that the target program has certain key properties that can be verified using a small amount of information about the source program.

The idea of checking individual compilations instead of verifying the compiler also appears in the work of Pnueli [PSS98], though in the simpler instance of a non-optimizing compiler from the SIGNAL [BGJ91] asynchronous language to the C programming language. Similarly, Cimatti et al. [C$^+$97] have implemented a certifying compiler from an expression language without loops or function calls to an RTL-like language. In this latter case, and also in recent work of Kozen [Koz98], the certification step is simplified considerably by restricting the optimizations such that each source-level construct is compiled to a given pattern in the target code. Then the certifier must recognize the patterns and must only check that adjacent patterns are assembled properly. On the other hand, the limitations in optimizations and the simplicity of the certifier permit these certifying compilers to be more ambitious and attempt to verify not just the type safety of the target code but also its equivalence to the source program.

The approach to a certifying compiler presented in this paper is inspired by the need to have an automatic front-end to a proof-carrying system. If integration with PCC and the generality and simplicity of the certifier were not important, I could have chosen from several alternate implementation approaches.

One alternative is suggested by the fact that the verification conditions emitted by VCGen from the output of Touchstone are guaranteed to be provable automatically. Thus, one can incorporate parts of the theorem prover in VCGen and prove the safety predicates as they are created, without actually generating a verification condition and maybe not even a proof. This might be particularly practical when optimizations like array bounds-checking elimination or induction-variable elimination are not allowed and thus only a very simple type-based theorem prover is necessary. The Java bytecode verifier [LY97] can be viewed as taking this approach, as can the type-checker in the typed assembly language of Morrisett, et al. [MWCG98].

To illustrate this point let us compare in more depth a certifying compiler together with a theorem prover with a Java [GJS96] compiler together with a bytecode verifier [LY97]. The similarity is that both systems produce code that is annotated for the purpose of enabling a certification system (the bytecode verifier, in the Java case) to verify type safety. The difference is that our certifier has a more flexible annotation language that permits the verification of arbitrarily optimized assembly language while necessitating fewer annotations. (PCC requires annotations only at the backward-branch targets while Java bytecodes contain annotations in every single instruction.) The Java bytecode verifier works only on a specially designed bytecode intermediate language where typing annotations are contained in the instruction codes themselves. Furthermore, the Java bytecode verifier prevents the compiler from doing several important optimizations, such as array bounds-checking elimination and global register allocation.

The compilation approach presented here also resembles in many respects the compilation strategy of the TIL [TMC$^+$96] compiler for Standard ML, which uses a typed intermediate language that can be easily type-checked to achieve an independent validation of optimizations. However, the TIL type-system does not guarantee memory safety in the presence of certain optimizations such as array bounds-checking elimination, and furthermore, it cannot be used after the register allocation phase when some variables (registers) are reused to hold values of different types in the body of the same function. For this reason, types are dropped in the TIL compiler before the register allocation phase, and thus no type-checking is possible at the level of the compiler output. The problems related to register allocation are solved by Morrisett et al. [MWCG98] by choosing a more expressive type system, but the issue of memory-safety of TIL programs in the presence of optimizations such as array bounds-checking elimination still remains a problem.

This chapter presents the class of type-based safety policies and the design of a certifying compiler that generates the necessary loop invariants for such policies when compiling programs written in a type-safe subset of the C programming language. This takes care of one of the challenging tasks facing a code producer that wants to use a proof-carrying code system. The other difficult task is to generate proofs of the verification conditions that arise from the code annotated with loop invariants. In the next chapter I describe a theorem prover that is powerful enough to prove all verification conditions arising from the output of the Touchstone certifying compiler and is also able to emit their proofs in the format required by the proof checker described in Chapter 5.

Chapter 7

The Proof-Generating Theorem Prover

There are two difficult tasks in front of a code producer wishing to use proof-carrying code to interact by mobile code with a code receiver. The first, and the more difficult one, is to generate the loop invariants. This is accomplished in the system presented in this dissertation by the Touchstone compiler described in Chapter 6. The second difficult task is to prove the verification conditions using the axiomatization provided by the code receiver as part of the safety policy.

The verification conditions are predicates in an extension of first-order predicate logic with application-specific function symbols, such as the `saferd` predicate constructor used to express memory safety. Therefore, the proof producer can be a theorem prover for first-order logic, possibly one of the many already implemented and discussed in the literature [BM79, CAB+86, CH85, Det96, Gor85, ORS92]. To my knowledge, all of these are able to prove typical verification conditions, sometimes with the help of additional tactics. However, for some safety properties, automatic decision procedures do not exist or are not effective. In such cases it is more practical to use a semi-interactive theorem prover guided by a person with a deep understanding of the reasons underlying the safety of the untrusted code.

To be usable as a PCC proof producer, a theorem prover must not only be able to prove verification conditions but must be also capable of generating detailed proofs of them. Furthermore these proofs must be expressed in the particular logic (i.e., using the axioms and inference rules specified as part of the safety policy) used by the code receiver and encoded in the LF_i framework so that the code receiver can use the LF_i type checker to verify them. The major difficulty here is to make the theorem prover output the detailed proof in any form, because once we have all the proof details, it is generally easy to transform them in the LF_i format expected by the receiver.

In my implementations of PCC, I have used two different theorem provers so far. The first, and most primitive, theorem prover that I used was suggested by the decision to represent proofs as LF expressions. This theorem prover uses the Elf [Pfe94] implementation of LF. The powerful type-reconstruction algorithm used in the Elf system makes it able to find LF expressions having a given LF type with respect to a specified signature. This feature can be used for theorem proving by loading the signature that encodes the axiomatization of the logic and then giving to Elf a type of the form `pf` P, where P is the LF representation of the predicate to be proved. Given this query, Elf searches for an LF term M of the required type and shows this term if it can find it. Because of the adequacy of the representation of proofs in LF (Theorem 5.2) we know that the given term is the encoding of a valid proof of P. Furthermore, this term is exactly in the form required by the code receiver.

The advantage of the Elf approach to theorem proving is that it is trivial to implement. We just need the Elf system to which we give the encoding of the logic as an LF signature. Note that this encoding is provided by the code receiver anyway. Not only does Elf prove the verification conditions but it also emits the proof in the exact format required by PCC. However, the major problem with the Elf approach to theorem proving is that Elf uses a relatively simple depth-first search algorithm that is inappropriate for theorem proving in many realistic logics. This means that, in order to use this approach, the inference rules have to be written in such a way and in such an order that the search is goal directed. In some cases, mostly having to do with integer arithmetic, I had to add redundant inference rules to the safety policy so that Elf could find a proof.

While the Elf approach to theorem proving was of extreme importance for the early validation of the proof-carrying code technique, there was a need for a real theorem prover using state-of-the-art decision procedures for theories such as equality and arithmetic. In order to be able to experiment more easily with different decision procedures and different proof-generating strategies, I chose to implement my own version of a theorem prover based on the Nelson-Oppen architecture for cooperating decision procedures [NO79], also implemented in the Stanford Pascal Verifier [LGvH+79] and the Extended Static Checking [Det96] systems. I believe however, that the proof-generating techniques presented in this chapter can be adapted to other theorem prover architectures and can even be retrofitted to existing theorem provers.

Before we start discussing the technical details, I want to point out that the ability to generate proof representations in a theorem prover is beneficial even when the theorem prover is not intended as a front end to proof-carrying code. A theorem prover that generates an explicit proof object for each successfully proved predicate enables a distrustful user to verify the validity of the subject theorem by checking the proof object. This effectively eliminates the need to trust the soundness of the theorem prover at the relatively small expense of having to trust a much simpler proof checker. The theorem prover is a complicated system implementing complex algorithms, so it is of great practical importance that we do not have to rely on its soundness. The generated proofs and the proof checker are also of great software engineering benefit as they can lead to the timely discovery of soundness bugs that are introduced during the development or maintenance of the theorem prover.

The rest of this chapter is structured as follows. In Section 7.1 I review the Nelson-Oppen architecture for a theorem prover and I show how my implementation adheres to it. Then in Section 7.2 I describe using pseudo-code algorithms the control part of the theorem prover responsible for handling the first-order logical connectives, the backtracking and the integration of the individual decision procedures. Then, in Section 7.3 I describe the general interface that a decision procedure must implement. In the same section I show how standard decision procedures such as congruence closure and Simplex can be adapted to the given interface. For illustration purposes, I show in Section 7.3.3 the operation of the theorem prover when proving a simple theorem involving linear inequalities and equalities. Also, as another example of a proof-generating decision procedure, I show in Section 7.3.4 a simple decision procedure for proving the typing predicates arising from the output of the Touchstone compiler. This chapter is concluded with a discussion of the many possible improvements to the algorithms for producing proofs.

7.1 The Nelson-Oppen Prover Architecture

In this section I describe, at a high level, the general strategy for combining decision procedures for several theories into a decision procedure for the combined theory. This strategy, first proposed by Nelson and Oppen [NO79], is at the basis of the proof-generating theorem prover described in this chapter. It is this architecture that makes the theorem prover easily extensible with new theories such as the Touchstone typing theory.

Informally, a *theory* consists of a set of function symbols, which are called the *free functions* of the theory, together with a set of axioms constraining the interpretation of the function symbols. The axioms are written in the first-order predicate calculus with function symbols and equality; thus every theory gets equality automatically, without having to define the congruence rules for its free functions.

To illustrate these concepts consider the theory of arrays, first introduced in Section 4.3, with the free functions "`sel`" and "`upd`" and the following two axioms:

$$\forall m.\forall e.\forall v.\ \mathtt{sel}(\mathtt{upd}(m,e,v),e) = v$$
$$\forall m.\forall e.\forall e'.\forall v.\ e \neq e' \supset \mathtt{sel}(\mathtt{upd}(m,e,v),e') = \mathtt{sel}(m,e')$$

A *literal* or an *atomic formula* in a theory is a formula that contains only free functions from the corresponding theory and the equality symbol. A formula is *valid* in the theory if it is satisfied by every interpretation of the function symbols in F that satisfies all the axioms A. A set of literals (called the hypotheses) *entails* another literal (called the goal) if the goal literal is satisfied by all interpretations that satisfy both the axioms of the theory and the hypotheses literals. A set of literals is *satisfiable* if there exists an interpretation that satisfies all the axioms and all the literals. A *decision procedure* for a theory is an algorithm that attempts to discover whether the conjunction of a set of literals entails another literal.

In many cases, a decision procedure can be implemented using a *satisfiability procedure* that discovers whether a set of literals is satisfiable or not. This is because the set of literals $\mathcal{H}$ entails the literal A if and only if the set "$\mathcal{H} \cup \{\overline{A}\}$" is not satisfiable, where $\overline{A}$ is a literal that is the negation of A.

Consider the theory $\mathbb{Q}$ of rational numbers with the free symbols $+$, $-$, $\geq$ and the numerals with the usual axioms of rational arithmetic. Consider also the theory $\mathbb{E}$ with one uninterpreted unary function symbol "`f`" (the function symbol is uninterpreted because there are no axioms pertaining to it, except for the congruence axiom). The satisfiability problems for each of these theories considered separately were solved long ago by Fourier for $\mathbb{Q}$ and by Ackermann for $\mathbb{E}$ [Ack54]. Although necessary, it is not sufficient to have decision procedures for these isolated theories because most theorem proving goals combine literals from multiple theories. We need therefore a method to combine multiple satisfiability procedures into one for the combined theory. This is not as straightforward as it might seem because of unexpected interactions between theories that, at the first sight, have nothing in common.

To illustrate the difficulties and to provide an example for presenting the Nelson and Oppen solution, consider the following set of four literals[1] from the combined theory $\mathbb{Q} + \mathbb{E}$:

$$\mathtt{f}(\mathtt{f}(x) - \mathtt{f}(y)) \neq \mathtt{f}(z) \;\wedge\; y \geq x \;\wedge\; x \geq y + z \;\wedge\; z \geq 0 \tag{7.1}$$

Informally, to demonstrate that the above set of literals is not satisfiable, we would first use the two literals in the middle to infer that "$0 \geq z$" and then the last literal to demonstrate that "$z = 0$" and then also that "$x = y$". Note that these inference steps were entirely within $\mathbb{Q}$. Then, we use the congruence rule of $\mathbb{E}$ to infer that "$\mathtt{f}(x) = \mathtt{f}(y)$". Then we move again in $\mathbb{Q}$ to prove that "$\mathtt{f}(x) - \mathtt{f}(y) = z$" and then back to $\mathbb{E}$ to prove that "$\mathtt{f}(\mathtt{f}(x) - \mathtt{f}(y)) = \mathtt{f}(z)$". This allows $\mathbb{E}$ to detect the contradiction with the first literal and to declare that the set of literals is not satisfiable. This example demonstrates that, in general, the decision procedures must interact in a non-obvious way to detect unsatisfiability. Nelson [Nel81] defines the exact way in which decision procedures must cooperate, as follows:

Definition 7.2 *In order to detect the unsatisfiability of a set F of literals with free functions from two theories $\mathcal{S}$ and $\mathcal{T}$, with satisfiability procedures* `S` *and* `T` *respectively, we must first rewrite F into an equivalent pair of sets F_S and F_T such that each set contains only literals from the corresponding theory. Then, each satisfiability procedure must detect unsatisfiability in its own set of literals and must also propagate to the other procedure* **all equalities between variables that it detects**. *If the two theories are convex then this cooperation procedure is complete for the theory $\mathcal{S} \cup \mathcal{T}$ provided that* `S` *is complete for $\mathcal{S}$ and* `T` *is complete for $\mathcal{T}$.*

[1] This example is taken from [Nel81].

As an example, consider how the Nelson-Oppen strategy behaves for the set of literals shown in 7.1. First, we introduce the variables g_1, g_2 and g_3 to stand for '$\mathtt{f}(x)$, $\mathtt{f}(y)$ and "$\mathtt{f}(x) - \mathtt{f}(y)$" respectively. With these auxiliary variables we can separate the set of literals into F_Q and F_E shown below:

F_Q			F_E		
y	$\geq$	x	$\mathtt{f}(g_3)$	$\neq$	$\mathtt{f}(z)$
x	$\geq$	$y+z$	$\mathtt{f}(x)$	$=$	g_1
z	$\geq$	0	$\mathtt{f}(y)$	$=$	g_2
g_3	$=$	$g_1 - g_2$			

Neither of F_Q or F_E are unsatisfiable in isolation but F_Q entails the equality "$x = y$". (It also entails the equality "$z = 0$" but this one is not propagated because it is not an equality between variables only. In fact, it would not be correct to propagate this equality because the free function "0" does not belong to the theory $\mathbb{E}$.) The addition of "$x = y$" to F_E makes this set to be able to infer and propagate the equality "$g_1 = g_2$". This in turn, when added to F_Q, entails "$g_3 = z$", which added to F_E makes the set F_E unsatisfiable. This example, complete with the proof-generating part, is revisited in detail in Section 7.3.3.

The Nelson-Oppen strategy is complete only for convex theories. A theory $\mathcal{T}$ is *not convex* if there exists a formula F and $2n$ variables $x_1, \ldots, x_n, y_1, \ldots, y_n$ with $n \geq 2$ such that F implies the disjunction:

$$\bigvee_{1 \leq i \leq n} (x_i = y_i)$$

The theories $\mathbb{Q}$ and $\mathbb{E}$ considered in our example are convex. But the theory of arrays and the theory $\mathbb{Z}$ of integer linear arithmetic are not convex, as demonstrated by the example below:

$$y = z + 1 \;\wedge\; y \geq x \;\wedge\; x \geq z \quad \text{entails} \quad x = y \;\vee\; x = z \tag{7.3}$$

The Nelson-Oppen architecture can be adapted to deal with non-convex theories by performing a case-split whenever a conjunction of literals entails a disjunction. Informally, the prover tries to guess which one of the disjuncts holds and asserts it to all decision procedures. If this does not lead to unsatisfiability then the next disjunct is tried. For this procedure to be correct there are additional technical requirements that the theories must satisfy. I do not mention these requirements here but note that any theory that we are likely to use satisfies them. These requirements along with a formal proof of soundness of the algorithm is given by Nelson in [Nel81].

The structure of the theorem prover is shown in Figure 7.1. At the top level we have a module that knows how to break predicates of first-order logic into literals. This module

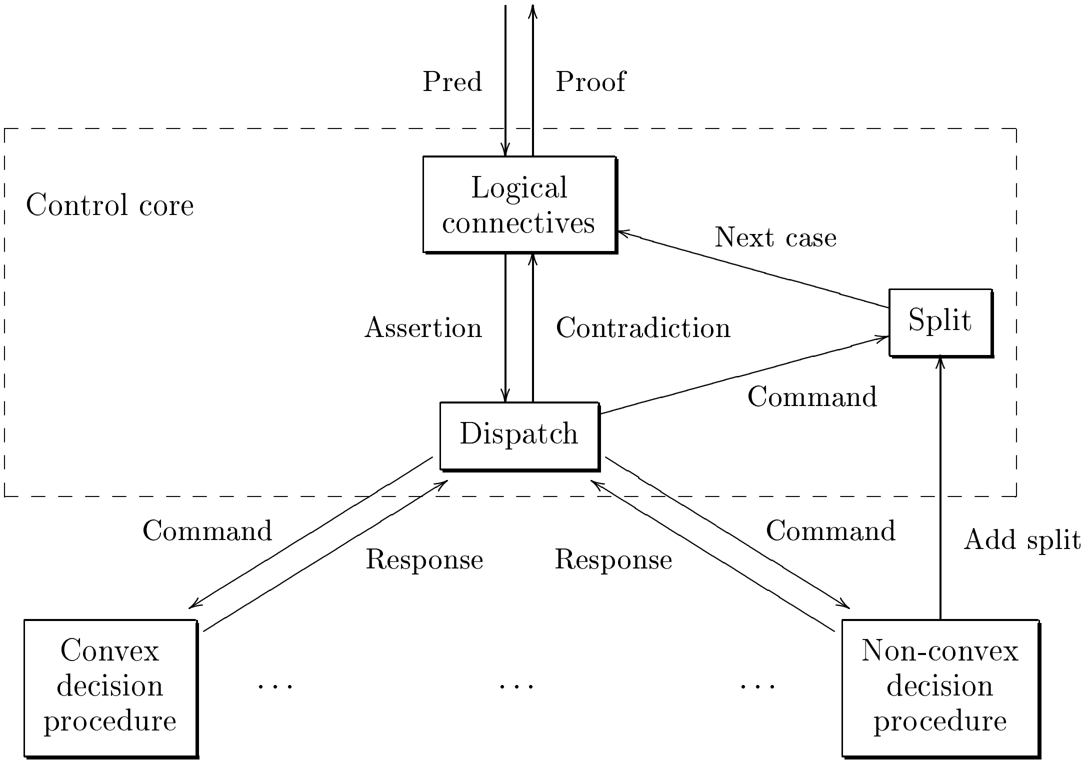


Figure 7.1: The overall structure of the theorem prover.

receives in input a predicate and returns a proof or an indication of failure. The "Logical Connectives" module is also responsible for implementing the backtracking necessary for proof search. The result of breaking a predicate into literals is a set of hypothesis literals (typically obtained from the left-hand side of implications) and a goal literal. If the predicate contains multiple goals, they are proved separately. These literals are given to the "Dispatch" module, which is responsible for propagating them to all the decision procedures. Just as in the example discussed above, the decision procedures try to find contradictions, in which case they return a contradiction. Or, a decision procedure might discover new equalities between variables, in which case it gives it to the "Dispatch", who in turn broadcasts it to all decision procedures. Additionally, a non-convex decision procedure might also discover a case split, which must be given to a special module dealing with case splits. This latter module is queried by the "Logical Connectives" module when the current goal is about to fail. If there are case splits, they are tried in order by the "Logical Connectives" module. Each of these modules and several decision procedures are discussed in the following sections.

7.2 The Control Core of the Theorem Prover

The control core of the theorem prover consists of the "Logical Connectives" module together with the "Dispatch" and the "Split" modules. This is the part that decides when and how the individual decision procedures must be invoked. In this section I describe and give pseudo-code algorithms for each of these modules.

7.2.1 Handling the Logical Connectives

In order to make proof search more efficient I restrict the combination of logical connectives that the prover must deal with to those described by the grammar of Figure 7.2. In addition to the absence of the existential quantifier, the other major restriction in this fragment of first-order logic is that the left-hand side of the implication can only contain conjunctions of literals, universal quantifications and a restricted form of implication. The set of literals contains at least equalities and disequalities between variables or, in general, between expressions. The language of expressions and other possible literals are determined by the free functions of the theories implemented by the decision procedures. Even though this might seem a very restrictive fragment of logic, note that the verification-condition generator emits only predicates from within this subset, provided the function specifications and the loop invariants are themselves from P^a. This is, in fact, true for all annotations emitted by the Touchstone compiler.

$$
\begin{array}{llll}
\text{Goals} & P^g & ::= & \mathtt{true} \mid \mathtt{false} \mid P_1^g \wedge P_2^g \mid L \mid P_1^g \vee P_2^g \\
 & & & \mid P_1^a \supset P_2^g \mid \forall x.P^g \\
\text{Assumptions} & P^a & ::= & \mathtt{true} \mid \mathtt{false} \mid L \mid P_1^a \wedge P_2^a \mid P_1^g \supset P_2^a \mid \forall x.P^a \\
\text{Literals} & L & ::= & e_1 = e_2 \mid e_1 \neq e_2 \mid \ldots
\end{array}
$$

Figure 7.2: The fragment of first-order logic handled by the that the theorem prover. For this fragment proving can be goal directed and thus more efficient.

The fragment of first-order logic presented here is a subset of the *first-order hereditary Harrop formulas* and is an extension of Horn logic. These fragments of first-order logic admit a complete sequent-style goal-directed proof system where the declarative meaning of logical connectors coincides with their search-related reading [MNPS91]. The resulting proofs are called uniform. The implementation of uniform-proof search for this logic is shown in Figure 7.3, as a pair of recursive functions. The function `fol` is the main entry point. It takes a predicate and produces either an LF_i representation of a proof of the input predicate or raises the exception `Failure` to signal that it cannot find a proof. The function `assert` is a helper function and it provides the interface with the "Dispatch" module.

$$
\begin{array}{lcll}
\texttt{fol(true)} & = & \boxed{\texttt{truei}} & \\
\texttt{fol(false)} & = & \underline{\textbf{raise}}\ \texttt{Failure} & \\
\texttt{fol}(P_1 \wedge P_2) & = & \boxed{\texttt{andi}(\texttt{fol}(P_1), \texttt{fol}(P_2))} & \\
\texttt{fol}(P_1 \vee P_2) & = & \underline{\textbf{try}} & \\
 & & \quad \boxed{\texttt{oril}(\texttt{fol}(P_1))} & \\
 & & \underline{\textbf{handle}}\ \texttt{Failure} & \\
 & & \quad \boxed{\texttt{orir}(\texttt{fol}(P_2))} & \\
\texttt{fol}(\forall x.P) & = & \boxed{\texttt{alli}(x, \texttt{fol}(P))} & x \text{ is a new variable} \\
\texttt{fol}(P_1 \supset P_2) & = & \underline{\textbf{try}} & \\
 & & \quad \texttt{snapshot()} & \\
 & & \quad \underline{\textbf{try}} & \\
 & & \qquad \texttt{assert}(P_1, \boxed{u}) & u \text{ is a new variable} \\
 & & \qquad \boxed{\texttt{impi}(u, \texttt{fol}(P_2))} & \\
 & & \quad \underline{\textbf{handle}}\ \texttt{Contradiction}(\boxed{prf}) & \\
 & & \qquad \boxed{\texttt{impi}(u, \texttt{falsee}(prf))} & \\
 & & \underline{\textbf{finally}} & \\
 & & \quad \texttt{undo()} & \\
\texttt{fol}(L) & = & \underline{\textbf{try}} & \\
 & & \quad \texttt{snapshot()} & \\
 & & \quad \underline{\textbf{try}} & \\
 & & \qquad \texttt{assert}(\neg L, \boxed{u}) & u \text{ is a new variable} \\
 & & \qquad \texttt{tryAllSplits()} & \\
 & & \qquad \underline{\textbf{raise}}\ \texttt{Failure} & \\
 & & \quad \underline{\textbf{handle}}\ \texttt{Contradiction}(\boxed{prf}) & \\
 & & \qquad \boxed{\texttt{contra}(u, prf)} & \\
 & & \underline{\textbf{finally}} & \\
 & & \quad \texttt{undo()} & \\
\texttt{assert}(\texttt{true}, \boxed{D}) & = & \underline{\textbf{return}} & \\
\texttt{assert}(\texttt{true}, \boxed{D}) & = & \underline{\textbf{raise}}\ \texttt{Contradiction}(\boxed{D}) & \\
\texttt{assert}(P_1 \wedge P_2, \boxed{D}) & = & \texttt{assert}(P_1, \boxed{\texttt{andel}(D)}) & \\
 & & \texttt{assert}(P_2, \boxed{\texttt{ander}(D)}) & \\
\texttt{assert}(\texttt{true}, \boxed{D}) & = & \texttt{dispatch}(L, \boxed{D}) & \\
\texttt{tryAllSplits()} & = & \underline{\textbf{while}}(\neg\texttt{Split.isEmpty()})\ \ \underline{\textbf{do}} & \\
 & & \quad (sg, ptrans) = \texttt{Split.next()} & \\
 & & \quad \underline{\textbf{try}} & \\
 & & \qquad \underline{\textbf{raise}}\ \texttt{Contradiction}(ptrans(\texttt{fol}(sg))) & \\
 & & \quad \underline{\textbf{handle}}\ \texttt{Failure} & \\
 & & \qquad \underline{\textbf{continue}} & \\
\end{array}
$$

Figure 7.3: The definition of the functions responsible for the handling of the logical connectives of first-order logic. The boxed components contain LF_i terms representing proofs.

Before we get to the details of these two functions, let me say a few words about proof representations. As described in detail in Chapter 5, proofs are represented in LF_i as expressions in a language whose constructors correspond directly to the axioms and inference rules of the logic. For example, a proof by conjunction introduction is denoted by the expression "$\mathtt{andi}(\mathtt{D}_1, \mathtt{D}_2)$", where $\mathtt{D}_1$ and $\mathtt{D}_2$ are the expressions that represent the proofs of the two conjuncts respectively. The axiomatization for the fragment of logic considered here is shown in Figure 4.8 on page 77. As a general rule, we consider that for each inference rule having n hypotheses we have an n-ary proof constructor having the same name as the rule. Exceptions are made for the hypothetical and parametric judgments, where the representation also contains a fresh variable whose uses in the hypothetical judgment denote the uses of the hypothesis. For example, a given use of the implication introduction rule can be represented as "$\mathtt{impi}(x, \mathtt{D})$", where x is considered bound in `D` and it marks the places where the left-hand side of the implication is assumed true. To distinguish the proof objects in the pseudo-code presented in this section, they are enclosed in boxes.

The algorithms used by the theorem prover are described in a pseudo-language with exceptions. An exception is raised using a "`raise`" statement and can be handled using a "`try` S_1 `handle` e S_2", where, in the event that the statement S_1 raises the exception e, the exception is consumed and statement S_2 is executed. There is also a statement "`try` S_1 `finally` S_2" that guarantees to run S_2 no matter how the statement S_1 terminates, normally or exceptionally. The exception is not consumed in the latter case. There are two exceptions used by the theorem prover. The exception `Failure` can only be raised by the `fol` function to signal a failed proving attempt. The other exception, `Contradiction`, can be raised by the decision procedures and by the `tryAllSplits` function, which implements case splitting, to signal that a contradiction was found. This exception carries a proof of `false`. Such a proof can be constructed, for example, from a proof of equality and a proof of disequality of the same pair of expressions. From a proof of `false` any predicate can be proved using the false-elimination rules. These rules are shown below:

$$\frac{e_1 = e_2 \qquad e_1 \neq e_2}{\mathtt{false}}\,\mathbf{falsei} \qquad \frac{\mathtt{false}}{P}\,\mathbf{falsee}$$

Back to Figure 7.3, the function `fol` returns a proof of its input predicate. If the input predicate is `false`, no proof exists and the exception `Failure` is raised. In the case of a conjunction, both conjuncts are proved separately and the resulting proof is constructed using `andi`. Note that if any of the two recursive invocations of `fol` raises the exception `Failure`, then the whole proof fails. In the case of a disjunction, the proof of the first disjunct is attempted first. If it succeeds, the resulting proof is build using the or-introduction-left rule. Otherwise, the second disjunct is tried. Note that this case fails only if both proofs fail.

In the case of an implication, the left-hand side must be asserted. For this purpose a new variable u is created to stand for the assumed proof of the left-hand side and the helper function `assert` is called. This function is passed both the assumption to be asserted and its proof. Then, the proof of the right-hand side is attempted and the whole proof is constructed using `impi`. However, we must account for the case when the left-hand side of the implication is a contradiction. In such a case, the invocation of the `assert` function will raise the `Contradiction` exception with the proof of `false`. The innermost exception handler takes care of this case by generating a proof of the implication using the false-elimination rule. Before the execution continues we must retract all the assertions that were made for the purpose of proving the implication. This is done implicitly by a pair of calls to `snapshot`, which instructs all decision procedures to save their current state, and to `undo`, which instructs all decision procedures to revert to the state corresponding to the most recent undone `snapshot`. The `try-finally` statement is used to ensure that the assertions are retracted even if the call to `assert` or to `fol` raise exceptions.

In the case of a literal, its negation is asserted and a contradiction is expected. If the assertion does not generate a contradiction, then the case splits are tried also expecting a contradiction. Otherwise the proof fails and the `Failure` exception is raised. The negation operation can be implemented either by changing the top-level literal constructor, such as changing the equality in a disequality, or simply by having a unary-negation predicate constructor.

The function `assert` takes a predicate and its proof and asserts it. It does not return a result, except that it might raise (or propagate from a decision procedure) the `Contradiction` exception. In the case of a conjunction, both conjuncts are processed in order. In the case of a literal the dispatcher is called. The cases when the assertion is a universal quantification or an implication are not discussed here because they require a moderate amount of extra machinery.

The last function of this module is `tryAllSplits` that is invoked when the proof of a given goal is about to fail. In such a case, the "Split" module is queried and if there are subgoals they are tried in order. Each subgoal is accompanied by a function, called a proof transformer, that given a proof of the subgoal produces a proof of `false`. Section 7.2.3 discusses in more detail case splits and proof transformers.

If the proof of a subgoal succeeds, then the proof transformer is applied to it to produce a proof of `false` and a `Contradiction` exception is raised. If, however, the proof fails, the exception is consumed and the next case is tried. If none of the cases is successful, the function terminates normally. The set of current subgoals can be viewed as a disjunction that must be proved.

Although the functions discussed here are described in a language that supports exceptions, they can be adapted to a language without exceptions by making each function return two results: an indication of whether the return is normal or exceptional and the return value. Also, the same functions can be implemented quite elegantly in a continuation-passing style using a success and a failure continuation. Finally, the only reason we need the functions `snapshot` and `undo` is because the `dispatch` function can have side effects on the state of the decision procedures. A more elegant implementation can use purely-functional decision procedures whose state is passed around explicitly by the `fol` and `assert` functions.

$$
\begin{array}{lll}
\texttt{dispatch}(L,D) & = & al\ = \{(L,D)\} \\
 & & \underline{\textbf{while}}(al\ \neq \emptyset)\ \ \underline{\textbf{do}} \\
 & & \quad accum\ = \emptyset \\
 & & \quad \underline{\textbf{foreach}}\ (L,D) \in al\ \ \underline{\textbf{do}} \\
 & & \quad\quad \underline{\textbf{foreach}}\ i \in \texttt{DecisionProcedures}\ \ \underline{\textbf{do}} \\
 & & \quad\quad\quad accum\ = accum\ \cup \texttt{decproc}_i(L,D) \\
 & & \quad al\ = accum \\
 & & \\
\texttt{snapshot}() & = & \texttt{Split.snapshot}() \\
 & & \underline{\textbf{foreach}}\ i \in \texttt{DecisionProcedures}\ \ \underline{\textbf{do}} \\
 & & \quad \texttt{snapshot}_i() \\
 & & \\
\texttt{undo}() & = & \texttt{Split.undo}() \\
 & & \underline{\textbf{foreach}}\ i \in \texttt{DecisionProcedures}\ \ \underline{\textbf{do}} \\
 & & \quad \texttt{undo}_i()
\end{array}
$$

Figure 7.4: The definition of the dispatcher functions. The functions `snapshot` and `undo` are dispatched to all the decision procedures to instruct them to save or undo their internal state. The function `fixpoint` invokes the all the decision procedures until no new assertions are produced.

7.2.2 The Dispatcher Module

The implementation of the "Dispatch" module is quite simple. It needs only to ensure that assertions along with their proofs are broadcast to all decision procedures. Also, while processing an assertion, a decision procedure might return a set of new equality assertions together with their proofs. Such assertions must also be propagated back to all decision procedures. The process must continue until no new assertion is generated. Thus the `dispatch` function, whose implementation is shown in Figure 7.4, is essentially a fixpoint computation.

In addition to the `dispatch` function, the "Dispatch" module also implements the system-wide `snapshot` and `undo` functions. Their implementations simply forward the messages to the "Split" module and to all decision procedures. Thus, the "Dispatch" module is the only module that needs to know which decision procedures are present in the system.

7.2.3 Handling Case Splits

As discussed in Section 7.1 and illustrated in the example 7.3, for certain decision procedures a conjunction of literals does not entail any new equalities between variables but it does entail a disjunction of such equalities. To allow such decision procedures in the theorem prover I introduce the "Split" module. In fact, the "Split" module can perform a more general form of case split. Whenever a decision procedure cannot prove a contradiction and cannot generate any new equalities between variables, it should look for subgoals that, if proved, would entail a contradiction. Such subgoals can be arbitrary predicates from the class P^g. These subgoals, together with a function that, when given a proof of the subgoal, produces a proof of `false`, must be given to the "Split" module using the function `Split.add`. The proof transformer function can be viewed as a proof with a hole inside that can be later filled with the proof of the subgoal when it is proved.

To illustrate how this more general approach to case splits can deal with the disjunction of equalities entailed by non-convex decision procedures, consider again the disjunction $x = y \lor x = z$ from the example 7.3. The decision procedure that discovers this disjunction has implicitly discovered that its negation, $x \neq y \land x \neq z$, is a subgoal that, if proved, can lead to a contradiction. The proof transformer for this case is shown below. The outermost rule used is the disjunction elimination rule, containing two hypothetical judgments, marked here with u and v, for the two disjuncts. The box stands for the hole where a proof of the subgoal must be plugged in.

$$\dfrac{x = y \lor x = z \qquad \dfrac{\dfrac{}{x = y}\,u \qquad \dfrac{\boxed{}}{x \neq y}}{\texttt{false}} \qquad \dfrac{\dfrac{}{x = z}\,v \qquad \dfrac{\boxed{}}{x \neq z}}{\texttt{false}}}{\texttt{false}}$$

The "Split" module exports several functions to all other modules. We have seen already that the "Logical Connectives" module uses `isEmpty` and `next` to extract all of the splits one by one. Also, the "Dispatch" module invokes the functions `snapshot` and `undo`. In addition to these, there is also the `add` function that is used by the non-convex decision procedures that wish to add case splits.

Figure 7.5 shows a possible implementation of the storage and manipulation of splits. The main data structure is a stack of sets of splits. The purpose of the stack is to record the state at the time of a `snapshot`. For a clearer presentation, both the `Stack` and the `Set` manipulation functions are fully functional. This stack data structure is held in a variable, called `splits`, that is initialized to a one-element stack containing an empty set. Non-convex decision procedures can add new subgoals by calling the function `Split.add` with a subgoal and a proof transformer. When this happens, the pair of the subgoal and the proof transformer are added to the set that is on top of the stack. Note that the function `Stack.top` just returns the set that is on the top of the stack without removing it.

The implementation of `snapshot` just makes a copy of the current set of splits (the one on top of the stack) and pushes it on top of the stack. The `undo` function just pops and discards the element that is on top of the stack. Finally, the `next` function returns one arbitrary element from the current set of splits and removes it from the set. For this purpose

$$
\begin{array}{lll}
\texttt{var}\ splits & : & \texttt{Stack of (Set of (pred} * \texttt{(proof} \rightarrow \texttt{proof)))} \\
 & & \qquad = \texttt{Stack.push (Stack.empty, Set.empty)} \\
\texttt{add}(sg, ptrans) & = & splits = \texttt{Stack.replaceTop}(splits, \texttt{Set.add}(\texttt{Stack.top}(splits), \\
 & & \qquad (sg, ptrans))) \\
\texttt{snapshot}() & = & splits = \texttt{Stack.push}(splits, \texttt{Stack.top}(splits)) \\
\texttt{undo}() & = & splits = \texttt{Stack.pop}(splits) \\
\texttt{isEmpty}() & = & \texttt{Set.isEmpty}(\texttt{Stack.top}(splits) \\
\texttt{next}() & = & ((sg, ptrans), rest) = \texttt{Set.pickOne}(\texttt{Stack.top}(splits)) \\
 & & splits = \texttt{Stack.replaceTop}(splits, rest) \\
 & & \underline{\textbf{return}}\ (sg, ptrans)
\end{array}
$$

Figure 7.5: The definition of the functions implementing the Split module of the theorem prover.

it uses the function `Set.pickOne` that returns both an element of the set and the rest of the set. In my implementation `Set.pickOne` returns the most recently added split.

This completes the description of the modules responsible with the control of the decision procedures. In the next section, I describe the general requirements of a proof-generating decision procedure and then several particular decision procedures used in the theorem prover.

7.3 The Decision Procedures

The control core of the theorem prover decomposes predicates into literals and passes them to the decision procedures, by means of the "Dispatch" module. The decision procedures must analyze the literals passed to them and must discover either a contradiction or new equalities between variables or a case split. The major advantage of this architecture is that it is easily extensible. To add a new theory all we must do is to write a cooperating decision procedure for the theory and plug it in the system.

In this section I describe first in general terms what are the requirements on the decision procedures. Then, in separate subsections, I discuss several examples of decision procedures used in realistic theorem provers. The major innovation with respect to the Nelson and Oppen approach is that the decision procedures and the control core are adapted to emit natural deduction proofs of all predicates that are proved.

All of the decision procedures in the theorem prover use the same internal representation of the literals, namely a global *expression directed acyclic graph*, which I refer to as the E-DAG. In the E-DAG, there is a node corresponding to each expression or subexpression in the predicate to be proved. Each node is labeled with an expression constructor and it has as many descendants as the arity of the constructor. When building the E-DAG care must be taken to ensure that all common subexpressions are shared. Then, the E-DAG

```
procedure snapshot()
procedure undo()
function decproc(a : pred, d : proof) : Set of (pred * proof)
         raises Contradiction (proof)
```

Figure 7.6: The interface that all decision procedures must implement.

node corresponding to an expression can be used throughout the system to refer to that expression.

The first thing that must be done with an assertion coming from the "Logical Connectives" module is to internalize all of its constituent expressions in the E-DAG. Only then can the assertion be rephrased in terms of E-DAG nodes and propagated to the decision procedures. The internalization is done by the "Dispatch" module in my implementation, right before the fixpoint operation is initiated.

In addition to the global E-DAG, each decision procedure is free to maintain its own internal data structures. Each decision procedure is required to implement the interface shown in Figure 7.6. These functions are discussed next.

The control core of the theorem prover ensures that for each `snapshot` there is a matching `undo`, although there might be several properly nested pairs of `snapshot` and `undo` in between. The intended semantics of the `undo` operation is that all decision procedures should adjust their internal data structures so that they do not reflect assertions that were made chronologically after the matching `snapshot` operation. This ensures that assertions are properly retracted from the system at the time of the `undo`.

There are several ways to implement the `snapshot` and `undo` operations. A decision procedure can maintain a stack of assertions. Each time a new assertion arrives it is considered with respect to the assertions memorized on the stack. Each `snapshot` places a marker on the stack and each `undo` pops the stack up to the nearest marker. This simple strategy is appropriate for those decision procedures that do not require the maintenance of separate data structures, except the undo stack. An example of such a decision procedure is described in Section 7.3.4 for dealing with the Touchstone typing rules.

Another decision procedure style uses an internal data structure where it records relationships between the current assertions. In such a case, however, the implementation of `snapshot` and `undo` is more involved. One possibility is for `snapshot` to make a complete copy of the state of the internal data structures, so that `undo` can recreate an identical state. This is not only too expensive but, in most cases, unnecessary. The `undo` operation needs only to recreate an equivalent state, not an identical one. This can usually be accomplished with minor changes to the current state, if the choice of the internal data structures is made wisely. An indication of the particular changes that must be performed at each `undo` is stored by `snapshot` in an *undo stack*. A decision procedure that is implemented in this style is called *incremental* and *undoable*. Such a decision procedure is more efficient than a non-incremental one but it requires considerably more care from the implementor. Examples of such decision procedures are the congruence closure decision procedure (described in Section 7.3.1) for dealing with equalities and uninterpreted function symbols, and Simplex (described in Section 7.3.2) for dealing with linear arithmetic.

The main operation exported by a decision procedure is the `decproc` function, whose type is shown in Figure 7.6. This is the function that processes each new assertion. First, it must scan the assertion predicate for occurrences of the free functions of the theory that the decision procedure is implementing. Because the assertion predicate is internalized, the E-DAG nodes serve as convenient names for subexpressions. According to Definition 7.2, the decision procedure must either detect a contradiction, in which case it must raise the `Contradiction` exception with a proof of `false`, or it must detect new equalities between E-DAG nodes. A set of such equality assertions, together with their proofs, is returned in case of normal return.

In addition to detecting a contradiction or returning a set of equality assertions, a decision procedure can also generate case splits. It can create subgoals that, when proven, would allow the decision procedure to generate a contradiction. Each such subgoal is passed, together with a function that when given a proof of the subgoal creates a proof of `false`, to the `Split.add` function. Even though it is the decision procedure who identifies the subgoals, it is up to the "Logical Connectives" and "Split" modules to decide when to try the subgoals. In fact, the subgoals are tried only if a goal is about to fail.

In the next few sections I will describe the main decision procedures present in my implementation of the proof-generating theorem prover.

7.3.1 Handling Equality with Congruence Closures

A central theory in any implementation of the Nelson-Oppen architecture is the theory of equality. This is because each decision procedure in the system must be able to discover and manipulate equality predicates. The free functions of the theory $\mathbb{E}$ are "=" and "$\neq$" along with any uninterpreted function symbols. The axioms of the theory are those shown in Figure 7.7. There is one inference rule for each function symbol in the system.

The theory $\mathbb{E}$ was first shown decidable by Ackermann [Ack54]. The problem of finding whether an equality is a consequence of other equalities can be tackled from several points of view. Downey, Sethi and Tarjan [DST80] view it as a variation of the common subexpression problem and Kozen [Koz77] as a word problem in finitely presented algebras. Decision procedures for $\mathbb{E}$ are also given by Shostak [Sho78] and by Nelson and Oppen [NO80].

All of these problems can be reduced to the problem of constructing the congruence closure of a relation on a graph. If $\mathcal{R}$ is an equivalence relation over a set of terms, we say that two terms $\mathtt{f}(t_1, \ldots, t_n)$ and $\mathtt{f}(t'_1, \ldots, t'_n)$ are *congruent* if t_i is related to t'_i by $\mathcal{R}$ for all $i = 1, \ldots, n$. $\mathcal{R}$ is *closed under congruences* if all congruent terms are also related in $\mathcal{R}$.

$$\frac{}{e = e}\textbf{eqid} \quad \frac{e_2 = e_1}{e_1 = e_2}\textbf{eqsym} \quad \frac{e_1 = e_2 \qquad e_2 = e_3}{e_1 = e_3}\textbf{eqtr} \quad \frac{e_1 = e_2 \qquad e_1 \neq e_2}{\texttt{false}}\textbf{falsei}$$

$$\frac{e_1 = e_1' \qquad \cdots \qquad e_n = e_n'}{\texttt{f}(e_1, \ldots, e_n) = \texttt{f}(e_1', \ldots, e_n')}\textbf{congr}$$

Figure 7.7: The axioms of the theory $\mathbb{E}$ of equality. There are congruence inference rules for each function symbol in the system.

The *congruence closure* of a relation $\mathcal{R}$ is the smallest extension of $\mathcal{R}$ that is an equivalence relation closed under congruences.

To see if a given equality "$t = u$" follows from a set of equalities $\mathcal{R}$, we first construct the congruence closure $\mathcal{R}'$ of $\mathcal{R}$ and then check to see if $t = u \in \mathcal{R}'$. This is the sense in which an algorithm for computing the congruence closure of a set of equalities can be at the base of a decision procedure for $\mathbb{E}$.

In this section I describe an implementation of the congruence closure algorithm adapted to work as a proof-generating decision procedure in a Nelson-Oppen theorem prover. This implementation is inspired by that shown by Nelson [Nel81] although the emphasis here is not on efficiency but on simplicity and on the proof generation aspect of proving.

In addition to the E-DAG, the congruence closure algorithm uses its own internal data structures described here as functions defined on nodes. For each node e in the E-DAG, the decision procedure maintains the root of its equivalence class $\texttt{root}(e)$. The equivalence class of e, denoted by $class(e)$, is defined as the set of nodes having the same `root` as e. Then, $\texttt{parents}(e)$ is a set of nodes that have arguments within the equivalence class of e. Finally, $\texttt{forbid}(e)$ is the set of nodes that are specified to be not equal to e, by means of disequality assertions. In addition to these functions, the decision procedure maintains also a stack of assertions, either equalities or disequalities, along with their proofs. The latter data structure is called the `undoStack` and it plays two roles. First, it stores enough information to enable the decision procedure to undo the changes to the internal state when requested by the "Dispatch" module. Second, the assertions on the `undoStack` are used to produce the proofs of equalities propagated by the congruence closure decision procedure or the proofs required for contradictions.

Figure 7.8 describes the main invariants that must be maintained by the implementation of the congruence closure. The invariant CC1 just says that the root of an equivalence class belongs to the class itself. (Remember that the equivalence class is defined indirectly as the set of elements with the same value of `root`.) The invariant CC2 says that each entry in the `undoStack` is either an equality or a disequality assertion along with its proof.[2] The

[2] Actually, an implementation might store additional information on the `undoStack` for the purpose of

CC1. $\texttt{root}(\texttt{root}(e)) = \texttt{root}(e)$

CC2. for all $(p, d) \in$ `undoStack`, d is the representation of a proof of p, where p can be either an equality or a disequality between E-DAG nodes

CC3. $\texttt{root}(a) = \texttt{root}(b)$ if and only if $a = b$ or there exist $a_1, \ldots, a_{n+1}$ such that

- $a = a_1$, and
- $b = a_{n+1}$, and
- $(a_i = a_{i+1}, \texttt{pf}_i) \in$ `undoStack` or $(a_{i+1} = a_i, \texttt{pf}_i) \in$ `undoStack` for all $i = 1, \ldots, n$

CC4. a $\in \texttt{parents}(\texttt{root}(\text{b}))$ if and only if there exist $e \in \mathit{class}(b)$ and i such that $e = \texttt{arg}_i(a)$

CC5. $\mathit{class}(\text{a}) \cap \texttt{forbid}(\texttt{root}(\text{b})) \neq \emptyset$ if and only if there exist $a' \in \mathit{class}(a)$ and $b' \in \mathit{class}(b)$ such that $(a' \neq b', \text{neqab}) \in$ `undoStack`.

Figure 7.8: The invariants maintained by the congruence closure decision procedure, where *class*(a) is a shorthand notation for the set $\{e \mid \texttt{root}(e) = \texttt{root}(a)\}$.

invariant CC3 says that two nodes are in the same equivalence class if and only if they are in the equivalence relation induced by the equality assertions from the `undoStack`. The "if" part of the invariant ensures the completeness of the representation and the "only if" part ensures soundness. The invariant CC4 says that the parents of a root node are exactly those elements that have an argument within the equivalence class of the node. Note that the `parents` function is only specified for the root nodes. The `parents` set is used to speed-up the search for node pairs that might become congruent due to changes in equivalence classes. Finally, the invariant CC5 specifies that the `forbid` function of an equivalence class contains a member of another class if and only if the two classes are specified not-equal in the `undoStack`. Just like `parents`, the `forbid` function is specified only for the root nodes because it is a property of a class rather than of an individual node.

Figure 7.9 shows the core of the implementation of congruence closure. This core consists of two main entry points: `merge` that is used to merge two equivalence classes in response to an external equality assertion or when a congruence is discovered, and `forbidMerge` that is used to forbid the equality of two equivalence classes in response to an external disequality assertion. Then there is the helper function `checkCongr` that detects and propagates congruences and two functions, `prfEq` and `mkEqContra`, used to generate proofs.

The function `merge` is called with two nodes and a proof of their equality for the purpose of combining their equivalence classes. The root of the combined class will be the root of

implementing the `undo` operation.

the second input class. But first, the function checks whether either class is forbidden to be merged with the other. If it is, then a contradiction is raised with a proof of `false` produced by the helper function `mkEqContra`. Note how the symmetry of equality is used in the second invocation to convert a proof of "$a = b$" into a proof of "$b = a$". If the contradiction checks are unsuccessful then a series of changes must be made to the internal data structures and an entry in the `undoStack` is made to record the changes. First, the `forbid` set for the new class is updated, then the `parents` set of the new class is computed. Finally, the first input class is re-rooted to the new root. The result of the `merge` function is a set of assertions with their proofs. One of these assertions is that the recently combined classes are equal. To this assertion we add all congruences discovered by the `checkCongr` function.

The `checkCongr` function receives two sets of nodes and must find a pair of congruent nodes among them. When two nodes are found congruent, a proof of equality is constructed using the `congr` rule and the merge function is called recursively to combine the equivalence classes. Note that only `merge` is changing the internal data structures.

When an external disequality assertion is encountered, the function `forbidMerge` is invoked with the two nodes that must not be equal along with a proof of their disequality. This time a contradiction is found when the two nodes are in the same equivalence class. In this case a contradiction is raised with a proof of `false` constructed using the false-introduction rule along with a proof of equality and disequality of the same two nodes.

The proof of equality of two nodes from the same equivalence class is constructed using the helper function `prfEq`. It is here where the invariant CC3 becomes significant. It guarantees that for any two nodes in the same equivalence graph there is a path in the undirected graph defined by the equality assertions in the `undoStack`. This path leads immediately to a proof of equality build using transitivity from the proof of equality of the individual hops. Occasionally, the symmetry rule must be used to adjust the order of the variables in individual hops.

Finally, the function `mkEqContra` creates a proof of `false` from a proof of equality of nodes that are also forbidden to be equal. Here we rely on the invariant CC5 to find the disequality assertion from the `undoStack` that is responsible for the contradiction. Then a series of equalities are constructed to contradict the disequality from `undoStack`.

Note that congruence closure is a convex decision procedure and thus, it does not need to create case splits. Also, as proven in [NO80] and [Nel81] for similar implementations, the congruence closure algorithm is a sound and complete decision procedure for $\mathbb{E}$.

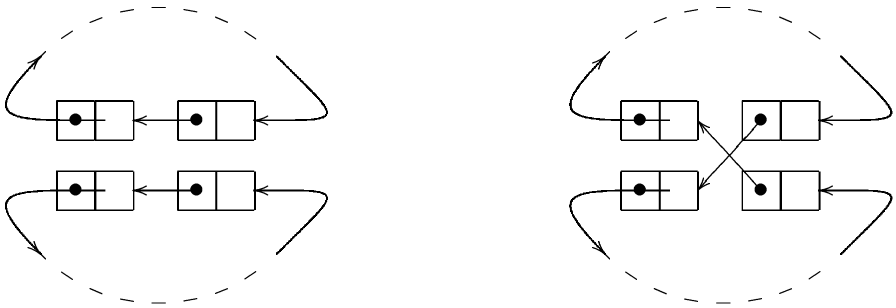
In addition to the functions shown in Figure 7.9, the decision procedure must also implement the `snapshot` and `undo` function. A typical implementation of `snapshot` is to place a special marker on the `undoStack` so that `undo` knows how many operations to undo. There are two places where the internal state is changed. One is in the `merge` function, for which an equality assertion is put in the `undoStack`, and the other is in the `forbidMerge` function, where a disequality assertion is pushed on the `undoStack`. As a general rule, each of these

```
merge(a : node, b : node, eqab : proof) =                     /* eqab is a proof of a = b */
if root(a) ≠ root(b) then
   if class(a) ∩ forbid(root(b)) ≠ ∅ then
      raise Contradiction(mkEqContra(a, b, eqab))
   else if class(b) ∩ forbid(root(a)) ≠ ∅ then
      raise Contradiction(mkEqContra(b, a, eqsym(eqab)))
   else
      undoStack = Stack.push(undoStack, (a = b, eqab))
      forbid(root(b)) = forbid(root(b)) ∪ forbid(root(a)))
      pa = parents(root(a))
      pb = parents(root(b))
      parents(root(b)) = pa ∪ pb
      foreach a' ∈ class(a)
         root(a') = root(b)
      return {(a = b, eqab)} ∪ checkCongr(pa, pb)

checkCongr(pa : node set, pb : node set) : (pred * proof) set =
   accum = ∅
   foreach a ∈ pa, b ∈ pb
      if head(a) = head(b) ∧ nrarg(a) = nrarg(b) = n ∧ root(arg_i(a)) = root(arg_i(b))
         eqab = congr(head(a), prfEq(arg_1(a), arg_1(b)), ... , prfEq(arg_n(a), arg_n(b)))
         accum = accum ∪ merge(a, b, eqab)
   return accum

forbidMerge(a : node, b : node, neqab : proof) =
   if root(a) = root(b) then
      raise Contradiction(falsei(prfEq(a, b), neqab))
   else
      undoStack = Stack.push(undoStack, (a ≠ b, neqab))
      forbid(root(b)) = forbid(root(b)) ∪ {a}

prfEq(a : node, b : node) =
   let a_1, ... , a_{n+1} as in the invariant CC3
```

$$pf'_i = \begin{cases} pf_i & \text{if } (a_i = a_{i+1}, pf_i) \in \texttt{undoStack} \\ \texttt{eqsym}(pf_i) & \text{if } (a_{i+1} = a_i, pf_i) \in \texttt{undoStack} \end{cases}$$

```
   return eqtr(pf'_1, eqtr(pf'_2, ... , eqtr(pf'_{n-1}, pf'_n) ...))

mkEqContra(a, b, eqab) =                         /* class(a) ∩ forbid(root(b)) ≠ ∅ */
   let a' ∈ class(a), b' ∈ class(b) such that (a' ≠ b', neqab) ∈ undoStack
   return falsei(eqtr(prfEq(a', a), eqtr(eqab, prfEq(b, b'))), neqab)
```

Figure 7.9: The main algorithms defining the congruence closure decision procedure.



(a) (b)

Figure 7.10: An easily undoable implementation of set union can be obtained by representing sets as circular lists and then swapping the contents of two `cdr` cells. This operation can be undone by swapping again the contents of the same `cdr` cells.

kinds of entries in the `undoStack` can be enriched with additional information for use by the `undo` operation. For illustration purposes, in Section 7.3.3 I show the behavior of the congruence closure decision procedure in the case of a concrete example.

Nelson [Nel81] shows a complete implementation of congruence closure, complete with the `undo` operation. The basic trick they recommend is to implement sets as circular lists. This representation supports well the scanning required for the `forbid` and `parents` sets and has the advantage that the union operation can be done by simply swapping the contents of two `cdr` cells, as shown in Figure 7.10. This operation, called splicing, can be easily undone by swapping again the same `cdr` cells, which incidentally are exactly those corresponding to the nodes mentioned in the `undoStack` assertion.

To complete the section on the congruence closure algorithm I mention that its complexity is determined by the method used to discover the congruent pairs of nodes. The function `checkCongr` shown in Figure 7.9 uses a nested iteration over the two predecessor lists, yielding an algorithm of complexity $\mathcal{O}(n^2)$, where n is the number of nodes in the E-DAG. If the `parents` lists are maintained sorted lexicographically then we can obtain an algorithm of complexity $\mathcal{O}(n \log n)$.

7.3.2 Handling Linear Arithmetic with Simplex

Let $\mathbb{Z}$ be the theory of integers with the free functions containing the integer numerals and the usual arithmetic and comparison operators $\{+, -, \geq, >, \leq, <\}$. As a notational convenience we also allow multiplication by integer numerals. The decision problem for $\mathbb{Z}$ is essentially the problem of deciding whether one linear inequality is a consequence of several other inequalities. This is equivalent to the general linear programming problem but, because the instances of the problem that arise in program verification differ significantly from those arising in operations research, verification researchers have devised special new methods and have modified traditional linear programming methods to handle the cases they encountered in practice.

For example, Pratt [Pra77] reports that most inequalities encountered in program verification are of the simple form "$x \leq y + k$", where k is an integer numeral while x and y are variables. A decision procedure for this fragment can be obtained by detecting negative cycles in weighted directed graphs, as we did for example in Section 6.5.9 as part of the Touchstone optimizer. Shostak [Sho81] generalizes this method to inequalities of the form "$ax + by \leq k$", where a and b are integer numerals. Decision procedures for the same fragment are also described by Nelson [Nel78] and by Aspvall and Shiloach [AS80]. There is also a variety of decision procedures for the general case of linear inequalities. Some theorem provers have used the Fourier-Motzkin elimination method, some have used the Sup-Inf method due to Bledsoe [Ble74] and others have used variations of the Simplex linear programming algorithm [Nel81].[3]

In this section I describe the use of a variant of the Simplex linear programming algorithm as a proof-generating decision procedure for linear arithmetic. The emphasis in this presentation is on how the proofs are generated and not so much on the internals of the algorithm. The same variant of Simplex, although without the proof-generating component, is described in detail by Nelson [Nel81]. The curious reader can find there the implementation details that are missing here.

As any decision procedure in a Nelson-Oppen theorem prover, the linear arithmetic decision procedure first isolates from the incoming assertions the subexpressions that involve only the free functions of $\mathbb{Z}$, namely the integers and any of the integer operators $\{+, -, \geq, >, \leq, <\}$. I will use the letter e to denote expressions built using only variables, numerals and the additive operators $+$ and $-$. In addition, the notation $n \cdot e$ is a shorthand for the n-way addition $e + \cdots + e$.

If ρ is a mapping of variables to integer numerals, then I write $\rho(e)$ to denote the integer value obtained by evaluating e after all variables have been replaced with their value in ρ. I

[3] All of these decision procedures, except for Pratt's loop residue, are actually decision procedures for $\mathbb{Q}$ and not for $\mathbb{Z}$. However, given the kinds of inequalities arising in the practice of program verification, the integer case can be handled quite satisfactorily by a combination of a decision procedure for rationals along with heuristics.

write $e_1 \simeq e_2$ when, for any mapping ρ of variables to integers, we have that $\rho(e_1) = \rho(e_2)$.

The internal data structure used by the Simplex algorithm is the *tableau*, which can be depicted in the form of a matrix, as shown below:

$$
\begin{array}{c|cccccc}
 & & C(1) & \cdots & C(j) & \cdots & C(c) \\
\hline
R(1) & T_{10} & T_{11} & \cdots & T_{1j} & \cdots & T_{1c} \\
\vdots & \vdots & \vdots & & \vdots & & \vdots \\
R(i) & T_{i0} & T_{i1} & \cdots & T_{ij} & \cdots & T_{ic} \\
\vdots & \vdots & \vdots & & \vdots & & \vdots \\
R(r) & T_{r0} & T_{r1} & \cdots & T_{rj} & \cdots & T_{rc}
\end{array}
$$

All of the entries T_{ij} are rational numbers. Except for column 0, all other rows and columns are *owned* by expressions. The expression that owns the row i is denoted by $R(i)$ and the expression that owns the column j is denoted by $C(j)$.

The Simplex tableau as described so far encodes only the linear relationships between the owning expressions. The actual encoding of the input assertions is by means of *restrictions* on the values that the owning expressions can take. There are two kinds of restrictions that Simplex must maintain. A row or a column can be either *+-restricted*, which means that the owning expression can take only values greater or equal to zero, or *$*$-restricted*, in which case the owning expression can only be equal to zero. As explained in [Nel81], the reason the $*$-restricted rows and columns are not simply deleted from the tableau is to support an inexpensive implementation of the `undo` operation. Also, it turns out that maintaining the $*$-restricted rows and columns is useful for proof generation.

I will describe the operation of the Simplex algorithm indirectly, by means of a set of invariants stated below as Invariant 7.5. To understand this invariant, consider that the tableau encodes a set of inequality constraints, one "≥ 0" inequality for each restricted owner. In fact, Invariant 7.5c says that each restricted owner has a proof that it is positive. The actual tableau entries encode only the linear relationships between the owners involved. If n is the number of variables among the owners of rows and columns (and by Invariant 7.5a, the set of variables in the current set of literals), then it is well known that the set of points in $\mathbb{R}^n$ that satisfy a set of linear inequations is a convex polyhedron, also called the *solution set*. As long and the polyhedron is non-empty, the set of inequations is satisfiable in $\mathbb{R}$, and also in $\mathbb{Q}$ if the coefficients are rational numbers. This does not necessarily mean that it is also satisfiable in $\mathbb{Z}$, but an empty polyhedron in $\mathbb{R}^n$ means that the inequations are not satisfiable in $\mathbb{Z}$. This means that the Simplex method is sound but not complete for integer inequalities.

We make the convention, following [Nel81], that the current state of the tableau represents a unique point, called the *sample point*, in $\mathbb{Q}^n$ where all expressions that own columns have value zero. This means that the expression that owns the row i has value T_{i0} at the sample point. Invariant 7.5 and the operation of the Simplex decision procedure relies on the fact that at any given moment the sample point of the tableau is in the solution set. This is the motivation behind Invariant 7.5d. A $*$-restricted row is one whose owner expression has value zero in all solution points. Such rows and columns are detected using the notion of a maximized row, defined below.

Definition 7.4 *A tableau row i is said to be manifestly maximized at T_{i0} if and only if its non-zero entries are either in $*$-restricted columns or are negative and in $+$-restricted columns.*

The intuition behind Definition 7.4 is that a linear combination of restricted expressions, such that those that are $+$-restricted have negative factors, is guaranteed to be at most zero. The owner of row i in Definition 7.4 is such an expression. Taking into account the constant entry in row i, we see that the maximal value that $R(i)$ can take in the solution set is T_{i0}.

Now, if a $+$-restricted row is maximized at zero, it means that it can only take the value zero, and it is thus safe to make it a $*$-restricted row. When we do that we can also make $*$-restricted all of the $+$-restricted columns that have non-zero entries in row i. This is the mechanism by which rows and columns become $*$-restricted, hence the Invariants 7.5e and 7.5f.

Invariant 7.5

(a) All variables occurring in the current set of literals own either exactly one row or, if not, exactly one column.

(b) For any row $i = 1, \ldots, r$ in a Simplex tableau, there is a positive integer constant m_i such that $m_i T_{ij}$ is an integer for all $j = 0, \ldots, c$ and such that:

$$m_i \cdot R(i) \simeq m_i T_{i0} + \sum_{j=1}^{c} (m_i T_{ij}) \cdot C(j)$$

(c) If row i is restricted (either $+$-restricted or $*$-restricted) then there is a proof of $R(i) \geq 0$. We refer to this proof as Proof$(R(i))$. A similar fact holds for restricted columns.

(d) If row i is $+$-restricted then $T_{i0} \geq 0$

(e) If row i is $*$-restricted then $T_{i0} = 0$ and all non-zero T_{ij} are in columns that are $*$-restricted

(f) If column j is $*$-restricted then there exists a $*$-restricted row i such that $T_{ij} < 0$ and $T_{ik} \leq 0$ for all $k > j$. Such a row i is called the restrictor of column j.

The main operation performed on the Simplex tableau is *pivoting*, or Gaussian elimination. To pivot row u and column v (where $T_{uv} \neq 0$) means to rewrite the contents of the tableau so that the expressions that own the row u and column v are swapped. The invariant Invariant 7.5b for row u is used to extract the value of $C(v)$ in terms of $R(u)$ and the other columns. Then, the other rows are rewritten by replacing $C(v)$ with the computed expression and carrying out the required algebraic simplifications. The new values for the tableau entries are shown below. Note that the owners of row u and column v have been swapped.

$$
\begin{array}{rr|ccccccc}
 & & & & & & & \text{column } v & \\
 & & & & & & & \downarrow & \\
 & & & \cdots & C(j) & \cdots & & R(u) & \cdots \\
\hline
 & \vdots & \vdots & & \vdots & & & \vdots & \\
 & R(i) & T_{i0} - \dfrac{T_{u0} \cdot T_{iv}}{T_{uv}} & \cdots & T_{ij} - \dfrac{T_{uj} \cdot T_{iv}}{T_{uv}} & \cdots & & \dfrac{T_{iv}}{T_{uv}} & \cdots \\
 & \vdots & \vdots & & \vdots & & & \vdots & \\
\text{row } u \rightarrow & C(v) & -\dfrac{T_{u0}}{T_{uv}} & \cdots & -\dfrac{T_{uj}}{T_{uv}} & \cdots & & \dfrac{1}{T_{uv}} & \cdots \\
 & \vdots & \vdots & & \vdots & & & \vdots &
\end{array}
$$

The core of the Simplex decision procedure is concerned with choosing the pivots. Not all combinations of rows and columns are valid pivots and some of them lead to better performance. The definition of validity of pivots is given below. The algorithm used for choosing valid pivots is described in detail in [Nel81].

Definition 7.6 *A pivot (u, v) is valid if and only if*

1. $T_{uv} \neq 0$, *and*

2. *both row u and column v are not $*$-restricted, and*

3. *the tableau after pivoting satisfies Invariant 7.5d.*

The main significance of a valid pivot operation is that it preserves the Invariant 7.5. The 7.5a part is preserved trivially because the set of row and column owners does not change through pivoting. The preservation of Invariant 7.5b can be verified through some simple but tedious algebraic manipulation. This is not surprising given that this is the criterion behind the definition of the pivoting operation. Invariant 7.5d holds trivially because of the definition of valid pivots and the parts 7.5e and 7.5f of Invariant 7.5 are true because all $*$-restricted rows i are left untouched by pivoting, because v is not a restricted column and by Invariant 7.5e we know that T_{iv} is zero. Another operation that does preserve Invariant 7.5

$$\frac{}{e_1 = e_2}\,\mathbf{arith}(e_1,\, e_2) \quad \text{if } e_1 \simeq e_2$$

$$\frac{}{n \geq 0}\,\mathbf{geqct}(n) \quad \text{if } n \text{ is a constant greater or equal to } 0$$

Figure 7.11: The kernel rules for arithmetic introduced in Section 5.5.

is a permutation of columns such that the $*$-restricted columns do not change their relative order.

Before we proceed to discuss the implementation of the Simplex decision procedure, we have to establish the set of axioms and inference rules that Simplex can use to build proofs. As described in detail in Section 5.5, it is convenient to deal with integer numerals directly and not by means of a unary or binary encoding. But this means that the theory is not finitely axiomatizable and hence, not directly implementable in LF_i. To overcome this problem, I have shown in Section 5.5 how to extend LF_i type reconstruction so that it can check proofs involving two special proof constructors `arith` and `geqct` shown in Figure 7.11. Recall that when the proof checker encounters the proof object "$\mathtt{arith}(e_1, e_2)$" it checks that $e_1 - e_2$ can be reduced to 0 by using simple algebraic simplification rules, or equivalently that $e_1 \simeq e_2$. If this is the case then the proof term is considered to be a valid representation of a proof of the predicate $e_1 = e_2$. Similarly, the proof object "$\mathtt{geqct}(n)$" is considered to be a valid representation of a proof of "$n \geq 0$", if the integer numeral n is indeed greater or equal to zero.

In addition to these kernel proof rules, we define the set of axioms shown in Figure 7.12. At this point in the presentation, it might not be obvious how Simplex is using these rules. So, for the time being, we should only be concerned with their soundness. The first four rules shown in the top row are used to convert the various inequality literals to the canonical form "$e \geq 0$". Similarly, the rule "`eqgeq`" is used to convert an equality to an inequality. The rules "`geqadd`" along with "`geq0`" are used to construct a proof that a linear combination of positive expressions with positive factors is also positive. For this purpose, we can use the "`geqadd`" rule as many times as is the length of the linear combination and then use "`geq0`" to terminate the chain. The two remaining rules are the top level rules used by Simplex to produce the proofs of `false` in case of a contradiction and the proof of equality of two variables. Although both rules are sound, their form is somewhat strange. This is because they were carefully formulated to fit precisely the situations when Simplex needs them. To verify that rule "`falsei`" is indeed sound, notice that we have the sequence of inequalities "$m \cdot e_1 = n - e_2 \leq n \leq -1$". This contradicts the first hypothesis "$e_1 \geq 0$". Finally, note that in the rule `eqi` the first group of hypotheses proves that "$e_2 \geq e_1$" while the second that "$e_1 \geq e_2$", thus it is valid to infer that "$e_1 = e_2$". Having settled the axiomatization of the theory $\mathbb{Z}$, I proceed with the description of the Simplex algorithm.

$$\frac{e_1 \geq e_2}{e_1 - e_2 \geq 0}\textbf{geqgeq} \qquad \frac{e_1 > e_2}{e_1 - e_2 - 1 \geq 0}\textbf{gtgeq} \qquad \frac{e_1 \leq e_2}{e_2 - e_1 \geq 0}\textbf{leqgeq} \qquad \frac{e_1 < e_2}{e_2 - e_1 - 1 \geq 0}\textbf{ltgeq}$$

$$\frac{e_1 = e_2}{e_1 \geq e_2}\textbf{eqgeq} \qquad \frac{e_1 \geq 0 \qquad e_2 \geq 0 \qquad n \geq 0}{e_1 + n \cdot e_2 \geq 0}\textbf{geqadd}$$

$$\frac{}{0 \geq 0}\textbf{geq0} \qquad \frac{e_1 \geq 0 \qquad e_2 \geq 0 \qquad m \cdot e_1 = n - e_2 \qquad m \geq 0 \qquad -1 - n \geq 0}{\texttt{false}}\textbf{falsei}$$

$$\frac{\begin{array}{ll} m_1 \cdot (e_1 - e_2) = -e_1' & m_2 \cdot (e_2 - e_1) = -e_2' \\ e_1' \geq 0 & e_2' \geq 0 \\ m_1 \geq 0 & m_2 \geq 0 \end{array}}{e_1 = e_2}\textbf{eqi}$$

Figure 7.12: The proof rules that the Simplex decision procedure uses to construct proofs.

Figure 7.13 shows the skeleton of the Simplex decision procedure. The entry point `decproc` processes a literal along with its proof and returns a set of equality assertions together with their proofs. Also, if it finds a contradiction it raises the `Contradiction` exception with a proof of `false`. The first step performed by the decision procedure is to convert the literal to the canonical form "$e \geq 0$". Note how the proof is also adjusted accordingly so that Invariant 7.7 always holds when `simplex` is invoked.

Invariant 7.7 (of function `simplex`) *Any invocation "`simplex`(e, prf)" is such that prf is a proof of $e \geq 0$. The result of such an invocation is a set of equalities along with their proofs, or the* `Contradiction` *exception is raised with a proof of* `false`*.*

The execution of the function `simplex` starts by adding a new row to the tableau owned by the expression e. The row entries are computed so that Invariant 7.5b holds. If the expression e is a variable that owns a column, then the value 1 is added to that column in the added row. If it is a variable that owns a row then the owned row is added to the added row. If e is the sum of two expressions, the added row will contain the sum of the entries in two rows corresponding to the two expressions. Before execution proceeds, `simplex` checks whether the added row is identical with some other existing row i or if it is zero except for an entry equal to one in one column j. In the former situation, the added row is deleted and the row i is considered to be owned by e. In the latter situation, the column j is pivoted into a row position i and we continue as in the case of equal rows.

The next step in the execution of `simplex` is to represent the assertion that e is greater or equal to zero. As I explained before, this must be done by making the row that owns e

(i in this case) to be +-restricted. If the row i is already restricted then nothing else must be done. If not, then Invariant 7.5d allows a restriction only if T_{i0} is at least zero. Thus, Simplex tries through a series of valid pivoting operations to increase the value of T_{r0} to more than zero. Choosing the pivots so that they are valid and achieve the required effect in fewer steps is the most delicate issue in the implementation of Simplex. One possible algorithm for choosing pivots is shown in [Nel81].

The only situation when the value of T_{i0} cannot be made positive through pivoting is when row i is maximized at a value less than zero. According to the earlier discussion of maximization this means that $R(i)$, that is e, is always less than zero, which contradicts the proof that e is greater or equal to zero. In this case the `Contradiction` exception is raised. A proof of `false` is generated using the function `mkContraProof` whose definition, shown in Figure 7.14, is discussed later in this section.

If, in fact, the sample value of row i can be increased to be at least zero, then the implementation of `simplex` proceeds by making it a +-restricted row and by setting the corresponding proof. Next, `simplex` detects which new rows and columns are maximized at zero and must be made $*$-restricted. The reordering step is necessary to preserve Invariant 7.5f, which in turn is necessary to prove easily that certain proof-generating procedures terminate.

The last step in the execution of `simplex` is concerned with discovering new equalities between owner expressions. The owners of rows i_1 and i_2 are equal if the entries in the two rows are equal except possibly for those in $*$-restricted columns. A similar situation involves the owners of a row i and a column j such that all entries in row i are zero except for a 1 in column j and possibly other non-zero entries in $*$-restricted columns. In both of these situations, the equality of the two expressions follows from Invariant 7.5b. For reasons explained in detail in [Nel81], if the algorithm ensures that all owners that are zero throughout the solution set are $*$-restricted, then these two situations and the only ones when owners can be equal. This guarantees that Simplex discovers all equalities that must be propagated. In both of these situations the proof of equality is obtained using the auxiliary function `mkEqProof`, described later.

This concludes the description of that part of the Simplex decision procedure that is concerned with discovering new equalities or contradictions. Now, we turn to the novel proof-generating aspect of the decision procedure. The defining property of proof generation in Simplex is that all the required information exists in the tableau, except for the mapping *Proof* from restricted owners to their proofs. The proof generating components of Simplex are shown in Figure 7.14. I describe these functions by means of their input-output invariants.

The function `mkContraProof` is used to create a proof of `false` from a proof that the owner of a maximized row is greater or equal to zero, when the row is maximized to a negative value.

$$
\begin{array}{lcl}
\texttt{decproc}(e_1 \geq e_2, prf) & = & \texttt{simplex}(e_1 - e_2, \texttt{geqgeq}(prf)) \\
\texttt{decproc}(e_1 > e_2, prf) & = & \texttt{simplex}(e_1 - e_2 - 1, \texttt{gtgeq}(prf)) \\
\texttt{decproc}(e_1 \leq e_2, prf) & = & \texttt{simplex}(e_2 - e_1, \texttt{leqgeq}(prf)) \\
\texttt{decproc}(e_1 < e_2, prf) & = & \texttt{simplex}(e_2 - e_1 - 1, \texttt{ltgeq}(prf)) \\
\texttt{decproc}(e_1 = e_2, prf) & = & \texttt{decproc}(e_1 \geq e_2, \texttt{eqgeq}(prf)) \\
 & & \texttt{decproc}(e_2 \geq e_1, \texttt{eqgeq}(\texttt{eqsym}(prf)))
\end{array}
$$

simplex$(e, prf) =$
  $r = r + 1$
  fill row r such that $m \cdot e \simeq mT_{r0} + \sum mT_{rj} \cdot C(j)$
  if row r is identical to row i **then**
    $r = r - 1$
  else if row r is zero except for $T_{rj} = 1$ **then**
    $r = r - 1$
    pivot column j with some row i
  else
    $i = r$
  if row i is already restricted **then**
    return $\{\}$
  try to increase the value of T_{i0} to more than 0 through valid pivoting
  if row i is maximized and $T_{i0} < 0$ **then**
    raise `Contradiction`$($`mkContraProof`$(i, prf))$
  else
    make i a +-restricted row with $\mathit{Proof}(R(i)) = prf$
    foreach l a +-restricted row maximized at 0 **do**
      make l a $*$-restricted row
      reorder columns so that those with $T_{lj} > 0$ come before those with $T_{lj} < 0$,
        without changing the relative order of $*$-restricted columns
      foreach k a +-restricted column such that $T_{lk} \neq 0$ **do**
        make k a $*$-restricted column (l is the restrictor)
    $accum = \emptyset$
    foreach rows i_1, i_2 differing only in $*$-restricted columns **do**
      $accum = accum \cup \{(R(i_1) = R(i_2), \texttt{mkEqProof}(R(i_1), R(i_2)))\}$
    foreach i, j such that row i is non-zero only in $*$-restricted columns and in column j,
        and either j is $*$-restricted or $T_{ij} = 1$ **do**
      $accum = accum \cup \{(R(i) = C(j), \texttt{mkEqProof}(R(i), C(j)))\}$
  return $accum$

Figure 7.13: The core of the Simplex decision procedure.

Invariant 7.8 (of function `mkContraProof`) *Any invocation "*`mkContraProof`(i, prf)*" is such that i is a row that is maximized at a negative value ($T_{i0} < 0$) and prf is a proof that $R(i) \geq 0$.*

The function `mkContraProof` is implemented in terms of the helper function `maxProof`, described below, and the proof rule `falsei`. Note that the numeral n from the `falsei` inference rule schema is instantiated with mT_{i0} and m with the positive integer constant returned by `maxProof`.

The function `maxProof` is called to produce a proof that the owner of a maximized row i is at most equal to T_{i0}. More precisely, the behavior of `maxProof` is described by Invariant 7.9 below.

Invariant 7.9 (of function `maxProof`) *Any invocation "*`maxProof`(i)*" is such that i is a row maximized at value T_{i0}. The return of such an invocation is a triple "(m, e, prf)", such that m is a positive integer numeral such that mT_{i0} is an integer and "$m \cdot R(i) \simeq mT_{i0} - e$". Also, prf is a proof that "$e \geq 0$".*

Before I discuss in more detail how `maxProof` is implemented, consider the implementation, also in terms of `maxProof`, of the function `mkEqProof` that produces a proof that two row owners are equal when their entries differ only in $*$-restricted columns. Invariant 7.10 describes the input state to this function.

Invariant 7.10 (of function `mkRowEqProof`) *Any invocation "*`mkEqProof`(e_1, e_2)*" is such that the expressions $e_1 - e_2$ and $e_2 - e_1$, when represented in the tableau, correspond to rows that are zero except possibly in $*$-restricted columns.*

The trick used in this case is to create a new temporary row whose owner to be "$e_1 - e_2$". Because of Invariant 7.10, this new row is maximized at zero and `maxProof` can be safely invoked. The same row is then rewritten to own "$e_2 - e_1$" and for similar reasons `maxProof` can be invoked again. Then, the results of the two invocations to `maxProof` are used to construct the hypotheses needed for the rule `eqi`.

To complete the discussion of proof generation in Simplex we need to discuss the implementation of `maxProof` satisfying Invariant 7.9. For this purpose I introduce the notion of a map Ψ from expressions to rational numbers. It is convenient to view such a map as a sequence of pairs, as described by the BNF form below:

$$\Psi ::= \cdot \quad | \quad \Psi; e \mapsto f$$

where "$\cdot$" is the empty map. For convenience I also define the operation "$\Psi(e) \mathrel{+}= f$" on maps to be equivalent to Ψ:=Ψ[e $\mapsto$ Ψ(e) + f], where $\Psi[e \mapsto f]$ is like Ψ but with the expression e mapping to f. If e was already mapped in Ψ, the old mapping is overridden.

mkContraProof$(i, geqe_1) =$
  $(m, e_2, geqe_2) = \texttt{maxProof}(i)$
  return $\texttt{falsei}(geqe_1, geqe_2, \texttt{arith}(m \cdot R(i), mT_{i0} - e_2),$
    $\texttt{geqct}(m), \texttt{geqct}(-1 - mT_{i0}))$

mkEqProof$(e_1, e_2) =$
  $r = r + 1$
  fill row r with coefficients for $e_1 - e_2$
  $(m_1, e_1', prf_1) = \texttt{maxProof}(r)$
  fill row r with coefficients for $e_2 - e_1$
  $(m_2, e_2', prf_2) = \texttt{maxProof}(r)$
  $r = r - 1$
  return $\texttt{eqi}(\texttt{arith}(m_1 \cdot (e_1 - e_2), -e_1'), prf_1, \texttt{geqct}(m_1),$
    $\texttt{arith}(m_2 \cdot (e_2 - e_1), -e_2'), prf_2, \texttt{geqct}(m_2))$

maxProof$(i) =$
  $\Psi = \cdot$
  foreach $k = 1 \,..\, c$ `such that` $T_{ik} < 0$ **do**
    $\Psi(C(k)) \mathrel{+}= T_{ik}$
  foreach $k = 1 \,..\, c$ `such that` $T_{ik} > 0$ **do**
    $\Psi = \texttt{computeCol}(k, T_{ik}, \Psi)$
  $m = \texttt{SCM}(\Psi, T_{i0})$
  return $(m, \texttt{mapProof}(\Psi, m)$

computeCol$(j, f, \Psi) =$
  let row i be the restictor of j as in Invariant 7.5f:
    – i is $*$-restricted, and
    – $T_{ij} < 0$, and
    – $T_{ik} \leq 0$ for all $k > j$, and
    – $T_{ik} \neq 0$ only if k is $*$-restricted
  $\Psi(R(i)) \mathrel{+}= f / T_{ij}$
  foreach $k \neq j$ such that $T_{ik} < 0$ **do**
    $\Psi(C(k)) \mathrel{+}= -f \cdot T_{ik} / T_{ij}$
  foreach k such that $T_{ik} > 0$ **do**
    $\Psi = \texttt{computeCol}(k, -f \cdot T_{ik}/T_{ij}, \Psi)$
  return Ψ

mapProof$(\cdot, m) =$ **return** $(0, \texttt{geq0})$
mapProof$(\Psi; e \mapsto f, m) =$
  $(e_1, d) = \texttt{mapProof}(\Psi, m)$
  return $(e_1 + (-mf) \cdot e, \texttt{geqadd}(d, \mathit{Proof}(e), \texttt{geqct}(-mf)))$

Figure 7.14: The proof-generating components of the Simplex decision procedure.

Definition 7.11 *A map Ψ is* valid *if all of the following conditions holds:*

1. *All $e \in Dom(\Psi)$ are restricted expressions for which there is a proof that $e \geq 0$. This proof is referred to as $Proof(e)$, and*
2. *For all $e \in Dom(\Psi)$ we have that $\Psi(e) \leq 0$.*

The intuition behind the implementation of `maxProof` is that a maximized row can always be written as a linear combination, with only negative coefficients, of restricted expressions. The function `maxProof` computes these coefficients as a valid map Ψ such that there exists an integer m that makes all of the products mT_{i0} and $m\Psi(e)$ for all $e \in Dom(\Psi)$ integers and such that

$$m \cdot R(i) \simeq mT_{i0} + \sum_{e \in Dom(\Psi)} (m\Psi(e)) \cdot e$$

Note that row i is maximized, thus its only non-zero entries are in restricted columns. Those columns whose entries are negative can be added directly to the map. Those whose entries are positive must be $*$-restricted entries and are added to the map by means of the helper function `computeCol`. Then, the function `SCM` (not shown here) is called to compute the smallest positive common multiple of all denominators of entries in Ψ and of T_{i0} and finally, the function `mapProof` is called to construct the actual proof.

Invariant 7.12 (of function `computeCol`) *Any invocation "`computeCol`(j, f, Ψ)" is for a $*$-restricted column j and a positive rational factor f. If we denote by Ψ' the resulting valid map then there exists an integer m such that*

$$mf \cdot C(j) + \sum_{e \in Dom(\Psi)} m\Psi(e) \cdot e \simeq \sum_{e' \in Dom(\Psi')} m\Psi'(e') \cdot e'$$

The idea behind the implementation of `computeCol` is that each $*$-restricted column can be written as a linear combination, with only negative factors, of restricted expressions. To find this linear combination we search first for the restrictor row and express the value of the column using Invariant 7.5b for this row. Note that the recursive invocation of `computecol` is guaranteed to terminate because $k < j$ as stated by Invariant 7.5f.

Finally, the function `mapProof` simply builds a proof that the negation of a linear combination, with only negative coefficients, of restricted expressions is greater or equal to zero. It returns both this proof and the negated expression.

This completes the description of the implementation of the proof-generating decision procedure for linear arithmetic using Simplex. In this case, unlike for congruence closure, the proof-generating component of the decision procedure is about as complicated as the contradiction-generating component. In the actual implementation, the proof-generating part is about one third of the whole decision procedure. For a better understanding of the congruence closure and the Simplex decision procedures, the interested reader is encourage to read the next section, where I revisit these decision procedures showing their operation in the context of a simple theorem proving task.

7.3.3 An Example with Congruence Closure and Simplex

To illustrate the cooperation of the congruence closure and Simplex decision procedures consider again the example introduced at the beginning of this chapter. The predicate that must be proved in this case is:

$$P \equiv (\mathtt{f}(\mathtt{f}(y) - \mathtt{f}(x)) \neq \mathtt{f}(z) \ \wedge \ y \geq x \ \wedge \ x \geq y + z) \ \supset z < 0$$

To simplify the presentation, I use the following abbreviations

$$\begin{array}{lll} t_1 & \text{for} & \mathtt{f}(y) \\ t_2 & \text{for} & \mathtt{f}(x) \\ t_3 & \text{for} & \mathtt{t}_1 - t_2 \\ s_4 & \text{for} & y - x \\ s_6 & \text{for} & x - y - z \end{array}$$

The prover is invoked as $\mathtt{fol}(\mathrm{P})$. (The definition of `fol` is given in Figure 7.3 on page 162.) Because P is an implication, a new variable u is created, to stand for the assumed proof of the left-hand side of the implication, as follows:

$$u : \mathtt{pf}\ ((\mathtt{f}(t_3) \neq \mathtt{f}(z) \wedge y \geq x) \wedge x \geq y + z)$$

Then the function `assert` is invoked on the left-hand side. (The definition of `assert` is also shown in Figure 7.3.) This leads to three invocations of the `dispatch` function, as follows:

$$\mathtt{dispatch}(\mathtt{f}(t_3) \neq \mathtt{f}(z),\ \mathtt{andel}(\mathtt{andel}(u))) \tag{7.13}$$
$$\mathtt{dispatch}(y \geq x,\ \mathtt{ander}(\mathtt{andel}(u))) \tag{7.14}$$
$$\mathtt{dispatch}(x \geq y + z,\ \mathtt{ander}(u)) \tag{7.15}$$

I assume in what follows that the "Dispatch" modules invokes first the congruence closure decision procedure and then the Simplex decision procedure. While responding to the invocation 7.13, the "Dispatch" modules creates nodes in the E-DAG corresponding to all subexpressions in the literal involved. Then the congruence closure forbids the merge of the class of "$\mathtt{f}(t_3)$" and the class of "$\mathtt{f}(z)$". The resulting state is shown in Figure 7.15(1), where the nodes and the solid lines belong to the E-DAG and the interrupted lines mark either equivalence relations or forbid relations belonging to the congruence closure. Also, the current value of the congruence closure `undoStack` is "$\{(\mathtt{f}(t_3) \neq \mathtt{f}(z), \mathtt{andel}(\mathtt{andel}(u)))\}$".

The same first literal is then asserted to Simplex, which introduces in the tableau the expression $t_3 = t_1 - t_2$. The resulting Simplex tableau is shown in Figure 7.15(2). The `simplex` function itself is not yet invoked because the free function $\neq$ does not belong to the theory $\mathbb{Z}$.

Because no contradiction was detected, the invocation 7.14 is initiated. First, the dispatcher adds to the E-DAG the node labeled s_4 in Figure 7.15(3). Then it passes the assertion to the congruence closure algorithm with no effect on the state of algorithm. Then, Simplex receives the assertion, recognizes the constructor $\geq$, and invokes

$$\texttt{simplex}(y - x, \texttt{geqgeq}(\texttt{ander}(\texttt{andel}(u)))) \tag{7.16}$$

Let s_4 be the name associated with the expression "$y - x$". After Simplex adds this expression to the tableau, the tableau is like that shown in Figure 7.15(4). Simplex tries to increase the sample value of s_4 but it cannot because it would have to pivot the row s_4 itself. Thus, Simplex makes s_4 a $+$-restricted state as shown in Figure 7.15(4) and it sets

$$\mathit{Proof}(s_4) = \texttt{geqgeq}(\texttt{ander}(\texttt{andel}(u)))$$

Finally, Simplex tries without success to $*$-restrict rows and columns and to propagate equalities.

Now the invocation 7.15 is started and the dispatcher adds the necessary nodes to the E-DAG, as shown in Figure 7.15(5). The congruence closure does not find any new congruences and leaves this state unchanged. When Simplex encounters this new literal it first adds the variable z to an empty column (so that all variables own either a row or a column) and then processes the expression $x - y - z$, which we are going to call s_6 using the invocation:

$$\texttt{simplex}(x - y - z, \texttt{geqgeq}(\texttt{ander}(u))) \tag{7.17}$$

After adding the row corresponding to s_6 the tableau is as shown in Figure 7.15(6). Simplex tries now to increase the sample value of s_6 past zero and for that purpose it pivots on (s_4^+, x), with the resulting tableau shown in Figure 7.16(7). The sample value of s_6 is still zero but it cannot be pivoted anymore without cycling through already encountered states, so Simplex makes s_6 a $+$-restricted row with

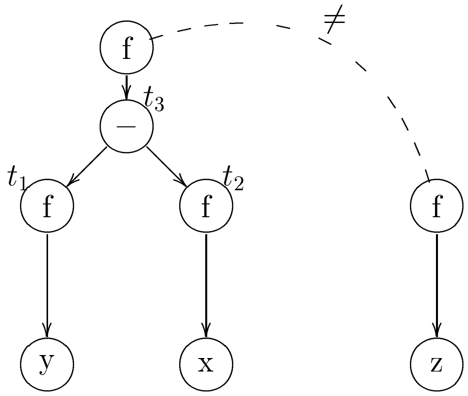
$$\mathit{Proof}(s_6) = \texttt{geqgeq}(\texttt{ander}(u))$$

All three literals from the left-hand side of the implications have been asserted and no contradiction was found. The "Logical connectives" modules continues the evaluation of the function `fol` for the top-level implication by trying to prove the right-hand side. Thus "$\texttt{fol}(z < 0)$" is invoked, which results in a new proof variable v being created for the negation of the goal literal and the dispatcher being invoked as follows:

$$\texttt{dispatch}(z \geq 0, v)$$

where:

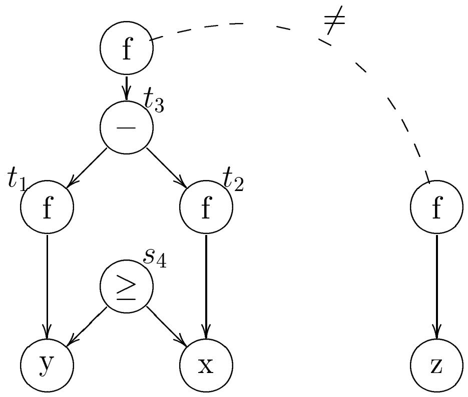
$$v : \texttt{pf}\ (z \geq 0)$$



(1)

	t_1	t_2
t_3	1	-1

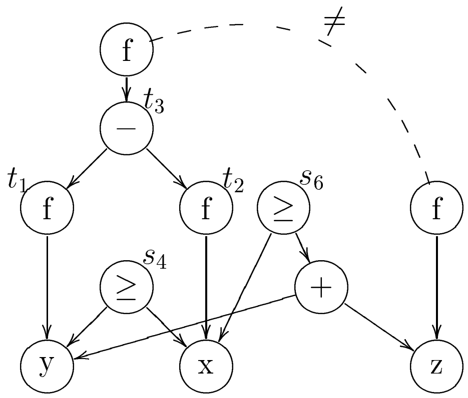
(2)



(3)

	t_1	t_2	x	y
t_3	1	-1		
s_4^+			-1	1

(4)



(5)

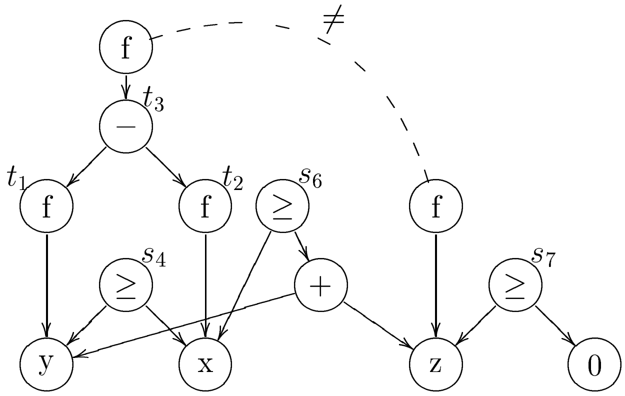
	t_1	t_2	x	y	z
t_3	1	-1			
s_4^+			-1	1	
s_6			1	-1	-1

(6)

Figure 7.15: State history of the congruence closure and Simplex decision procedures (I).

	t_1	t_2	s_4^+	y	z
t_3	1	-1			
x			-1	1	
s_6^+			-1		-1

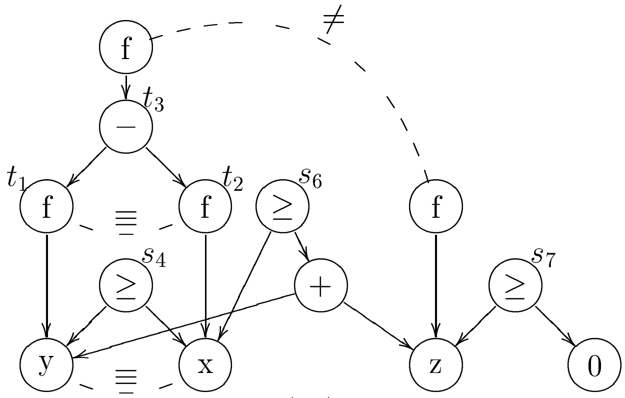
(7)



(8)

	t_1	t_2	s_4^*	y	s_6^*
t_3	1	-1			
x			-1	1	
z^*			-1		-1

(9)



(10)

	t_1	t_2	s_4^*	y	s_6^*
t_3^+	1	-1			
x			-1	1	
z^*			-1		-1

(11)

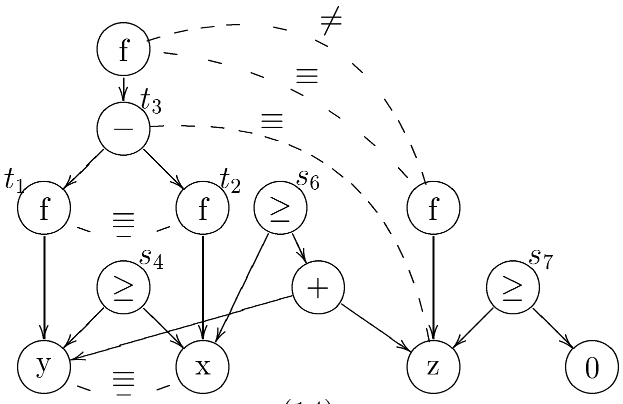
	t_1	t_2	s_4^*	y	s_6^*
t_3^+	1	-1			
x			-1	1	
z^*			-1		-1
s_8	-1	1			

(12)

Figure 7.16: State history of the congruence closure and Simplex decision procedures (II).

	t_1	t_3^*	s_4^*	y	s_6^*
t_2	1	-1			
x			-1	1	
z^*			-1		-1
s_8^*		-1			

(13)



(14)

Figure 7.17: State history of the congruence closure and Simplex decision procedures (III).

The dispatcher expands the E-DAG, as shown in Figure 7.16(8), as passes the assertion along to congruence closure. The congruence closure is once more inactive. The Simplex decision procedure is then invoked as

$$\texttt{simplex}(z, v) \tag{7.18}$$

Simplex tries to add the expression "z" to the tableau and discovers immediately that this expression owns a column. Thus it brings z into a row by pivoting on (s_6+, z). The resulting tableau is shown in Figure 7.16(9) (the $*$-restrictions on s_4, s_6 and z, which are explained next). Note that in this state the sample value of z cannot be increased anymore and it is thus $+$-restricted with

$$\mathit{Proof}(z) = v$$

Then, Simplex discovers that z is both $+$-restricted and maximized at zero and thus it makes it $*$-restricted together with the columns s_4 and s_6. The state of the Simplex tableau is now as shown in Figure 7.16(9). These restrictions enable Simplex to discover the equalities "$x = y$" (because row x is zero everywhere except in $*$-restricted columns and in the column of y, where it has a one), and "$z = s_4 = s_6$". Only the first equality is interesting because the others involve expressions ("$y - x$" and "$x - y - z$") that do not exist in the original predicate but were introduced by Simplex for its own purposes.

To produce a proof of "$x = y$", Simplex invokes "$\texttt{mkEqProof}(x, y)$". (The definition of `mkEqProof` is shown in Figure 7.14 on page 184.) This results in a new temporary row being created to represent "$x - y$". Given the current columns, this row will be zero except for a -1 in the column corresponding to s_4. The function `maxProof` builds the map "$\Psi = \cdot; s_4 \mapsto -1$", without needing to call `computeCol` because there are no positive coefficients in the new row.

From this map, mapProof returns

$$\begin{aligned} m_1 &= 1 \\ e'_1 &= 0 + 1 \cdot s_4 = 0 + 1 \cdot (y - x) \\ prf_1 &= \texttt{geqadd}(\texttt{geq0}, \texttt{geqgeq}(\texttt{ander}(\texttt{andel}(u))), \texttt{geqct}(1)) = \mathcal{D}_{x-y} \end{aligned}$$

Then, the same row is rewritten to encode "$y-x$", meaning that there will be a coefficient 1 in the column corresponding to s_4 and zero everywhere else. Now maxProof invokes computeCol to compute the map Ψ for the column s_4. The restrictor of s_4 is z, and the result of "computeCol$(s_4, 1, \cdot)$" is "$\Psi = \cdot; z \mapsto -1; s_6 \mapsto -1$". This makes mapProof to return the following values:

$$\begin{aligned} m_2 &= 1 \\ e'_2 &= (0 + 1 \cdot z) + 1 \cdot s_6 = (0 + 1 \cdot z) + 1 \cdot (x - y - z) \\ prf_2 &= \texttt{geqadd}(\texttt{geqadd}(\texttt{geq0}, v, \texttt{geqct}(1)), \texttt{geqgeq}(\texttt{ander}(u)), \texttt{geqct}(1)) = \mathcal{D}_{y-x} \end{aligned}$$

Thus the final proof of $x = y$ constructed by mkEqProof is:

$$\mathcal{D}_{x=y} = \texttt{eqi}(\texttt{arith}(1 \cdot (x - y), -e'_1), \mathcal{D}_{x-y}, \texttt{geqct}(1), \texttt{arith}(1 \cdot (y - x), -e'_2), \mathcal{D}_{y-x}, \texttt{geqct}(1))$$

Now, the equality "$x = y$" discovered by Simplex is propagated to the congruence closure decision procedure and the function "merge$(x, y, \mathcal{D}_{x=y})$" is called. The undoStack becomes "$\{(\texttt{f}(t_3) \neq \texttt{f}(z), \texttt{andel}(\texttt{andel}(u))), (x = y, \mathcal{D}_{x=y})\}$". The singleton classes of x and of y are merged and the procedure checkCongr discovers a congruence between "f(x)" and "f(y)". To produce a proof of the congruence, we need a proof of $x = y$ for which purpose the function prfEq (defined in Figure 7.9 on page 173) is called. This is a simple case when the equality exists in an identical form in the undoStack. The function checkCongr terminates by calling recursively "merge$(\texttt{f}(x), \texttt{f}(y), \texttt{congr}(\texttt{f}, \mathcal{D}_{x=y}))$". This latter call does not discover any new equalities and the final state of the congruence closure decision procedure is shown in Figure 7.16(10). The result of the congruence closure decision procedure is the new equality "$t_1 = t_2$" and the value of the undoStack at this point is:

$$\{(\texttt{f}(t_3) \neq \texttt{f}(z), \texttt{andel}(\texttt{andel}(u))), (x = y, \mathcal{D}_{x=y}), (t_1 = t_2, \texttt{congr}(\texttt{f}, \mathcal{D}_{x=y}))\}$$

Next, the equality "$t_1 = t_2$" is propagated to Simplex, which leads to two calls to simplex as follows

$$\begin{aligned} &\texttt{simplex}(t_1 - t_2,\ \texttt{geqgeq}(\texttt{eqgeq}(\texttt{congr}(\texttt{f}, \mathcal{D}_{x=y})))) \\ &\texttt{simplex}(t_2 - t_1,\ \texttt{geqgeq}(\texttt{eqgeq}(\texttt{eqsym}(\texttt{congr}(\texttt{f}, \mathcal{D}_{x=y}))))) \end{aligned}$$

Let $\mathcal{D}_{t_1-t_2}$ and $\mathcal{D}_{t_2-t_1}$ be the two proofs with which `simplex` is invoked. As Simplex tries to add the expression "$t_1 - t_2$" to the tableau it discovers that it is already there, owning the row of t_3. Thus, it tries to +-restrict the row t_3. This row cannot be pivoted further and Simplex goes ahead and makes t_3 a +-restricted row, as shown in Figure 7.16(11). The $Proof$ map is modified such that:

$$Proof(t_3) = \mathcal{D}_{t1_-t_2} = \mathtt{geqgeq}(\mathtt{eqgeq}(\mathtt{congr}(\mathtt{f}, \mathcal{D}_{x=y})$$

Then, Simplex asserts "$t_2 - t_1 \geq 0$" and, for that purpose, adds to the tableau a new row representing "$s_8 = t_2 - t_1$". The state of the tableau after adding the row s_8 is shown in Figure 7.16(12). Simplex tries to increase the sample value of s_8 and pivots on (t_3^+, t_2). The resulting tableau is shown in Figure 7.17(13). Because s_8 cannot be pivoted further, Simplex makes it +-restricted with:

$$Proof(s_8) = \mathcal{D}_{t_2-t_1} = \mathtt{geqgeq}(\mathtt{eqgeq}(\mathtt{eqsym}(\mathtt{congr}(\mathtt{f}, \mathcal{D}_{x=y}))$$

The Simplex discovers that s_8 is maximized at zero and makes both s_8 and t_3 $*$-restricted. At this point Simplex discovers the new equalities "$z = t_3 = s_4 = s_6 = s_8$" (all $*$-restricted). Of these equalities only "$z = t_3$" is of interest to other decision procedure and it is thus propagated.

In order to produce the proof of "$z = t_3$", the function `mkEqProof` is called, which in turn calls `maxProof` to calculate "$z - t_3$" and "$t_3 - z$" as a linear combination with negative coefficients of restricted expressions. The results are:

$$\begin{aligned} z - t_3 &= -t_3 - s_4 - s_6 \\ t_3 - z &= t_3 + s_4 + s_6 = -s_8 - z \end{aligned}$$

From here the following proof of "$z = t_3$" is produced:

$$\begin{aligned} \mathcal{D}_{z=t_3} = \mathtt{eqi}(&\mathtt{arith}(1 \cdot (z - t_3), -(((0 + 1 \cdot t_3) + 1 \cdot s_4) + 1 \cdot s_6)), \\ &\mathtt{geqadd}(\mathtt{geqadd}(\mathtt{geqadd}(\mathtt{geq0}, Proof(t_3), \mathtt{geqct}(1)), \\ &\qquad\qquad Proof(s_4), \mathtt{geqct}(1)), \\ &\qquad Proof(s_6), \mathtt{geqct}(1)), \\ &\mathtt{geqct}(1), \\ &\mathtt{arith}(1 \cdot (t_3 - z), -((0 + 1 \cdot s_8) + 1 \cdot z)), \\ &\mathtt{geqadd}(\mathtt{geqadd}(\mathtt{geq0}, Proof(s_8), \mathtt{geqct}(1)), \\ &\qquad Proof(z), \mathtt{geqct}(1)), \\ &\mathtt{geqct}(1)) \end{aligned}$$

When the equality "$z = t_3$" is propagated to the congruence closure it results in the merging of the singleton classes of z and t_3, which in turn leads to the discovery of the congruence "$\mathtt{f}(z) = \mathtt{f}(t_3)$", with the resulting state shown in Figure 7.17(14). The congruence closure `undoStack` at this point is:

$$\begin{array}{rcl} \texttt{undoStack} & = & \{(\texttt{f}(t_3) \neq \texttt{f}(z), \texttt{andel}(\texttt{andel}(u))), \\ & & (x = y, \mathcal{D}_{x=y}), \\ & & (t_1 = t_2, \texttt{congr}(\texttt{f}, \mathcal{D}_{x=y})), \\ & & (z = t_3, \mathcal{D}_{z=t_3}), \\ & & (\texttt{f}(z) = \texttt{f}(t_3), \texttt{congr}(\texttt{f}, \mathcal{D}_{z=t_3}))\} \end{array}$$

When trying to merge the equivalence classes of $\texttt{f}(z)$ and $\texttt{f}(t_3)$, the congruence closure algorithm detects a contradiction and invokes the function `mkEqContra` to build a proof of `false`.

It searches the `undoStack` to find the disequality that is responsible for the contradiction, and it finds the assertion "$\texttt{f}(t_3) \neq \texttt{f}(z)$" with the proof "$\texttt{andel}(\texttt{andel}(u))$". Thus is builds the proof

$$\mathcal{D}_{\texttt{false}} = \texttt{falsei}(\texttt{eqtr}(\texttt{eqid}, \texttt{eqtr}(\texttt{eqsym}(\texttt{congr}(\texttt{f}, \mathcal{D}_{z=t_3})), \texttt{eqid})), \texttt{andel}(\texttt{andel}(u)))$$

The function `merge` raises the exception `Contradiction` with the proof $\mathcal{D}_{\texttt{false}}$. This exception is caught by the `fol` function that was trying to prove the right-hand side of the implication. Thus, the complete proof of the predicate P is

$$\mathcal{D}_P = \texttt{impi}(u, \texttt{contra}(v, \mathcal{D}_{\texttt{false}}))$$

This concludes the step-by-step example of the operation of the congruence closure and the Simplex decision procedures. Even for the simple example discussed here, the global operation of the theorem prover is quite complicated. This only demonstrates how difficult it would be to write a single decision procedure to handle the combination of the theories of interest. Instead, we rely on the clean separation between the decision procedures provided by the Nelson-Oppen architecture, to develop and reason about one decision procedure at a time. In the next section, I describe one more decision procedure, this time for handling the typing predicate that arise in the verification conditions of programs generated with the Touchstone compiler described in Chapter 6.

7.3.4 Handling the Touchstone Typing Rules

If we want to use the theorem prover to be able to prove type safety for the output of the Touchstone certifying compiler then we must have a module that handles the inference rules shown in Figure 6.5 on page 126. This typing module must prove predicates of the forms "$E : T$", "$\texttt{saferd}(M, A)$", and "$\texttt{safewr}(M, A, V)$". In this section I will sketch a proof-generating decision procedure for these typing predicates. Unlike the convex decision procedures considered before, the typing decision procedure makes extensive use of the case-split capability of the theorem prover.

The only internal data structure used by the typing decision procedure is a stack of typing assertions. This stack is pushed whenever a typing assertion is dispatched and is popped during the execution of the `undo` operation. In addition, the typing decision procedure has access to the EDAG data structure.

The typing decision procedure simply collects on its stack the typing assertions that are dispatched to it. This is until it encounters the negation of one of the three forms of predicates that it can prove. In that case, it tries to prove the direct form of the asserted literal based on the accumulated typing assertions and if it succeeds it generates a contradiction using the asserted negation. All of the proofs discussed in this section refer to the axiomatization shown in Figure 6.5.

When trying to prove memory-safety predicates of the form "$\texttt{saferd}(M, A)$" or of the form "$\texttt{safewr}(M, A, V)$", the decision procedure will use the rules `read` and `write` and will attempt to find a type T and a proof that "$A : \texttt{ptr}(T)$". This subgoal is proved using a helper function `findPtrTypes` which is discussed later in this section. The other form of assertion that can be proved is "$E : T$". If the type T is any base type except a `ptr` type, it is handled in a straightforward manner using the syntax-directed rules shown in the "Other Types" section of Figure 6.5. To simplify the typing decision procedure, whenever a recursive type "$\texttt{mu}\ \lambda t.T$" is encountered it is replaced with "$[{}^{\texttt{mu}\ \lambda t.T}/_{t}]T$", according to the rule `mu` of Figure 6.5.

In order to handle the rules related to the `sizeOf` predicate constructor, the typing decision procedure contains a procedure "$\texttt{sizeOf}(T, S) \rightarrow D$" that given a structured type T, computes the integer constant S and the proof D of the predicate $\texttt{sizeOf}(T, S)$. Two other helper functions are "$E - E' \rightarrow E''$" and "$E \div N \rightarrow (E', N')$" that succeed whenever, using only trivial arithmetic rules, the expression E can be rewritten as "$E' + E''$" or, respectively, as "$E' * N + N'$" where N' is a constant such that "$0 \leq N' < N$". If this is not possible then the functions abort and their result is interpreted as falsehood when they are used in a conditional context. These two helper functions are implemented by scanning the EDAG node corresponding to E.

In order to handle the rules shown in the "Memory Updates" section of Figure 6.5, the typing decision procedure uses the function `proveMem` shown at the top of Figure 7.18. This function tries to use the rules `upd` or `updoa` to prove that a modified memory state is still well-typed. For this purpose, it first finds all types T such that the last update address A can be proved to have type "$\texttt{ptr}(T)$". This is done with the helper function `findPtrTypes`, whose return value is a list of elements of the form "$<A : \texttt{ptr}(T); G, D>$", where G is a list of subgoals and D is a proof of "$A : \texttt{ptr}(T)$" possibly referring to the proofs of the not-yet-proved subgoals. The list of subgoals and the proof D are as required by the `Split.add` function described on page 167. Each element in the returned list is scanned and if it is a base type (as verified by "$\texttt{sizeOf}(T, W)$") then the rule `upd` is tried. Otherwise, if T is an open-array type then the rule `updoa` is tried. There rules are tried by registering with the "Split" module of the theorem prover a list of goals named g_1 and g_2 along with a proof of $\texttt{upd}(M, A, E) : \texttt{Mem}$.

The only remaining detail is the handling of the rules shown in the section labeled "PointerTypes" in Figure 6.5. Most of these rules are handled using the helper function `findPtrTypes` whose pseudo-code description is shown in Figure 7.18. When trying to find types T such that "$A : \texttt{ptr}(T)$", `findPtrTypes` first tries all expressions E such that A can be rewritten as "$E + E'$" for some E'. Then, using the helper function `findSelAssTypes`, it collects all types T such that "$E : T$" is among the assumptions or, in the case when E is of the form "$\texttt{sel}(M, A)$", by using the rules `sel` or `seloa`. The pseudo-code for the helper function `findSelAssTypes` is also shown in Figure 7.18.

Returning to `findPtrTypes`, if the type of E is an array type and if E' can be rewritten as "$E'' * S + I$" where "$\texttt{sizeOf}(T_1, S)$" then the `array` rule is tried. First, the `getField` helper function is called with the index I and a proof of "$E + E'' * S : \texttt{ptr}(T_1)$". This proof is constructed with the `array` rule and the proofs of the subgoals $g_1 \triangleright\ E'' * S \geq 0$ and $g_2 \triangleright E'' * S \leq L$. The result from a successful execution of `getField` is the type T_1' that is at offset I in the structured type T_1 and a proof that $(E + E'' * S) + I : \texttt{ptr}(T_1')$. The pseudo-code for the function `getField` is shown at the bottom of Figure 7.18. Upon return from `getField`, the function `findPtrTypes` adds to the result list the assertion "$A : \texttt{ptr}(T_1')$", with the original list of goals G and the proof obtained from D' by congruence of the typing predicate.

The other cases analyzed by `findPtrTypes` are when the type of E is either a pointer option or a pointer type itself. In these cases the `getField` is called again to find the base type or the open array type that A points to.

There are several important observations related to the typing decision procedure. The first is that the decision procedure is not complete. There are instances when it cannot prove predicates for which there is a derivation in the logic. Take for example the following predicate:

$$a : \texttt{array}(\texttt{int}, l) \wedge e \geq 0 \wedge e \leq l \wedge e \bmod \texttt{W} = 0 \quad \supset \quad \texttt{sel}(\texttt{upd}(m, a, a + e), a) : \texttt{ptr}(\texttt{int})$$

This predicate arises when the compiler generates code that does the bounds checking for index e and then stores the address of the indexed array element in the first location of the array. Then, it loads back the pointer and tries to use it as a pointer to integers. This predicate is provable using the McCarthy axiom of our logic saying that "$\texttt{sel}(\texttt{upd}(m, a, a + e), a) = a + e$". This predicate cannot be proved by the decision procedure outlined in this section because it tries to rewrite the term "$\texttt{sel}(\texttt{upd}(m, a, a + e), a)$" to "$a + e'$" using only trivial axioms of arithmetic. One solution to this completeness problem would be to ensure that the helper function "$A - E \to E'$" succeeds whenever $A = E + E'$ is provable, for any E and E'. Although this is possible it is significantly more expensive.

$$
\begin{array}{l}
\textbf{proveMem}(\mathtt{upd}(M, A, E)) = \\
\qquad \underline{\textbf{foreach}}\ \langle A : \mathtt{ptr}(T); G, D_A\rangle \in \mathtt{findPtrTypes}(A)\ \ \underline{\textbf{do}} \\
\qquad\quad \underline{\textbf{if}}\ \mathtt{sizeOf}(T, W) \to D_S\ \underline{\textbf{then}} \\
\qquad\qquad \mathtt{Split.add}(G, g_1 \triangleright\ M : \mathtt{Mem}, g_2 \triangleright\ E : T;\ \mathtt{upd}(g_1, D_A, D_S, g_2)) \\
\qquad\quad \underline{\textbf{if}}\ T \equiv \mathtt{openarray}(T') \wedge M \equiv \mathtt{upd}(M', A + W, V)\ \underline{\textbf{then}} \\
\qquad\qquad \mathtt{Split.add}(G, g_1 \triangleright\ M' : \mathtt{Mem}, g_2 \triangleright\ V : \mathtt{array}(T', E);\ \mathtt{updoa}(g_1, D_A, g_2)) \\
\\
\textbf{findPtrTypes}(A) = \\
\qquad \underline{\textbf{foreach}}\ E \text{ such that } A - E \to E'\ \ \underline{\textbf{do}} \\
\qquad\quad \underline{\textbf{foreach}}\ \langle E : T; G, D_E\rangle \in \mathtt{findSelAssTypes}(E)\ \ \underline{\textbf{do}} \\
\qquad\qquad \underline{\textbf{if}}\ T \equiv \mathtt{array}(T_1, L) \wedge \mathtt{sizeOf}(T_1, S) \to D_S \wedge E' \div S \to (E'', I) \wedge \\
\qquad\qquad\qquad\qquad\qquad \mathtt{getField}(T_1, I, \mathtt{array}(D_E, D_S, g_1, g_2, \mathtt{mod0})) \to (T_1', D')\ \underline{\textbf{then}} \\
\qquad\qquad\quad \text{add } \langle A : \mathtt{ptr}(T_1'); G, g_1 \triangleright\ E'' * S \geq 0, g_2 \triangleright\ E'' * S \leq L, \\
\qquad\qquad\qquad\quad g_3 \triangleright\ A = (E + E'' * S) + I;\ \mathtt{ofcongr}(D', g_3)\rangle \\
\qquad\qquad \underline{\textbf{if}}\ T \equiv \mathtt{ptropt}(T_1) \wedge E' \text{ is a constant } \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{getField}(T_1, E', \mathtt{ptropt}(D_E, g_1)) \to (T', D')\ \underline{\textbf{then}} \\
\qquad\qquad\quad \text{add } \langle A : \mathtt{ptr}(T_1'); G, g_1 \triangleright\ E \neq 0, g_2 \triangleright\ A = E + E';\ \mathtt{ofcongr}(D', g_2)\rangle \\
\qquad\qquad \underline{\textbf{if}}\ T \equiv \mathtt{ptr}(T_1) \wedge E' \text{ is a constant } \wedge \mathtt{getField}(T_1, E', D_E) \to (T', D')\ \underline{\textbf{then}} \\
\qquad\qquad\quad \text{add } \langle A : \mathtt{ptr}(T_1'); G, g_1 \triangleright\ A = E + E';\ \mathtt{ofcongr}(D', g_1)\rangle \\
\\
\textbf{findSelAssTypes}(E) = \\
\qquad \underline{\textbf{foreach}}\ \mathtt{assumption}\ u \triangleright\ E : T\ \ \underline{\textbf{do}} \\
\qquad\quad \text{add } \langle E : T; \cdot; u\rangle \\
\qquad \underline{\textbf{if}}\ E \equiv \mathtt{sel}(M, A)\ \underline{\textbf{then}} \\
\qquad\quad \underline{\textbf{foreach}}\ \langle A : \mathtt{ptr}(T); G, D_A\rangle \in \mathtt{findPtrTypes}(A)\ \ \underline{\textbf{do}} \\
\qquad\qquad \underline{\textbf{if}}\ \mathtt{sizeOf}(T, W) \to D_S\ \underline{\textbf{then}} \\
\qquad\qquad\quad \text{add } \langle E : T; G, g_1 \triangleright\ M : \mathtt{Mem};\ \mathtt{sel}(g_1, D_A, D_S)\rangle \\
\qquad\quad \underline{\textbf{foreach}}\ \langle A' : \mathtt{ptr}(T); G, D_A\rangle \in \mathtt{findPtrTypes}(A - W)\ \ \underline{\textbf{do}} \\
\qquad\qquad \underline{\textbf{if}}\ T \equiv \mathtt{openarray}(T')\ \underline{\textbf{then}} \\
\qquad\qquad\quad \text{add } \langle E : \mathtt{array}(T', \mathtt{sel}(M, A - W)); G, g_1 \triangleright\ M : \mathtt{Mem};\ \mathtt{seloa}(g_1, D_A)\rangle \\
\\
\textbf{getField}(T, I, D) = \\
\qquad \underline{\textbf{if}}\ I = 0 \wedge \mathtt{sizeOf}(T, W) \to D_S\ \underline{\textbf{then}}\ \ \underline{\textbf{return}}\ \ (T, D) \\
\qquad \underline{\textbf{if}}\ T \equiv \mathtt{pair}(T_1, T_2) \wedge \mathtt{sizeOf}(T_1, S_1) \to D_S\ \underline{\textbf{then}} \\
\qquad\quad \underline{\textbf{if}}\ I < S_1\ \underline{\textbf{then}}\ \ \underline{\textbf{return}}\ \ \mathtt{getField}(T_1, I, \mathtt{pairl}(D)) \\
\qquad\quad \underline{\textbf{else}}\ \underline{\textbf{return}}\ \mathtt{getField}(T_2, I - S_1, \mathtt{pairr}(D, D_S)) \\
\qquad \underline{\textbf{return}}\ \mathtt{false}
\end{array}
$$

Figure 7.18: Functions for proving Touchstone typing predicates.

Fortunately, the above predicate cannot arise from an agent produced by the Touchstone compiler because it would be against the typing rules to use the contents of an element of an array of integers as a pointer. In addition, the Touchstone optimizer would not eliminate the bounds checks for an index value loaded from memory. If such an optimization is ever added to the compiler, then we must improve the theorem prover to recognize those cases and prove the verification conditions that can arise.

7.4 Discussion

In this chapter I have shown how known decision procedures can be modified to produce proofs in a natural deduction style that can be verified using a simple proof checker. I have also shown that the proof-generating decision procedures can be integrated in a modified Nelson-Oppen architecture for a modular theorem prover. This allows easy extension of the theorem prover with decision procedures for various logics.

To simplify the presentation in this chapter I have given low priority to various performance aspects of the proof generation process. In the rest of this section I discuss briefly some optimizations that I have determined to be effective at reducing either the running time of the theorem prover or the size of the generated proofs.

One possibility for improving the running time of the theorem prover is to modify the "Split" module discussed in Section 7.2.3 so that a case split is never tried more than once. Note that the definition of `Split.snapshot` shown in Figure 7.5 copies the current set of splits. This is not necessary if the `next` function was already invoked on the current set because the backtracking mechanism built in the `tryAllSplits` function shown in Figure 7.3 ensures that the entire set will be tried if needed. This means that we can use a marker for each set of splits to record if `next` was invoked and copy only those sets that were not invoked.

Another mechanism that I have implemented relies on the fact that many subgoals that are proved are not used in the final proof. Consider the case when, during a case split, the last subgoal fails. Then the proofs generated for the preceding subgoals are not needed. This suggests that it is worth generating proofs lazily, only when they are needed. In my implementation, instead of generating a proof I generate a closure that when invoked will produce the proof. The closure structure records minimal amount of information, such as what module is responsible for generating the proof, and is much easier to construct than the proof itself. This is especially true for decision procedures where proof generation is a non-trivial task, such as Simplex. Note, that the internal data structures of various decision procedures might change from the moment the subgoal is proved and the closure is created to the moment when the closure is invoked and the proof is generated. However, provided that no `undo` operation was invoked meanwhile all of the information needed for proof generation is still present.

Next I discuss a couple of optimizations meant to reduce the size of the proofs generated by the theorem prover. The most obvious opportunity for proof optimization is the elimination of trivial uses of the contradiction rule. This is beneficial because the theorem prover proves a literal by assuming its negation and generating a contradiction. I have implemented a proof optimization that scans the proof and eliminates all uses of contradictions. The effect is that the size of proofs is reduced by 20%. Another opportunity for easy optimization is the elimination of trivial uses of the `ofcongr` rule in the typing decision procedure. These uses arise because the typing decision procedure indiscriminately wraps each typing derivation in a congruence rule. This optimization can be implemented either as a post pass or by changing the typing decision procedure.

Consider for example the two proof transformations shown below:

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ L\end{array} \qquad \dfrac{}{\neg L}u}{\texttt{false}}\,\textbf{falsei}}{L}\,\textbf{contra}^u \quad \equiv \quad \mathcal{D} \qquad\qquad \dfrac{\begin{array}{c}\mathcal{D}\\ E : T\end{array} \qquad \dfrac{}{E = E}\,\textbf{eqid}}{E : T}\,\textbf{ofcongr} \quad \equiv \quad \mathcal{D}$$

On the left-hand size we have an instance of a trivial use of contradiction. In order to prover the literal L, the theorem prover asserts its negation u. In this case the contradiction is generated by obtaining a proof $\mathcal{D}$ of L and thus the whole contradiction proof can be replaced with $\mathcal{D}$. The typing decision procedure always generates proofs of this form because it tries to contradict the negation of a goal by proving the goal itself. Even when the assumption u is used deeper within the proof of `false` it is a simple exercise to write the proof transformation that eliminates the contradiction with a direct proof. On the right-hand side we see the elimination of a trivial use of the `ofcongr` rule.

This concludes Part II of this dissertation. I have discussed two useful tools for use by proof-carrying code producers. One is the Touchstone compiler that can be used to translate agents written in a type-safe subset of the C programming language to optimized machine code annotated with loop invariants and function specifications. This otherwise undecidable task is possible in this case because the specifications are restricted to typing predicates and because the source language is itself typed. The second tool that I described is a theorem prover that is both powerful enough to prove many verification conditions and is also capable of generating proofs of them. The theorem prover, when used with the Touchstone compiler, acts as an effective referee signaling all compilation errors manifested as unsafe assembly language output. This use of the theorem prover has proved invaluable when debugging and maintaining the compiler.

In the third and final part of this dissertation I discuss various experiments that I have designed and performed to gain experience with proof-carrying code.

Part III

Evaluation of Proof-Carrying Code

In this last part of the dissertation, I present the experiments that I have done with proof-carrying code for the purpose of gaining experience with using the technique and of understanding its significant practical. I discuss three sets of experiments. One is designed to compare proof-carrying code with other techniques that can be used for ensuring the safety of untrusted code. Another set of experiments is designed to reveal the practical significance of various costs of using proof-carrying code, starting from the cost of certifying compilation and theorem proving and ending with the cost of storing and checking proofs. The purpose of the final set of experiments is to validate the effectiveness of the proof optimizations discussed in Chapter 5.

Chapter 8

Experimental Validation of Proof-Carrying Code

This chapter presents experimental evidence attesting that proof-carrying code is a practical approach to safe execution of untrusted code. In order to gain more experience with PCC and to compare it with other approaches to code safety, I have performed a series of experiments with safe network packet filters. These experiments, discussed in detail in Section 8.1, demonstrate that the run-time performance of PCC agents can be up to an order of magnitude better than that obtained with other techniques.

Then, in Section 8.2, I discuss a set of experiments using the Touchstone certifying compiler to produce proof-carrying code for a type-based safety policy. As part of these experiments, I measure the various costs of using proof-carrying code, starting from the cost of certifying compilation and theorem proving and ending with the cost of storing and checking the proofs.

The final set of experiments, discussed in Section 8.3, is designed to gauge the practical benefits that could be gained from using the implicit representation of proofs and the various proof optimizations discussed in Chapter 5. These experiments show that the savings due to LF_i representation of proofs over the plain LF representation can be as high as two orders of magnitude, sometimes making the difference between impractical proofs of several megabytes and manageable proofs of tens of kilobytes. Also in Section 8.3 I share my observations of the correlation between the size of the proofs and the time required to validate them using LF_i type checking.

8.1 Proof-Carrying Code for Packet Filters

A network packet filter is an application-provided subroutine that scans each incoming network packet and decides whether the user application is interested in receiving it or not. The packet filter is executed in the kernel-address space for each incoming packet. The overall effect is that only a small number of accepted packets have to be copied to the user address space. Packet filters are supported by most of today's workstation operating systems. Since their first introduction in [MRA87], packet filters have been used successfully in network monitoring and diagnosis.

Malicious or buggy packet filters can exploit the high-level of privilege they are executed at and corrupt the kernel's internal data structures. To prevent this, various safety techniques, including proof-carrying code, can be used. In this section, after describing the safety policy and the overall implementation of packet filters with proof-carrying code, I show quantitative comparisons with other techniques that can achieve a similar level of safety. Another description of these experiments has appeared in the *"Second Symposium on Operating System Design and Implementation"*[NL96].

8.1.1 The Safety Policy

Following the standard procedure for using proof-carrying code, the kernel administrator must establish first a safety policy for packet filters. To simplify the comparisons with alternative techniques, the chosen safety policy models that enforced by the Berkeley Packet Filter architecture (BPF) [MJ93] implemented in many Unix operating system kernels. The BPF safety policy allows the filter to examine the current network packet and to write to a kernel-provided scratch memory area. Informally, the safety policy requires that: (1) memory reads are restricted to the packet and the scratch memory; (2) memory writes are limited to the scratch memory; (3) all branches are forward; and (4) reserved and callee-saves registers are not modified. These rules establish memory safety and termination assuming that the kernel calls the packet filter with valid packet and scratch memory addresses.

The BPF safety policy can be obtained easily as a type-based safety policy, as described in Section 6.2. Each filter agent has three input arguments, passed in registers as follows. The registers $\mathbf{r}_1$ and $\mathbf{r}_2$ contain the start address of the current packet along with the index of the last valid word, and $\mathbf{r}_3$ contains the address of a 16-byte scratch memory area. The filter must return its boolean result in register $\mathbf{r}_0$. Furthermore, the safety policy designer specifies that, since the network packets are all Ethernet packets, they are are at least 64-bytes long.

Unfortunately, the type-based safety policies introduced in Section 6.2 are not quite sufficient to encode the BPF safety policy. We also need to extend the type system to distinguish between read-only arrays (used to represent the inspected packet) and regular read-write arrays (used to represent the scratch memory). Let us say that the notation "$\tau[l]^{\mathtt{ro}}$" denotes a read-only array and that the axiomatization of type-based safety is extended appropriately. To be fully compliant with the BPF safety policy we modify VCGen to disallow backward-branches. With these changes, the specification part of the safety policy can be expressed as follows (recall that the length component of an array type denotes the last accessible index):

$$\begin{array}{rcl} \mathtt{Pre}_{\mathtt{pf}} & = & \mathbf{r}_1 : \mathtt{int}[\mathbf{r}_2]^{\mathtt{ro}} \wedge \mathbf{r}_3 : \mathtt{int}[12] \wedge \mathbf{r}_2 \geq 60 \\ \mathtt{Pre}_{\mathtt{pf}} & = & \mathbf{r}_0 : \mathtt{bool} \end{array}$$

8.1.2 Performance Comparisons with Other Techniques

In order to experiment with proof-carrying code packet filters, I have implemented four typical packet filters in the DEC Alpha assembly language. To illustrate a simple case of packet filter programming, I show below the equivalent SAL code for Filter 1.

$$\begin{array}{lll} & \mathbf{r}_1 \leftarrow M[\mathbf{r}_1 + 12] & \#\ \text{Load bytes at offset 12–15} \\ & \mathbf{r}_1 \leftarrow \mathtt{extr}(\mathbf{r}_1, 0, 2) & \#\ \text{Extract 2 bytes at offset 12–13} \\ & \mathbf{r}_0 \leftarrow 0 & \#\ \text{Failure code} \\ & \mathtt{cond}\ \mathbf{r}_1 \neq 8, \mathrm{L}_0 & \#\ \text{Compare to } \mathtt{ETHER\_IP} \\ & \mathbf{r}_0 \leftarrow 1 & \\ \mathrm{L}_0: & \underline{\mathbf{return}} & \end{array}$$

The "$\mathtt{extr}(e, i, l)$" expression operator, which is supported directly in the DEC Alpha instruction set, is used to extract a sequence of l bytes starting at index i in the 4-byte expression e. A typical packet filter accepts or rejects network packets by examining the contents of various fields in the packet header. The location and the valid values of these fields are defined by the network protocol standards. For example, in an Ethernet packet the bytes located at offset 12 and 13 in the packet must contain the values 8 and 0 respectively if the packet belongs to the IP protocol. Filter 1 shown above accepts exactly the IP packets, assuming that the host machine (a DEC Alpha in the experiments) is a "little-endian" machine, where memory words are stored in memory in the least-significant-first order. Note that the array bounds checks are not necessary for safety because the packets are at least 64-bytes long.

Filter 2 accepts IP packets originating from a given network (with number 128.2.206). This involves checking a 24-bit value (starting at offset 26 in the packet) in addition to the work done by Filter 1.

Filter 3 accepts IP or ARP packets exchanged between two given networks (128.2.206 and 128.2.209). This is the filter with the most complex control-flow among those considered in my experiments because the source and destination fields are located at different offsets in an IP and an ARP packet and also because these fields sometimes spread across word boundaries. This filter first branches depending on whether the packet is an IP packet or an ARP packet (the packet is rejected if neither is true). Then the originating network address (3 bytes starting at offset 26 for IP packets and at offset 28 for ARP packets) is compared to 128.2.206 and then to 128.2.209. If this test succeeds, it is repeated for the destination address (3 bytes starting at offset 30 for IP packets and at offset 38 for ARP packets).

Packet Filter		1	2	3	4
Instructions		8	15	47	28
Proof Size (bytes)		132	260	1008	688
Validation Cost	Time (μs)	93	234	891	593
	Space (kilobytes)	1.1	1.4	2.1	3.8

Figure 8.1: Proof size and validation cost for PCC packet filters.

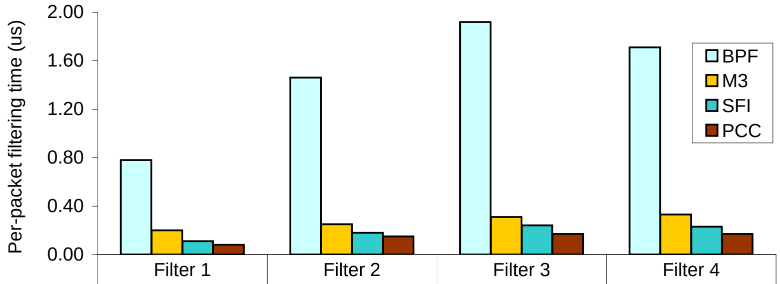
Finally, filter 4 accepts all TCP packets with a given destination port. This filter has to check that the Ethernet packet is an IP packet, then that it is a TCP packet, then that this is the first packet in a sequence and lastly, that the destination port matches a given value. The interesting feature of this filter over the others is that the offset of the TCP destination-port field is computed based on the value of a header field (the length of the IP header). Because it is not guaranteed that this computation yields an offset less than the minimum packet size (64 bytes), filter 4 contains a run-time bounds check. Furthermore the possible lack of alignment of the TCP port byte requires a few extra instructions.

As a matter of procedure, I wrote the packet filters by hand in DEC Alpha assembly language. Because, they do not contain loops or helper functions, there was no need for additional annotations. For the same reasons, a simpler verification-condition generator can be used in a PCC system dedicated to packet filters. The verification conditions are then proved using the theorem prover discussed in Chapter 7 and the proofs are checked using the LF_i proof-checker, as discussed in Chapter 5.

Let us start with a discussion of the proof-related costs of proof-carrying code for packet filters. The other costs, such as theorem proving cost, are discussed for a similar but larger set of agents in the next section. Figure 8.1 shows, for each of the four packet filters, the size of the proof and the time required for checking it on a 175-MHz DEC Alpha processor. The last line in the figure shows that the memory requirements during proof validation are relatively modest. We shall see in Section 8.3 that the scratch memory used during type checking increases very slowly with the size of the proof.

Let us focus next on the benefit of PCC packet filters, namely the high run-time performance characteristic of hand-optimized assembly-language programs. To gauge the significance of this benefit, I have implemented the same four filters using three other techniques that can be used to enforce the same safety policy. These alternative techniques are described next.

The standard way to ensure safe execution of packet filters is to interpret the filter program and to perform extensive run-time checks. This approach is best exemplified by the BSD Packet Filter architecture [MJ93], commonly referred to as BPF, that was also the inspiration source for this set of experiments. In the BPF approach the filter is encoded in a restricted accumulator-based language. According to the BPF semantics, a filter that



	Filter 1	Filter 2	Filter 3	Filter 4
BPF	0.78	1.46	1.92	1.71
M3	0.20	0.25	0.31	0.33
SFI	0.11	0.18	0.24	0.23
PCC	0.08	0.15	0.17	0.17

Figure 8.2: Comparison of average per-packet run time.

attempts to read outside the packet or the scratch memory, or to write outside the scratch memory, is terminated and the packet rejected.

The BPF interpreter makes a simple static check of the packet filter code to verify that all instruction codes are valid and all branches are forward and within code limits. I measured this one-time overhead to be a few microseconds, which is negligible. BPF packet filters, however, are about 10 times slower than the PCC filters. In the PCC approach all checks are moved to the validation stage, allowing for much faster execution.

In order to collect data for the BPF packet filters, I extracted the BPF interpreter as implemented by the OSF/1 kernel and compiled it as a user library.

Another approach to safe code execution is Software Fault Isolation (SFI) [WLAG93]. SFI is an inexpensive method for parsing binaries and inserting run-time checks before memory operations. There are many flavors of SFI depending on the desired level of memory safety. If the entire code runs in a single protection domain whose size is a power of two, and if only memory stores are checked, then the run-time cost of SFI is relatively small. If, on the other hand, the untrusted code interacts frequently with the code receiver or other untrusted components residing in different protection domains and the read operations must be checked also, the overhead of the run-time checks can amount to 20% [WLAG93]. A more serious disadvantage of SFI is that it can only ensure memory safety. In order to better accommodate SFI for packet filters, the safety policy must be relaxed slightly. For example, I assumed that each packet is aligned on a 2048-byte boundary and the entire 2048-byte block is accessible to the filter.

Usually, SFI is performed by a trusted component of the code receiver. But SFI can also be performed at the code-producer site, possibly as part of the code-generation phase of a modified compiler. In such a case, the code receiver must still inspect the code and verify that all run-time checks are present. Such a validator is reportedly simple if it does not try to eliminate redundant checks [WLAG93]. On the other hand, a relatively more complex

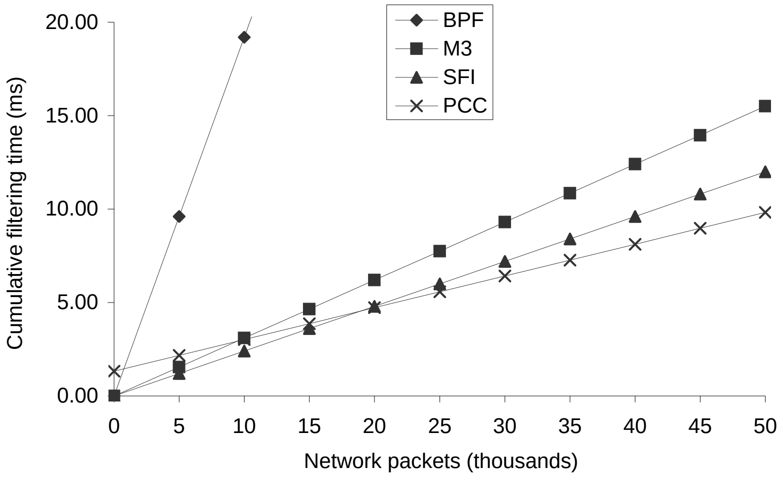


Figure 8.3: Startup cost amortization for Filter 3.

and slower validator is required if we want the validator to optimize the placement of the run-time checks. To my knowledge, such an SFI validator does not presently exist.

In order to collect data for SFI packet filters, I have inserted run-time checks for all the memory operations in the assembly language filters. The cost of these checks is a 25% slowdown of the filtering process. As part of the SFI experiment, I have also produced safety proofs attesting that the resulting SFI packet filters adhere to the safety policy. These proofs can also be viewed as proofs that SFI was performed correctly with the effect that proof-checking replaces a SFI validator as that described above.

The last alternative technique examined here is to write the filters in a type-safe language and to rely on the correctness of the compiler to ensure that the execution is type safe. This approach to untrusted-code safety, using the Modula-3 language [Nel91], is taken in the SPIN extensible operating system [BSP$^+$95]. In an attempt to maximize the run-time performance of the resulting packet filters, I use the VIEW language extension [SSP$^+$96] developed for the purpose of the SPIN project to allow efficient manipulation of multi-byte fields in arrays of bytes. Unfortunately, there is no way to communicate to the Modula-3 compiler that most bounds checks can be eliminated because the packets are at least 64-bytes long. This, coupled with a moderate amount of register shuffling and spilling introduced by the compiler, makes the packet filters obtained through compilation from Modula-3 source code a factor of two slower than the hand-optimized packet filters.

Figure 8.2 shows a comparison of the per-packet running time of the four filters discussed above implemented using BPF, Modula-3, SFI and PCC. All performance measurements were done on a DEC Alpha 3000/600 with a 175-MHz processor, a 2-MByte secondary cache and 64-MByte main memory, running OSF/1. All measurements were performed off-line using a 200,000-packet trace from a busy Ethernet network at Carnegie Mellon University.

From a per-packet latency point of view, the PCC packet filters outperform filters developed using any other considered approach. They are ten times faster than interpreted BPF filters, two times faster than filters compiled from Modula-3 and 25% faster than filters using SFI.

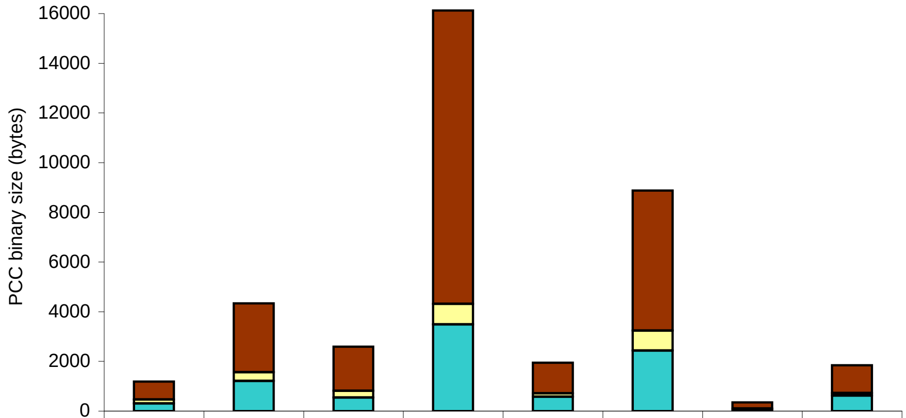
The PCC method has a one-time startup cost consisting of the proof-validation cost. Despite this relatively high validation cost, the run-time benefits of PCC packet filters are large enough to amortize the startup cost after processing a reasonable number of packets. Figure 8.3 shows the overall running time, including the startup cost, as a function of the number of packets processed, for Filter 3 (the filter with the largest proof-validation cost). In this particular case, the cost of proof validation is amortized after 800 packets when compared to the BPF version of the filter, after 10,000 packets when compared to the Modula-3 version and after 19,000 packets when compared to the SFI packet filter. Note that I counted about 1000 Ethernet packets per second at the time when I collected the packet trace used for the experiments.

8.2 Experiments with the Touchstone Compiler

The next set of experiments I am discussing here attempts to present a global quantitative picture of a proof-carrying code system using the Touchstone certifying compiler as a front-end. These experiments exercise all the parts of the system discussed in this dissertation. I have measured several quantitative aspects of the system. First, as a follow-up to the experiments using packet filters, I show what are the relative sizes of executable code, loop invariants and proofs, in PCC binaries obtained with the Touchstone compiler. Then, I show the distribution of time in a typical PCC session consisting of compilation, verification-condition generation, proving and proof checking. Finally, to support the claim that certification can coexist with optimizations, I show that the code produced by the Touchstone compiler is competitive with code produced with optimizing traditional compilers for C, such as DEC `cc` and GNU `gcc` [Fou93]. To complete the comparison with the traditional C compilers I also compare the code sizes and the compilation times.

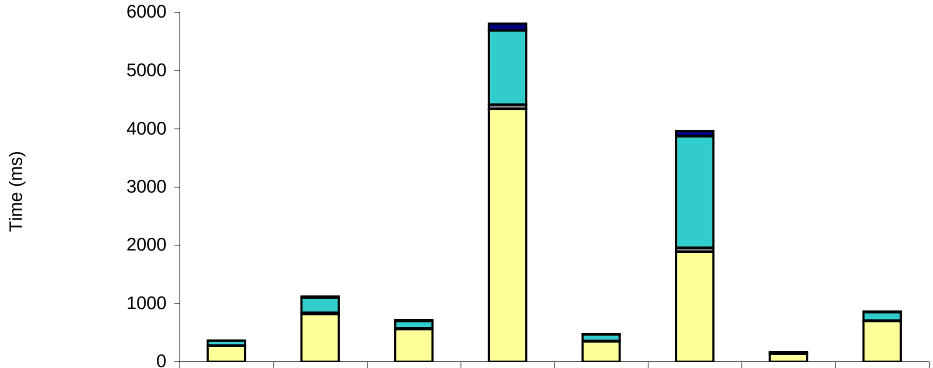
The benchmark programs used for the experiments contain only those language features that are currently implemented in the Touchstone certifying compiler (this rules out floating-point benchmarks, for example) with a bias towards programs for which array bounds-checking elimination could make a significant difference in the running time. I furthermore preferred programs that might be useful as native-code components in a safe mobile-code system, in order to evaluate the certifying compiler as a front-end to a system for safe execution of Proof-Carrying Code.

These considerations led to eight benchmarks. Three of them, `blur`, `sharpen`, and `edge` are bidimensional convolutions used as image processing filters in the `XV` image-manipulation program [Bra97]. `qsort` is an implementation of the quicksort algorithm for an array of integers. `simplex` is the linear programming algorithm implemented for rational numbers. `kmp` (an implementation of the KMP search algorithm) and `unpack` (one of the `gzip` decompression algorithms and the core of the Unix utility with the same name) were chosen



	blur	sharp	qsort	simplex	kmp	unpack	bcopy	edge
Proof (bytes)	718	2774	1778	11810	1234	5636	250	1122
Annot (bytes)	162	342	272	825	132	804	36	102
Code (bytes)	304	1216	544	3488	576	2432	64	624

Figure 8.4: The relative sizes (in bytes) of proofs, invariants and the machine code.



	blur	sharp	qsort	simplex	kmp	unpack	bcopy	edge
Proof Checking (ms)	7.7	23.5	16.3	120.3	9.6	92.5	3.6	14.7
Proving (ms)	81.0	257.0	127.0	1272.0	108.0	1912.0	25.0	143.0
VC Generation (ms)	6.3	20.9	11.5	73.9	8.4	70.6	3.5	9.7
Code Generation (ms)	271.0	818.0	560.0	4340.0	348.0	1885.0	136.0	697.0

Figure 8.5: The distribution of time spent for the compilation and certification of several benchmarks. The data in the table is expressed in milliseconds.

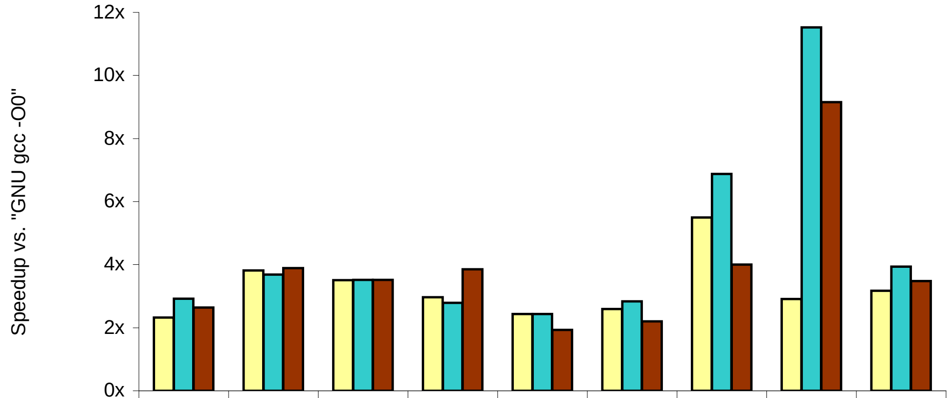
as examples of cases where array bounds-checking elimination is not effective. The `bcopy` program is an implementation of string copy for non-overlapping strings. It is worth noting that some of these C programs are fairly realistic in both size and complexity, and none required anything more than minor syntactic modifications to conform to the safe C dialect compiled by Touchstone. The main changes involved replacing the use of pointer arithmetic with array indexing. All results are the average of at least 1000 runs on a DEC Alpha 21064 running at 175MHz.

Figure 8.4 shows the size of the safety proofs and of the annotations as compared to the size of the machine code for each benchmark. The annotations are only 30% of the size of the code, on the average. The average ratio of proof size to code size is 2.5, which is consistent with observations in experiments with PCC using hand-written assembly language. Similar results are shown in Section 8.3 for a similar but larger set of experiments.

Figure 8.5 displays graphically the distribution of the time spent for compilation and certification. On the average, 72% of the time is spent compiling, 22% is used for theorem proving and the rest of 6% is split evenly between VC generation and proof checking. Based on these results we can make two observations. First, the cost of certification is only about a third of the cost of compilation, meaning that it is reasonable to use the certifier throughout the life of the compiler, and not just during compiler development. Second, not only are VCGen and the proof checker much simpler than the compiler and the theorem prover, but they are also much faster. Hence, the safety-critical PCC infrastructure is both small and fast. This is important in situations when the infrastructure is executed on systems with limited computational power, such as smart cards.

Figure 8.6 shows the effect of optimizations on the running time of the benchmark programs for the GNU `gcc` compiler, the DEC `cc` compiler, and the Touchstone certifying compiler. The C compilers were invoked with all optimizations enabled (`-O4`). The running times are reported as speedups over the running time of the unoptimized code as compiled with `gcc -O0`. The last set of bars in Figure 8.6 is the geometric mean of the speedups for each compiler. On the average, the Touchstone compiler performs slightly better than `gcc` (by about 10%) and not quite as well as `cc` (the difference being about 12%). The programs for which Touchstone is not quite as good as the C compilers are `kmp` and `unpack`, due to the bounds checks that cannot be eliminated, and `bcopy`, because of the lack of loop-unrolling In addition to array bounds-checking elimination, the inter-procedural register allocation and the common-subexpression elimination played a major role in making the quality of code generated by Touchstone comparable to that produced by the other C compilers.

In my experiments, the C compilers compile the programs unsafely (that is, without any bounds checking), while Touchstone has the handicap of having to implement (and then hopefully remove) the array bounds checks. The array bounds-checking elimination described in Section 6.5.9 is able to eliminate most of those checks whose proof of redundancy is local to the current function, but is ineffective when the elimination requires global information. This weakness is a problem in all of our benchmarks except for `blur`, `edge` and `bcopy`. To alleviate the cost of array bounds checking in these cases, I have added to each of these

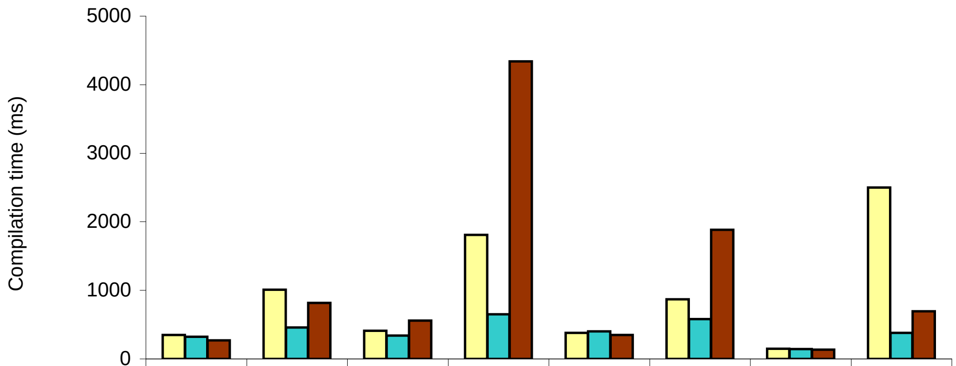


	blur	sharp	qsort	simplex	kmp	unpack	bcopy	edge	GMEAN
GNU gcc -O4	2.33	3.82	3.51	2.97	2.44	2.59	5.50	2.92	3.17
DEC cc -O4	2.92	3.68	3.52	2.79	2.44	2.84	6.88	11.52	3.94
Touchstone	2.64	3.89	3.52	3.86	1.93	2.20	4.00	9.16	3.48

Figure 8.6: The effect of optimizations in the certifying compiler, expressed as the ratio between the running time of the optimized code to the running time of the same code compiled with "GNU gcc -O0". For comparison, we also show for each benchmark the effect of the optimizations in the GNU C compiler ("GNU gcc -O4") and the vendor C compiler ("DEC cc -O4"). The last column is the geometric mean over all the benchmarks.

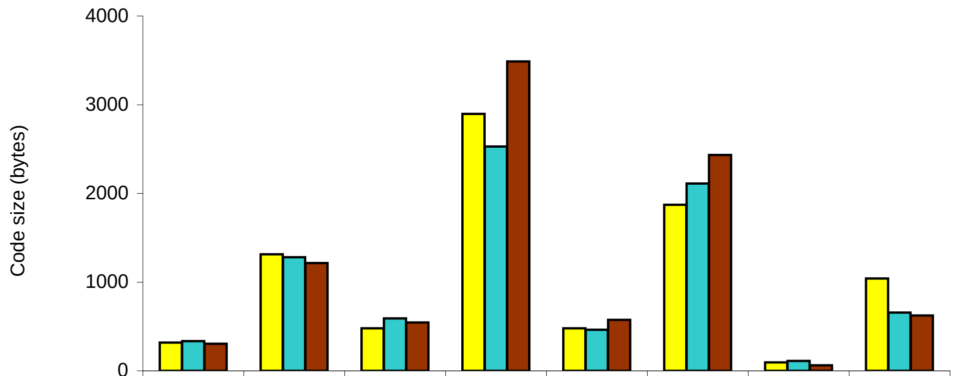
benchmarks assert conditionals that enable Touchstone to eliminate all bounds checking inside loop bodies. Two other benchmarks, `kmp` and `unpack`, compute array indices based on the contents of some auxiliary data structures. The formal safety argument for these array operations involves the proof of complicated global program invariants, and thus it is probably not reasonable to expect a compiler to be able to eliminate these bounds checks automatically.

Due to the fact that Touchstone is an early prototype, the compilation time is significantly larger than that of the C compilers used in the performance comparisons. Figure 8.7 shows the compilation times (not including the time for VC generation, proof generation or proof checking) of Touchstone and of the C compilers (with all optimizations enabled) for our set of benchmarks. On the average, Touchstone is 20% slower than GNU `gcc` and 72% slower than DEC `cc`. Figure 8.8 shows the comparison of the machine-code sizes of programs compiled with Touchstone and the C compilers. Unlike the compilation times, the sizes of machine code emitted by Touchstone are within 5% of that emitted by the C compilers. Note, however, that there is no fundamental reason why a certifying compiler should emit code that is larger than that emitted by a traditional compiler. With respect to the compilation time, the certifying compiler must incur the extra cost of emitting the loop invariants and type specifications. This cost, however, should negligible when compared to the rest of the compilation effort.



	blur	sharp	qsort	simplex	kmp	unpack	bcopy	edge
GNU gcc -O4	350	1010	410	1810	380	870	150	2500
DEC cc -O4	325	460	340	650	400	580	145	380
Touchstone	271.0	818.0	560.0	4340.0	348.0	1885.0	136.0	697.0

Figure 8.7: Comparison of the compilation time for Touchstone and the GNU `gcc` and DEC `cc` compilers with all optimizations enabled. The times in the table are shown in milliseconds. On the average, Touchstone is 20% slower than `gcc` and 72% slower than `cc`. Note that the compilation time does not include VC generation, proof generation or proof checking.



	blur	sharp	qsort	simplex	kmp	unpack	bcopy	edge
GNU gcc -O4	320	1312	480	2896	480	1872	96	1040
DEC cc -O4	336	1280	592	2528	464	2112	112	656
Touchstone	304	1216	544	3488	576	2432	64	624

Figure 8.8: Comparison of the target code sizes for programs compiled with Touchstone and the GNU `gcc` and DEC `cc` compilers with all optimizations enabled. The sizes in the table are shown in bytes of machine code. On the average, Touchstone is within 5% of the sizes of code emitted by the C compilers.

8.3 Experimental Validation of LF$_i$

In this section I describe a set of experiments whose purpose is to quantify the advantages of implicit representation of proofs and the benefits of the various optimizations discussed in Section 5.6. The experimental data was collected for a set of 85 proofs of memory safety and type safety produced using the theorem prover described in Chapter 7 and the Touchstone compiler described in Chapter 6. I measured for all these programs, the sizes of the fully explicit and bimodal representations of proofs and the time and memory required for proof reconstruction of the fully explicit form and of the bimodal representations. All measurements are done on a Pentium II machine running at 300MHz. In all cases, except for a few fully explicit representations, the proofs fit in the first-level cache whose size is 512Kbytes.

But before I even start with analyzing the optimizations, I show the typical proof sizes, proof reconstruction time and the memory usage during proof reconstruction, for the proof representation and the set of proof optimizations that are presently used in the proof-carrying code system. Figure 8.9 shows the proof reconstruction time as a function of the proof representation size. In the worst case, the complexity of reconstruction can be expected to be non-elementary by similarity with the problem of normalizing terms in typed λ-calculus [Sta79]. The correlation that I observed experimentally is almost linear and the reason is that all of the experiments presented here are with variants of first-order logic, where the use of quantification and higher-order syntax is very limited. Furthermore, the proofs considered in this experiment state type safety in a type system whose type checking is syntax directed.

Figure 8.10 shows the scratch memory usage during proof reconstruction. In the largest example, it uses 13 kilobytes of working memory when reconstructing a proof whose bimodal representation is 12.7 kilobytes and whose explicit representation is 1.6 megabytes. We observe from this graph that the reconstruction algorithm not only does not need to recreate the full proof before checking it, but it uses even less memory than the implicit proof itself.

A common characteristic of the experimental results is that the optimizations perform better for larger problem sizes, meaning that they are more effective just when we need them more. Because of this characteristic it is misleading to try to capture the effect of an optimization as an average improvement percentage. Instead, I show scatter plots of the improvement due to the optimization as a function of the problem size.

Figure 8.11 shows the ratio of the size of bimodal LF$_i$ representations and the fully-explicit LF representations, as a function of the size of the LF representation. The size of a representation is measured in bytes of the external form of LF$_i$ terms, as discussed in Section 5.6. Note that the improvement ranges from 4 to over 100 for the larger problems. For the biggest problem size the size of the fully explicit LF representation is over 1.6 megabytes while the LF$_i$ representation is just 12.7 kilobytes, a reduction that has a huge impact on the practicality of proof-carrying code.

Figure 8.12 shows the improvement in the time required for the reconstruction of the bimodal representation with respect to the time required for type checking fully-explicit LF

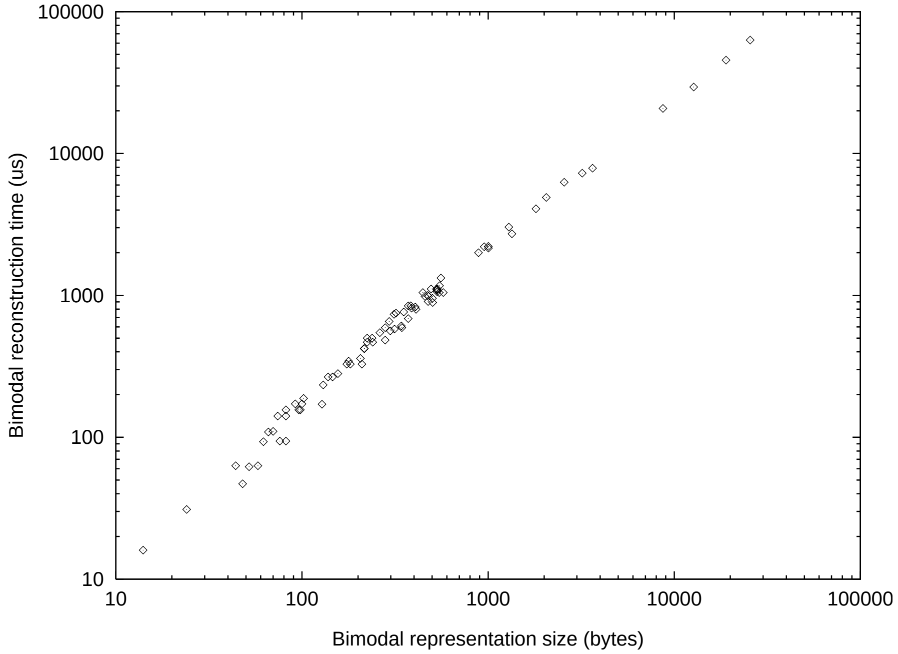


Figure 8.9: The correlation of the reconstruction time with the size of the bimodal representation, on a logarithmic scale.

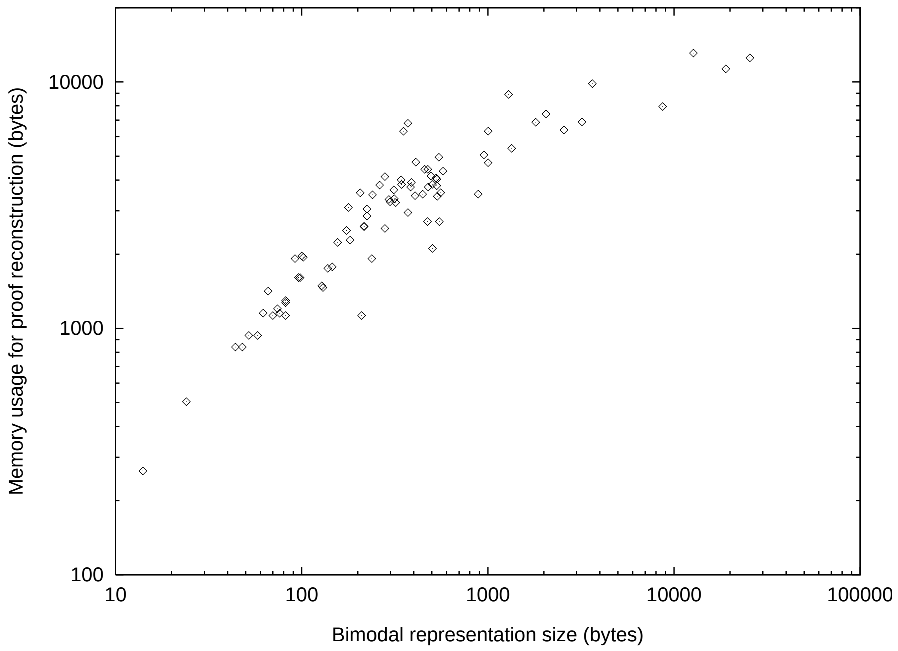


Figure 8.10: The correlation of the memory usage during reconstruction with the size of the bimodal representation, on a logarithmic scale.

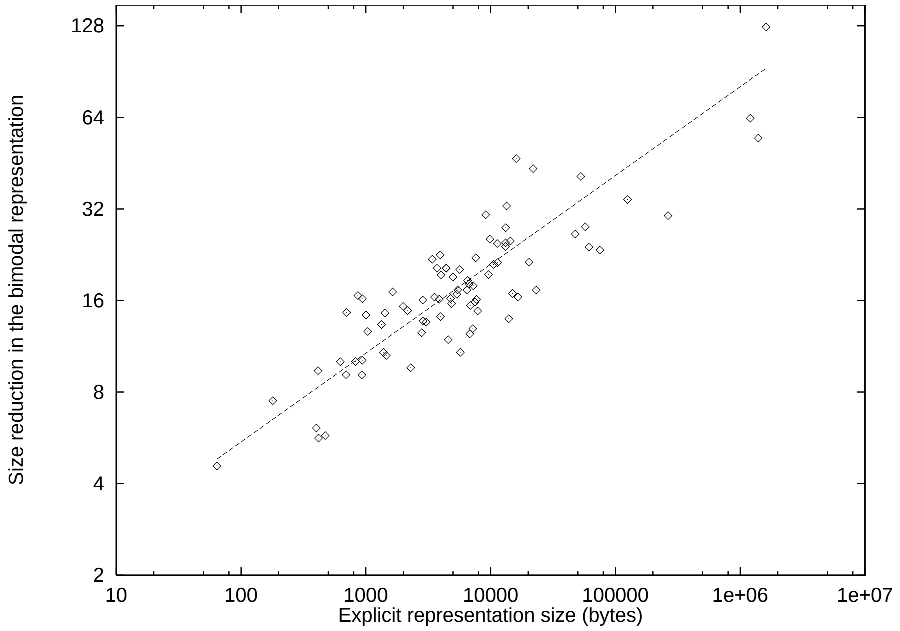


Figure 8.11: The ratio between the size of the explicit representation and the size of the bimodal representation as a function of the original size.

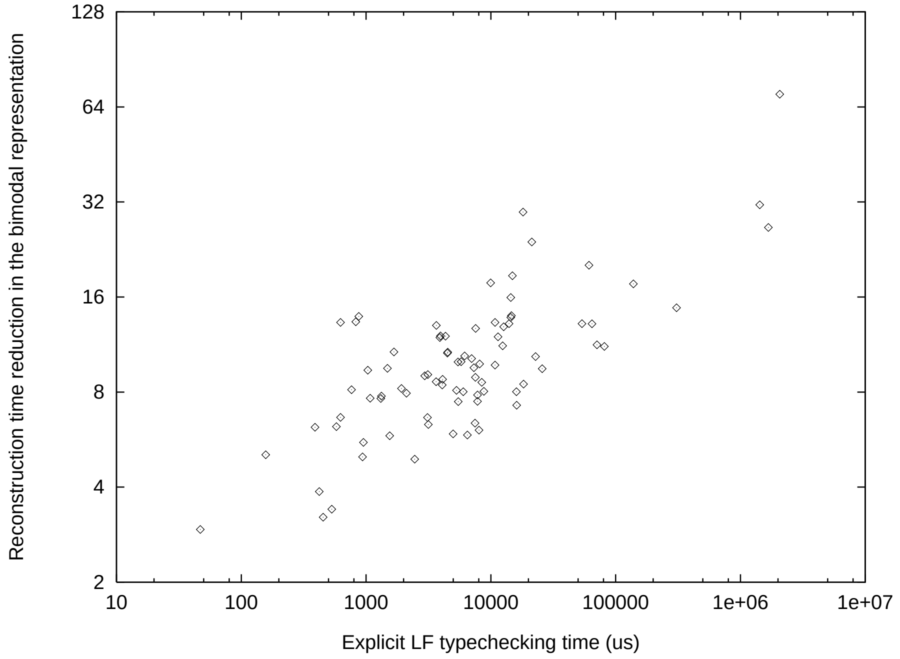


Figure 8.12: The ratio between the time required for the checking the explicit representation and the time required for the reconstruction of the bimodal representation, as a function of the original checking time, on a logarithmic scale.

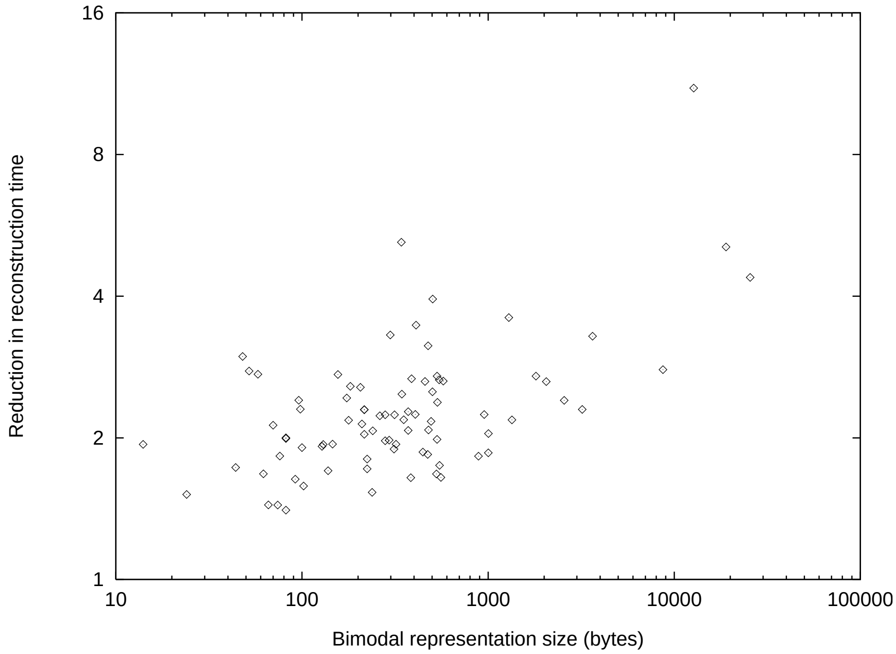


Figure 8.13: The speed-up of the reconstruction due to occurs-check optimization as a function of the bimodal representation size, on a logarithmic scale.

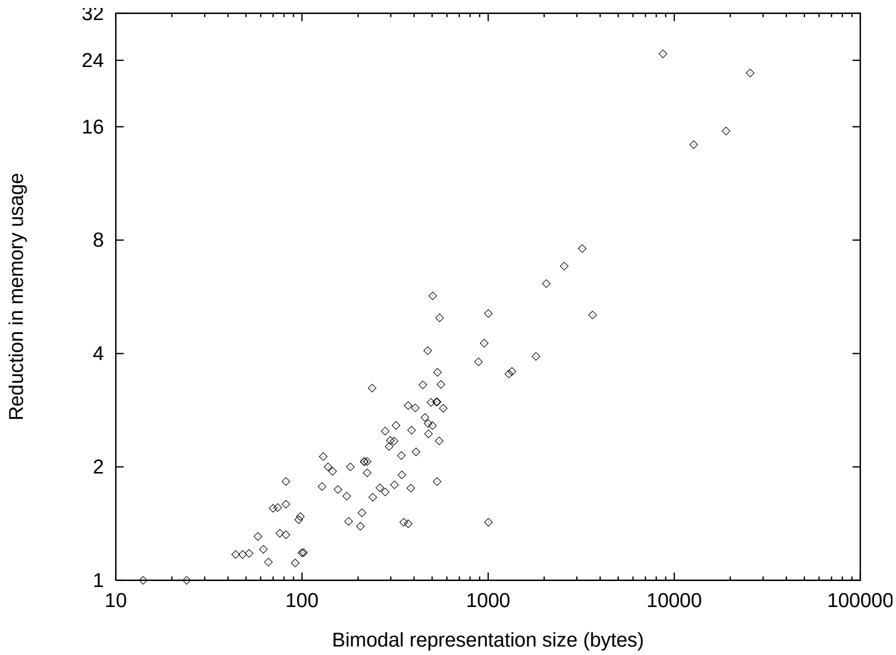


Figure 8.14: The reduction in the memory usage due to the memory optimization as a function of the bimodal representation size, on a logarithmic scale.

terms, on a logarithmic scale. These results demonstrate that, indeed, by using LF_i we can save not only in terms of proof size but also in terms of proof-checking time. In the largest experiment, the reduction factor is over 70, from an LF type-checking time of over 2 seconds to an LF_i reconstruction time of just 70 milliseconds.

Finally, both the occurs-check optimization and the memory usage optimization, although simple, have a major impact on the reconstruction time and the memory usage, as shown in Figure 8.13 and Figure 8.14. The occurs-check optimization eliminates more than 90% of the occurs checks with an overall reduction in the reconstruction time ranging from 50% for the smaller problems to over 10 times in the larger cases. Similarly, the reduction in memory usage ranges from 30% to over 20 times. In both of these cases the optimizations are more effective for the larger problem sizes.

8.4 Discussion

The experiments presented in this chapter demonstrate that there are practical instances where the proof-carrying code technique can be more efficient that other techniques based on interpretation or run-time checking. The performance benefit is due in large part to the static checking of safety and also to involved proof engineering targeted at maintaining a low cost for storing and validating the proofs of safety.

Chapter 9

Conclusions and Future Work

In this final chapter I summarize the main technical contributions of this dissertation and I discuss several promising directions for extending this research.

9.1 Contributions

One of the major challenges of building software systems is to ensure that the various components fit together in a well-defined manner. This problem is exacerbated by the recent advent of software components whose origin is unknown or inherently untrusted, such as mobile code or user extensions for operating-system kernels or database servers. Such extensions are useful for implementing an efficient interaction model between a client and a server because several data exchanges between them can be saved at the expense of a single code exchange.

In this dissertation, I propose to tackle such system integrity and security problems with techniques from mathematical logic and programming language semantics, in the framework provided by **proof-carrying code**. Proof-carrying code requires the extension provider to send along with the extension code a representation of a formal proof that the code meets certain safety and correctness requirements. Then, all the code receiver has to do to ensure the safety of executing the extension, is to validate the attached proof.

Proof-carrying code has several key characteristics that, in combination, give it an advantage over previous approaches to safe execution of foreign untrusted code. The following advantages can be claimed:

1. *PCC is general.* PCC can be used to enforce more than memory safety, more even than type safety. At an extreme, PCC can be used to verify any code property for which there exists a logic capable of expressing it. This includes many code properties that would otherwise be undecidable to infer from the code alone. PCC has been tested with safety properties ranging from memory and type safety to bounded resource usage.

2. *PCC receiver infrastructure is low-risk and automatic.* The proof-checking process used by the code receiver to determine code safety is completely automatic, and can be implemented by a program that is relatively simple and easy to trust. Thus, the safety-critical infrastructure that the code receiver must rely upon is reduced to a minimum.

3. *PCC is efficient.* In practice, the proof-checking process runs quickly. Furthermore, in contrast to previous approaches, the code receiver does not modify the code in order to insert costly run-time safety checks, nor does the code receiver perform any other checking or interpretation once the proof itself has been validated and the code installed.

4. *PCC does not require trust relationships.* The code receiver does not need to trust the producer. In other words, the receiver does not have to know the identity of the code producer, nor does it have to know anything about the process by which the agent code was produced. All of the information needed for determining the safety of the code is included in the annotated agent code and its proof.

5. *PCC is flexible.* The proof-checker does not require that a particular programming language be used. PCC can be used for a wide range of languages, even machine languages. Furthermore, a code receiver can support multiple agent languages and safety policies with a minimal duplication of the infrastructure components.

6. *PCC producer can be automated in special cases.* If the safety properties can be decided statically or enforced through systematic run-time checks, a certifying compiler together with a matching theorem prover can be used on the code-receiver side to automate the process of producing the annotations and proofs.

Many of the advantages of PCC are due to the imbalance between the difficulty of the tasks assigned to the code producer (the generation of annotations and proving of verification conditions) and the code receiver (the generation of verification conditions and the validation of proofs). This dissertation also contributes the concept of a certifying compiler and the design of proof-generating decision procedures, whose purpose is to assist code producers with their attributions in a proof-carrying code system.

A **certifying compiler** is a compiler that emits, in addition to target code, function specifications and loop invariants that enable a theorem proving agent to prove non-trivial properties of the target code, such as type safety. Such a certifying compiler, along with a proof-generating theorem prover, is not only a convenient producer of proof-carrying code but also a powerful software-engineering tool. The certifier acts as an effective referee for the correctness of each compilation simplifying considerably compiler testing and maintenance. The following advantages can be claimed for a certifying compiler:

1. Certifying compilation, when used in conjunction with proof-generating theorem proving, is a practical method for producing, in an automatic manner, the loop invariants and the proofs required in a proof-carrying code system for type safety.

2. Certifying compilation is significantly easier to employ than a formal verification of the compiler, even if the formal verification is restricted to proving only that the target code is type safe. This is because it is easier in general to verify the correctness of the result of a computation than to prove the correctness of the computation itself. Furthermore, with this approach, most compiler revisions and improvements do not require any change to the certifier.

3. This method can be applied to optimizing compilers, because the design of the certifier does not restrict the optimizations that the compiler is allowed to perform. The Touchstone optimizing compiler generates code that, for many programs, matches or is within 15% of the performance of both `gcc` and `cc` with all optimizations enabled.

4. The presence of the certifier drastically improves the effectiveness of compiler testing because, for each test case, it statically signals compilation errors that might otherwise require many executions to detect. Even though this approach does not ensure full compiler correctness, in my experience the vast majority of compiler bugs lead the compiler to generate unsafe target programs for at least one of the test cases.

5. This method is applicable to the compilation of any type-safe language, as well as for certifying other properties of the target programs beyond type safety. Also, a significant benefit of the design is that it requires relatively few modifications to a traditional compiler, and hence it should be possible to adapt existing compilers to this technique.

A complete system for proof-carrying code must also contain a **proof-generating theorem prover** for the purpose of producing the attached proofs of safety. This dissertation shows how standard decision procedures can be adapted so that they can produce detailed proofs of the proved predicates. Just like in the case of the certifying compiler, a proof-generating theorem prover has significant software-engineering advantages over a traditional prover. In this case, a simple proof checker can ensure the soundness of each successful proving task and indirectly assist in testing and maintenance of the theorem prover.

9.2 Future Work

The concepts presented in this thesis are broad and general even though, for feasibility reasons, the actual implementations that I describe covers only a small part of the design space. In this section, I discuss a few of the most promising directions for extending the ideas discussed in this dissertation.

One of the most notable limitations of the implementation of proof-carrying code described here is that it cannot handle liveness properties and general security policies. Recall from Chapter 3, that the general security policies are those that can be expressed as predicates on the set of all possible executions of a program. Information flow is an example of such a security property. Then there are the security properties that can be expressed as predicates on individual executions. Among the security properties we have the safety properties that disallow certain "bad things" to happen during the execution, and the liveness properties stipulating that certain "good things" must happen. As explained in detail in Chapter 3, the VCGen implementation of the proof-carrying code is targeted to enforcing safety properties.

However, as we have already seen in the case of type-based safety policies, described in Section 6.2, the VCGen-based approach can be used to enforce data abstraction. Data abstraction is not a safety property, not even a security property. Data abstraction is a security policy according to the definition of Schneider [Sch98]. This is because we cannot tell that a program violates the abstraction boundaries by looking only at one of its executions. We have to look at the source code or, equivalently, at the set of all possible executions, to be able to distinguish between programs truly respecting abstraction and those that violate the abstraction boundaries by "guessing" the concrete representations.

The experience with type-based safety policies suggests that a promising technique for extending proof-carrying code to new flavors of security properties is to use non-standard type systems as the basis for the security policy and to adapt VCGen and theorem proving to act as a type-checking system. What makes this strategy of special interest is that it would position proof-carrying code to gain leverage from a large body of work in the area of type-based static analyses. For example, several research groups have developed non-traditional type systems for expressing and checking information flow [HR98, ML97, VS97] or non-interference [VS97], and for performing control-flow analysis [Hei95, PS95, PO95], strictness analysis [KM89], or binding-time analysis [Hen91].

One of the main advantages of using VCGen in a proof-carrying code system is that it is insensitive to many code transformations such as variable renaming, common-subexpression elimination and dead-code elimination. While this is a feature in most cases, it can also prevent the use of VCGen for certain safety policies. For example, the new generation of EPIC processors allow even non-valid memory addresses to be read as long as the result of the read operation is not used in the subsequent computation [HHG+95]. There is no natural way to capture such behavior with a VCGen, because the property of a value being used in the subsequent computation is not a property of the current state of the memory and the registers. In order to handle such code properties, we must either extend the notion of the state of execution in creative ways or else, use a more conventional syntax-based type checking approach.

More opportunities for future research are suggested by the limitations that I have imposed on the source language compiled by the Touchstone compiler. Many of these limitations are imposed by VCGen. Currently, VCGen cannot handle programming paradigms that result in the target code being enriched with runtime data structures that affect its behavior. Examples of such problematic data structures are the function closures for implementing higher-order functions, dynamic-dispatch tables for implementing object-oriented languages, exception-dispatch tables for implementing exceptions and root tables for assisting a garbage collector. In fact, VCGen does currently support one such runtime data structure, namely the stack. Other data structures can be supported in a similar manner, although at the cost of a more complicated VCGen.

A last direction for future research that I mention here is in the area of proof optimizations. Recall from Chapter 5, that when transitioning from the LF representation of proofs to the implicit LF_i representations, the proof sizes and validation times are decreased due to the elimination of redundant components. However, there are still significant common sub-proofs that are not shared in the LF_i representation. Such common sub-proofs cannot be considered redundant, because it is not always possible to reconstruct them if they are missing.

Consider, for example, the situation where the same subgoal occurs several times in a verification condition. Then, it is very likely that the same proof of the common subgoal will appear multiple times in the proof. In such situations it might be beneficial to apply the same strategy that humans use for handling large proofs: state and prove lemmas that are then used in proofs as building blocks. Lemma extraction is the process by which a proof is scanned and common sub-proofs are identified and factored out, in a manner similar to common-subexpression elimination. Lemma extraction can also be used to scan the safety proofs of multiple agents for the same safety policy. The lemmas detected in this case can be added as derived proofs rules to the axiomatization of the safety policy. Then, all proofs can be simplified by simply referring to these lemmas instead of proving them each time they arise.

To conclude this dissertation, I note that the research described here demonstrates that ideas from logic and programming languages can and should be used for solving problems posed by real software systems. The design and implementation of proof-carrying code shows how, once we identify a practical problem to address, it is helpful to focus on the underlying abstract phenomenon and to analyze and understand thoroughly the supporting theory. By working at a higher level of abstraction, we can propose solutions that are elegant, rigorous and general so that they can outlast the particular artifacts that initially motivated them. Finally, to complete the research, it is important to implement the abstract-level solutions in the concrete practical setting in order to validate them through experimentation.

Bibliography

[ACC93] M. Abadi, L. Cardelli, and P.-L. Curien. Formal parametric polymorphism. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 157–170. ACM New York, NY, 1993.

[ACCL91] Martin Abadi, Luca Cardelli, P.-L. Curien, and J.-J Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, October 1991.

[Ack54] Wilhelm Ackermann. *Solvable Cases of the Decision Problem.* Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.

[AS80] Bengt Aspvall and Yossi Shiloach. A polynomial time algorithm for solving systems of linear inequalities with two variables per inequality. *SIAM Journal on Computing*, 9(4):827–845, April 1980.

[AS85] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[BGJ91] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103–149, September 1991.

[Ble74] W.W. Bledsoe. The Sup-Inf method in Presurger arithmetic. Technical report, University of Texas Math Dept., December 1974.

[BM79] Robert Boyer and J. Strother Moore. *A Computational Logic.* Academic Press, 1979.

[BM81] R. S. Boyer and J. S. Moore. *The Correctness Problem in Computer Science.* Academic Press, New York, NY, USA, 1981.

[Bra97] John Bradley. XV : Interactive image display for the X window system. Available at *ftp://ftp.cis.upenn.edu/pub/xv/*, 1997.

[BSP+95] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Symposium on Operating System Principles*, pages 267–284, December 1995.

[C+97] Alessandro Cimatti et al. A provably correct embedded verifier for the certification of safety critical software. In *Computer Aided Verification. 9th International Conference. Proceedings*, pages 202–213. Springer-Verlag, June 1997.

[CAB+86] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 234–252, 1977.

[CH85] Thiery Coquand and Gerard Huet. Constructions: A higher order proof system for mechanizing mathematics. In *Proc. European Conf. on Computer Algebra (EUROCAL'85), LNCS 203*, pages 151–184. Springer-Verlag, 1985.

[DeB72] N. DeBruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Mat.*, 34:381–392, 1972.

[Det96] David Detlefs. An overview of the Extended Static Checking system. In *Proceedings of the First Formal Methods in Software Practice Workshop*, 1996.

[DFH+93] G. Dowek, Amy Felty, H. Herbelin, Gérard P. Huet, Chet Murthy, C. Parent, Christine Paulin-Mohring, and B. Werner. The Coq proof assistant user's guide. Version 5.8. Technical report, INRIA – Rocquencourt, May 1993.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminancy and formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[DST80] Peter J. Downey, Ravi Sethi, and Robert E. Tarjan. Variations on the common subexpressions problem. *Journal of the ACM*, 27(4):758–771, 1980.

[Dyb85] P. Dybjer. Using domain algebras to prove the correctness of a compiler. In Kurt Mehlhorn, editor, *Proceedings of the 2nd Annual Symposium on Theoretical Aspects of Computer Science (STACS '85)*, volume 182 of *LNCS*, pages 98–108, Saarbrücken, FRG, January 1985. Springer.

[Fou93] Free Software Foundation. GCC - The GNU C compiler. Available at *http://www.gnu.org/software/gcc/gcc.html*, 1993.

[GJS96] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification.* The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[GLB75] Donald I. Good, Ralph L. Londona, and W. W. Bledsoe. An interactive program verification system. *IEEE Transactions on Software Engineering*, 1(1):59–67, March 1975.

[Gor85] Michael Gordon. HOL: A machine oriented formulation of higher-order logic. Technical Report 85, University of Cambridge, Computer Laboratory, July 1985.

[GRW95] Joshua Guttman, John Ramsdell, and Mitchell Wand. VLISP: A verified implementation of scheme. *Lisp and Symbolic Computation*, 8(1/2):5–32, 1995.

[Hei95] Nevin Heintze. Control-flow analysis and type systems. *Lecture Notes in Computer Science*, 983:189–206, 1995.

[Hen91] Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, 5th ACM Conference*, volume 523 of *Lecture Notes in Computer Science*, pages 448–472. Springer, Berlin, Heidelberg, New York, 1991.

[HHG+95] W. W. Hwu, R. E. Hank, D. M. Gallagher, S. A. Mahlke, D. M. Lavery, G. E. Haab, J. C. Gyllenhaal, and D. I. August. Compiler technology for future microprocessors. In *Proceedings of the IEEE*, pages 1625–1640, December 1995.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[HR98] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with security and integrity. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 365–377, San Diego, California, January 1998.

[ILL73] Shigeru Igarashi, Ralph L. London, and David C. Luckham. Automatic program verification I: a logical basis and its implementation. Technical Report CS-TR-73-365, Stanford University, Department of Computer Science, May 1973.

[Int97] Intel Corp. *Intel Architecture Software Developer's Manual. Vol. 1: Basic Architecture.* Intel Corporation, 1997.

[KF72] J. C. King and R. W. Floyd. An interpretation oriented theorem prover over integers. *Journal of Computer and System Sciences*, 6(4):305–323, August 1972.

[Kin71] J. C. King. Proving programs to be correct. *IEEE Transactions on Computers*, 20:1331–1336, November 1971.

[KM89] T-M. Kuo and P. Mishra. Stricness analysis: A new perspective based on type inference. In *Functional Programming Languages and Computer Architecture*, pages 260–272. Springer Verlag, 1989.

[Koz77] Dexter Kozen. Complexity of finitely presented algebras. In *Conference Record of the Ninth Annual ACM Symposium on Theory of Computing*, pages 164–177, Boulder, Colorado, 2–4 May 1977.

[Koz98] Dexter Kozen. Efficient code certification. Technical Report TR 98–1661, Cornell University, January 1998.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Engineering*, 3(2):125–143, 1977.

[Lev84] Hank Levy. *Capability-Based Computer Systems.* Digital Press, 1984.

[LGvH$^+$79] David C. Luckham, Steven M. German, Friedrich W. von Henke, Richard A. Karp, P. W. Milne, Derek C. Oppen, Wolfgang Polak, and William L. Scherlis. Stanford pascal verifier user manual. Technical Report CS-TR-79-731, Stanford University, Department of Computer Science, 1979.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, Reading, MA, USA, January 1997.

[McC63] John McCarthy. Towards a mathematical theory of computation. In Cicely M. Popplewell, editor, *Proceedings of the International Congress on Information Processing*, pages 21–28. North-Holland, 1963.

[Mic96] Microsoft Corporation. Proposal for authenticating code via the Internet. http://www.microsoft.com/security/tech/authcode/authcode-f.htm, April 1996.

[Mil87] Dale A. Miller. A compact representation of proofs. *Studia Logica*, 46(4):347–370, 1987.

[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.

[MJ93] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *The Winter 1993 USENIX Conference*, pages 259–269. USENIX Association, January 1993.

[ML97] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31,5 of *Operating Systems Review*, pages 129–142, New York, October 1997. ACM Press.

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[Moo89] J. Strother Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5(4):461–492, December 1989.

[Mor73] F. Lockwood Morris. Advice on structuring compilers and proving them correct. In *Proceedings of the First ACM Symposium on Principles of Programming Languages*, pages 144–152, 1973.

[MP67] John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. In J. T. Schwartz, editor, *Proceedings of Symposia in Applied Mathematics*. American Mathematical Society, 1967.

[MP92] Spiro Michaylov and Frank Pfenning. An empirical study of the runtime behavior of higher-order logic programs. In D. Miller, editor, *Proceedings of the Workshop on the λProlog Programming Language*, pages 257–271, July 1992. Available as Technical Report MS-CIS-92-86.

[MP93] Spiro Michaylov and Frank Pfenning. Higher-order logic programming as constraint logic programming. In *PPCP'93: First Workshop on Principles and Practice of Constraint Programming*, pages 221–229, Newport, Rhode Island, April 1993. Brown University.

[MRA87] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *ACM Symposium on Operating Systems Principles*, pages 39–51. ACM Press, November 1987. An updated version is available as DEC WRL Research Report 87/2.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[MWCG98] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *The 25th Annual ACM Symposium on Principles of Programming Languages*. ACM, January 1998. To appear.

[Nec97] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, Paris, France, January 1997.

[Nel78] Greg Nelson. An $n^{\log n}$ algorithm for the two-variable-per-constant linear programming satisfiability problem. Technical Report STAN-CS-78-689, Stanford University, 1978.

[Nel81] Greg Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox Palo Alto Research Center, 1981.

[Nel91] Greg Nelson. *Systems Programming with MODULA-3*. Prentice-Hall, 1991.

[NL96] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In USENIX, editor, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 229–243, Berkeley, CA, USA, October 1996. USENIX.

[NL98a] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the 1998 ACM SIGPLAN Conference on Prgramming Language Design and Implementation (PLDI)*, pages 333–344, 1998.

[NL98b] George C. Necula and Peter Lee. Efficient representation and validation of proofs. In *Thirteenth Annual Symposium on Logic in Computer Science*, pages 93–104, Indianapolis, June 1998. IEEE Computer Society Press.

[NL98c] George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In Giovanni Vigna, editor, *LNCS 1419: Special Issue on Mobile Agent Security*, pages 61–91. Springer-Verlag, June 1998.

[NO79] Greg Nelson and Derek Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[NO80] Greg Nelson and Derek C. Oppen. Fast decision procedures bases on congruence closure. *Journal of the Association for Computing Machinery*, 27(2):356–364, April 1980.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[ORW95] Dino P. Oliva, John D. Ramsdell, and Mitchell Wand. The VLISP verified PreScheme compiler. *Lisp and Symbolic Computation*, 8(1/2):111–182, 1995.

[Pfe91a] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[Pfe91b] 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.

[Pfe94] Frank Pfenning. Elf: A meta-language for deductive systems (system description). In Alan Bundy, editor, *12th International Conference on Automated Deduction*, LNAI 814, pages 811–815, Nancy, France, June 26–July 1, 1994. Springer-Verlag.

[PO95] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. *ACM Transactions on Programming Languages and Systems*, 17(4):576–599, July 1995.

[Pol90] Robert Pollack. Implicit syntax. Informal Proceedings of First Workshop on Logical Frameworks, Antibes, May 1990.

[Pra77] Vaughan R. Pratt. Two easy theories whose combination is hard. Unpublished manuscript, 1977.

[PS95] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference. *Information and Computation*, 118(1):128–141, April 1995.

[PSC] Frank Pfenning, Carsten Schürman, and Iliano Cervesato. The Twelf project. Available online at http://www.cs.cmu.edu/~twelf.

[PSS98] Amir Pnueli, M. Siegel, and Eli Singerman. Translation validation. In Bernhard Steffen, editor, *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98*, volume LNCS 1384, pages 151–166. Springer, 1998.

[PT98] Benjamin C. Pierce and David N. Turner. Local type inference. In *The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 252–265, January 1998.

[Sch98] Fred B. Schneider. Enforceable security policies. Computer Science Technical Report TR98-1644, Cornell University, Computer Science Department, September 1998.

[SESS96] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Second Symposium on Operating Systems Design and Implementations*, pages 213–227. Usenix, October 1996.

[Sho78] Robert E. Shostak. An algorithm for reasoning about equality. *Communications of the ACM*, 21(7):583–585, July 1978.

[Sho81] Robert Shostak. Deciding linear inequalities by computing loop residues. *Journal of the ACM*, 28(4):769–779, October 1981.

[Sit92] Richard L. Sites. *Alpha Architecture Reference Manual.* Digital Press, 1992.

[SSP+96] Emin Gun Sirer, Stefan Savage, Przemyslaw Pardyak, Greg P. DeFouw, and Brian N. Bershad. Writing an operating system with Modula-3. In *The Inaugural Workshop on Compiler Support for Systems Software*, pages 134–140, February 1996.

[Sta79] R. Statman. The typed λ-calculus is not elementary recursive. *Theoretical Computer Science*, 9:73–81, 1979.

[TMC+96] David Tarditi, J. Gregory Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *PLDI'96 Conference on Programming Language Design and Implementation*, pages 181–192, May 1996.

[TWW79] James W. Thatcher, Eric G. Wagner, and Jesse B. Wright. More on advice on structuring compilers and proving them correct. In Hermann A. Maurer, editor, *Automata, Languages and Programming, 6th Colloquium*, volume 71 of *Lecture Notes in Computer Science*, pages 596–615, Graz, Austria, 16–20 July 1979. Springer-Verlag.

[VS97] Dennis Volpano and G. Smith. A type-based approach to program security. *Lecture Notes in Computer Science*, 1214:607–621, 1997.

[Wad71] C. P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, Oxford University, 1971.

[Wad89] Philip Wadler. Theorems for free. In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989.

[WLAG93] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *14th ACM Symposium on Operating Systems Principles*, pages 203–216. ACM, December 1993.

[You89] William D. Young. A mechanically verified code generator. *Journal of Automated Reasoning*, 5(4):493–518, December 1989.

Appendix A

Soundness of Verification Condition Generation

The main result of this appendix is that a valid verification condition for an agent guarantees the safety of all invocations of the agent functions. Because the agent functions might invoke system calls, we also need to assume that the system calls are safe. This is stated formally as the soundness Theorem 4.4 on page 74 and restated below:

Theorem A.1 (Soundness of VCGen) *If all system calls are safe, i.e., $Safe(\Phi^S)$, and if the agent's verification condition is valid, i.e., $\models VC(\Phi^A)$, then all functions in the system are safe, i.e., $Safe(\Phi)$.*

The proof of the soundness theorem is by induction on the length of the execution. At each step (in any reachable state) we show that the execution has either returned from the current invocation or else it can make further progress. During the induction we must propagate the information that the execution was initiated in a state satisfying the precondition and that the verification condition is valid. This is customarily done by means of an *induction hypothesis*. In our case the induction hypothesis for an execution state $<<F, i>, \rho, \mathcal{H}_0 + \rho_0>$ is that there exists a related state of the symbolic evaluator at the same point in function F. Let this state be $SE_{F,\sigma_0,i_0}(i, \sigma, \mathtt{so}, \mathcal{L})$. The key property of the run-time and symbolic states is that they are related, in a complicated sense that is stated formally later in this section. One of the defining issues of the relation is that there exists a substitution τ of the variables occurring in the symbolic execution state σ with values, such that $\tau \circ \sigma = \rho$. Without disclosing yet the formal definition of the invariant let us just say that it is a relation among eleven parameters written as $IH_{F,\sigma_0,i_0,\mathcal{H}_0,\rho_0}(i, \sigma, \mathtt{so}, \mathcal{L}, \tau, \rho)$. Just as for the symbolic evaluation function, the subscripts are constant within a given function's body and are omitted most of the time.

Lemma A.2 (Soundness) *If all system calls are safe, i.e., Safe*(Φ^S)*, and if the agent's verification condition is valid, i.e.,* $\models VC(\Phi^A)$*, then for any function* F *and initial state* $<<F, 0>, \rho_0, \mathcal{H}_0 + \rho_0>$ *that satisfies the precondition, i.e.,* $Pre_F(\rho_0)$ *and whose entire frame fits on the runtime stack, then* $\underline{\textit{assuming}}$ $\underline{\textit{that}}$ $\underline{\textit{all}}$ $\underline{\textit{invoked}}$ $\underline{\textit{agent}}$ $\underline{\textit{functions}}$ $\underline{\textit{are}}$ $\underline{\textit{safe}}$ *we have that the execution does not get stuck and for any reachable state* $\Sigma = <<F, i>, \rho, \mathcal{H}_0 + \rho_0>$ *within the current invocation of* F*, there exists* σ_0*,* i_0*,* σ*,* $\mathtt{so}$*,* $\mathcal{L}$ *and* τ *such that the induction hypothesis holds, i.e.,* $IH_{F,\sigma_0,i_0,\mathcal{H}_0,\rho_0}(i, \sigma, \mathtt{so}, \mathcal{L}, \tau, \rho)$ *and either*

1. $F_i = \mathtt{ret}$ *and* $SafeRET_F(\rho_0, \rho)$*, or*

2. $F_i = \mathtt{call}\ G$ *and the execution is safe by assumption, or*

3. *There exists* $\Sigma' = <<F, i'>, \rho', \mathcal{H}_0 + \rho_0>$ *such that* $\Sigma \to \Sigma'$ *and the induction hypothesis holds in the new state.*

The proof of Theorem 4.4 is by means of the Lemma A.2. Note that the soundness lemma considers only the reachable states that are within the current invocation. If the current invocation makes function calls then the safety property follows from the assumption that system calls are safe and that all invoked agent functions are also safe. There seems to be a problem here in the case of recursive agent functions because the soundness lemma assumes that its own conclusion holds. This can be fixed with a technical trick. We introduce an additional definition of safety involving a parameter k such that only executions that do not lead to more than k nested function calls are considered. The other executions are treated as non-terminating and are thus considered safe. Then, we also change the statement of the soundness lemma to assume that all agent functions are safe up to k nested executions and to conclude that all agent functions are safe up to $k + 1$ nested executions. The base case is for invocations that do not invoke other functions. Thus the modified soundness lemma constitutes both the base case and the inductive case of an inductive argument on k. Thus, the soundness lemma is true for all values of k, which is the kind of safety that we desire. Having argued informally that it is possible to fix the apparent bug in the statement of the soundness lemma, I proceed with it in the simpler form.

The termination part of the soundness theorem follows from the soundness lemma because of Property 3.3 that requires that "`ra`" is a preserved register.

We are now left with proving the Lemma A.2. This lemma is proved by using four other lemmas: one stating that the induction hypothesis is established upon invocation of an agent function (Lemma A.3), one stating that if the induction hypothesis holds and the current instruction is neither a function call nor a return, then the execution can make progress to a state that also satisfies the induction hypothesis (Lemma A.4), one to deal with the function call case stating that executing a function call (for a safe function) to its (potential) completion preserves the induction invariant (Lemma A.5) , and one to deal with the function return case (Lemma A.6). The four lemmas are stated below.

Lemma A.3 (Invocation) *If function F has a valid verification condition, i.e.,* $\models VC(F)$ *and if* ρ_0 *is a safe invocation state for* F*, i.e.,* $Pre_F(\rho_0)$ *and* $Stack(a)$ *for all "a" such that* $\rho_0(\mathtt{sp}) + Arg_F - Local_F \leq a \leq \rho_0(\mathtt{sp}) + Arg_F - 1$*, then for any call history* $\mathcal{H}_0$ *there exist* σ_0*,* τ_0 *such that* $IH_{F,\sigma_0,i_0,\mathcal{H}_0,\rho_0}(0, \sigma_0, Arg_F, \emptyset, \tau_0, \rho_0)$*, where* i_0 *is a new variable.*

Lemma A.4 (Progress) *If* $\models IH_{F,\sigma_0,i_0,\mathcal{H}_0,\rho_0}(i, \sigma, \mathtt{so}, \mathcal{L}, \tau, \rho)$ *and the current instruction is neither a call nor a return instruction then the interpreter can make progress, i.e., there exist* i',σ',$\mathtt{so}'$, $\mathcal{L}'$,τ',ρ' *such that* $<<F, i>, \rho, \mathcal{H}_0 + \rho_0> \rightarrow <<F, i'>, \rho', \mathcal{H}_0 + \rho_0>$ *and also* $\models IH_{F,\sigma_0,i_0,\mathcal{H}_0,\rho_0}(i', \sigma', \mathtt{so}', \mathcal{L}', \tau', \rho')$.

Lemma A.5 (Function call) *If* $\models IH_{F,\sigma_0,i_0,\mathcal{H}_0,\rho_0}(i, \sigma, \mathtt{so}, \mathcal{L}, \tau, \rho)$ *and the current instruction is a function call, i.e.,* $F_i = \mathtt{call}\ G$*, such that* $Safe(G)$*, then the execution can make progress until possibly returning to the state* $<i{+}{+}, \rho', \mathcal{H}_0 + \rho_0>$*. In this case there exist* σ' *and* τ' *such that* $\models IH_{F,\sigma_0,i_0,\mathcal{H}_0,\rho_0}(i{+}{+}, \sigma', \mathtt{so}, \mathcal{L}, \tau', \rho')$.

Lemma A.6 (Function return) *If* $\models IH_{F,\sigma_0,i_0,\mathcal{H}_0,\rho_0}(i, \sigma, \mathtt{so}, \mathcal{L}, \tau, \rho)$ *and the current instruction is a return then the return is safe, i.e.,* $SafeRET_F(\rho_0, \rho)$.

Before we can attempt to prove the four lemmas we have to state formally the definition of the induction hypothesis. This is given in Figure A.1. In the definition of the induction hypothesis and the proofs to follows I use the letter τ to denote substitutions of values in $\mathcal{U}$ for logical variables. The notation $\tau + \tau'$ is used when the domains of the substitutions τ and τ' are disjoint and denotes the extension of the substitution τ with the substitution τ'. Also, during the course of the proof it will be often necessary to use the congruence lemma stated below.

Lemma A.7 (Congruence) *If* P *is a predicate whose free variables are within* $Regs_F$*, and* ρ *and* ρ_0 *are evaluator states and* τ *is a mapping from symbolic registers* $Regs_F$ *to values such that* $\models \tau(r) = S_{\rho_0,Arg_F}(\rho, r)$ *for all* $r \in Regs_F \cap FV(P)$*, then we have that* $\models \tau(P)$ *if and only if* $\models S_{\rho_0,Arg_F}(\rho, P)$.

Proof of the congruence Lemma A.7: The proof of the congruence lemma is an easy induction on the structure of the judgment of the predicate P.

□

Now I proceed with the formal proofs of the lemmas A.3–A.6.

Proof of invocation Lemma A.3: Because the verification condition is valid for F we have from Definition 4.3 that $\models \forall i_0.y_1 \ldots y_k.\sigma_0(\mathtt{Pre}_F) \supset SE_{F,\sigma_0,i_0}(0, \sigma_0, Arg_F, \emptyset)$, where k is the number of registers in $Regs_F = \{r_1, \ldots, r_k\}$, i_0 and $y_1, \ldots, y_k$ are new variables and the initial symbolic state is $\sigma_0 = [r_1 \mapsto y_1, \ldots, r_k \mapsto y_k]$. From the definition of validity for

Definition A.8 (Induction Hypothesis) *Consider a simultaneous execution of the* SAL *interpreter in the state* $<<F, i>, \rho, \mathcal{H}_0 + \rho_0>$ *and of the VCGen symbolic evaluator in the state* $SE_{F,\sigma_0,i_0}(i, \sigma, \mathtt{so}, \mathcal{L})$. *Consider that the current instantiation of logic variables is* τ. *We say that the induction hypothesis holds in this situation, written* $\models IH_{F,\sigma_0,i_0,\mathcal{H}_0,\rho_0}(i, \sigma, \mathtt{so}, \mathcal{L}, \tau, \rho)$, *when all of the following conditions hold:*

IH1. The verification condition is valid, i.e., $\models \tau(SE_{F,\sigma_0,i_0}(i, \sigma, \mathtt{so}, \mathcal{L}))$, *and*

IH2. so *is correct, i.e.,* $0 \leq \mathtt{so} \leq Local_F$ *and* $\rho(\mathtt{sp}) + \mathtt{so} = \rho_0(\mathtt{sp}) + Arg_F$, *and*

IH3. i_0 *is correct, i.e.,* $<F, 0> = \tau(i_0)$, *and*

IH4. σ *and* σ_0 *are correct, i.e., for all* $r \in Regs_F$ *we have* $\models \tau(\sigma(r)) = S_{\rho_0, Arg_F}(\rho, r)$, *and also that* $\models \tau(\sigma_0(r)) = S_{\rho_0, Arg_F}(\rho_0, r)$, *and*

IH5. The stack is preserved, i.e., for all a *such that* $Stack(a)$ *and* $a \geq \rho_0(\mathtt{sp}) + Arg_F$ *we have* $\rho_0(\mathtt{mem})(a) = \rho(\mathtt{mem})(a)$, *and*

IH6. The frame fits in the stack, i.e., $Stack(a)$ *holds for all addresses "a" that satisfy the inequalities* $\rho_0(\mathtt{sp}) + Arg_F - Local_F \leq a \leq \rho_0(\mathtt{sp}) + Arg_F - 1$, *and*

IH7. τ *is correct with respect to* $\mathcal{L}$, *i.e., either*

IH7.1. $\mathcal{L}$ *is empty and for all* $r \in Regs_F$, *we have*

IH7.1.1. $\sigma_0(r) \in Dom(\tau)$, *and*

IH7.1.2. $\models \tau(\sigma_0(r)) = S_{\rho_0, Arg_F}(\rho_0, r)$.

IH7.2. or if $\mathcal{L} = \mathcal{L}_1 + (i, \sigma_1')$ *with* $\sigma_1' = \sigma_1[r_1 \mapsto y_1, \ldots r_k \mapsto y_k]$, *then*

IH7.2.1. $F_i = \mathtt{inv}\ P, n, \{r_1, \ldots, r_k\}$, *and*

IH7.2.2. y_j *are new variables, i.e.,* $y_j \notin (Dom(\tau_1) \cup FV(\mathcal{L}_1) \cup FV(\sigma_1))$, *and*

IH7.2.3. $\tau = \tau_1 + [y_1 \mapsto v_1, \ldots, y_k \mapsto v_k]$, *and*

IH7.2.4. $\models \tau_1(\sigma_1(P))$, *and*

IH7.2.5. $\models \tau_1(\forall y_1 \ldots y_k.\sigma_1'(P) \supset SE_{F,\sigma_0,i_0}(i{+\!+}, \sigma', n, \mathcal{L})$, *and*

IH7.2.6. τ_1 *is correct with respect to* $\mathcal{L}_1$, *as defined by IH7.*

Figure A.1: The inductive invariant that is used for proving the soundness of verification-condition generation.

universal quantification we further have that $\models \tau_0(\sigma_0(\texttt{Pre}_F) \supset SE_{F,\sigma_0,i_0}(0, \sigma_0, Arg_F, \emptyset))$, where $\tau_0 = [i_0 \mapsto <F,0>, y_1 \mapsto S_{\rho_0,Arg_F}(\rho_0, r_1), \ldots, y_k \mapsto S_{\rho_0,Arg_F}(\rho_0, r_k)]$. We want to prove that $\models IH_{F,\sigma_0,i_0,\mathcal{H}_0,\rho_0}(0, \sigma_0, Arg_F, \emptyset, \tau_0, \rho_0)$. Of all of the clauses of the induction hypothesis the only one that is not trivial is IH1. This clause is proved by first inferring that $\models (\tau_0 \circ \sigma_0)(\texttt{Pre}_F)$, which holds because of clause IH4 and of the correctness of preconditions (Property 4.2 on page 68) together with the congruence lemma. Note that clause IH7 is proved in this case using option IH7.1.

□

PROOF OF THE PROGRESS LEMMA A.4: The proof is by case analysis on the current instruction, or more precisely on the last case used for symbolic evaluation. We start with the most difficult cases, those for the loop invariants.

Case: The last rule used during the symbolic evaluation is the rule for a loop invariant "$\texttt{inv}\ P, n, \{r_1, \ldots r_k\}$" that is encountered for the first time. This is an annotation and hence there are no safety checks and progress is guaranteed to the state $<i{+}{+}, \rho, \mathcal{H}_0{+}\rho_0>$. We have to prove the invariant:

$$IH_{F,\sigma_0,i_0,\mathcal{H}_0,\rho_0}(i{+}{+}, \sigma', \texttt{so}, \mathcal{L} + (i, \sigma'), \tau', \rho) \tag{A.9}$$

$$\begin{aligned} \textbf{where}:& \\ & y_1 \ldots y_k \text{ are new variables} \\ & \tau' = \tau + [y_1 \mapsto S(\rho, r_1), \ldots, y_k \mapsto S(\rho, r_k)] \\ & \sigma' = \sigma[r_1 \mapsto y_1, \ldots r_k \mapsto y_k] \end{aligned}$$

The clauses IH2, IH5 and IH6 follow immediately from their induction hypothesis counterparts because the state of the registers does not change. Clause IH3 also follows from the induction hypothesis because $\tau'(i_0) = \tau(i_0)$.

Because of the way τ' is defined and because of clause IH4 of the induction hypothesis we can verify that the clause IH4 of Equation A.9 holds:

$$\models \tau'(\sigma'(r)) = S(\rho, r) \qquad \text{for all } r \in Regs_F \tag{A.10}$$

In order to prove IH1 we notice that from clause IH1 of the induction hypothesis we have:

$$\models \tau(\sigma(P)) \tag{A.11}$$

$$\models \tau'(\sigma'(P)) \supset \tau'(SE(i{+}{+}, \sigma', \texttt{so}, \mathcal{L} + (i, \sigma'))) \tag{A.12}$$

From Equation A.11 and clause IH4 of the induction hypothesis, by using the congruence lemma we deduce that $\models S(\rho, P)$. Using again the congruence lemma but this time with

Equation A.10 we deduce that $\models \tau'(\sigma'(P))$ and hence from the definition of validity for implication we conclude that clause IH1 holds.

The last step in the proof of this case is to prove clause IH7.2. The subclauses IH7.2.1–IH7.2.3 are true by construction, and IH7.2.6 follows from clause IH7.2.6 of the induction hypothesis. Clause IH7.2.5 follows from clause IH1 of the induction hypothesis with the observation that $n = \texttt{so}$ as verified by VCGen. Finally, clause IH7.2.4 is Equation A.11.

Case: The next case that we consider is that of a loop invariant "$\texttt{inv}\ P, n, \{r_1, \ldots, r_k\}$" that has been encountered before ($i \in Dom(\mathcal{L})$). In this case progress is guaranteed and the next execution state is $<i{+}{+}, \rho, \mathcal{H}_0 + \rho_0>$. The trick is to find the corresponding state for the symbolic evaluator. It is here where we need the clause IH7.2 of the induction hypothesis. Because of that clause and because $i \in Dom(\mathcal{L})$ we know that $\mathcal{L} = \mathcal{L}_1 + (i, \sigma') + \cdots$ and $\tau = \tau_1 + [y_1 \mapsto v_1, \ldots, y_k \mapsto v_k] + \cdots$, where $\sigma' = \sigma_1[r_1 \mapsto y_1, \ldots, r_k \mapsto y_k]$, and y_j are variables that do not occur in $\mathcal{L}_1$ or in σ_1. We also know from clause IH7.2 of the induction hypothesis that

$$\models \tau_1(\sigma_1(P)) \tag{A.13}$$

$$\models \tau_1(\forall y_1 \ldots y_k.\sigma'(P) \supset SE_{F,\sigma_0,i_0}(i{+}{+}, \sigma', n, \mathcal{L}_1 + (i, \sigma')) \tag{A.14}$$

This suggests that we should prove the induction hypothesis stated as follows:

$$IH_{F,\sigma_0,i_0,\mathcal{H}_0,\rho_0}(i{+}{+}, \sigma', \texttt{so}, \mathcal{L} + (i, \sigma'), \tau', \rho) \tag{A.15}$$

where $\tau' = \tau_1 + [y_1 \mapsto S(\rho, r_1), \ldots, y_k \mapsto S(\rho, r_k)]$. We first notice that τ' was chosen such that $\models \tau'(\sigma'(r) = S(\rho, r)$ for all $r \in \{r_1, \ldots, r_k\}$. For the registers that are not modified by the loop, we know from clause IH1 of the induction hypothesis that $\models \tau(\sigma(r)) = \tau(\sigma'(r)$. (This is because of the $\texttt{CheckEq}$ part of the verification condition with $\mathcal{L}(i) = \sigma'$.) Then, because of the clause IH4 of the induction hypothesis we know that $\models \tau(\sigma'(r)) = S(\rho, r)$. Because r is not modified by the loop and because τ is an extension of τ_1 we have the sequence of equalities $\tau'(\sigma'(r)) = \tau'(\sigma_1(r)) = \tau_1(\sigma_1(r)) = \tau(\sigma_1(r)) = \tau(\sigma'(r)$, which concludes the proof of clause IH4 of Equation A.15:

$$\models \tau'(\sigma'(r) = S(\rho, r) \;\text{ for all } r \in Regs_F \tag{A.16}$$

From clause IH1 of the induction hypothesis we know that $\models \tau(\sigma(P))$. From Equation A.16 and clause IH4 of the induction hypothesis, we get by using the congruence lemma that $\models \tau'(\sigma'(P))$. This, used in conjunction with the definition of the validity for implication and universal quantification in Equation A.14, leads to the conclusion that clause IH1 of the Equation A.15 holds. Clauses IH2, IH3, IH5 and IH6 hold because the register state has not changed. Finally, note that clause IH7.2 holds because of clause IH7.2 of the induction hypothesis.

This concludes the case of loop invariants. Note that in the other cases the loop map does not change and in fact it is not used at all. This implies that the substitution τ does not change. Because of this the clauses IH3, IH6 and IH7 are trivial to prove from the induction hypothesis directly.

Case: If the last rule used for symbolic evaluation is the register move rule then we must prove that $IH(i{+\!+}, \sigma[r \mapsto \sigma(r')], \texttt{so}, \mathcal{L}, \tau, \rho[r \mapsto \rho(r')])$. The only clause of interest in this case is IH4, in which case we have to prove that $\models \tau(\sigma(r')) = \rho(r')$, which follows from clause IH4 of the induction hypothesis. The case of register initialization is similar.

Case: If the last rule used for symbolic evaluation is the expression operand rule then we must first prove that $\mathit{SafeEOP}(\rho(r'), \rho(r''))$ so that we know that the execution makes progress. But from clause IH1 of the induction hypothesis and the definition of validity for conjunction we have that $\models \texttt{safeeop}(\tau(\sigma(r')), \tau(\sigma(r'')))$. Because of clause IH4 we can use the congruence lemma to infer that $\models \texttt{safeeop}(\rho(r'), \rho(r''))$ and hence, because of the definition of validity for "`safeeop`" we know that $\mathit{SafeEOP}(\rho(r'), \rho(r''))$.

To conclude this case of the proof we must prove that the induction invariant holds in the new state $IH(i{+\!+}, \sigma[r \mapsto \texttt{eop}(\sigma(r'), \sigma(r''))], \texttt{so}, \mathcal{L}, \tau, \rho[r \mapsto \underline{\texttt{EOP}}(\rho(r'), \rho(r''))])$. Clause IH1 follows from clause IH1 of the induction hypothesis. The only other interesting clause in this case is IH4, in which case we must prove that $\models \texttt{eop}(\tau(\sigma(r')), \tau(\sigma(r''))) = \underline{\texttt{EOP}}(\rho(r'), \rho(r''))$. This follows from the definition of valuation for the expression constructor "`eop`" and from clause IH4 of the induction hypothesis by using the congruence lemma.

Case: If the last rule used for symbolic evaluation is the conditional branch rule then progress is guaranteed if $\mathit{SafeCOP}(\rho(r))$ holds. This is proved as in the case of expression operators from the clauses IH1 and IH4 of the induction hypothesis, the congruence lemma and the definition of validity for the predicate constructor "`safecop`". Once we know that the execution can make progress we have to consider two cases, depending on the outcome of the comparison.

If the comparison is successful then the next state is $<n + i{+\!+}, \rho, \mathcal{H}_0 + \rho_0>$ and thus the invariant to prove is $IH(i{+\!+}, \sigma, \texttt{so}, \mathcal{L}, \tau, \rho)$. Only clause IH1 is interesting because so few parameters of the induction hypothesis are changed. Clause IH1 follows from clause IH1 of the induction hypothesis if we can prove that $\models \texttt{cop}(\tau(\sigma(r)))$. This follows immediately from the definition of validity of "`cop`" and clause IH4 of the induction hypothesis, because we know that the control flow test "$\underline{\texttt{COP}}(\rho(r))$" succeeds. The fall-through case of the branch is similar.

Case: In the case of a memory write instruction "$M[r'] \mapsto r$" we have to verify first that the safety condition $\mathit{SafeWr}(\rho(\texttt{mem}), \rho(r'), \rho(r))$ holds so that the execution can make progress. This follows from the clauses IH1 and IH4 of the induction hypothesis by using the congruence lemma and the definition of validity for the "safewr" predicate constructor. Next we have to prove that the following form of the induction hypothesis holds $IH(i{+}{+}, \sigma[\mathtt{mem} \mapsto \mathtt{upd}(\sigma(\mathtt{mem}), \sigma(r'), \sigma(r))], \mathtt{so}, \mathcal{L}, \tau, \rho[\mathtt{mem} \mapsto \rho(\mathtt{mem})[\rho(r') \mapsto \rho(r)]])$. Clause IH1 follows from clause IH1 of the induction hypothesis. In this case the memory changes so we must verify clause IH5. Because of Property 3.1 we know that $\neg Stack(\rho(r'))$ and hence clause IH5 follows directly from clause IH5 of the hypothesis.

To prove clause IH4 we must prove that the following equality holds

$$\models \mathtt{upd}(\tau(\sigma(\mathtt{mem})), \tau(\sigma(r')), \tau(\sigma(r))) = \rho(\mathtt{mem})[\rho(r') \mapsto \rho(r)]$$

This follows directly from the definition of valuation from the "upd" constructor. To finish the proof of clause IH4 we must also verify that the memory that contains the values of the local pseudo-registers has not change. This follows from the fact that all such memory locations are on the stack (clause IH6 of the induction hypothesis) and the stack is not changed by generic memory writes. The case of memory reads is very similar to that of expression operators.

Case: In the case of a stack pointer advance instruction we must prove that $Stack(n + \rho(\mathtt{sp}))$. We know from clause IH2 that $\rho(\mathtt{sp}) + n = \rho_0(\mathtt{sp}) + Arg_F - \mathtt{so} + n$ and we also know from the check performed by VCGen that the following inequalities hold $\rho_0(\mathtt{sp}) + Arg_F - Local_F \leq \rho(\mathtt{sp}) + n \leq \rho_0(\mathtt{sp}) + Arg_F - 1$. Now we can see that the safety condition follows from clause IH6 of the induction hypothesis. Once progress is established we have to verify that the induction hypothesis holds $IH(i{+}{+}, \sigma, \mathtt{so} - n, \mathcal{L}, \tau, \rho[\mathtt{sp} \mapsto \rho(\mathtt{sp}) + n])$. All of the clauses follow directly from the induction hypothesis, except for IH2, in which case simple arithmetic is sufficient.

Case: In the case of a read from the stack, we argue the progress just as for the stack pointer advance instruction. Then, we have to prove that the induction hypothesis holds $IH(i{+}{+}, \sigma[r \mapsto \sigma(l_{\mathtt{so}-n})], \mathtt{so}, \mathcal{L}, \tau, \rho[r \mapsto \rho(\mathtt{mem})(\rho(\mathtt{sp}) + n)])$. The only clause that does not follow directly from the induction hypothesis is IH4, in which case we have to prove that $\models \tau(\sigma(l_{\mathtt{so}-n})) = \rho(\mathtt{mem})(\rho(\mathtt{sp}) + n)$. Note that $\rho(\mathtt{mem})(\rho(\mathtt{sp}) + n) = \rho(\mathtt{mem})(\rho_0(\mathtt{sp}) + Arg_F - (\mathtt{so} - n)) = S_{\rho_0, Arg_F}(\rho, l_{\mathtt{so}-n})$ and thus the desired conclusion follows from clause IH4 of the induction hypothesis. The check that VCGen performs ensures that $l_{\mathtt{so}-n}$ is a valid local pseudo-register in the frame of F.

Case: In the case of a write to the stack, progress and clause IH4 are argued in a manner similar to the case of reads from the stack. In this case we have the extra obligation to verify clause IH5. This is easy to verify because the address written to is smaller than $\rho_0(\mathtt{sp}) + Arg_F$ due to the checks performed by VCGen and because of clause IH2 of the induction hypothesis.

This concludes the proof of the progress lemma.

$\square$

PROOF OF THE FUNCTION CALL LEMMA A.5: From the induction hypothesis in the case of a function call to G we have that $\models \tau(\sigma_1^G(\mathtt{Pre}_G))$. The safety requirement for the call instruction is $\mathit{Pre}_G(\rho)$. This is proved easily by using Property 4.2 and the congruence lemma once we prove that $\models \tau(\sigma_1^G(r)) = S_{\rho,\mathit{Arg}_G}(\rho, r)$ for all $r \in \mathit{Regs}_G \cap \mathit{FV}(\mathtt{Pre}_G)$. To prove this property I am doing a case analysis on the register r, as follows:

$$\models \tau(\sigma_1^G(r)) = S_{\rho,\mathit{Arg}_G}(\rho, r) \quad \text{if} \quad r \in \{\mathbf{r}_1, \ldots, \mathbf{r}_R, \mathtt{ra}, \mathtt{mem}\} \tag{A.17}$$

In this case $\sigma_1^G(r) = \sigma^F(r)$ (see Figure 4.7 for the definition of $\mathtt{CopyIn}$) and $S_{\rho,\mathit{Arg}_G}(\rho, r) = \rho(r)$ and Equation A.17 follows from clause IH4 of the induction hypothesis. The other possible case for r is:

$$\models \tau(\sigma_1^G(r)) = S_{\rho,\mathit{Arg}_G}(\rho, r) \quad \text{if} \quad r \equiv l_i \in \{l_1, \ldots, l_{\mathit{Arg}_G}\} \tag{A.18}$$

In this case $\sigma_1^G(r) = \sigma^F(l_{\mathtt{so}-\mathit{Arg}_G+i})$ and we have from clause IH4 of the induction hypothesis that the sequence of equalities $\models \tau(\sigma^F(l_{\mathtt{so}-\mathit{Arg}_G+i})) = S_{\rho_0,\mathit{Arg}_F}(\rho, l_{\mathtt{so}-\mathit{Arg}_G+i}) = S_{\rho,\mathit{Arg}_G}(\rho, l_i)$ holds. (The latter equality follows from the definition of substitution and clause IH2). Note that $l_{\mathtt{so}-\mathit{Arg}_G+i}$ is a valid local registers in Regs_F because of clause IH2 and of the check on "$\mathtt{so}$" that VCGen performs for a function call.

Once we proved that $\mathit{Pre}_G(\rho)$ holds, we are guaranteed that the execution of the interpreter makes progress. If the stack overflow check fails then the execution never returns, which is one of the possible benign outcomes in this lemma. If, however, the stack overflow check succeeds, then we know from the continuity of the stack (Property 3.2) that the entire stack frame of G fits on the stack. Hence from the assumption that G is safe (as stated in Definition 3.5) we know that the execution makes progress and if it returns the current state is $<\rho(\mathtt{ra}), \rho', \mathcal{H}_0 + \rho_0>$ such that:

$$\mathit{SafeRET}_F(\rho, \rho') \tag{A.19}$$

We know from clause IH4 that $\models \tau(\sigma^F(\mathtt{ra})) = \rho(\mathtt{ra})$. Because VCGen checks the value of the return address register at the time of the call (see Figure 4.6) we know that $\tau(\sigma^F(\mathtt{ra})) = \mathtt{offset}(\tau(i_0), i{+}{+})$ hence by clause IH3 we know that the return point is the instruction following the call $\rho(\mathtt{ra}) = <F, i{+}{+}>$.

To complete the proof of this lemma we need to show that the induction hypothesis holds in the state following the return from the function for some symbolic state and some substitution of variables with values. Let the symbolic state be σ_2^F as defined in Figure 4.6 and let $\tau_2 = \tau + [y_1 \mapsto S_{\rho,\mathit{Arg}_G}(\rho', r_1), \ldots, y_k \mapsto S_{\rho,\mathit{Arg}_G}(\rho', r_k), z_{\mathtt{so}+1} \mapsto S_{\rho_0,\mathit{Arg}_F}(\rho', l_{\mathtt{so}+1}), \ldots z_{\mathit{Local}_F} \mapsto S_{\rho_0,\mathit{Arg}_F}(\rho', l_{\mathit{Local}_F})]$. We have to prove that:

$$\mathit{IH}_{F,\sigma_0,i_0,\mathcal{H}_0,\rho_0}(i{+}{+}, \sigma_2^F, \mathtt{so}, \mathcal{L}, \tau_2, \rho') \tag{A.20}$$

I start with proving the clause IH4. We have to show that $\models \tau_2(\sigma_2^F(r)) = S_{\rho_0,Arg_F}(\rho', r)$ and also that $\models \tau_2(\sigma_0(r)) = S_{\rho_0,Arg_F}(\rho_0, r)$ for all registers $r \in Regs_F$. The second part of the clause follows immediately from clause IH4 of the induction hypothesis once we realize that for all such registers $\tau_2(\sigma_0(r)) = \tau(\sigma_0(r)$. This is because all variables y_i and z_j are new. The first part of the clause is proved by a case analysis on the register r. We distinguish four cases labelled below as A.21–A.24, as follows:

$$\models \tau_2(\sigma_2^F(r)) = S_{\rho_0,Arg_F}(\rho', r) \quad \text{if} \quad r \in \{\mathbf{r}_1, \ldots, \mathbf{r}_R, \mathtt{ra}, \mathtt{mem}\} \tag{A.21}$$

In this case $\sigma_2^F(r) = \sigma_2^G(r)$ (by definition of `CopyOut` from Figure 4.7). We now consider two subcases, depending on whether $r \in CS_G$ or not. If r is saved by G, then $\sigma_2^G(r) = \sigma_1^G(r) = \sigma^F(r)$ and from Equation A.19 we get that $S_{\rho_0,Arg_F}(\rho', r) = \rho(r)$. The desired conclusion follows from clause IH4 if we note that $\tau_2(\sigma^F(r)) = \tau(\sigma^F(r))$ because τ_2 is an extension of τ. In the case when r is not saved then $\tau_2(\sigma_2^F(r)) = \tau_2(\sigma_2^G(r)) = \tau_2(y_j) = S_{\rho,Arg_G}(\rho', r) = S_{\rho_0,Arg_F}(\rho', r)$.

The next case is when r is a local register that sits above G's stack frame:

$$\models \tau_2(\sigma_2^F(r)) = S_{\rho_0,Arg_F}(\rho', r) \quad \text{if} \quad r \equiv l_i \in \{l_1, \ldots, l_{\mathtt{so}-Arg_G}\} \tag{A.22}$$

In this case $\models \tau_2(\sigma_2^F(r)) = \tau(\sigma^F(r)) = S_{\rho_0,Arg_F}(\rho, r)$ by the definition of `CopyOut` and clause IH4. By definition of substitution $S_{\rho_0,Arg_F}(\rho, r) = \rho(\mathtt{mem})(a)$ where $a = \rho_0(\mathtt{sp}) + Arg_F - i$. Note that $a \geq \rho(\mathtt{sp}) + Arg_G$ and therefore by Equation A.19 we have that $S_{\rho_0,Arg_F}(\rho, r) = S_{\rho_0,Arg_F}(\rho', r)$, which concludes this case of the proof.

The next case is when r is a local register that is an argument of G:

$$\models \tau_2(\sigma_2^F(r)) = S_{\rho_0,Arg_F}(\rho', r) \quad \text{if} \quad r \equiv l_i \in \{l_{\mathtt{so}-Arg_G+1}, \ldots, l_{\mathtt{so}}\} \tag{A.23}$$

In this case we have that $\sigma_2^F(r) = \sigma_2^G(l_{i-\mathtt{so}+Arg_G})$ and we have to consider again two subcases depending whether $l_{i-\mathtt{so}+Arg_G}$ is a callee-save of G or not. If it is saved then $\sigma_2^G(l_{i-\mathtt{so}+Arg_G}) = \sigma_1^G(l_{i-\mathtt{so}+Arg_G}) = \sigma^F(l_i)$ (by definition of `CopyIn`). Also, $S_{\rho_0,Arg_F}(\rho', l_i) = S_{\rho,Arg_G}(\rho', l_{i-\mathtt{so}+Arg_G}) = S_{\rho,Arg_G}(\rho, l_{i-\mathtt{so}+Arg_G}) = S_{\rho_0,Arg_F}(\rho, l_i)$. (We used two times clause IH2 of the induction hypothesis and the callee-save condition of Equation A.19.) From here we get the desired conclusion by using clause IH4 of the induction hypothesis if we observe that $\tau_2(\sigma^F(l_i)) = \tau(\sigma^F(l_i))$. The second subcase is when $l_{i-\mathtt{so}+Arg_G}$ is not saved by G. Then $\tau_2(\sigma_2^G(l_{i-\mathtt{so}+Arg_G})) = \tau_2(y_j) = S_{\rho,Arg_G}(\rho', l_{i-\mathtt{so}+Arg_G}) = S_{\rho_0,Arg_F}(\rho', l_i)$, which concludes this case.

The last case in the proof of clause IH4 is when r is a local register sitting below the arguments of G:

$$\models \tau_2(\sigma_2^F(r)) = S_{\rho_0,Arg_F}(\rho', r) \quad \text{if} \quad r \equiv l_i \in \{l_{\mathtt{so}+1}, \ldots, l_{Local_F}\} \tag{A.24}$$

In this case $\tau_2(\sigma_2^F(l_i)) = \tau_2(z_i) = S_{\rho_0,Arg_F}(\rho', l_i)$, which concludes this case and the whole proof of clause IH4.

Next we prove clause IH1 of Equation A.20. From clause IH1 of the induction hypothesis we have that $\models \tau_2(\sigma_2^G(\mathtt{Post}_G) \supset SE_{F,\sigma_0,i_0}(i{+}{+}, \sigma_2^F, \mathtt{so}, \mathcal{L}))$ and we need to prove that $\models \tau_2(SE_{F,\sigma_0,i_0}(i{+}{+}, \sigma_2^F, \mathtt{so}, \mathcal{L}))$. For this it is sufficient to prove that $\models \tau_2(\sigma_2^G(\mathtt{Post}_G))$. From Equation A.19 we have that $Post_G(\rho')$ and we could get our conclusion by using Property 4.2 if we knew that $\models \tau_2(\sigma_2^G(r)) = S_{\rho,Arg_G}(\rho', r)$ for all $r \in Regs_G \cap FV(\mathtt{Post}_G)$. We distinguish two cases, as follows:

$$\models \tau_2(\sigma_2^G(r)) = S_{\rho,Arg_G}(\rho', r) \quad \text{if} \quad r \in \{\mathbf{r}_1, \ldots, \mathbf{r}_R, \mathtt{ra}, \mathtt{mem}\}$$

In this case $\sigma_2^G(r) = \sigma_2^F(r)$ and $S_{\rho,Arg_G}(\rho', r) = S_{\rho_0,Arg_F}(\rho', r)$ and hence we can reuse the proof of Equation A.21. The other case is when r is a argument local register (there are no other cases because of Property 4.1).

$$\models \tau_2(\sigma_2^G(r)) = S_{\rho,Arg_G}(rho', r) \quad \text{if} \quad r \equiv l_i \in \{l_1, \ldots, l_{Arg_G}\}$$

In this case we can reuse the proof of Equation A.23 after we convert the local names from the frame of G to that of F. This concludes the proof of clause IH1.

The proof of clause IH2 follows directly from Equation A.19 and clause IH2 of the induction hypothesis.

The proof of clause IH3 follows directly from clause IH3 of the induction hypothesis.

The proof of clause IH5 follows from Equation A.19 and the clause IH5 of the induction hypothesis because everything that is above the frame of F is also above the frame of G and thus could not have been modified by the call to G. This is ensured by the check of "so" performed by VCGen.

The proof of clause IH6 follows directly from clause IH6 of the induction hypothesis.

The proof of clause IH7 follows from clause IH7 of the induction hypothesis by noticing that the substitution τ_2 is an extension of τ with only new variables and the loop map does not change. This concludes the proof of the function call lemma.

□

Proof of the function return Lemma A.6: Note that Figure 3.2 defines a return to be safe if the history is not empty and if $SafeRET_F(\rho_0, \rho)$. The former condition is trivially true because of the induction hypothesis. The proof of the latter safety condition has four parts as requested by the Definition 3.4, as follows:

1. From clause IH1 of the induction hypothesis we deduce that $\tau(\sigma(\mathtt{Post}_F))$. From this and from clause IH4 of the induction hypothesis it follows that $\models S(\rho, \mathtt{Post}_F)$. Hence

from the correctness of postcondition specifications we have that $Post_F(\rho)$, which is what we need to prove.

2. From clause IH1 of the induction hypothesis we deduce that $\models \tau(\sigma_0(r)) = \tau(\sigma(r))$ for all $r \in CS_F$. Now we can use clause IH4 for σ and σ_0 to deduce that $S(\rho_0, r) = S(\rho_0, r)$ for all callee-save registers.

3. $\rho(\mathtt{sp}) = \rho_0(\mathtt{sp})$ follows immediately from the IH2 clause of the induction hypothesis given that $\mathtt{so} = Arg_F$ (as checked by the verification-condition generator).

4. The preservation of the stack follows immediately from the IH5 clause of the induction hypothesis.

$\square$

Appendix B

Soundness of LF$_i$ Proof Checking

In this appendix I prove that the reconstruction algorithm presented in Section 5.3 is adequate for checking the validity of proofs. This proof of correctness is not just an exercise in type-theoretic proofs. Besides the obvious purpose of ensuring the correctness of the type reconstruction algorithm—which is not obvious by inspection only—the correctness proof also constitutes a thorough analysis of the strengths and limitations of the algorithm.

The limitations of the algorithm are imposed mostly by the side-conditions present in some of the rules, most notably the collection of explicit parameters, the typing of abstractions and the instantiation rules. The correctness proof can serve as a reference documenting why each of these side-conditions is needed. In many practical cases the reconstruction algorithm is used in circumstances when some of the side-conditions do not need to be enforced. These circumstances can be discovered only after a deep understanding of the purpose of the side conditions in the correctness proof.

The proof is given in two stages. In the first stage I prove that the reconstruction succeeds only when there is a LF$_i$ typing derivation of the subject implicit proof. This is stated as Theorem 5.5 on page 97 and restated below:

Theorem B.1 (Correctness of proof reconstruction) *If M is an LF$_i$ object such that* $\mathrm{UVF}(M)$ *and* $\cdot \overset{r}{\vdash} M : \mathtt{pf}\ \ulcorner P \urcorner \Rightarrow \cdot$ *then* $\cdot \overset{i}{\vdash} M : \mathtt{pf}\ \ulcorner P \urcorner$.

The second stage of the proof is to show that whenever there is a LF$_i$ typing for an implicit proof, then there exists a fully-explicit well-typed reconstruction of the proof. This is Theorem 5.4 on page 92 restated below:

Theorem B.2 Soundness of LF$_i$ typing *If* $\Gamma \overset{i}{\vdash} M : A$ *and* $\mathrm{PF}(\Gamma)$, $\mathrm{PF}(A)$, *then there exists* M' *such that* $M \nearrow M'$ *and* $\Gamma \overset{LF}{\vdash} M' : A$.

I start with the proof of Theorem B.1 and then continue on 257 with the proof of Theorem B.2. But first, I must introduce additional notation to the notation that is introduced in Section 5.3.

Notation and Conventions

I write $\Psi(\Gamma)$ to denote the result of applying the substitution Ψ to a type environment Γ. The resulting type environment is defined on $Dom(\Psi(\Gamma)) = Dom(\Gamma) \backslash Dom(\Psi)$, as follows:

$$\Psi(\Gamma)(x) = \Psi(\Gamma(x)) \qquad\qquad \forall x \in Dom(\Gamma) \backslash Dom(\Psi)$$

I write $\Gamma \vdash \Psi$ to denote that the substitution Ψ is well-typed according to a type environment Γ, using the LF$_i$ typing rules, as follows:

$$\Gamma \vdash \Psi \ \text{ iff } \ \forall u \in Dom(\Psi) \ \text{ then } \ u \in Dom(\Gamma) \ \text{ and } \ \Psi(\Gamma) \overset{i}{\vdash} \Psi(u) : \Psi(\Gamma(u))$$

In order to simplify the presentation of the correctness proofs in this section I make the convention that *all of the types involved are placeholder-free.* This applies to types given as part of the theorem hypotheses or types mentioned in the conclusion of helper lemmas and theorems. Whenever new types are created, the proof shall verify this property even though it is not expressed explicitly in the statements of the theorems. Because of this convention, it is not necessary to check the PF side-conditions in the LF$_i$ typing judgments involved in the following proofs. In order to keep the presentation focused I have separated a number of helper lemmas in Section B.3 at the end of this appendix.

B.1 Correctness of LF$_i$ Type Reconstruction

The type reconstruction algorithm is expressed as five mutually recursive judgments. Not surprisingly the correctness proof of the reconstruction algorithm consists of five mutually dependent correctness proofs for the constituent judgments. These proofs are each by induction on the structure of the corresponding judgment. Occasionally, a proof invokes the theorem for a related judgment, in the same way as the corresponding judgment invokes a related judgment. For the five proofs at hand the chain of theorem invocations is circular but the derivations involved are structurally smaller, therefore the induction is well-founded.

Correctness of the Type Reconstruction Judgment

First is the correctness theorem for the main reconstruction judgment. We cannot prove the Theorem B.1 directly. Instead we have to strengthen the statement of the theorem so that the induction succeeds. In particular we have to allow for arbitrary typing environments Γ and types A, possibly containing unification variables but no placeholders. The correctness theorem is stated formally bellow:

Theorem B.3 (Correctness of the Type Reconstruction) *If Γ and A are a type environment and a type respectively such that $\mathrm{PF}(\Gamma)$ and $\mathrm{PF}(A)$ and $\Gamma \overset{i}{\vdash} A : \mathtt{Type}$ and M is a term such that $\mathrm{UVF}(M)$ and if $\Gamma \overset{r}{\vdash} M : A \Rightarrow \Psi$ then*

- $\Gamma \vdash \Psi$*, and*

- PF(Ψ), *and*
- $\Psi(\Gamma) \overset{i}{\vdash} M : \Psi(A)$.

From the Theorem B.3 we can immediately prove Theorem B.1 for the empty type environment and the empty substitution, if we note that PF($\texttt{pf} \ulcorner P \urcorner$) by the definition of the representation function.

Proof of Theorem B.3: The proof is by induction on the structure of the derivation $\mathcal{D} :: \Gamma \overset{r}{\vdash} M : A \Rightarrow \Psi$. There are two cases, depending on the last rule used in $\mathcal{D}$.

Case: If M is an abstraction:

$$\mathcal{D} = \cfrac{\begin{array}{c}\mathcal{D}_1 \\ \Gamma, x : A \overset{r}{\vdash} M : B \Rightarrow \cdot\end{array}}{\Gamma \overset{r}{\vdash} \lambda x.M : \Pi x\!:\!A.B \Rightarrow \cdot}$$

It is obvious that the empty substitution is well-typed and placeholder-free. Because PF(Γ) and PF($\Pi x : A.B$), it follows that we can apply the induction hypothesis on $\mathcal{D}_1$ and infer that $\Gamma, x : A \overset{i}{\vdash} M : B$. Therefore we can also infer that $\Gamma \overset{i}{\vdash} \lambda x.M : \Pi x\!:\!A.B$, which is the desired conclusion.

Case: If M is not an abstraction:

$$\mathcal{D} = \cfrac{\begin{array}{c}\mathcal{D}_1 \\ \Gamma \overset{r}{\vdash} M \Rightarrow (\Delta \,;\, C \,;\, B)\end{array} \qquad \begin{array}{c}\mathcal{D}_2 \\ \Gamma, \Delta \overset{r}{\vdash} C, A \approx_a B \Rightarrow \Psi\end{array} \qquad Dom(\Delta) \subseteq Dom(\Psi)}{\Gamma \overset{r}{\vdash} M : A \Rightarrow \Psi}$$

We follow the sequence of deduction steps shown below:

1. $\Gamma \overset{i}{\vdash} A : \texttt{Type}$ (hypothesis),
2. PF(Γ) and PF(A) and UVF(M) (hypothesis),
3. Using Theorem B.6 on page 250 on $\mathcal{D}_2$ we infer that
4. For all $N : A'$ from C there exist Ψ_1 and Ψ_2 such that $\Psi_1(\Gamma, \Delta) \overset{i}{\vdash} \Psi_1(N) : \Psi_1(A') \Rightarrow \Psi_2$
5. With 2, 4 we apply Theorem B.7 on $\mathcal{D}_1$ (with $\Delta' = \emptyset$) and infer that
6. $\Gamma, \Delta \overset{i}{\vdash} B : \texttt{Type}$, and
7. PF(Δ), and
8. PF(B), and
9. For all $N : A'$ in C we have PF(A') and UVF(N) and $\Gamma, \Delta \overset{i}{\vdash} A' : \texttt{Type}$
10. From 1 we infer that $\Gamma, \Delta \overset{i}{\vdash} A : \texttt{Type}$.

11. With 2, 6, 7, 8, 9 and 10 we apply Theorem B.8 on $\mathcal{D}_2$ and infer that
12. $\Gamma, \Delta \vdash \Psi$, and
13. $\Psi(A) \equiv_\beta \Psi(B)$, and
14. $\mathrm{PF}(\Psi)$, and
15. For all $N : A'$ in C we have $\Psi(\Gamma, \Delta) \overset{i}{\vdash} N : \Psi(A')$. Also using Lemma B.22 with $Dom(\Delta) \subseteq Dom(\Psi)$ we infer that $\Psi(\Gamma) \overset{i}{\vdash} N : \Psi(A')$.
16. With 9, 12, 14, 15 we apply Theorem B.9 and infer that
17. $\Psi(\Gamma) \overset{i}{\vdash} M : \Psi(B)$
18. Because all unification variables from Δ are new, they cannot occur in Γ or in B. Therefore $\Psi(Dom(\Gamma)) = \Psi\big|_{Dom(\Gamma)}(\Gamma)$ and $\Psi(B) = \Psi\big|_{Dom(\Gamma)}(B)$. Therefore we get one of the desired conclusions.
19. With 12 and using Lemma B.22 we prove that $\Gamma \vdash \Psi\big|_{Dom(\Gamma)}$. Also from 14 we can easily prove that $\mathrm{PF}(\Psi\big|_{Dom(\Gamma)})$.

$\square$

Correctness of Unification

The most important judgments of the type reconstruction algorithm presented in the previous section are the unification judgments. Their properties are crucial for the correctness of the reconstruction and their implementation determine the performance of the reconstruction. The key property of the unification judgments is that the resulting substitution preserves the types of the unification variables, and as a consequence, the algorithm does not need to type check the resulting substitution.

The properties of interest of the unification judgments are expressed in Theorem B.4. The first two parts of the theorem deal with atomic unification of types and objects respectively. The last part deals with normal unification of objects.

Theorem B.4 (Correctness of Unification) *All of the types mentioned in the statement below are assumed to be placeholder-free.*

(a) If $A_1 \approx_a A_2 \Rightarrow \Psi$ such that $\Gamma \overset{i}{\vdash} A_1 : K_1$ and $\Gamma \overset{i}{\vdash} A_2 : K_2$ then

- $\Gamma \vdash \Psi$, *and*
- $\mathrm{PF}(\Psi)$, *and*
- $\Psi(A_1) \equiv_\beta \Psi(A_2)$, *and*
- $\Psi(K_1) \equiv_\beta \Psi(K_2)$

(b) *If* $M_1 \approx_a M_2 \Rightarrow \Psi$ *and* $\mathrm{PF}(M_1)$ *and* $\mathrm{PF}(M_2)$ *such that* $\Gamma \overset{i}{\vdash} M_1 : A_1$ *and* $\Gamma \overset{i}{\vdash} M_2 : A_2$ *then*

- $\Gamma \vdash \Psi$, *and*
- $\mathrm{PF}(\Psi)$, *and*
- $\Psi(M_1) \equiv_\beta \Psi(M_2)$, *and*
- $\Psi(A_1) \equiv_\beta \Psi(A_2)$

(c) *If* $M_1 \approx M_2 \Rightarrow \Psi$ *and* $\mathrm{PF}(M_1)$ *and* $\mathrm{PF}(M_2)$ *such that* $\Gamma \overset{i}{\vdash} M_1 : A$ *and* $\Gamma \overset{i}{\vdash} M_2 : A$ *then*

- $\Gamma \vdash \Psi$, *and*
- $\mathrm{PF}(\Psi)$, *and*
- $\Psi(M_1) \equiv_\beta \Psi(M_2)$

Such a complicated statement is required in order to prove the theorem by induction. All of the actual uses of the Theorem B.4 are in the form of a much simpler corollary shown below. The corollary follows immediately from the case (a) of Theorem B.4 by taking $K_1 = K_2 = \mathtt{Type}$.

Corollary B.5 *If* $A_1 \approx_a A_2 \Rightarrow \Psi$ *such that* $\Gamma \overset{i}{\vdash} A_1 : \mathtt{Type}$ *and* $\Gamma \overset{i}{\vdash} A_2 : \mathtt{Type}$ *then*

1. $\Gamma \vdash \Psi$, *and*
2. $\mathrm{PF}(\Psi)$, *and*
3. $\Psi(A_1) \equiv_\beta \Psi(A_2)$.

PROOF OF THEOREM B.4: The proof is by induction on the structure of the unification derivations. We only show here the cases for the unification of objects (cases b and c). The proof for atomic unification of types follows the same patterns.

Case:

$$\mathcal{D} = \frac{}{c \approx_a c \Rightarrow \cdot}$$

The empty substitution is well-typed in any environment and is also placeholder-free. By hypothesis we know that $\Gamma \overset{i}{\vdash} c : A_1$ and $\Gamma \overset{i}{\vdash} c : A_2$. From Lemma B.16 (stated on page 258) we conclude that $A_1 \equiv_\beta A_2$. The rest of the conclusion follows immediately.

Case:

$$\mathcal{D} = \frac{}{x \approx_a x \Rightarrow \cdot}$$

Again, the typing condition on the resulting substitution is vacuously true. By hypothesis we know that $\Gamma \overset{i}{\vdash} x : A_1$ and $\Gamma \overset{i}{\vdash} x : A_2$. From Lemma B.16 we infer that $A_1 \equiv_\beta A_2$, which concludes the proof of this case.

Case:

$$\mathcal{D} = \cfrac{\begin{array}{c}\mathcal{D}_1 \\ [{}^{N}\!/\!_{x}]M \approx_a M' \Rightarrow \Psi\end{array}}{(\lambda x.M)\ N \approx_a M' \Rightarrow \Psi}$$

For the purpose of the correctness proof we only consider the β-reduction case with only one argument ($n = 1$). The general case is proved in a similar way. By hypothesis we have that $\Gamma \overset{i}{\vdash} (\lambda x.M)\ N : A_1$. From Lemma B.16 we infer that $\Gamma \overset{i}{\vdash} \lambda x.M : \Pi x:A.B$ and $\Gamma \overset{i}{\vdash} N : A$ and $[{}^{N}\!/\!_{x}]B \equiv_\beta A_1$. Because PF$((\lambda x.M)\ N)$ we infer that PF$(\lambda x.M)$ and PF(N). Therefore PF$([{}^{N}\!/\!_{x}]M)$. Now we can use the Lemma B.17 to infer that $\Gamma \overset{i}{\vdash} [{}^{N}\!/\!_{x}]M : [{}^{N}\!/\!_{x}]B$ and therefore $\Gamma \overset{i}{\vdash} [{}^{N}\!/\!_{x}]M : A_1$. Now we apply the induction hypothesis on $\mathcal{D}_1$ and infer that $\Gamma \vdash \Psi$ and $\Psi(A_1) \equiv_\beta \Psi(A_2)$ and that PF(Ψ). We also conclude from the induction hypothesis that $\Psi([{}^{N}\!/\!_{x}]M) \equiv_\beta \Psi(M')$. But we have $\Psi([{}^{N}\!/\!_{x}]M) = [{}^{\Psi(N)}\!/\!_{x}]\Psi(M) \equiv_\beta \Psi((\lambda x.M)N)$, which completes the proof of this case.

Case:

$$\mathcal{D} = \cfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ M \approx_a M' \Rightarrow \Psi & \Psi(N) \approx \Psi(N') \Rightarrow \Psi'\end{array}}{M\ N \approx_a M'\ N' \Rightarrow \Psi' \circ \Psi}$$

We follow the following sequence of deductions:

1. From $\Gamma \overset{i}{\vdash} M\ N : A_1$ (hypothesis) and Lemma B.16 we get
2. $\Gamma \overset{i}{\vdash} M : \Pi x\!:\!A.B$, and
3. $\Gamma \overset{i}{\vdash} N : A$, and
4. $[{}^{N}\!/\!_{x}]B \equiv_\beta A_1$.
5. From $\Gamma \overset{i}{\vdash} M'\ N' : A_2$ (hypothesis) and Lemma B.16 we get
6. $\Gamma \overset{i}{\vdash} M' : \Pi x\!:\!A'.B'$, and
7. $\Gamma \overset{i}{\vdash} N' : A'$, and
8. $[{}^{N'}\!/\!_{x}]B' \equiv_\beta A_2$.
9. Using 2 and 6 we apply the induction hypothesis on $\mathcal{D}_1$. We conclude:
10. $\Gamma \vdash \Psi$, and
11. PF(Ψ), and
12. $\Psi(\Pi x\!:\!A.B) \equiv_\beta \Psi(\Pi x\!:\!A'.B')$, and
13. $\Psi(M) \equiv_\beta \Psi(M')$.
14. With 3, 10 and 11 we use Corollary B.19 and deduce that $\Psi(\Gamma) \overset{i}{\vdash} \Psi(N) : \Psi(A)$.

15. With 7, 10 and 11 we use Corollary B.19 and deduce that $\Psi(\Gamma) \overset{i}{\vdash} \Psi(N') : \Psi(A')$, and using 12 we also deduce that $\Psi(\Gamma) \overset{i}{\vdash} \Psi(N') : \Psi(A)$.
16. From 11 we infer that $\mathrm{PF}(\Psi(N))$ and $\mathrm{PF}(\Psi(N'))$. This together with 14 and 15 let us apply the induction hypothesis on $\mathcal{D}_2$. We deduce that:
17. $\Psi(\Gamma) \vdash \Psi'$, and
18. $\mathrm{PF}(\Psi')$, and
19. $\Psi'(\Psi(N)) \equiv_\beta \Psi'(\Psi(N'))$.
20. From 10, 11, 17 and 18 with Lemma B.21 we get the first part of the conclusion: $\Gamma \vdash \Psi' \circ \Psi$ and $\mathrm{PF}(\Psi' \circ \Psi)$.
21. From 13 and using Lemma B.23 we deduce that $\Psi' \circ \Psi(M) \equiv_\beta \Psi' \circ \Psi(M')$. This and 19 allow us to prove more of the conclusion: $\Psi' \circ \Psi(M\ N) \equiv_\beta \Psi' \circ \Psi(M'\ N')$.
22. From 4 and using Lemma B.23 we deduce that $\Psi' \circ \Psi(A_1) \equiv_\beta [^{\Psi' \circ \Psi(N)}/_x]\Psi' \circ \Psi(B)$. Similarly from 8 we get that $\Psi' \circ \Psi(A_2) \equiv_\beta [^{\Psi' \circ \Psi(N')}/_x]\Psi' \circ \Psi(B')$. Now we can use Lemma B.24 with 12 and 19 to get the last part of the conclusion: $\Psi' \circ \Psi(A_1) \equiv_\beta \Psi' \circ \Psi(A_2)$.

This concludes all of the cases regarding the atomic unification. The rest of the cases are for normal unification. The only interesting cases here are the abstraction and instantiation cases shown below.

Case:

$$\mathcal{D} = \cfrac{\begin{matrix}\mathcal{D}_1 \\ M \approx M' \Rightarrow \cdot\end{matrix}}{\lambda x.M \approx \lambda x.M' \Rightarrow \cdot}$$

The empty substitution is well-typed and also placeholder-free. By hypothesis $\Gamma \overset{i}{\vdash} \lambda x.M : A$ and $\Gamma \overset{i}{\vdash} \lambda x.M' : A$. From Lemma B.16 we have that $A = \Pi x\!:\!A_1.A_2$ and $\Gamma, x : A_1 \overset{i}{\vdash} M : A_2$ and also $\Gamma, x : A_1 \overset{i}{\vdash} M' : A_2$. Now we can use the induction hypothesis on $\mathcal{D}_1$ and infer the required conclusion: $M \equiv_\beta M'$.

Case:

$$\mathcal{D} = \cfrac{u \notin FV(M)}{u \approx M \Rightarrow u \mapsto M}$$

Let $\Psi = u \mapsto M$. By hypothesis we have that $\Gamma \overset{i}{\vdash} u : A$ and $\Gamma \overset{i}{\vdash} M : A$ and $\mathrm{PF}(M)$. From Corollary B.20 we deduce that $\Gamma \vdash \Psi$. Because $\mathrm{PF}(M)$ we can also infer that $\mathrm{PF}(\Psi)$. The rest of the conclusion is trivial: $\Psi(u) \equiv_\beta \Psi(M)$ because $\Psi(M) = M$.

□

Correctness of Constraint Collection and Solving

Due to the presence of dependent types there is a close relationship between constraint collection and constraint solving, and thus we cannot consider them separately. Intuitively, the constraint solving judgment makes sense only if all typing constraints contain well-typed types since, due to the dependent types, the constaint types might contain objects that are reconstructed while solving other constraints.

Things are further complicated by not being able to assume that the constraints are solved in the order in which they were produced. If we could assume this, then by the time a constraint is about to be solved we could prove that the type involved is well-formed of kind `Type`. We want to prove correctness for an arbitrary order of solving the constraints because much of the power of the reconstruction originates in the ability to solve the constraints out-of-order.

This circular dependency of the judgments complicates the correctness proof substantially. Instead of being able to prove one correctness theorem for each of the two judgments, I must state correctness as a sequence of four theorems, where each theorem establishes the assumptions for the next. In this sequence the first (Theorem B.6) and third theorem (Theorem B.8) refer to constraint solving and the other two (Theorem B.7 and Theorem B.9) to constraint collection.

The first theorem says that during constraint list solving each typing constraint is eventually solved. However, depending on the particular order of solving some unification variables may be already instantiated, hence the Ψ_1.

Theorem B.6 *If* $\Gamma \overset{r}{\vdash} C \Rightarrow \Psi$ *then for each* $N : A'$ *from* C *there exist* Ψ_1 *and* Ψ_2 *such that* $\Psi_1(\Gamma) \overset{r}{\vdash} \Psi_1(N) : \Psi_1(A') \Rightarrow \Psi_2$.

Proof of Theorem B.6: The proof is by induction on the structure of the derivation $\mathcal{D} :: \Gamma \overset{r}{\vdash} C \Rightarrow \Psi$. The conclusion is vacuously true if C is empty. Also, if the last rule in $\mathcal{D}$ is the reordering rule, the induction hypothesis proves the conclusion directly. The other two cases are similar so I only show here the case when the last rule in $\mathcal{D}$ is solving a typing constraint.

Case:

$$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Gamma \overset{r}{\vdash} M : A \Rightarrow \Psi} \qquad \overset{\mathcal{D}_2}{\Psi(\Gamma) \overset{r}{\vdash} \Psi(C) \Rightarrow \Psi'}}{\Gamma \overset{r}{\vdash} C, M : A \Rightarrow \Psi' \circ \Psi}$$

I show the conclusion separately for $M : A$ and then for all of the other typing constraints $N : A'$ from C. If $M : A$ is the considered constraint in $C, M : A$ then the conclusion follows immediately with $\Psi_1 = \cdot$ and $\Psi_2 = \Psi$. Otherwise let $N : A'$ be a typing constraint in C. Then $\Psi(N) : \Psi(A')$ is a constraint in $\Psi(C)$. By induction hypothesis on $\mathcal{D}_2$ we get that there exists Ψ'_1 and Ψ'_2 such that $\Psi'_1(\Psi(\Gamma)) \overset{r}{\vdash} \Psi'_1(\Psi(N)) : \Psi'_1(\Psi(A')) \Rightarrow \Psi'_2$. This proves our conclusion with the required substitutions $\Psi_1 = \Psi'_1 \circ \Psi$ and $\Psi_2 = \Psi'_2$.

$\square$

I continue now with a theorem about the constraint collection judgment. Using the result of the previous theorem I show that all of the types in the resulting constraint list are well-formed and of kind `Type`.

Theorem B.7 *If* $\Gamma \overset{r}{\vdash} M \Rightarrow (\Delta \,;\, C \,;\, B)$ *with* $\mathrm{PF}(\Gamma)$ *and* $\mathrm{UVF}(M)$ *and for all* $N : A'$ *in* C *there exist* Ψ_1, Ψ_2 *and* Δ' *such that* $\Psi_1(\Gamma, \Delta, \Delta') \overset{r}{\vdash} \Psi_1(N) : \Psi_1(A') \Rightarrow \Psi_2$ *then*

- $\Gamma, \Delta \overset{i}{\vdash} B : \mathtt{Type}$, *and*
- $\mathrm{PF}(\Delta)$, *and*
- $\mathrm{PF}(B)$, *and*
- *For all* $N : A'$ *in* C *we have*
  - $\mathrm{PF}(A')$, *and*
  - $\mathrm{UVF}(N)$, *and*
  - $\Gamma, \Delta \overset{i}{\vdash} A' : \mathtt{Type}$.

This theorem establishes crucial properties required by the unification invoked from the constraint solving judgment. It is here that side-condition on the application to explicit parameter is used. The main purpose of the side-condition is to allow us to infer that the type of an explicit parameter that becomes part of dependent type makes that type well-formed.

Proof of Theorem B.7: The proof is by induction on the structure of the derivation $\mathcal{D} : \Gamma \overset{r}{\vdash} M \Rightarrow (\Delta \,;\, C \,;\, B)$. There are 4 cases depending on the last rule used in $\mathcal{D}$. The cases for constants and variables follow immediately as the Δ and the list of constraints are empty and because $\mathrm{PF}(\Gamma)$. The only interesting cases are those that deal with application. In the case of an explicit application, we need the helper Theorem B.10 on page 256 that states the reconstruction is reduced to LF$_i$ typing if the objects and types involved are fully explicit.

Case:

$$\mathcal{D} = \cfrac{\begin{array}{c}\mathcal{D}_1\\ \Gamma \overset{r}{\vdash} M \Rightarrow (\Delta \,;\, C \,;\, \Pi x{:}A.B)\end{array}}{\Gamma \overset{r}{\vdash} M * \Rightarrow (\Delta, u : A \,;\, C \,;\, [{}^{u}\!/\!{}_{x}]B)} \qquad u \text{ is a new unification variable}$$

We can immediately apply the induction hypothesis on $\mathcal{D}_1$ and follows the sequence of steps shown below:

1. $\Gamma, \Delta \overset{i}{\vdash} \Pi x\!:\!A.B : \mathtt{Type}$, and
2. $\mathrm{PF}(\Delta)$, and
3. $\mathrm{PF}(\Pi x\!:\!A.B)$, and
4. For all $N : A'$ in C we have $\Gamma, \Delta \overset{i}{\vdash} A' : \mathtt{Type}$, and $\mathrm{PF}(A)$ and $\mathrm{UVF}(N)$.
5. Because u is new we can transform 4 in $\Gamma, \Delta, u : A \overset{i}{\vdash} A' : \mathtt{Type}$, which proves part of the conclusion.
6. Similarly, from 1 we deduce that $\Gamma, \Delta, u : A \overset{i}{\vdash} [{}^{u}\!/_{x}]B : \mathtt{Type}$, which is another part of the conclusion.
7. From 2 and 3 we can show that $\mathrm{PF}(\Delta, u : A)$
8. From 3 we show that $\mathrm{PF}([{}^{u}\!/_{x}]B)$, which concludes the proof of this case.

Case:

$$\mathcal{D} = \cfrac{\begin{array}{c}\mathcal{D}_1 \\ \Gamma \overset{r}{\vdash} M \Rightarrow (\Delta \;;\; C \;;\; \Pi x\!:\!A.B)\end{array} \qquad \begin{array}{r} x \in FV(B) \supset (\mathrm{PF}(N) \textit{ and } \mathrm{UVF}(A) \\ \textit{and } \mathrm{UVF}(\Gamma(FV(N))))\end{array}}{\Gamma \overset{r}{\vdash} M\ N \Rightarrow (\Delta \;;\; C, N : A \;;\; [{}^{N}\!/_{x}]B)}$$

From the induction hypothesis on $\mathcal{D}_1$ we infer the following:

1. $\Gamma, \Delta \overset{i}{\vdash} \Pi x\!:\!A.B : \mathtt{Type}$, and
2. $\mathrm{PF}(\Delta)$ (part of the conclusion), and
3. $\mathrm{PF}(\Pi x\!:\!A.B)$, and
4. For all $N' : A'$ in C we have $\Gamma, \Delta \overset{i}{\vdash} A' : \mathtt{Type}$ and $\mathrm{PF}(A')$ and $\mathrm{UVF}(N')$ (part of the conclusion).
5. From 3 we can deduce that $\mathrm{PF}([{}^{N}\!/_{x}]B)$ (part of the conclusion). Note that we have used the fact that if $x \in FV(B)$ then $\mathrm{PF}(N)$.
6. The only part of the conclusion that remains to be proved is that $\Gamma, \Delta \overset{i}{\vdash} [{}^{N}\!/_{x}]B : \mathtt{Type}$. If $x \notin FV(B)$ then this follows immediately from 1. Otherwise is suffices to prove that $\Gamma \overset{i}{\vdash} N : A$.
7. By hypothesis we have that there exist Ψ_1, Ψ_2 and Δ' such that $\Psi_1(\Gamma, \Delta, \Delta') \overset{r}{\vdash} \Psi_1(N) : \Psi_1(A) \Rightarrow \Psi_2$.
8. But $\mathrm{UVF}(A)$ therefore we get $\Psi_1(A) = A$.
9. Because $\mathrm{UVF}(M\ N)$ (hypothesis) we know that $\mathrm{UVF}(N)$.
10. From 9 we conclude that $\Psi_1(N) = N$.

11. With 8 and 10 we transform 7 to $\Psi_1(\Gamma, \Delta, \Delta') \overset{r}{\vdash} N : A \Rightarrow \Psi_2$
12. But because UVF(N) then none of the unification variables from Δ and Δ' occur in N. Therefore we can transform 11 to $\Psi_1(\Gamma) \overset{r}{\vdash} N : A \Rightarrow \Psi_2$.
13. We know that UVF($\Gamma(FV(N))$) therefore we can transform 12 to $\Gamma \overset{r}{\vdash} N : A \Rightarrow \Psi_2$.
14. With 3 we apply Theorem B.10 on 13 and deduce that $\Gamma \overset{i}{\vdash} N : A$. As motivated at 6 this concludes the proof of this case.

□

Next is the theorem that shows that the constraint solving judgment, when presented with constraints whose types are well-formed, produces a well-typed substitution that satisfies the constraints.

Theorem B.8 *If* $\Gamma \overset{r}{\vdash} C \Rightarrow \Psi$ *with*

- PF(Γ), *and*
- *For* $A \approx_a B$ *in* C *we have* PF(A), PF(B) *and* $\Gamma \overset{i}{\vdash} A : \texttt{Type}$ *and* $\Gamma \overset{i}{\vdash} B : \texttt{Type}$, *and*
- *For all* $N : A'$ *in* C *we have* UVF(N) *and* PF(A') *and* $\Gamma \overset{i}{\vdash} A' : \texttt{Type}$, *and*

then the following are true:

- $\Gamma \vdash \Psi$, *and*
- PF(Ψ), *and*
- *For* $A \approx_a B$ *in* C *we have* $\Psi(A) \equiv_\beta \Psi(B)$, *and*
- *For all* $N : A'$ *in* C *we have* $\Psi(\Gamma) \overset{i}{\vdash} N : \Psi(A')$

Proof of Theorem B.8: The proof is by induction on the structure of the derivation $\mathcal{D} : \Gamma \overset{r}{\vdash} C, A \approx_a B \Rightarrow \Psi$.

The case when the last rule in $\mathcal{D}$ is the reordering rule poses no problems. Similarly the case of an empty constraint list is trivial. I show next the case when the last rule in $\mathcal{D}$ is solving a typing constraint.

Case:

$$\mathcal{D} = \dfrac{\begin{matrix}\mathcal{D}_1 \\ \Gamma \overset{r}{\vdash} M : A \Rightarrow \Psi\end{matrix} \qquad \begin{matrix}\mathcal{D}_2 \\ \Psi(\Gamma) \overset{r}{\vdash} \Psi(C) \Rightarrow \Psi'\end{matrix}}{\Gamma \overset{r}{\vdash} C, M : A \Rightarrow \Psi' \circ \Psi}$$

Because PF(Γ), PF(A) and UVF(M) and $\Gamma \overset{i}{\vdash} A : \texttt{Type}$ we can apply Theorem B.3 on $\mathcal{D}_1$ and infer that $\Gamma \vdash \Psi$, PF(Ψ) and $\Psi(\Gamma) \overset{i}{\vdash} M : \Psi(A)$.

Because the substitution Ψ is without placeholders and is well-typed, the constraint list $\Psi(C)$ satisfies all of the conditions for applying the induction hypothesis on $\mathcal{D}_2$ with respect to the type environment $\Psi(\Gamma)$. We deduce from the induction hypothesis that $\Psi(\Gamma) \vdash \Psi'$ and $\mathrm{PF}(\Psi')$ and for all $N : A'$ in C that $\Psi' \circ \Psi(\Gamma) \overset{i}{\vdash} N : \Psi' \circ \Psi$. The final step is to use Lemma B.21 to conclude the proof of this case.

The case when a unification constraint is solved is very similar with the difference that Theorem B.4 is invoked to show that the unification returns a well-typed substitution.

□

The last theorem in the correctness proof of the reconstruction algorithm shows why the existence of a well-typed substitution defined on all locally introduced unification variables is enough to guarantee the well-typedness of the original term.

Theorem B.9 *If* $\Gamma \overset{r}{\vdash} M \Rightarrow (\Delta \,;\, C \,;\, B)$ *and*

- $\Gamma, \Delta \vdash \Psi$*, and*
- $\mathrm{PF}(\Psi)$*, and*
- $Dom(\Delta) \subseteq Dom(\Psi)$*, and*
- *For all* $N : A'$ *in* C *we have* $\mathrm{UVF}(N)$ *and* $\Psi(\Gamma) \overset{i}{\vdash} N : \Psi(A')$

then $\Psi(\Gamma) \overset{i}{\vdash} M : \Psi(B)$.

Proof of Theorem B.9: The proof is by induction on the structure of the derivation $\mathcal{D} :: \Gamma \overset{r}{\vdash} M \Rightarrow (\Delta \,;\, C \,;\, B)$. The cases of a constant or a variable follow immediately from the hypothesis. I consider next the cases of application to explicit terms and to placeholders.

Case:

$$\mathcal{D} = \cfrac{\begin{array}{c}\mathcal{D}_1 \\ \Gamma \overset{r}{\vdash} M \Rightarrow (\Delta \,;\, C \,;\, \Pi x\!:\!A.B)\end{array} \qquad \begin{array}{r} x \in FV(B) \supset (\mathrm{PF}(N) \textit{ and } \mathrm{UVF}(A) \\ \textit{and } \mathrm{UVF}(\Gamma(FV(N))))\end{array}}{\Gamma \overset{r}{\vdash} M\ N \Rightarrow (\Delta \,;\, C, N : A \,;\, [^N\!/_x]B)}$$

We apply the induction hypothesis on $\mathcal{D}_1$ and conclude that $\Psi(\Gamma) \overset{i}{\vdash} M : \Psi(\Pi x : A.B)$. From the hypothesis we have that $\mathrm{UVF}(N)$ and $\Psi(\Gamma) \overset{i}{\vdash} N : \Psi(A)$. From the application rule in LF$_i$, and because $\mathrm{UVF}(N)$ we deduce that $\Psi(\Gamma) \overset{i}{\vdash} M\ N : \Psi([^N\!/_x]B)$. Note that the newly introduced type $[^N\!/_x]B$ is without placeholders because if x occurs in B then $\mathrm{PF}(N)$.

Case:

$$\mathcal{D} = \cfrac{\begin{matrix}\mathcal{D}_1 \\ \Gamma \overset{r}{\vdash} M \Rightarrow (\Delta \,;\, C \,;\, \Pi x\!:\!A.B)\end{matrix}}{\Gamma \overset{r}{\vdash} M * \Rightarrow (\Delta, u : A \,;\, C \,;\, [{}^{u}\!/\!{}_{x}]B)} \qquad u \text{ is a new unification variable}$$

Let $\Psi' = \Psi|_{\Gamma,\Delta}$. We follow a sequence of deductions as follows:

1. From $\Gamma, \Delta, u : A \vdash \Psi$ (hypothesis) and $Dom(\Delta, u : A) \subseteq Dom(\Psi)$ (hypothesis) we deduce:
2. $\Psi(\Gamma, \Delta, u : A) \overset{i}{\vdash} \Psi(u) : \Psi(A)$, and
3. $Dom(\Delta) \subseteq Dom(\Psi')$.
4. Because $Dom(\Delta, u : A) \subseteq Dom(\Psi)$ we have that $\Psi(\Gamma, \Delta, u : A) = \Psi(\Gamma)$.
5. From 2 and 4 we infer that $\Psi(\Gamma) \overset{i}{\vdash} \Psi(u) : \Psi(A)$.
6. From Lemma B.22 and 1 we have that $\Gamma, \Delta \vdash \Psi'$. Also because u is new it cannot appear in C or in Γ. Therefore we can deduce that for all $N : A'$ in C we have $\Psi'(\Gamma) \overset{i}{\vdash} N : \Psi'(A')$. Thus we can apply the induction hypothesis on $\mathcal{D}_1$ and conclude that
7. $\Psi'(\Gamma) \overset{i}{\vdash} M : \Pi x\!:\!\Psi'(A).\Psi'(B)$.
8. Again, because u is new it cannot appear in Γ, M, A or B. Therefore, from 7 we deduce that $\Psi(\Gamma) \overset{i}{\vdash} M : \Pi x\!:\!\Psi(A).\Psi(B)$.
9. Recall that we assume that all types involved are without placeholders. Thus, we have PF($\Psi'(A)$) and therefore PF($\Psi(A)$).
10. Now we can use the implicit application rule of LF$_i$ with 5, 8 and 9 and with the placeholder replacement being $\Psi(u)$. The resulting type is $[{}^{\Psi(u)}\!/\!{}_{x}]\Psi(B) = \Psi([{}^{u}\!/\!{}_{x}]B)$.
11. This case is not complete until we verify that the newly introduced type $\Psi([{}^{u}\!/\!{}_{x}]B)$ is without placeholders. This follows immediately from the PF($\Psi(\Pi x\!:\!A.B)$) and PF(Ψ).

Note that this is the place where we require the property that Ψ be well-typed and defined for all variables in Δ.

□

This concludes the skeleton of the correctness proof for the reconstruction algorithm. Next are the proofs of the helper lemmas used above in the correctness proofs. I start with a family of theorems mirroring the correctness proof but in the special case when the terms involved are fully-reconstructed.

Correctness in the Fully-Explicit Case

The correctness of the reconstruction algorithm use the fact that reconstruction is correct in the special case when both the LF term and type involved do not contain placeholders or unification variables. This is stated below as Theorem B.10. The correctness proof in the fully-explicit form follows the same pattern as the proof in the general case, with some simplifications. We do not show here a complete proof of this case. We just state the lemmas involved.

Theorem B.10 *If* $\Gamma \overset{r}{\vdash} M : A \Rightarrow \Psi$ *and* $\mathrm{PF}(\Gamma)$, $\mathrm{PF}(M)$, $\mathrm{PF}(A)$, $\mathrm{UVF}(M)$, $\mathrm{UVF}(A)$ *and* $\mathrm{UVF}(\Gamma(FV(M)))$ *then* $\Psi = \cdot$ *and* $\Gamma \overset{i}{\vdash} M : A$.

PROOF: The proof of this theorem is done similarly to that of Theorem B.1 by induction on the derivation $\mathcal{D} : \Gamma \overset{r}{\vdash} M : A \Rightarrow \Psi$. The abstraction case is simple. For the application case we need a series of auxiliary lemmas about the constraint collection and solving judgments in the case of fully-explicit terms and types. These lemmas are stated without proof below.

□

Lemma B.11 *If* $\Gamma \overset{r}{\vdash} M \Rightarrow (\Delta \; ; \; C \; ; \; B)$ *such that* $\mathrm{PF}(\Gamma)$, $\mathrm{PF}(M)$, $\mathrm{UVF}(M)$ *and also* $\mathrm{UVF}(\Gamma(FV(M)))$ *then*

- $\Delta = \cdot$, *and*
- $\mathrm{PF}(B)$ *and* $\mathrm{UVF}(B)$, *and*
- *For all* $N : A'$ *in* C *we have that* $\mathrm{PF}(N)$, $\mathrm{UVF}(N)$, $\mathrm{PF}(A')$ *and* $\mathrm{UVF}(A')$ *and also that* $\mathrm{UVF}(\Gamma(FV(N)))$.

The intuition behind Lemma B.11 is that because M does not have placeholders, no unification variables are introduced, hence $\Delta = \cdot$. Also the terms in C are subterms of M and therefore do not contain placeholders or unification variables and also all their free variables have types that do not contain unification variables. The types in C and B are constructed from fully-explicit types (because $\mathrm{PF}(\Gamma)$ and $\mathrm{UVF}(\Gamma(FV(M)))$) with subterms of M, hence the lack of a condition on types.

Lemma B.12 *If* $A \approx_a B \Rightarrow \Psi$ *and* $\mathrm{UVF}(A)$ *and* $\mathrm{UVF}(B)$ *then*

- $\Psi = \cdot$, *and*
- $A \equiv_\beta B$

The proof of Lemma B.12 is induction on the structure of the unification judgment. The intuition behind this lemma is that if the terms to be unified do not contain unification variables then the resulting substitution must be empty. In this case the two terms are β-equivalent.

Lemma B.13 *If* $\Gamma \overset{r}{\vdash} C, A \approx_a B \Rightarrow \Psi$ *and* $\mathrm{PF}(\Gamma)$, $\mathrm{PF}(A)$, $\mathrm{UVF}(A)$, $\mathrm{PF}(B)$, $\mathrm{UVF}(B)$ *and for all* $N : A'$ *in* C *we have* $\mathrm{PF}(N)$, $\mathrm{UVF}(N)$, $\mathrm{PF}(A')$, $\mathrm{UVF}(A')$ *and* $\mathrm{UVF}(\Gamma(FV(N)))$ *then*

- $\Psi = \cdot$, *and*
- $A \equiv_\beta B$, *and*
- $\Gamma \overset{i}{\vdash} N : A'$ *for all* $N : A'$ *in* C

The proof of Lemma B.13 is by induction on the structure of the derivation $\Gamma \overset{r}{\vdash} C, A \approx_a B \Rightarrow \Psi$. When a unification is solved we use Lemma B.12 to conclude that the resulting substitution is empty and that the unified types are β-equivalent. When a typing constraint is solved then the hypothesis provides all of the conditions necessary to apply Theorem B.10 and conclude again that the substitution is empty and that the typing constraints are satisfied.

Lemma B.14 *If* $\Gamma \overset{r}{\vdash} M \Rightarrow (\cdot\ ;\ C\ ;\ B)$ *and* $A \equiv_\beta B$ *with* $\mathrm{PF}(A)$ *and for all* $N : A'$ *in* C *we have* $\Gamma \overset{i}{\vdash} N : A'$ *and* $\mathrm{PF}(A')$ *then* $\Gamma \overset{i}{\vdash} M : A$.

The proof of Lemma B.14 is by induction on the structure of the collection derivation. Again, placeholders are disallowed and because the typing constraints from C are satisfied we can immediately prove the conclusion using the typing rules of LF_i.

B.2 Soundness of LF_i typing

In addition to proving that the reconstruction algorithm only succeeds when there is an LF_i typing derivation for the presented proof, we also need to show that the existence of an LF_i typing derivation guarantees the existence of a well-typed fully-explicit LF form for the proof. Only then we can use the adequacy of LF representation theorems that guarantee the provability of the verification condition. This theorem is first stated on page 92 as Theorem 5.4 and is proved in this section. For clarity I restate the theorem below:

Theorem B.15 Soundness of LF_i typing *If* $\Gamma \overset{i}{\vdash} M : A$ *and* $\mathrm{PF}(\Gamma)$, $\mathrm{PF}(A)$, *then there exists* M' *such that* $M \nearrow M'$ *and* $\Gamma \overset{LF}{\vdash} M' : A$.

PROOF: The proof is by induction on the structure of the derivation $\mathcal{D} : \Gamma \overset{i}{\vdash} M : A$. The case of constants or variables are trivial. In the case of the β-equivalence rule we use the induction hypothesis with the assumption that PF(A). Then we use the LF β-equivalence rule. In the case of an abstraction we use the assumption $PF(\Pi x\!:\!A.B)$ to ensure that we can apply the induction hypothesis. The remaining cases deal with the application to a term or a placeholder.

Case:

$$\mathcal{D} = \frac{\Gamma \overset{i}{\vdash} M : \Pi x\!:\!A.B \qquad \Gamma \overset{i}{\vdash} N : A \qquad \mathrm{PF}(A)}{\Gamma \overset{i}{\vdash} M * : [N/x]B}$$

Because of the hypothesis PF(A) we can apply the induction hypothesis on $\mathcal{D}_2$ and deduce that there exists N' such that $N \nearrow N'$ and $\Gamma \overset{LF}{\vdash} N' : A$.

From the theorem assumption we have that $\mathrm{PF}([N/x]B)$. From here we infer that PF(B) and then that $\mathrm{PF}(\Pi x\!:\!A.B)$. This justifies applying the induction hypothesis to $\mathcal{D}_1$ and inferring that there exists M' such that $M \nearrow M'$ and $\Gamma \overset{LF}{\vdash} M' : \Pi x\!:\!A.B$. Now using the LF application rule we infer that $\Gamma \overset{LF}{\vdash} M'\ N' : [N'/x]B$. It is evident that $M * \nearrow M'\ N'$. What remains to be proved is that $[N'/x]B \equiv_\beta [N/x]B$. This is that case if $x \notin FV(B)$. Otherwise, we know from the hypothesis that $\mathrm{PF}([N/x]B)$ which implies that PF(N) and then that $N = N'$.

The case when the last rule in $\mathcal{D}$ is an application to a term is very similar to the case presented above.

□

B.3 Auxiliary Lemmas

The following lemmas are used in the correctness proof of the type reconstruction algorithm in Section B.1. Most of them are trivial to prove and therefore we omit their proof. Recall also that we made the convention that all of the types involved in the statements of the theorems and lemmas are placeholder-free.

The first lemma establishes canonical forms of types in LF_i judgments.

Lemma B.16 *If $\Gamma \overset{i}{\vdash} M : A$ then the following are true:*

- *If $M = x$ then $\Gamma(x) \equiv_\beta A$*
- *If $M = c$ then $\Sigma(c) \equiv_\beta A$*
- *If $M = M_1\ M_2$ then $\Gamma \overset{i}{\vdash} M_1 : \Pi x\!:\!A_1.A_2$ and $\Gamma \overset{i}{\vdash} M_2 : A_1$ and $[M_2/x]A_2 \equiv_\beta A$*
- *If $M = \lambda x.N$ then $A = \Pi x\!:\!A_1.A_2$ and $\Gamma, x : A_1 \overset{i}{\vdash} M : A_2$.*

Next there is a lemma saying that β-reduction "preserves" the type of the expression.

Lemma B.17 *If* $\Gamma \overset{i}{\vdash} \lambda x.M : \Pi x : A.B$ *and* $\Gamma \overset{i}{\vdash} N : A$ *with* $\mathrm{PF}(N)$ *then* $\Gamma \overset{i}{\vdash} [{}^{N}\!/\!_{x}]M : [{}^{N}\!/\!_{x}]B$.

We continue with a crucial lemma used throughout the proof of correctness. This lemma says that if a substitution is well-typed on certain unification variables then it preserves the typing relation.

Lemma B.18 *Let* $D = Dom(\Psi) \cap FV(M)$ *if* $\mathrm{PF}(M)$ *and* $Dom(\Psi)$ *otherwise. If* $\mathrm{PF}(\Psi)$ *and* $\Psi(\Gamma) \overset{i}{\vdash} \Psi(u) : \Psi(\Gamma(u))$ *for all* $u \in D$ *and if* $\Gamma \overset{i}{\vdash} M : A$ *then* $\Psi(\Gamma) \overset{i}{\vdash} \Psi(M) : \Psi(A)$.

Proof of Lemma B.18: The proof is by induction on the structure of the derivation $\mathcal{D} :: \Gamma \overset{i}{\vdash} M : A$. The case of a constant is trivial because the constant and its type do not change by substitution.

More interesting cases are those when M is a variable or a unification variable. In the case of a normal variable, or a unification variable outside $Dom(\Psi)$, the conclusion follows immediately from the definition of $\Psi(\Gamma)$. In the case of a unification variable that is in $Dom(\Psi)$ the conclusion follows from the hypothesis because that variable is also in $FV(M)$ and therefore in D.

We consider the other cases below: **Case:**

$$\mathcal{D} = \frac{\overset{\mathcal{D}_1}{\Gamma \overset{i}{\vdash} M : A} \qquad A \equiv_\beta B \qquad \mathrm{PF}(A)}{\Gamma \overset{i}{\vdash} M : B}$$

In this case we can apply the induction hypothesis on $\mathcal{D}_1$ and infer that $\Psi(\Gamma) \overset{i}{\vdash} \Psi(M) : \Psi(A)$. Note that we have used the hypothesis $\mathrm{PF}(A)$ to ensure that our implicit convention about types is preserved. By Lemma B.23 we get that $\Psi(A) \equiv_\beta \Psi(B)$. Because $\mathrm{PF}(\Psi)$ and $\mathrm{PF}(A)$ we deduce that $\mathrm{PF}(\Psi(A))$. We can therefore use the LF_i rule for beta-equivalence and infer that $\Psi(\Gamma) \overset{i}{\vdash} \Psi(M) : \Psi(B)$.

Case:

$$\mathcal{D} = \frac{\Gamma, x : A \overset{i}{\vdash} M : B}{\Gamma \overset{i}{\vdash} \lambda x.M : \Pi x{:}A.B}$$

The set of unification variables D is the same for $\lambda x.M$ and M. We know that for all $u \in D$ we have $\Psi(\Gamma) \overset{i}{\vdash} \Psi(u) : \Psi(\Gamma(u))$. Because x cannot occur in Ψ we deduce that $\Psi(\Gamma, x : A) \overset{i}{\vdash} \Psi(u) : \Psi((\Gamma, x : A)(u))$ for all $u \in D$. We can therefore apply the induction hypothesis and conclude that $\Psi(\Gamma, x : A) \overset{i}{\vdash} \Psi(M) : \Psi(B)$. From here we can use the abstraction rule of LF_i and deduce the desired conclusion $\Psi(\Gamma) \overset{i}{\vdash} \Psi(\lambda x.M) : \Psi(\Pi x{:}A.B)$.

Case:

$$\mathcal{D} = \cfrac{\begin{matrix}\mathcal{D}_1 & & \mathcal{D}_2 & \\ \Gamma \overset{i}{\vdash} M : \Pi x\!:\!A.B & & \Gamma \overset{i}{\vdash} N : A & & \mathrm{PF}(A)\end{matrix}}{\Gamma \overset{i}{\vdash} M\ N : [{}^{N}\!/\!_{x}]B}$$

We know that $\mathrm{PF}(M\ N)$ iff $\mathrm{PF}(M)$ and $\mathrm{PF}(N)$. We can use the induction hypotheses both on $\mathcal{D}_1$ and $\mathcal{D}_2$ because the free variables of M and N are among those of $M\ N$. Because $\mathrm{PF}(A)$ and $\mathrm{PF}(\Psi)$ we have that $\mathrm{PF}(\Psi(A)$ which can be used together with the induction hypotheses to infer the desired conclusion $\Psi(\Gamma) \overset{i}{\vdash} \Psi(M\ N) : \Psi([{}^{N}\!/\!_{x}]B)$. Note that by our implicit convention on types we have that $\mathrm{PF}([{}^{N}\!/\!_{x}]B)$ which implies that $\mathrm{PF}(B)$ and also that $\mathrm{PF}(\Pi x\!:\!A.B)$.

Case:

$$\mathcal{D} = \cfrac{\begin{matrix}\mathcal{D}_1 & & \mathcal{D}_2 & \\ \Gamma \overset{i}{\vdash} M : \Pi x\!:\!A.B & & \Gamma \overset{i}{\vdash} N : A & & \mathrm{PF}(A)\end{matrix}}{\Gamma \overset{i}{\vdash} M\ * : [{}^{N}\!/\!_{x}]B}$$

Note that in this case we do not have $\mathrm{PF}(M\ *)$ and therefore we must have $\Gamma \vdash \Psi$. We can therefore apply the induction hypothesis for the derivation $\mathcal{D}_1$ and infer that $\Psi(\Gamma) \overset{i}{\vdash} \Psi(M) : \Pi x : \Psi(A).\Psi(B)$. From the induction hypothesis on $\mathcal{D}_2$ we infer that $\Psi(\Gamma) \overset{i}{\vdash} \Psi(N) : \Psi(A)$. From here we follow the same steps as in the previous case. Note that $\Psi(M\ *) = \Psi(M)\ *$.

□

The Lemma B.18 is not actually used in that form. All its uses are in the form of two corollaries stated and proved below.

Corollary B.19 *If* $\Gamma \overset{i}{\vdash} M : A$ *and* $\Gamma \vdash \Psi$ *with* $\mathrm{PF}(\Psi)$ *then* $\Psi(\Gamma) \overset{i}{\vdash} \Psi(M) : \Psi(A)$.

Proof of Corollary B.19: The corollary follows immediately from Lemma B.18 if we note that $D \subseteq \mathit{Dom}(\Psi)$ and that $\Gamma \vdash \Psi$ implies that for all $u \in \mathit{Dom}(\Psi)$ we have $\Psi(\Gamma) \overset{i}{\vdash} \Psi(u) : \Psi(\Gamma(u))$.

□

Corollary B.20 *If* $\Gamma \overset{i}{\vdash} u : A$ *and* $\Gamma \overset{i}{\vdash} M : A$ *with* $\mathrm{PF}(M)$ *and* $u \notin FV(M)$ *then* $\Gamma \overset{i}{\vdash} u \mapsto M$.

Proof of Corollary B.20: Let $\Psi = u \mapsto M$. Because $\mathrm{PF}(M)$ we can apply Lemma B.18 with $D = \mathit{Dom}(\Psi) \cap FV(M) = \emptyset$ and we infer that $\Psi(\Gamma) \overset{i}{\vdash} \Psi(M) : \Psi(A)$. But because $u \notin FV(M)$ we have that $\Psi(M) = M = \Psi(u)$. Therefore $\Gamma \vdash \Psi$.

□

We continue with two lemmas dealing with typing substitution. The first is concerned with the well-typedness of a composition of two substitutions. The second one deals with restricted substitutions.

Lemma B.21 *If* $\Gamma \vdash \Psi$ *and* $\Psi(\Gamma) \vdash \Psi'$ *with* $\mathrm{PF}(\Psi)$ *and* $\mathrm{PF}(\Psi')$ *then* $\Gamma \vdash \Psi' \circ \Psi$ *and* $\mathrm{PF}(\Psi' \circ \Psi)$.

PROOF OF LEMMA B.21: It is obvious that $\mathrm{PF}(\Psi' \circ \Psi)$. To prove that $\Gamma \vdash \Psi' \circ \Psi$ consider a unification variable $u \in Dom(\Psi' \circ \Psi)$. Then either $u \in Dom(\Psi)$, in which case $\Psi(Gamma) \stackrel{i}{\vdash} \Psi(u) : \Psi(\Gamma(u))$ and by Lemma B.18 we get the desired conclusion, or $u \in Dom(\Psi') \backslash Dom(\Psi)$, in which case $\Psi'(\Psi(\Gamma)) \stackrel{i}{\vdash} \Psi'(u) : \Psi'(\Psi(\Gamma(u)))$.

□

Lemma B.22 *If* $\Gamma, \Delta \vdash \Psi$ *and* $Dom(\Delta) \subseteq Dom(\Psi)$ *then* $\Psi(\Gamma, \Delta) = \Psi(\Gamma) = \Psi\big|_{Dom(\Gamma)} (\Gamma)$ *and* $\Gamma \vdash \Psi\big|_{Dom(\Gamma)}$.

PROOF OF LEMMA B.22: It is easy to prove that $\Psi(\Gamma, \Delta) = \Psi(\Gamma)$ using the definition of substitution applied to type environments. Also it must be the case that Γ does not contain any unification variable contained in Δ, thus we get $\Psi(\Gamma) = \Psi\big|_{Dom(\Gamma)} (\Gamma)$.

For the second part, let $u \in Dom(\Psi) \cap Dom(\Gamma)$. We have that $\Psi(\Gamma) \stackrel{i}{\vdash} \Psi(u) : \Psi(\Gamma(u))$. Because nothing in Δ can occur in Γ we conclude that $\Gamma \vdash \Psi\big|_{Dom(\Gamma)}$.

□

The last lemmas required are concerned with β-equivalence. Their proof is trivial so I state them here without proof.

Lemma B.23 *If* $M \equiv_\beta N$ *then* $\Psi(M) \equiv_\beta \Psi(N)$.

Lemma B.24 *If* $M \equiv_\beta N$ *and* $M' \equiv_\beta N'$ *then* $[{}^{M}\!/_{x}]M' \equiv_\beta [{}^{N}\!/_{x}]N'$.