A Compositional Logic for Control Flow and its Application in Foundational Proof-Carrying Code

Gang Tan

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
By the Department of
Computer Science

September 2005

Abstract

Proof-Carrying Code (PCC) is a static mechanism that mechanically verifies safety of machine-language programs. But the problem in conventional PCC is, who will verify the verifier (the type checker) itself? The Foundational Proof-Carrying Code (FPCC) project at Princeton verifies the soundness of the type checker from the smallest possible set of axioms — logic plus machine semantics. One challenge in the verification is that machine code, unlike high-level languages, contains unstructured control flow (due to arbitrary jumps). A piece of machine code can contain multiple entry points that jump instructions might jump to, and multiple exit points. Traditional Hoare logic and its variants either verify programs with only one entry and one exit, or need the whole program to verify jump instructions, which is not modular.

The major contribution of this dissertation is a program logic, $\mathcal{L}_c$, which modularly verifies properties of machine-code fragments. Unlike previous program logics, the basic reasoning units in $\mathcal{L}_c$ are multiple-entry and multiple-exit code fragments. $\mathcal{L}_c$ provides composition rules to combine code fragments and to eliminate intermediate entries/exits in the combined fragment. $\mathcal{L}_c$ is not only useful for reasoning about properties of machine code with unstructured control flow, but also useful for deriving rules for common control-flow structures such as while-loops, repeat-until-loops, among others. We also present a semantics for $\mathcal{L}_c$ and prove that $\mathcal{L}_c$ is both sound and complete.

As an application to the FPCC project, I have implemented $\mathcal{L}_c$ on top of the SPARC machine language and used $\mathcal{L}_c$'s rules to verify the soundness of the instruction rules of a full-fledged low-level typed assembly language. This demonstrates $\mathcal{L}_c$'s applicability of verifying properties of machine-language programs.

Acknowledgments

First, I would like to thank my advisor, Andrew Appel, for his guidance throughout my graduate career. I could never ask for a more patient and understanding advisor. He is always available for discussion, and his comments are always insightful. He sets a model example for me by his enthusiasm and persistence toward research, teaching, and life.

Thanks to my dissertation readers, David Walker and Karl Crary, for their helpful comments. David Walker has also been a mentor to me. I have learned a great deal from him through our research collaborations. He is always encouraging and eager to help. Thanks to Karl Crary for his valuable comments, which has helped to improve this dissertation significantly.

I have had the privilege of working with many talented fellow students and postdocs at Princeton. Thanks to Xinming Ou, who has made our research collaborations a memorable experience. Thanks to Daniel Wang, who is so energetic; it has always been pleasant to be around him. Thanks to everybody in the Foundational Proof-Carrying Code project: Amal Ahmed, Lujo Bauer, Juan Chen, Eun-Young Lee, Neophytos Michael, Xinming Ou, Chris Richards, Kedar Swadi, Roberto Virga, Daniel Wang, and Dinghao Wu.

I would also like to thank Yuqun Chen for mentoring me in my internship at Microsoft Research. Also thanks to Srimat Chakradhar, Anand Raghunathan, and Srivaths Ravi for mentoring me in my internship at Nec Laboratories America. These summer internships have provided me invaluable experiences.

Special thanks to our graduate secretary, Melissa Lawson, for many things she has taken care of for me. Thanks to all administrative and technical staffs at the department who were always friendly.

Finally, my work is supported by DARPA grant F30602-99-1-0519, NSF grant CCR-0208601, ARDA grant NBCHC030106, and by the Francis Upton Graduate Fellowship for 1999-2003, a Princeton University honorific Fellowship.

To my wife, Xiaolei, and my parents.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software systems are pervasive in all aspects of society. From online shopping to electronic voting, software has become an intrinsic part of business and our daily life over the past few decades. However, software systems are not secure and reliable. The media is full of reports of the catastrophic impact of software failures. These software failures are very expensive. According to a federal study [52], software failures are costing the U.S. economy an estimated $59.5 billion each year.

It is very difficult to develop secure and reliable software systems, because of so called "trinity of trouble" [37]: complexity, connectivity, and extensibility.

Complexity Software is complicated and will become more and more complicated. The size of software products nowadays is measured in terms of millions lines of code. For example, Windows XP has 40 million lines of code. Furthermore, software products consist of many small components. Understanding the interfaces between these components and tracking of their relationship is a very difficult task.

Connectivity The Internet connects more and more computers, from home PCs to control-critical infrastructures. This connectivity makes it possible to attack computer systems without physical access. It also allows small attacks to propagate and spread and cause vast damage. In March 1999, more than 100,000 individual hosts were infected by Melissa virus over a weekend [1]. One site reported receiving 32,000 copies of infected messages within 45 minutes.

Extensibility Many software products can accept extensions and updates to incrementally evolve system functionality. For example, an extensible browser can install new plug-ins to support new media formats. An operating system allows new device drivers to be installed and run in the kernel mode. However, without a proper design, the very nature of extensible systems also allows malicious extensions to be installed.

In a highly connected and extensible environment, an important research problem is how to guarantee security and safety of the software that we download from the Internet, especially when the author of the software is unknown, or untrusted. This problem has been dubbed the mobile-code security problem.

The first question is what kind of guarantee is desirable for mobile code. To the very extreme, we would want the code to be *correct*, that is, it computes what it supposed to do. However, practice has shown that it is extremely difficult to verify correctness of large programs. First of all, it is an daunting task to specify the correctness, and even if this is done, there is no guarantee it is specified correctly. “The correctness condition for a real program is vastly complicated; you’ll always be proving the wrong theorem!” [27] After the correctness is specified, verifying that the code meets the specification is also daunting and the proof is "unreadable and — literally — unspeakable" [27].

Fortunately, full correctness is usually not necessary for assuring the security of mobile code. We may be satisfied with, for example, that the code will not peek into other applications' memory. *Security policies* like these are much easier to specify and verify. Schneider [58] specifies a security policy as a predicate on sets of executions. Well-known security policies include access-control policies (about who can perform what actions on which objects) and information-flow policies (about what principals can infer about objects by observing system behavior).

Schneider [58] specifies a *security property* as a predicate on individual executions, and therefore a security property is also a security policy. Security properties include safety properties ("bad things" do not happen), and liveness properties ("good things" do happen). Alpern and Schneider [5] have proved that every security property can be decomposed into a safety property and a liveness property.

One important safety property that has been intensively studied is *type safety*. Cardelli defines type safety as the property that programs do not cause untrapped errors [15]. Typically, type safety of programming languages implies many other desirable safety properties. One is *memory safety*, which means that the code will access only appropriate memory locations (no errors such as buffer overrun). The other is *control-flow safety*, which means that the code will transfer control only to appropriate locations. Type safety of some languages such as ML also provides *type abstraction*, which can guarantee that behavior of clients of abstract types is independent of the particular implementation of abstract types.

Having discussed desirable security properties, we review some existing techniques for mobile-code security and investigate what kind of properties they guarantee.

Virus, worm, and spyware detection Viruses, worms, and spyware are all malicious software, or malware. Many tools (including Norton, Symantec, and Ad-Aware) can examine files to scan for bit patterns to detect malware. However, these tools scan for only known malware, not unknown malware. After scanning for a malware, say M, the only security guarantee we get is that the file being scanned is free of malware M; the guarantee does not apply to any other malware, or even the variants of M.

Code signing This is a method that is used in web browsers to authenticate publishers of mobile code. Mobile code is attached with a digital signature signed by the secret key of a software publisher. Before running the code, users verify the signature using the public key of the publisher. Code signing can authenticate the source of the mobile code. However, the security guarantee of code signing is predicated on the assumption that the software developer does not make programming mistakes. This assumption has been proved to be fragile. For example, in November 2002, Microsoft released an ActiveX control, signed with Microsoft's code-signing key, but it may lead to a security vulnerability. For users who desired a secure system, Microsoft later had to recommend to them removing *Microsoft* from Internet Explorer's Trusted Publisher List [2].

Hardware-based protection One way to prevent mobile code from reading from or writing to other processes' memory space is to put it into a separate address space. Virtual memory, managed by the kernel, maps each process's address space into physical memory. Each process has its own virtual address space and cannot directly accesses the memory belonging to other processes. Only through safe system calls can a user process switch to the privileged mode to access shared resources and communicate with other processes. Hardware-based protection is a widely used approach. However, when one process needs to communicate with other processes, this approach requires expensive context switches and therefore incurs a significant performance overhead. Furthermore, it can protect only resources accessed through system calls, and thus guarantee a limited set of security properties.

1.1 Language-Based Security

During recent years, a new approach to software security — language-based security — has attracted more and more attention. Schneider, Morrisett, and Harper [59] defines language-based security as any mechanism that "leverages program analysis and program rewriting to enforce security policies". It achieves secure software by analyzing program behavior using formal language semantics. It utilizes techniques in compilers, automated program analysis, type checking, and program rewriting to analyze semantics of programs during both static time and runtime.

Language-based security can be classified according to the level where the approach is applied: source-language level, intermediate-language level, and low level. In the following sections, we will discuss approaches at each level. During our discussion, we will analyze each approach's Trusted Computing Base (TCB), which is the set of components that has to be trusted to ensure the security of the system. In general, systems with smaller TCB are more secure.

1.1.1 Source-Language Level

Approaches in this category include safe languages, which are type safe by design; they also include program rewriting, which rewrite a program at the source-language level to make it safe.

Type-safe languages. Type-safe languages include ML [43], Java [12], and Modula-3 [50]. Type systems of these languages are designed to explicitly rule out bad behaviors. Jave's type system, in particular, has been proved type safe [28, 51]. There have also been proposals for type-safe dialects of unsafe languages. Cyclone [38] is a safe dialect of C. It has been designed to avoid common programming errors in C, such as buffer overflow, format string attacks, and memory-management errors.

Program rewriting. This approach performs source-to-source program transformation so that an unsafe program is rewritten to a safe version. CCured [49] is a source-to-source translator for C. By inserting run-time checks, it transforms a C program to a version that is memory safe, meaning that it will stop rather than overrun a buffer or scribble over memory that it should not touch. The Naccio system [29] allows users to specify a security policy, such as what files or directories may be read and written, and the system can transform a Java program into a version that conforms to the specified policy.

Since these approaches are applied at the source-language level, programs have to be fed into a compiler before they are run on a real machine. However, a compiler may compile safe source programs into unsafe ones, due to possible bugs in the compiler. Therefore, unless we can prove the compiler is correct, approaches in this category need to trust the compiler.

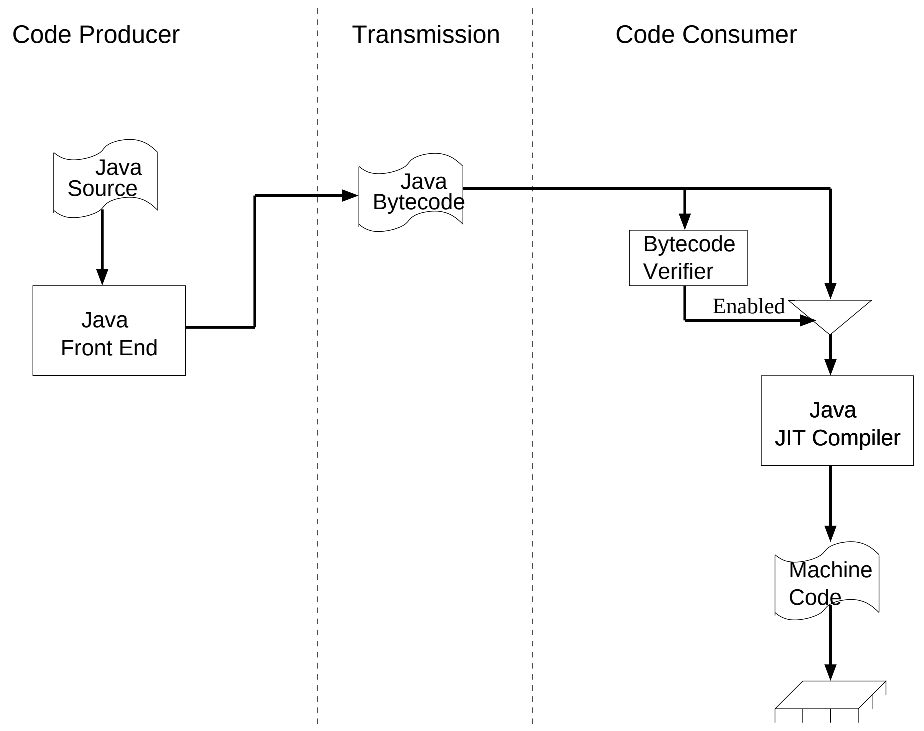


Figure 1.1: Java bytecode framework

1.1.2 Intermediate-Language Level

The Java platform (Figure 1.1) may be the most widely used approach in language-based security. A Java program is compiled by a compiler front end into a program at the intermediate-language level. In Java's terminology, the language is called Java Virtual Machine Language [41], or Java bytecode. The important thing about Java bytecode is that it is a typed intermediate language so that it can be independently type checked at the code-consumer side. If type checking of bytecode is successful, the bytecode is type safe and is then compiled by a Just-In-Time (JIT) compiler into machine code to run (or the bytecode is interpreted). The .NET framework uses a similar strategy. In .NET [18], different languages can be compiled into a common type-safe intermediate language, Microsoft Intermediate Language (MSIL).

The success of Java is a powerful witness to the applicability of language-based security. Java is designed to assume the role of mobile code in the internet age. Portability and security are its two biggest concerns. By pushing types into the level of bytecode, the Java framework eliminates the Java compiler front end from the TCB. On the other hand, the Java compiler back end, where the compilation from bytecode to machine code happens and where most optimizations happen, is still in the TCB.

1.1.3 Low Level

Because ultimately it is the native machine code that actually runs, approaches that are applied at either the source level or the intermediate level suffer from the drawback that a translation module needs to be trusted. In the case of the source level, the compiler has to function properly to compile a safe source-level program to safe machine code. In the case of intermediate level, the compiler back end such as Java's JIT compiler has to function properly. Both a full compiler and a compiler back end are complicated, big, and thus error prone. Verifying the correctness of the compiler can remove it from TCB, but it is generally prohibitive to do verification on an industrial-strength compiler due to its size and complexity.

From the point view of security, it is better to guarantee security properties at a level closer to the real machine, that is, at the assembly-code level[1] or directly at the machine-code level. By doing so, the compiler is no longer needs to be trusted.

[1] There is a difference between the assembly-code level and the machine-code level. In the former case, an assembler needs to be trusted to correctly assemble programs. But in this chapter, we ignore this issue.

Software-fault isolation. To avoid expensive context switches in hardware-based protection, Software-Fault Isolation (SFI) [68] loads code and data for an untrusted module into its own fault domain, a logically separate portion of the application's address space. Then, the object code of the untrusted module is modified to prevent it from writing or jumping to an address outside its fault domain. For example, for each store instruction, the target address is modified to be constrained in the protection domain. Through this sandboxing, program modules isolated in separate software-enforced fault domains cannot modify each other's data or execute each other's code except through an explicit cross-fault-domain RPC interface.

By putting untrusted modules into the same address space, SFI avoids expensive context switches. Experiments show its cross-fault-domain performance is two orders of magnitude more efficient than context switches. At the same time, only 4% execution-time overhead to the C SPEC92 benchmarks is incurred, when the security guarantee is that the untrusted module cannot destroy other modules' memory. In this case, only store and jump instructions need to be sandboxed.

However, in the case of general protection by sandboxing load instructions as well as store and jump instructions, the execution-time overhead is around 20%, which is not a small amount. Furthermore, theoretically SFI can protect a single memory address, but since its technique involves dedicated registers for each protection domain, it usually protects a large region of memory and thus its protection granularity is coarse.

Proof-carrying code. Necula and Lee introduced Proof-Carrying Code (PCC) [46, 47], which is a technique that can be used for safe execution of untrusted code. In a typical instance, a code consumer establishes a safety policy which can guarantee safe behavior of programs, and the code producer produces a formal safety proof. Then, the receiver is able to use a proof checker to check the validity of the proof and, if so, can be assured of the safety of the program. PCC separates proof generation and proof checking. The intuition is that proof checking is much easier than proof generation. Code consumers are responsible for proof checking, while code producers are responsible for proof generation, which is the hard part.

Theoretically PCC can be used for any safety policy. But PCC usually concentrates on type safety, for which the safety proof can be automatically generated. Necula and Lee have developed two experimental systems for type safety. The first system [48] is a certifying compiler that translates programs written in a type-safe subset of the C programming language into DEC Alpha assembly language. The second system, called Special J [20], certifies type safety of machine code compiled from Java programs.

Figure 1.2 depicts Special J's setup. Starting from type-safe Java programs, the system has a certifying compiler that emits machine code with annotations, which are essential to construct the type-safety proof for the machine code. Another component, called the certifier, includes a Verification-Condition Generator (VCGen), a prover, and a proof checker. The VCGen scans the code and the annotation, and produces a safety theorem such that the code is safe if and only if the safety theorem is true. The prover proves the safety predicate using a set of standard typing rules, and produces a proof if succeeds. The proof checker can check the proof against the safety theorem (generated by the VCGen at the consumer side), using the same set of typing rules.

PCC is a very appealing framework to guarantee safe execution of untrusted code. It has no runtime overhead. After checking the proof, the code is safe and

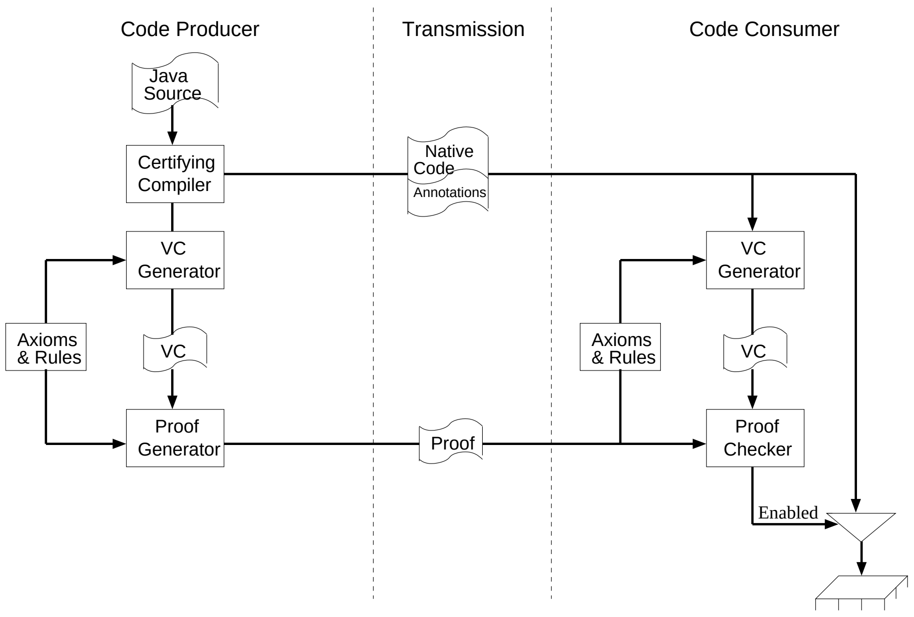


Figure 1.2: PCC framework for certified code

can be run as many times as we want. It is also tamper-proof. If the code and/or the safety proof is modified, either the proof will be rejected, or the modified code is still safe to run.

On the other hand, PCC systems by Necula et al. still have some limitations. The trusted computing base in PCC includes the VCGen, the proof checker, and the typing rules (typing rules are provided to the proof generator as a collection of inference rules). VCGen needs to understand machine semantics, calling conventions, and safety predicates to function correctly. Typing rules are also very complex for low-level types. Furthermore, their systems do not reason about memory management and trust an automatic garbage collector.

As an attempt to remove VCGen from TCB, Bernard and Lee [13] proposed to use temporal logic to model machine semantics and have code producers show directly that a particular temporal-logic safety property holds for the untrusted program. Since machine semantics, calling conventions, and safety properties are all modeled in temporal logic, there is no need to have a VCGen in Bernard and Lee's system. When using their framework to certify type safety, however, the low-level type system still needs to be trusted (as a set of inference rules in temporal logic).

Typed-assembly languages. In Java's framework, the compiler front end translates Java source into bytecode. At the consumer side, the bytecode checker type-checks the bytecode. Then the type information is discarded, and the compiler back end optimizes the code, translates it into a machine specific intermediate language, performs register allocation, and finally gets a machine-language program. In this framework, we need to trust the whole compiler back end, including the optimizer, the code generator and the register allocator. If any of them has bugs inside, the resulting machine code may be unsafe.

There has been a great deal of research efforts to push types further down to lower levels. Typed intermediate languages were pioneered by the TIL [66] research compiler. FLINT [61] is a research compiler for ML and Java that uses the typed-intermediate-language technology. At each level, the intermediate language has type information, and a type checker can run to verify the soundness of the program. The compiler compiles one typed intermediate-language programs into programs of another typed intermediate language.

In their work on Typed Assembly Language (TAL) [45], Morrisett et al. presented a type preserving compilation from System F down to a typed assembly language on a RISC machine. Their translation to TAL is specified as a series of type-preserving transformation, including continuation-passing style (CPS) and

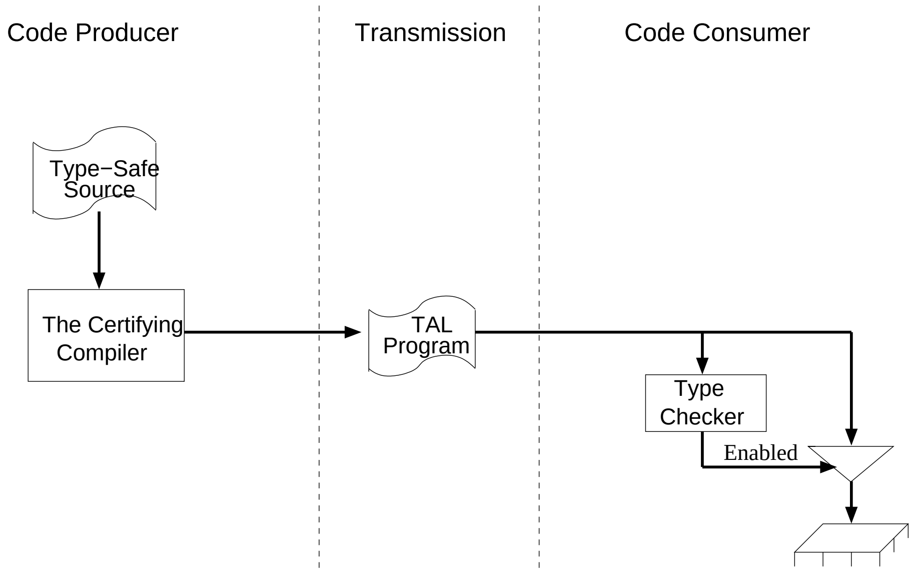


Figure 1.3: TAL framework for certified code

closure conversion; type-safe source programs are mapped to type-safe assembly-language programs.

The compiler in TAL provides a fully automatic way to certify safety of untrusted code. As shown in Figure 1.3, producers write their applications in type-safe source languages. Then the compiler translates type-safe source programs into type-safe assembly-language programs. TAL Programs are transmitted to the consumer side, and after being type checked, they are safe to run.

1.2 Foundational Proof-Carrying Code

If a compiler produces TAL, then by checking only the low-level code, we can guarantee safety — but only if there is no bug in the typing rule, or in the type checker. In TAL and PCC systems, typing rules are trusted axioms.

Take a typing rule from the TAL paper [45],

$$\frac{\Psi;\Delta;\Gamma\ \vdash\ r:\forall[\,].\Gamma' \quad \Delta\ \vdash\ \Gamma\le\Gamma'}{\Psi;\Delta;\Gamma\ \vdash\ \texttt{jmp}\ r} \tag{1.1}$$

The rule means that a "jump to register r" instruction type checks if the value in r is a code pointer with precondition Γ', and the current type environment Γ is a subtype of Γ'. Based on the semantics of the jump instruction, this rule is "correct". (In this dissertation we will not be concerned with the exact meaning of Ψ, Δ, etc.; we show this rule just to illustrate TAL's trusted axioms.)

In the TAL system and its variants [44, 24, 69, 25, 72], such typing rules are accepted as axioms. They are part of the TCB. However, low-level type systems tend to be complex because of intricate machine semantics. Any misunderstanding of the semantics could lead to errors in the type system. League et al. [40] found an unsound proof rule in Special J's type system. The Low-Level Typed Assembly Language (LTAL) [17] by Chen et al. has around eleven hundred constructors and rules. The author of this dissertation has been involved in proving the soundness of LTAL and routinely found and fixed bugs in its early versions.

In an effort to remove the type system from TCB, Appel and Felty proposed the idea of Foundational Proof-Carrying Code (FPCC) [8, 7]. In FPCC, a soundness proof at the level of the concrete machine accompanies the type system. The soundness proof states that if the type system type checks a program, then the program is safe on the machine. After checking the validity of the proof, the consumer is assured of the soundness of the type system.

In FPCC, we can imagine a two-stage process of running untrusted code. In the first stage (Figure 1.4), the code producer sends the consumer a type checker,

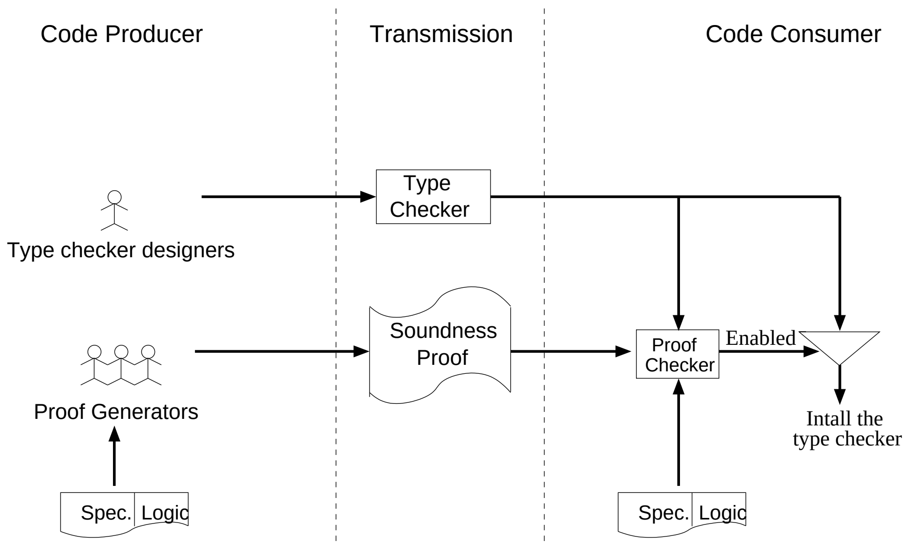


Figure 1.4: FPCC framework for certified code: the first stage

together with a soundness proof with respect to the specification of the concrete machine. The soundness proof is encoded in some suitable logic. After verifying the validity of the soundness proof, the consumer can install the type checker. For each type system, this stage happens only once.

The second stage is just like the one in TALs (Figure 1.3). The producer sends untrusted code with annotations in the type system to the consumer. After checking the safety of the code using the already installed type checker, the code is safe to run.

FPCC is more secure and more flexible than conventional PCC and TAL. It is more secure because its trusted base is smaller: the type system is not trusted; only the specification of the machine, the logic and the proof checker needs to be trusted. It is also more flexible because it makes no commitment to any particular type system, which is usually dependent on both source languages and the compiler. The code producer can explain a novel type system to the code consumer.

The semantic approach to FPCC. Appel and Felty's approach [8] to FPCC is called the semantic approach. The semantic approach gives denotational semantics to the syntactic constructs in the type system, including types and typing judgments. This semantics is encoded in a simple but expressive logic. Appel and Felty used higher-order logic for its expressiveness and its successful use in the HOL [32] system and Isabelle [54] system. Based on the denotational semantics for the type system and an operational semantics for the concrete machine, typing rules like 1.1 on page 14 are proved as lemmas, instead of being trusted as axioms. Finally, the safety theorem is also proved as a lemma: If a program type checks, then it is safe.

In the semantic approach, since each syntactic construct has a denotational semantics, a typing derivation in the type system is converted into a safety proof with respect to the smallest possible set of axioms — logic plus machine semantics.

This dissertation is part of the FPCC project at Princeton and follows the semantic approach. Before we move on, we discuss an alternative approach to FPCC.

The syntactic approach to FPCC. Hamid et al. [33], Crary and Sarkar [22, 23] use a syntactic approach to prove soundness of the type system in FPCC. The first stage of their approach gives an operational semantics to the typed assembly language on an abstract machine, then syntactic type-soundness theorems are proved on this abstract machine, following the scheme proposed by Wright and Felleisen [70]. The second stage proves a simulation relation between the abstract machine and the concrete machine. The simulation relation proves that whenever the abstract machine steps forward, the concrete machine steps forward.

The syntactic approach to FPCC does not need the building of denotational semantics for complicated types such as recursive types and mutable-reference types.

It has been very successful in delivering foundational proof-carrying code. On the other hand, the real system built by Crary and Sarkar uses the metatheory engine of Twelf [55] and has not so far produced a proof object (an independently checkable proof expressed in a general logic) that represents the soundness proof. (The system by Hamid deals with a toy architecture.) Furthermore, there is a possible difficulty for the syntactic approach when trying to relate results in different type systems. The syntactic approach treats the syntax of each type system abstractly. Since each system has its own syntax of terms and types, it is nearly impossible for the syntactic approach to derive general theorems that relate different type systems. The semantic approach embeds the meaning of terms and types into a common logic. A common semantic framework makes it possible to relate theorems in different type systems.

When prototype systems in the semantic and syntactic approaches evolve into a mature stage, it would be interesting to perform a detailed comparison between these two approaches. We believe that there are many connections between them.

1.3 Contributions and Dissertation Outline

In the semantic approach to FPCC, typing rules should be proved as lemmas based on the machine semantics and a denotational semantics for types and typing judgments. For example, the following rule from TAL we have seen should be proved based on the semantics of the $\texttt{jmp}\ r$ instruction and a suitable denotational semantics for the judgments such as "$\Psi;\Delta;\Gamma \vdash \texttt{jmp}\ r$".

$$\frac{\Psi;\Delta;\Gamma \vdash r : \forall[\,].\Gamma' \quad \Delta \vdash \Gamma \leq \Gamma'}{\Psi;\Delta;\Gamma \vdash \texttt{jmp}\ r}$$

To complete such a proof, we first need a systematical way to organize properties about machine instructions, so that raw semantics of machine instructions can be raised to the typed level and properties of machine instructions can be composed together to form a property about the combined instruction sequence.

One possible choice is Hoare Logic [35], which has long been used to verify properties of programs written in high-level programming languages. In Hoare Logic, a triple $\{p\}s\{q\}$ describes the relationship between exactly two states—the normal entry and exit states—associated with a program execution. That is, if the state before execution of s satisfies the assertion p, then the state after execution satisfies q. Hoare Logic also provides rules for combining program fragments.

However, in FPCC we need to prove properties about machine-language programs, which contain goto statements with unrestricted destinations. Therefore, a program fragment or a collection of statements possibly contains multiple exits and multiple entries to which goto statements might jump. In Hoare Logic, since a triple $\{p\}s\{q\}$ is tailored to describe the relationship between the normal entry and the normal exit states, it is not surprising that trouble arises in considering program fragments with more than one entry and/or more than one exit.

In Chapter 2, we will introduce a new program logic, $\mathcal{L}_c$, which is the major contribution of this dissertation. The strength of $\mathcal{L}_c$ is that it can modularly verify properties of machine-code fragments. Unlike previous program logics, the basic reasoning units in $\mathcal{L}_c$ are multiple-entry and multiple-exit code fragments. $\mathcal{L}_c$ provides composition rules to combine code fragments and to eliminate intermediate entries/exits in the combined fragment. $\mathcal{L}_c$ is not only useful for reasoning about properties of machine code with unstructured control flow, but also useful for deriving rules for common control-flow structures such as while-loops, repeat-until-loops, among others.

Also in Chapter 2, we will develop for $\mathcal{L}_c$ a semantics. We will show that a naive semantics for $\mathcal{L}_c$'s multiple-entry and multiple-exit instruction judgment will not work. We need to use a semantics based on approximations of computation steps. Based on this semantics, soundness and completeness of $\mathcal{L}_c$ are proved (with reasonable assumptions). This semantics is another contribution of this dissertation because it turns a derivation in $\mathcal{L}_c$ into a foundational safety proof.

The major goal of the FPCC project is to prove the soundness of a type-assembly language, called LTAL [17]. Our logic $\mathcal{L}_c$ plays an important role in this proving process. We have first proved the soundness of $\mathcal{L}_c$. Then, LTAL has been proved sound on top of $\mathcal{L}_c$. In Chapter 3, the step from $\mathcal{L}_c$ to LTAL will be demonstrated by constructing the soundness proof for a simple typed assembly language, called TAL_0. After that, we will add more and more complex features to TAL_0 to have a series of typed assembly languages, and show how $\mathcal{L}_c$ can justify their soundness.

In Chapter 4, we will discuss the implementation of $\mathcal{L}_c$ in the FPCC project. We have implemented $\mathcal{L}_c$ on top of the SPARC machine language, proved the soundness of LTAL's instruction rules from $\mathcal{L}_c$'s rules, and proved many typing lemmas about SPARC machine instructions. All the proofs have been encoded and machine-checked in Twelf [55], a theorem-developing system. In total, $\mathcal{L}_c$'s implementation has around 30,000 lines of Twelf code.

In Chapter 5, we will give some concluding remarks.

As a final note, the basic idea of Section 3.2 has been published in the Fifth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI '04), under the title of "Construction of a semantic model for a typed assembly language" [65].

Chapter 2

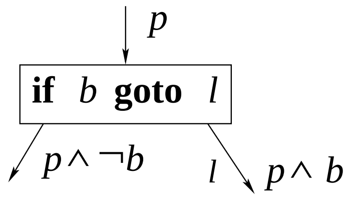
A Compositional Logic for Control Flow

Hoare Logic [35] has long been used to verify properties of programs written in high-level programming languages. But it is unsuitable to verify unstructured machine-language programs, because a Hoare triple $\{p\}s\{q\}$ is tailored to describe the relationship between exactly one entry state and exactly one exit state.

To address the problem of reasoning about control flow in machine-language programs, this chapter proposes a program logic, $\mathcal{L}_c$, which modularly reasons about machine-language program fragments.

2.1 Overview of $\mathcal{L}_c$ and its Related Work

$\mathcal{L}_c$'s modularity is because of two things: its judgment (form of specification) and its composition rules. The judgment in $\mathcal{L}_c$ is directly on multiple-entry and multiple-exit program fragments. For example, $\mathcal{L}_c$ treats a conditional-branch statement "**if** b **goto** l" as a one-entry and two-exit fragment. $\mathcal{L}_c$ then provides for "**if** b **goto** l" an axiom, which associates the entries and exits with appropriate invariants, depicted as follows:



The above graph associates the invariant p with the entry, and associates $p \wedge \neg b$ and $p \wedge b$ with two exits, respectively. As a note to our convention in such kind of graphs, we put invariants on the right of edges and put labels, when they exist, on the left.

$\mathcal{L}_c$ also provides a set of inference rules to compose judgments on program fragments. These inference rules reason about control flow in smaller steps than previous program logics. One example is the case of sequential composition. In Hoare Logic, sequential composition happens in one single step:

$$\frac{\{p_1\}s_1\{p_2\} \quad \{p_2\}s_2\{p_3\}}{\{p_1\}s_1; s_2\{p_3\}}$$

In contrast, $\mathcal{L}_c$ treats both s_1 and s_2 as single-entry and single-exit program fragments (Figure 2.1), and achieves sequential composition in a series of small steps: first combine the two fragments by collecting entries and exits; after this step, the intermediate label l_2 is both an entry and an exit; then eliminate l_2 as an exit; at last, eliminate l_2 as an entry. After these steps, the combined fragment has only one entry and one exit.

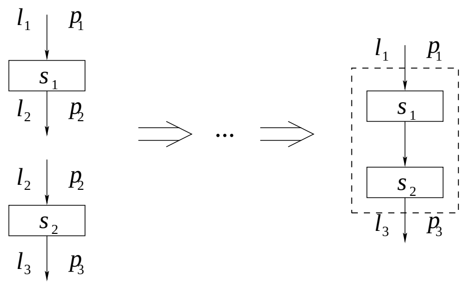


Figure 2.1: Sequential composition.

$$\begin{array}{ll} l: & \textbf{if } \neg b \textbf{ goto } l'; \\ l_1: & s; \\ l_2: & \textbf{goto } l \\ l': & \end{array}$$

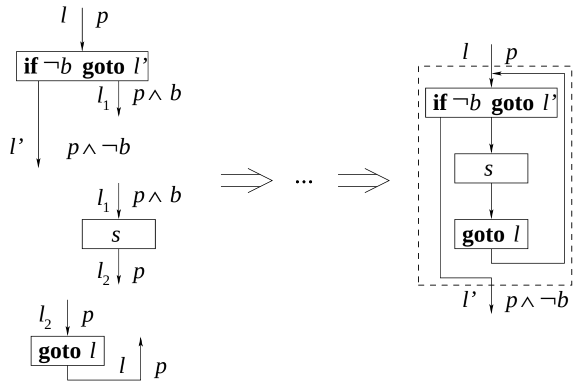


Figure 2.2: A while loop: **while** b **do** s.

The set of inference rules in $\mathcal{L}_c$ is *compositional* in the sense that it can derive not only the rule for sequential composition, but also rules for sequences of statements that implement while loops, repeat-until loops, and many other control-flow structures. Figure 2.2 presents one implementation of a while loop. Each statement in the sequence is a multiple-entry and multiple-exit fragment. After putting fragments together, intermediate entries/exits are eliminated. In the end, the only entry for the whole sequence is the label l and the only exit is l'.

$\mathcal{L}_c$ also supports reasoning about unstructured control flow in machine-language programs, since users of $\mathcal{L}_c$ can choose not to eliminate an intermediate entry, even if it points to the middle of a loop. For example, in Figure 2.2, the label l_1 can remain as an entry of the whole loop in the case that other fragments need to jump to it.

Related work on program logic for goto statements. Many researchers have also realized the difficulty of verifying properties of programs with goto statements in Hoare Logic [19, 39, 11, 26, 53]. Some of them have proposed improvements over Hoare Logic. Almost all of these works are at the level of high-level languages. For example, they treat a while loop as a separate syntactic construct and have a rule for it. In comparison, $\mathcal{L}_c$ derives rules for control-flow structures implemented by sequences of statements.

These previous works also differ from $\mathcal{L}_c$ in terms of the form of specification. The work by de Bruin [26] is a typical example. In his system, the judgment for a statement s is:

$$\langle L_1 : p_1, \ldots, L_n : p_n | \{p\} s \{q\} \rangle, \tag{2.1}$$

where $L_1, \ldots, L_n$ are labels in a program P; the assertion p_i is the invariant associated with the label L_i; the statement s is a part of the program P. Judgment (2.1) judges a triple $\{p\}s\{q\}$, but under all label invariants in a program. By explicitly supplying invariants for labels in the judgment, de Bruin's system can handle goto statements, and its rule for goto statements is $\langle L_1 : p_1, \ldots, L_n : p_n | \{p_i\} \textbf{goto } L_i \{\mathit{false}\} \rangle$.

Judgment (2.1) is sufficient for verifying properties of programs with goto statements. Typed Assembly Languages (TAL) by Morrisett et al. [45] and many of its variants [17, 22, 33] use a similar judgment to verify type safety of assembly-language programs.

However, judgment (2.1) assumes the availability of global information, because it judges a statement s under all label invariants of a program— $L_1 : p_1, \ldots, L_n : p_n$. Consequently, it is impossible for de Bruin's system or TAL to compose fragments with different sets of global label invariants. We believe that a better form of specification should judge s under only those label invariants associated with exits in s, as the form in $\mathcal{L}_c$. This new form of specification makes fewer assumptions (fewer label invariants) about the rest of the program and is more modular.

Furthermore, judgment (2.1) is a specification for a statement s with multiple exits, but with only one entry—the left side of s, or the normal entry. It cannot specify the case of multiple-entry statements. Since arbitrary composition may create multiple-entry statements, de Bruin's system supports only sequential composition. In contrast, $\mathcal{L}_c$ supports any composition of program fragments, even when they do not occupy a sequential region.

Related work on Floyd's flowchart verification. Floyd's work [30] on program verification associates a predicate for each arc in the flowchart representation of a program. If each statement in the program has been verified correct with respect to the predicates associates with the entry and exit arcs of the statement, then the program is correct. In Floyd's system, however, the composition of statements are based on flowcharts and are informal, and it has no principles of eliminating intermediate edges. Our $\mathcal{L}_c$ provides formal rules of combining statements; it can also eliminate intermediate entry and exit edges, so that internal control flow can be hidden from the outside.

When verifying properties of goto statements and labels, Floyd's system also assumes the availability of the complete program. In this sense, it is an informal version of later work by de Bruin. Therefore, our comments to de Bruin's system apply to Floyd's system as well.

Foundational semantics. After presenting $\mathcal{L}_c$ for a language with goto statements, Section 2.3 will show an operational semantics for the language and then a denotational semantics for $\mathcal{L}_c$. With respect to the semantics, we will prove soundness and completeness of $\mathcal{L}_c$. The soundness theorem guarantees there are no inconsistencies in $\mathcal{L}_c$. The completeness theorem shows that $\mathcal{L}_c$ is capable of deriving any judgment that is semantically true.

The semantics that we will present satisfies the "foundational" requirement in Foundational Proof-Carrying Code (FPCC) [7]. FPCC verifies safety of machine code from the smallest possible set of axioms—machine semantics plus logic. The safety proof must explicitly define, down to the foundations of mathematics, all required concepts and explicitly prove any needed properties of these concepts[1]. The semantics that we will present for $\mathcal{L}_c$ satisfies this requirement, because it explicitly gives a meaning to every concept in $\mathcal{L}_c$, and because every axiom and inference rule is proved as a lemma, all from machine semantics and higher-order logic. This semantics gives a way to convert a derivation in $\mathcal{L}_c$ to a safety proof at the machine level.

In fact, design of $\mathcal{L}_c$ has been influenced by the foundational requirement, which further favors a compositional program logic. For example, it would be easy to add an inference rule for the whole sequence of statements that implements a while loop

[1]For efficiency reasons, the FPCC project uses the 32-bit integer constraint domain in Twelf. But integers could be constructed.

operator symbols	*OPSym*	op		
relation symbols	*RSym*	re		
variables	*Var*	x, y, z		
labels	*Label*	l		
primitive statements	*PrimStmt*	t	$::=$	$x := e \mid \mathbf{goto}\ l \mid \mathbf{if}\ b\ \mathbf{goto}\ l$
statements	*Stmt*	s	$::=$	$t \mid l : s \mid (s_1; s_2)$
expressions	*Exp*	e	$::=$	$x \mid op(e_1, \ldots, e_{ar(op)})$
boolean expressions	*BExp*	b	$::=$	$\mathbf{true} \mid b_1 \vee b_2 \mid \neg b \mid re(e_1, \ldots, e_{ar(re)})$

Figure 2.3: Language syntax, where $ar(op)$ is the arity of the symbol op.

(Figure 2.2). However, since that rule has to be proved based on the semantics of the sequence of statements, a better strategy is to design a set of rules that compose proofs about primitive statements together. A small set of such rules saves proof effort when producing foundational proofs for machine-language programs.

2.2 Program Logic $\mathcal{L}_c$

We present $\mathcal{L}_c$ on a simple imperative language. Figure 2.3 presents the syntax of the language. Most of the syntax is self-explanatory, and we only stress a few points. First, since the particular set of primitive operators and relations does not affect the presentation, the language assumes a class of operator symbols, *OPSym*, and a class of relation symbols, *RSym*. For concreteness, *OPSym* could be $\{+, \times, 0, 1\}$ and *RSym* could be $\{=, <\}$. Second, boolean expressions include constructors **true**, $\vee$ and $\neg$; other standard constructors such as **false**, $\wedge$ and $\Rightarrow$ are used as abbreviations defined by **true**, $\vee$ and $\neg$.

The language in Figure 2.3 is tailored to imitate a machine language. For instance, the destination of a goto statement is unrestricted and may be a label in the middle of a loop. Furthermore, the language does not have control structures such

fragments	*Fragment*	f	$::=$	$l : (t) : l'$
fragment sets	*FragSet*	F	$::=$	$\{l_1 : (t_1) : l'_1,\ \ldots,\ l_n : (t_n) : l'_n\}$
assertions	*Assertion*	p	$::=$	$\mathbf{true} \mid p_1 \vee p_2 \mid \neg p$
			$\mid$	$re(e_1, \ldots, e_{ar(re)}) \mid \exists x.p$
label-continuation sets	*LContSet*	Ψ	$::=$	$\{l_1 \triangleright p_1, \ldots, l_n \triangleright p_n\}$

Figure 2.4: $\mathcal{L}_c$: Syntax

as **if** b **then** s, and **while** b **do** s. These control structures are implemented by a sequence of primitive statements.

To simplify the presentation, the language in Figure 2.3 differs from machine languages in several aspects. It uses abstract labels while machine languages use concrete addresses. This differences do not affect the major results of $\mathcal{L}_c$. The language also lacks indirect jumps (jump through a variable) and pc-relative jumps. Adding these control-transfer features will not affect the soundness result of $\mathcal{L}_c$, as shown in Section 3.3. However, the completeness result may only be true in the absence of indirect jumps.

2.2.1 Syntax and Rules of $\mathcal{L}_c$

The syntax of $\mathcal{L}_c$ is in Figure 2.4.

Program fragments. A program fragment is a primitive statement t with a start label l and an end label l':

$$l : (t) : l',$$

where l identifies the left side of t, the *normal entry*, and l' identifies the right side of t, the *normal exit*. We also use $l_1 : (s_1; s_2) : l_3$ as an abbreviation for two fragments:

$l_1 : (s_1) : l_2$ and $l_2 : (s_2) : l_3$, where l_2 is a new label. We use the symbol F for a set of fragments, and it belongs to the domain *FragSet*.

Assertions and label continuations. *Assertions* are meant to describe predicates on states. $\mathcal{L}_c$ can use any assertion language. We use first-order logic in this presentation (Figure 2.4); it is a superset of boolean expressions and thus it can accurately express the invariants after the execution of a conditional-branch statement. Conjunction and universal quantifiers can be defined by constructors in the assertion language: $p_1 \wedge p_2 = \neg(\neg p_1 \vee \neg p_2)$, and $\forall x.p = \neg\exists x.\neg p$.

$\mathcal{L}_c$ is parametrized over a deduction system, $\mathcal{D}$, which derives true formulas in the assertion language. We leave the rules of $\mathcal{D}$ unspecified, and assume that its judgment is

$$\vdash_{\mathcal{D}} p,$$

which is read as p is a true formula.

A label identifies a point in a program. To associate assertions with labels, $\mathcal{L}_c$ uses the notation $l \triangleright p$, pronounced "l with p". In Hoare Logic, when an assertion p is associated with a label l in a verified program, then whenever the control of the program reaches l, the assertion p is true. In $\mathcal{L}_c$, we interpret $l \triangleright p$ in a different way: If $l \triangleright p$ is true in a program, then whenever p is satisfied, it is safe to continue from l (or, jump to l). Therefore, we call p a *precondition* of the label l, and call $l \triangleright p$ a *label continuation*. We use symbol Ψ for a set of label continuations, and it belongs to the domain *LContSet*.

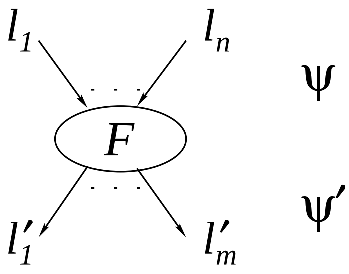
Form of specification. In $\mathcal{L}_c$, the judgment to specify properties of multiple-entry and multiple-exit program fragments has the syntax:

$$F\,;\,\Psi' \vdash \Psi,$$

where F is a set of program fragments; Ψ' and Ψ are sets of label continuations. The meaning of the judgment is explained as follows. Suppose

$$\Psi' = \{l'_1 \triangleright p'_1, \ldots, l'_m \triangleright p'_m\},$$
$$\Psi = \{l_1 \triangleright p_1, \ldots, l_n \triangleright p_n\}.$$

Labels $l'_1, \ldots, l'_m$ in Ψ' are exits of F, and $l_1, \ldots, l_n$ in Ψ are entries of F. The following graph depicts the relationship between F, Ψ and Ψ':



With this relationship in mind, a simplified interpretation of the judgment $F\,;\,\Psi' \vdash \Psi$ is as follows: for a set of fragments F, if *every* $l'_i \triangleright p'_i$ in Ψ' is a true label continuation, then *all* $l_j \triangleright p_j$ in Ψ are true label continuations. Note, however, the semantics we will develop for $F\,;\,\Psi' \vdash \Psi$ in Section 2.3 has an additional requirement: It takes at least one computation step from an entry to reach an exit. We ignore this issue in this section.

Using this judgment, $\mathcal{L}_c$ provides rules for primitive statements. In particular, the rule for the assignment statement is (also in Figure 2.5)

$$\frac{}{\{l : (x := e) : l'\}\,;\, \{l' \triangleright p\} \vdash \{l \triangleright p[e/x]\}}\ \textsf{assignment}$$

The fragment, $l : (x := e) : l'$, has one entry, namely l, and one exit, namely l'. The assignment rule states that $l \triangleright p[e/x]$ is a true label continuation, if $l' \triangleright p$ is a true label continuation. The truth of $l \triangleright p[e/x]$ can be informally argued like this: If a state, which has $\{l : (x := e) : l'\}$ loaded and $p[e/x]$ satisfied, starts from the label l, then the next statement to execute is $x := e$; after its execution, the new state will reach the label l', and the new state satisfies the assertion p because of the semantics of the assignment; because that $l' \triangleright p$ is a true label continuation, the new state is safe to continue; hence, the initial state can safely continue from l.

In Hoare Logic, the assignment rule is

$$\{p[e/x]\}\ x := e\ \{p\}.$$

This is essentially the same as the assignment rule in $\mathcal{L}_c$. In general, for any statement s that has only the normal entry and the normal exit, a Hoare triple $\{p\}s\{q\}$ has in $\mathcal{L}_c$ a corresponding judgment: $\{l : (s) : l'\}\,;\, \{l' \triangleright q\} \vdash \{l \triangleright p\}$.

But unlike Hoare triples, $F\,;\, \Psi' \vdash \Psi$ is a more general judgment, which is on multiple-entry and multiple-exit fragments. This capability is used in the rule for conditional-branch statements, **if** b **goto** l_1, in Figure 2.5. A fragment, l : (**if** b **goto** l_1) : l', has two exit labels: l' and l_1. Therefore, the $\textsf{if}$ rule assumes two exit label continuations.

$$\boxed{F\,;\,\Psi_1 \vdash \Psi_2}$$

$$\frac{}{\{l : (x := e) : l'\}\,;\,\{l' \triangleright p\} \vdash \{l \triangleright p[e/x]\}}\ \textsf{assignment}$$

$$\frac{}{\{l : (\mathbf{goto}\ l_1) : l'\}\,;\,\{l_1 \triangleright p\} \vdash \{l \triangleright p\}}\ \textsf{goto}$$

$$\frac{}{\{l : (\mathbf{if}\ b\ \mathbf{goto}\ l_1) : l'\}\,;\,\{l_1 \triangleright p \wedge b, l' \triangleright p \wedge \neg b\} \vdash \{l \triangleright p\}}\ \textsf{if}$$

$$\frac{F_1\,;\,\Psi_1' \vdash \Psi_1 \quad F_2\,;\,\Psi_2' \vdash \Psi_2}{F_1 \cup F_2\,;\,\Psi_1' \cup \Psi_2' \vdash \Psi_1 \cup \Psi_2}\ \textsf{combine}$$

$$\frac{F\,;\,\Psi' \cup \{l \triangleright p\} \vdash \Psi \cup \{l \triangleright p\}}{F\,;\,\Psi' \vdash \Psi \cup \{l \triangleright p\}}\ \textsf{discharge}$$

$$\frac{\vdash \Psi_1' \Rightarrow \Psi_2' \quad F\,;\,\Psi_2' \vdash \Psi_2 \quad \vdash \Psi_2 \Rightarrow \Psi_1}{F\,;\,\Psi_1' \vdash \Psi_1}\ \textsf{weaken}$$

$$\boxed{\vdash \Psi_1 \Rightarrow \Psi_2}$$

$$\frac{m \geq n}{\vdash \{l_1 \triangleright p_1, \ldots, l_m \triangleright p_m\} \Rightarrow \{l_1 \triangleright p_1, \ldots, l_n \triangleright p_n\}}\ \textsf{s-width}$$

$$\frac{\vdash_{\mathfrak{D}} p' \Rightarrow p}{\vdash \Psi \cup \{l \triangleright p\} \Rightarrow \Psi \cup \{l \triangleright p'\}}\ \textsf{s-depth}$$

Figure 2.5: $\mathcal{L}_c$: Rules

Composition rules. The strength of $\mathcal{L}_c$ is its composition rules. These rules can compose judgments on individual statements to form properties of the combined statement. By internalizing control flow of the combined statement, these composition rules allows modular reasoning.

Figure 2.5 presents $\mathcal{L}_c$'s composition rules. We use an example in Figure 2.6 to illustrate these composition rules. Assume we already have two individual statements, depicted in Figure 2.6. The first one is an increment-by-one statement. If

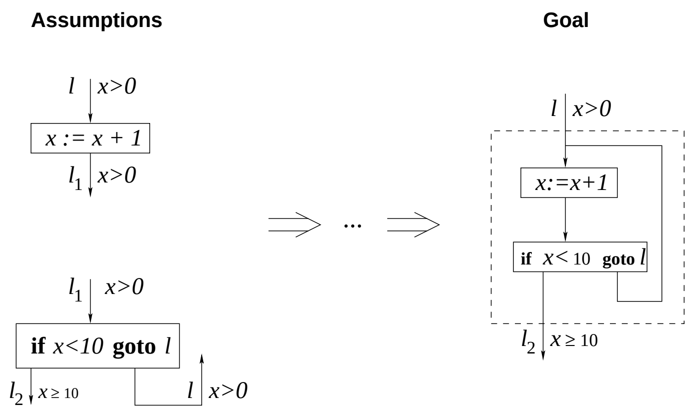


Figure 2.6: An example to illustrate $\mathcal{L}_c$'s composition rules

$x > 0$ when entering this statement, then $x > 0$ is still true after the completion of the statement. The second statement is **if** $x < 10$ **goto** l. It has one entry, but two exits. The entries and exits are also associated with assertions, which are shown in the figure. The goal is to combine these two statements together to form a property of the two-statement block. Notice that the two-statement block is effectively a repeat-until loop: it keeps incrementing x until x reaches 10. For this repeat-until loop, our goal is to prove that if $x > 0$ before entering the block, then $x \geq 10$ after the completion of the block.

The steps to derive the goal from the assumptions are presented in Figure 2.7.

In step 1, we use a rule called combine in Figure 2.5. When combining two fragment sets, F_1 and F_2, the combine rule makes the union of the entries of F_1 and F_2 the entries of the combined fragment; the same goes for the exits. For the example in Figure 2.7, since both statements have only one entry, we have two entries after the combine rule. Since the first statement has one exit, and the second statement has two exits, we have three exits after the combine rule.

After combining fragments, there may be some label that is both an entry and an exit. For example, the label l after the step 1 in Figure 2.7 is both an entry and an exit. Furthermore, the entry and the exit for l carry the same assertion: $x > 0$. In such a case, the discharge rule in Figure 2.5 can eliminate the label l as an exit. Formally, the discharge rule[2] states that if some $l \triangleright p$ appears on both the left and the right of the $\vdash$, then it can be removed from the left; Remember exits are on the left, so this rule removes an exit.

The label l_1 is also both an entry and an exit, and the entry and the exit carry the same assertion. The discharge rule can remove l_1 as an exit as well. Therefore, the step 2 in Figure 2.7 is to apply the discharge rule twice to remove both l and l_1 as exits. After this step, only one exit is left.

In the last step, we remove l_1 as an entry using the weaken rule. The weaken rule uses a relation between two sets of label continuations:

$$\vdash \Psi_1 \Rightarrow \Psi_2,$$

which is read as Ψ_1 is a stronger set of label continuations than Ψ_2.

The rule s-width in Figure 2.5 states that a set of label continuations is stronger than its subset. Therefore, $\vdash \{l_1 \triangleright (x > 0), l \triangleright (x > 0)\} \Rightarrow \{l \triangleright (x > 0)\}$ is derivable. Using this result and the weaken rule, the step 3 in Figure 2.7 removes the label l_1 as an entry.

After these steps, we have one entry and one exit left for the repeat-until loop, and we have proved the desired property for the loop.

[2] Careful readers may notice that the discharge rule, when read intuitively, seems wrong. In Section 2.3, we will develop for $F\,;\,\Psi' \vdash \Psi$ a semantics, which treats Ψ' and Ψ differently, so that the discharge rule can be justified.

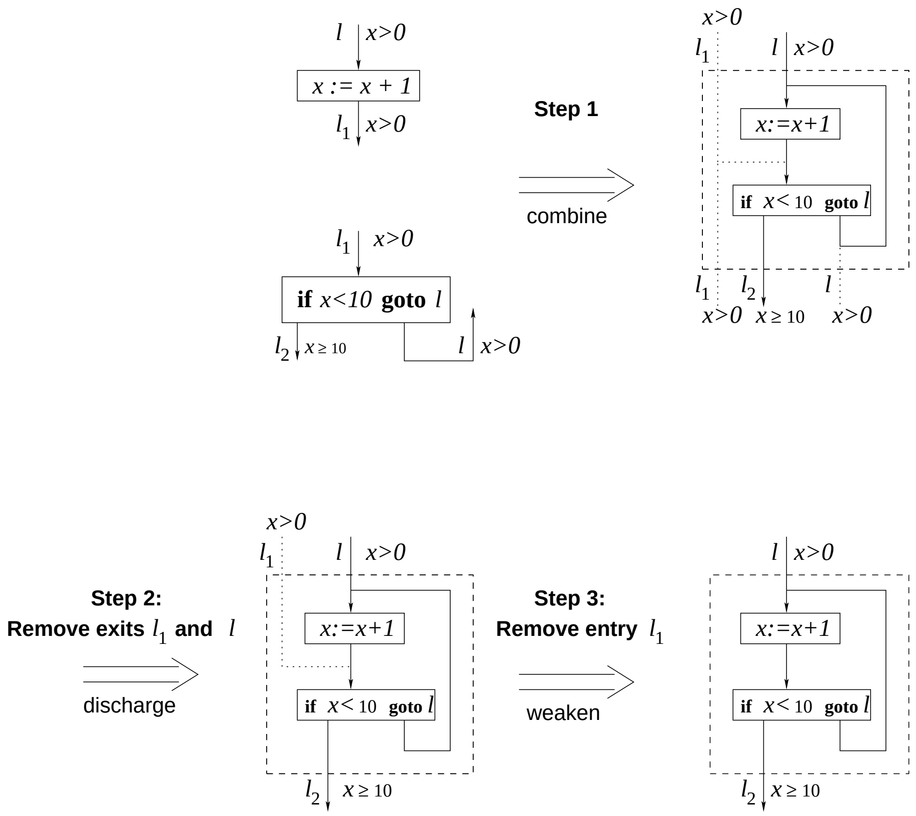


Figure 2.7: The steps to derive the example in Figure 2.6

The example in Figure 2.7 has used almost all composition rules, except for the s-depth rule. The s-depth rule states that a label continuation with a weaker precondition is stronger than the continuation with a stronger precondition. The rule is contravariant over the preconditions. An example of using this rule and the weaken rule is to derive $F\,;\,\Psi' \vdash \{l \triangleright p \wedge q\}$ from $F\,;\,\Psi' \vdash \{l \triangleright p\}$.

One issue to clarify is that, in step 2 of Figure 2.7, the discharge rule eliminates the label l_1 as an exit, and then in step 3, the label is eliminated as an entry. It may seem unnecessary to have separate steps for eliminating a label as an exit, and for eliminating the same label as an entry. However, the other label l in Figure 2.7 shows that when a label appears both as an entry and an exit, it may suggest

$$\dfrac{\dfrac{\dfrac{\overline{(1)}^{\,*} \quad \{l_1 : (s) : l_2\}\,;\,\{l_2 \rhd p\} \vdash \{l_1 \rhd p \wedge b\} \quad \overline{(2)}\;\textsf{goto}}{\begin{array}{c}\{l : (\textbf{while } b \textbf{ do } s) : l'\}\,;\,\{l \rhd p,\ l_1 \rhd p \wedge b,\ l_2 \rhd p,\ l' \rhd p \wedge \neg b\} \\ \vdash \{l \rhd p,\ l_1 \rhd p \wedge b,\ l_2 \rhd p\}\end{array}}\;\textsf{combine}}{\{l : (\textbf{while } b \textbf{ do } s) : l'\}\,;\,\{l' \rhd p \wedge \neg b\} \vdash \{l \rhd p,\ l_1 \rhd p \wedge b,\ l_2 \rhd p\}}\;\textsf{discharge}}{\{l : (\textbf{while } b \textbf{ do } s) : l'\}\,;\,\{l' \rhd p \wedge \neg b\} \vdash \{l \rhd p\}}\;\textsf{weaken}$$

(1)= $\{l : (\textbf{if } \neg b \textbf{ goto } l') : l_1\}\,;\,\{l' \rhd p \wedge \neg b,\ l_1 \rhd p \wedge b\} \vdash \{l \rhd p\}$
(2)= $\{l_2 : (\textbf{goto } l) : l'\}\,;\,\{l \rhd p\} \vdash \{l_2 \rhd p\}$

*The judgment is derived from the if rule and the weaken rule, assuming that $\vdash_{\mathfrak{D}} p \wedge \neg\neg b \Rightarrow p \wedge b$.

Figure 2.8: Derivation of a rule for "**while** b **do** s", whose definition is in Figure 2.2

a loop in the control flow; in this case, we need to preserve the label as the loop entry. Furthermore, we sometimes even want to keep l_1 as an entry. In unstructured control flow, it is possible to jump into the middle of a loop and exposing l_1 as an entry is necessary for other fragments to jump to it.

Rules for common control-flow structures. Figure 2.5 does not include a rule for sequential composition, since it is derivable by the composition rules in $\mathcal{L}_c$:

$$\dfrac{\dfrac{\dfrac{\{l_1 : (s_1) : l_2\}\,;\,\{l_2 \rhd p_2\} \vdash \{l_1 \rhd p_1\} \qquad \{l_2 : (s_2) : l_3\}\,;\,\{l_3 \rhd p_3\} \vdash \{l_2 \rhd p_2\}}{\{l_1 : (s_1; s_2) : l_3\}\,;\,\{l_2 \rhd p_2, l_3 \rhd p_3\} \vdash \{l_1 \rhd p_1, l_2 \rhd p_2\}}\;\textsf{combine}}{\{l_1 : (s_1; s_2) : l_3\}\,;\,\{l_3 \rhd p_3\} \vdash \{l_1 \rhd p_1, l_2 \rhd p_2\}}\;\textsf{discharge}}{\{l_1 : (s_1; s_2) : l_3\}\,;\,\{l_3 \rhd p_3\} \vdash \{l_1 \rhd p_1\}}\;\textsf{weaken}$$

In the same spirit, $\mathcal{L}_c$ can derive rules for many other control-flow structures such as **if** b **then** s_1 **else** s_2, **while** b **do** s and **repeat** s **until** b. Figure 2.8 presents the derivation of a rule for **while** b **do** s.

Name	Domain Construction
values, v	Val is a nonempty domain
addresses, n	$\mathit{Addr} = \mathbb{N}$
instr. memories, π	$\mathit{IM} = \mathit{Addr} \rightarrow \mathit{PrimStmt} \cup \{\textbf{illegal}\}$
data memories, m	$\mathit{DM} = \mathit{Var} \rightarrow \mathit{Val}$
states, σ	$\Sigma = \mathit{Addr} \times \mathit{IM} \times \mathit{DM}$
label maps, θ	$\mathit{LMap} = \mathit{Label} \rightarrow \mathit{Addr}$

where $\mathbb{N}$ is the domain of natural numbers.

Figure 2.9: Semantic domains

2.3 Foundational Semantics

In this section, we first present an operational semantics for the language in Figure 2.3. Then, we develop a continuation-style semantics for $\mathcal{L}_c$.

2.3.1 Operational Semantics for the Language

The operational semantics assumes an interpretation $\oint$ of the primitive symbols in *OPSym* and *RSym* in the following way: *Val* is a nonempty domain; for each *op* in *OPSym*, its semantics, $\underline{op}$, is a function in $(\mathit{Val}^{ar(op)} \rightarrow \mathit{Val})$; for each *re* in *RSym*, $\underline{re}$ is a relation $\subset \mathit{Val}^{ar(re)}$, where $ar(op)$ is the arity of the operator *op*.

A machine state is a triple, (pc, π, m): a program counter pc, which is an address; an instruction memory π, which maps addresses to primitive statements or to an **illegal** statement; a data memory[3] m, which maps variables to values. Figure 2.9 lists the relevant semantic domains.

Before presenting the operational semantics, we introduce some notation. For a state σ, the notation control(σ), i_of(σ), and m_of(σ) projects σ into its program

[3]We separate instruction memory and data memory for a simple presentation. Our implementation on SPARC stores instructions and data into the same memory, and maintains an invariant that instructions never get modified.

$(\mathrm{pc}, \pi, m) \mapsto_\theta \sigma$ where	
if $\pi(\mathrm{pc}) =$	then $\sigma =$
$x := e$	$(\mathrm{pc}+1, \pi, m[x \mapsto \mathcal{V}\,[\![e]\!]\,m])$
goto l	$(\theta(l), \pi, m)$
if b **goto** l	$\begin{cases} (\theta(l), \pi, m) & \text{if } \mathcal{B}\,[\![b]\!]\,m = \mathrm{tt} \\ (\mathrm{pc}+1, \pi, m) & \text{otherwise} \end{cases}$

where

$$\mathcal{V} : \mathit{Exp} \to \mathit{DM} \to \mathit{Val}$$
$$\mathcal{B} : \mathit{BExp} \to \mathit{DM} \to \{\mathrm{tt}, \mathrm{ff}\}$$

and their definitions are

$$\mathcal{V}\,[\![x]\!]\,m \triangleq m\,[\![x]\!]$$
$$\mathcal{V}\,[\![op(e_1, \ldots, e_{ar(op)})]\!]\,m \triangleq \underline{op}(\mathcal{V}\,[\![e_1]\!]\,m, \ldots, \mathcal{V}\,[\![e_{ar(op)}]\!]\,m).$$
$$\mathcal{B}\,[\![\mathbf{true}]\!]\,m \triangleq \mathrm{tt}$$
$$\mathcal{B}\,[\![b_1 \vee b_2]\!]\,m \triangleq \begin{cases} \mathrm{tt} & \text{if } \mathcal{B}\,[\![b_1]\!]\,m = \mathrm{tt} \text{ or } \mathcal{B}\,[\![b_2]\!]\,m = \mathrm{tt} \\ \mathrm{ff} & \text{otherwise} \end{cases}$$
$$\mathcal{B}\,[\![\neg b]\!]\,m \triangleq \begin{cases} \mathrm{tt} & \text{if } \mathcal{B}\,[\![b]\!]\,m = \mathrm{ff} \\ \mathrm{ff} & \text{otherwise} \end{cases}$$
$$\mathcal{B}\,[\![re(e_1, \ldots, e_{ar(re)})]\!]\,m \triangleq$$
$$\begin{cases} \mathrm{tt} & \text{if } \langle \mathcal{V}\,[\![e_1]\!]\,m, \ldots, \mathcal{V}\,[\![e_{ar(re)}]\!]\,m \rangle \in \underline{re} \\ \mathrm{ff} & \text{otherwise} \end{cases}$$

Figure 2.10: Operational semantics

counter, instruction memory, and data memory, respectively. For a mapping m, the notation $m[x \mapsto v]$ denotes a new mapping that maps x to v and leaves other slots unchanged.

The operational semantics for the language is presented in Figure 2.10 as a step relation $\sigma \mapsto_\theta \sigma'$ that executes the statement pointed by the program counter. The operational semantics is conventional, except that it is parametrized over a label map $\theta \in \mathit{LMap}$, which maps abstract labels to concrete addresses. When the next statement to execute is **goto** l, the label map θ is used to change the control to $\theta(l)$.

In the operational semantics, if the current statement in a state σ is an **illegal** statement, then σ has no next state to step to; such a state is called a stuck state. If a state σ will not reach a stuck state within k steps, it is *safe for k steps*:

$$\begin{aligned} &\mathrm{safe\_state}(\sigma, k) \triangleq \\ &\quad \forall \sigma' \in \Sigma. \forall j < k.\ \ \sigma \mapsto_{\theta}^{j} \sigma' \ \Rightarrow\ \exists \sigma''.\ \sigma' \mapsto_{\theta} \sigma'', \end{aligned}$$

where $\mapsto_{\theta}^{j}$ denotes j steps being taken.

2.3.2 Semantics of $\mathcal{L}_c$

The semantics of $\mathcal{L}_c$ is centered on an interpretation of the judgment $F\,;\,\Psi' \,\vdash\, \Psi$. Before giving a rigorous definition, we first present an informal overview.

We have discussed a simplified interpretation of $F\,;\,\Psi' \,\vdash\, \Psi$: for the set of fragments F, if Ψ' is true (according to some appropriate definition), then Ψ is true. However, this interpretation is not sufficient to justify the soundness of $\mathcal{L}_c$, because of the discharge rule. When both Ψ' and Ψ in the discharge rule are empty sets, it becomes

$$\frac{F\,;\,\{l \triangleright p\} \,\vdash\, \{l \triangleright p\}}{F\,;\,\emptyset \,\vdash\, \{l \triangleright p\}}$$

According to the simplified interpretation, the above rule is like stating "from $A \Rightarrow A$, derive A", which is clearly unsound.

The problem is not that $\mathcal{L}_c$ is intrinsically unsound, but that the simplified interpretation is too weak to utilize invariants implicitly in $\mathcal{L}_c$. The interpretation that we use is a much stronger one. The basic idea is based on a notion of label continuations being *approximately* true. The judgment $F\,;\,\Psi' \,\vdash\, \Psi$ is interpreted as, by assuming the truth of Ψ' at a lower approximation, Ψ is true at a higher approximation. In this inductive interpretation, Ψ' and Ψ are treated differently, and it allows the discharge rule to be justified by induction.

Label continuations being approximately true. We first introduce a semantic function, $\mathcal{A}$, which gives a meaning to assertions:

$$\mathcal{A} : \mathit{Assertion} \to \mathit{DM} \to \{\mathrm{tt}, \mathrm{ff}\}$$

$$\mathcal{A}\, [\![\exists x.p]\!]\, m \triangleq \begin{cases} \mathrm{tt} & \text{if } \exists d \in \mathit{Val}.\ \mathcal{A}\, [\![p[d/x]]\!]\, m = \mathrm{tt} \\ \mathrm{ff} & \text{otherwise.} \end{cases}$$

when $p = p_1 \vee p_2$, or $\neg p_1$, or $re(e_1, \ldots, e_{ar(re)})$, the definition of "$\mathcal{A}\, [\![p]\!]\, m$" is the same as the definition of $\mathcal{B}$ (in Figure 2.10), except every occurrence of $\mathcal{B}$ is replaced by $\mathcal{A}$.

Next, we present a notion, $\sigma; \theta \models_k l \triangleright p$, to mean that label continuation $l \triangleright p$ is k-approximately true in state σ relative to label map θ:

$$\begin{aligned} &\sigma; \theta \models_k l \triangleright p \ \triangleq \\ &\quad \forall \sigma' \in \Sigma. \\ &\qquad \sigma \mapsto_\theta^* \sigma' \ \wedge\ \mathrm{control}(\sigma') = \theta(l) \ \wedge\ \mathcal{A}\, [\![p]\!]\, (\mathrm{m\_of}(\sigma')) = \mathrm{tt} \\ &\qquad\quad \Rightarrow\ \mathrm{safe\_state}(\sigma', k) \end{aligned} \tag{2.2}$$

where $\mapsto_\theta^*$ denotes multiple steps being taken.

There are several points that need to be clarified about the definition of $\sigma; \theta \models_k l \triangleright p$. First, by this definition, $l \triangleright p$ being a true label continuation in σ to approximation k means that the state is safe to execute for k steps. In other words, the state will not get stuck within k steps.

Second, the definition is relative to a label map θ, which is used to translate the abstract label l to its concrete address.

Last, the definition quantifies over all future states σ' that σ can step to (including σ itself). The reason is that if $\sigma; \theta \models_k l \triangleright p$, provided that p is satisfied, it should be safe to jump to location l, not just now, but also in the future. In other words, if $l \triangleright p$ is true in the current state, it should also be true in all future states. Therefore, the definition of $\sigma; \theta \models_k l \triangleright p$ has to satisfy the following lemma:

Lemma 2.1 *If $\sigma \mapsto_\theta^* \sigma'$, and $\sigma; \theta \models_k l \triangleright p$, then $\sigma'; \theta \models_k l \triangleright p$.*

By quantifying over all future states, the definition of $\sigma; \theta \models_k l \triangleright p$ satisfies the above lemma. On this aspect, the semantics of $\sigma; \theta \models_k l \triangleright p$ is similar to the Kripke model [62, Ch 2.5] of intuitionistic logic: knowledge is preserved from current states to future states.

The semantics of a single label continuation is extended to a set of label continuations:

$$\sigma; \theta \models_k \Psi \;\triangleq\; \forall (l \triangleright p) \in \Psi.\ \sigma; \theta \models_k l \triangleright p$$

Loading statements. The predicate $\text{loaded}(F, \pi, \theta)$ describes the loading of a fragment set F into an instruction memory π with respect to a label mapping θ:

$$\begin{aligned}
&\text{loaded}(F, \pi, \theta) \;\triangleq \\
&\quad \forall (l : (t) : l') \in F. \\
&\qquad \pi(\theta(l)) = t \;\wedge\; \theta(l') = \theta(l) + 1 \;\wedge\; \big(\forall l \in \text{exits}(F).\ \theta(l) \notin \text{addr}(F, \theta)\big).
\end{aligned}$$

Note that some θ are not valid with respect to F. For example, if $F = \{l : (x := 1) : l'\}$, and θ maps l to address 100, then θ has to map l' to the address 101 to be consistent. This is the reason why the definition requires that $\theta(l') = \theta(l) + 1$.

In the definition of $\text{loaded}(F, \pi, \theta)$, the notation $\text{addr}(F, \theta)$ denotes the address space of F relative to θ:

Definition 2.2 *(Address space of F relative to θ.) The address space of a fragment set F relative to a label mapping θ is:*

$$\text{addr}(F, \theta) \triangleq \{ x \in \mathbb{N} \mid \exists (l : (t) : l') \in F.\ x = \theta(l) \}$$

Therefore, the last conjunct in the definition of $\text{loaded}(F, \pi, \theta)$ requires[4] that θ maps exit labels to addresses that are not occupied by F. For example, suppose $F = \{ l : (\textbf{goto}\ l_1) : l' \}$, and $\theta(l) = 100$. Since $l_1 \in \text{exits}(F)$, then θ cannot map l_1 to 100—the address has been occupied by F.

We define a relation, $F\,;\,\Psi' \models \Psi$, to model the semantic meaning of $F\,;\,\Psi' \vdash \Psi$.

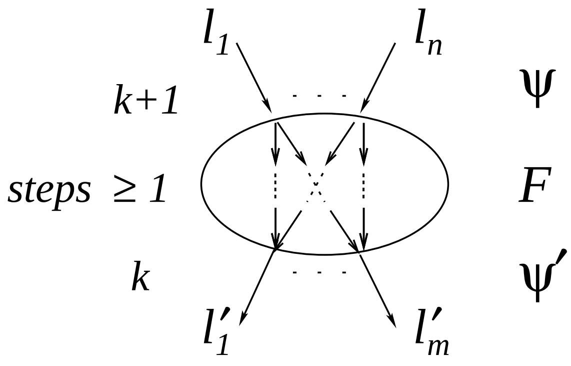
Semantics of the judgment $F\,;\,\Psi' \vdash \Psi$. We define a relation, $F\,;\,\Psi' \models \Psi$, which is the semantic modeling of $F\,;\,\Psi' \vdash \Psi$.

$$\begin{aligned} & F\,;\,\Psi' \models \Psi \triangleq \\ & \quad \forall \sigma \in \Sigma, \theta \in \mathit{LMap}.\ \text{loaded}(F, \text{i\_of}(\sigma), \theta) \Rightarrow \\ & \quad\quad \forall k \in \mathbb{N}.\ \big(\sigma; \theta \models_k \Psi' \ \Rightarrow\ \sigma; \theta \models_{k+1} \Psi\big). \end{aligned}$$

[4]This requirement is not needed for the proof of the soundness theorem.

The definition quantifies over all label maps θ and all states σ such that F is loaded in the state with respect to θ. It derives the truth of Ψ to approximation $k+1$, from the truth of Ψ' to approximation k. This inductive definition allows the discharge rule to be proved by induction over k.

We have given $F\,;\,\Psi' \models \Psi$ a strong definition. But the question is what about rules other than the discharge rule. Do they support such a strong semantics? The answer is yes for $\mathcal{L}_c$, because of one implicit invariant—for any judgment $F\,;\,\Psi' \vdash \Psi$ that is derivable, it takes at least one computation step from labels in Ψ to reach labels in Ψ'. In other words, it takes at least one step from entries of F to reach an exit of F. Because of this invariant, although Ψ' is assumed to be only true at a lower approximation, k, we can still show the truth of Ψ at a higher approximation, $k+1$. The following figure depicts the relationship between this invariant and $F\,;\,\Psi' \models \Psi$.



Finally, since $\mathcal{L}_c$ also contains rules for deriving $\vdash \Psi \Rightarrow \Psi'$ and $\vdash_{\mathcal{D}} p$, we define relations, $\models \Psi \Rightarrow \Psi'$ and $\models p$, to model their meanings, respectively. Their definitions are straightforward:

$$\begin{aligned}&\models \Psi \Rightarrow \Psi' \;\triangleq\\ &\quad \forall \sigma \in \Sigma, \theta \in \mathit{LMap}, k \in \mathbb{N}.\ (\sigma; \theta \models_k \Psi) \Rightarrow (\sigma; \theta \models_k \Psi')\end{aligned}$$

$$\models p \;\triangleq\; \forall m \in \mathit{DM}.\ \mathcal{A}\,[\![p]\!]\,m = \mathrm{tt}$$

2.4 Properties of $\mathcal{L}_c$

Theorems about the soundness and completeness of $\mathcal{L}_c$ are presented in this section. We will also briefly discuss the relationship between the continuation-style semantics for $\mathcal{L}_c$ and another semantics.

As a start, since $\mathcal{L}_c$ is parametrized by a deduction system $\mathcal{D}$, which derives formulas in the assertion language, it is necessary to assume properties of $\mathcal{D}$ before proving properties of $\mathcal{L}_c$.

Definition 2.3

- *If* $\vdash_{\mathcal{D}} p \;\Rightarrow\; \models p$*, then* $\mathcal{D}$ *is* sound*.*
- *If* $\models p \;\Rightarrow\; \vdash_{\mathcal{D}} p$*, then* $\mathcal{D}$ *is* complete*.*

2.4.1 Soundness

The soundness theorem for $\mathcal{L}_c$ is

Theorem 2.4 *(Soundness.) Assume* $\mathcal{D}$ *is sound. If* $F\,;\,\Psi \vdash \Psi'$*, then* $F\,;\,\Psi \models \Psi'$*.*

The proof is by induction over the derivation of $F\,;\,\Psi \vdash \Psi'$. The most difficult case is the discharge rule, which is proved as follows. Proofs of other rules are included in Appendix A.1.

We first present an auxiliary lemma.

Lemma 2.5 *For any $\sigma \in \Sigma$, $\theta \in \mathit{LMap}$, $k \in \mathbb{N}$ and $\Psi \in \mathit{FragSet}$,*

(i) $\sigma; \theta \models_0 \Psi$.

(ii) if $\sigma; \theta \models_k \Psi$, *then* $\sigma; \theta \models_j \Psi$ *for all* $j \leq k$.

Proof. Straightforward from the definition of $\sigma; \theta \models_k \Psi$.

Lemma 2.6

If $F\,;\,\Psi' \cup \{l \triangleright p\} \models \Psi \cup \{l \triangleright p\}$, *then* $F\,;\,\Psi' \models \Psi \cup \{l \triangleright p\}$.

Proof. To prove $F\,;\,\Psi' \models \Psi \cup \{l \triangleright p\}$, pick $\sigma \in \Sigma$ and $\theta \in \mathit{LMap}$. Assume

$$\mathrm{loaded}(F, \mathrm{i\_of}(\sigma), \theta). \tag{2.6.1}$$

The goal is $\forall k \in \mathbb{N}.\ \big(\sigma; \theta \models_k \Psi' \Rightarrow \sigma; \theta \models_{k+1} \Psi \cup \{l \triangleright p\}\big)$. We prove it by induction over the natural number k.

For the base case, assume

$$\sigma; \theta \models_0 \Psi', \tag{2.6.2}$$

and prove that $\sigma; \theta \models_1 \Psi \cup \{l \triangleright p\}$.

Lemma 2.5 (i) gives

$$\sigma; \theta \models_0 \{l \triangleright p\}. \tag{2.6.3}$$

Together with 2.6.2, we have

$$\sigma;\theta \models_0 \Psi' \cup \{l \triangleright p\}. \tag{2.6.4}$$

Now from the assumption, $F\,;\,\Psi' \cup \{l \triangleright p\} \models \Psi \cup \{l \triangleright p\}$, the result 2.6.1, and the result 2.6.4, we get $\sigma;\theta \models_1 \Psi \cup \{l \triangleright p\}$, which is the goal for the base case.

For the inductive case, assume the induction hypothesis is true for k:

$$\sigma;\theta \models_k \Psi' \ \Rightarrow \sigma;\theta \models_{k+1} \Psi \cup \{l \triangleright p\}. \tag{2.6.5}$$

The goal is to prove that the induction hypothesis is true for $k+1$:

$$\sigma;\theta \models_{k+1} \Psi' \ \Rightarrow \sigma;\theta \models_{k+2} \Psi \cup \{l \triangleright p\}.$$

Thus, assume

$$\sigma;\theta \models_{k+1} \Psi'. \tag{2.6.6}$$

Lemma 2.5 (ii) gives

$$\sigma;\theta \models_k \Psi'. \tag{2.6.7}$$

From the induction hypothesis 2.6.5 and the result 2.6.7, we have

$$\sigma;\theta \models_{k+1} \Psi \cup \{l \triangleright p\}, \tag{2.6.8}$$

from which the following is derivable:

$$\sigma;\theta \models_{k+1} \{l \triangleright p\}. \tag{2.6.9}$$

Together with 2.6.6, we have

$$\sigma; \theta \models_{k+1} \Psi' \cup \{l \triangleright p\}. \tag{2.6.10}$$

Now, use the assumption $F\,;\, \Psi' \cup \{l \triangleright p\} \ \models\ \Psi \cup \{l \triangleright p\}$, together with 2.6.1 and 2.6.10, to derive

$$\sigma; \theta \models_{k+2} \Psi \cup \{l \triangleright p\},$$

which is the goal for the inductive case. □

2.4.2 Connection with Conventional Hoare Logic Semantics

Before discussing the completeness of $\mathcal{L}_c$, we first present a connection between our semantic for $\mathcal{L}_c$ and the conventional Hoare Logic Semantics.

Our semantics of $F\,;\, \Psi' \vdash \Psi$ is in *continuation style*. For example, the meaning of $F\,;\, \{l' \triangleright q\} \ \models\ \{l \triangleright p\}$ assumes $l' \triangleright q$ is a true label continuation (to a lower approximation), and infer that $l \triangleright p$ is a true label continuation.

The conventional interpretation of a Hoare triple $\{p\}s\{q\}$ is that if the state before execution of s satisfies the assertion p, then the state after execution (if there is one) satisfies q. We call this style the *direct style*. The direct-style semantics positively asserts that the exit state satisfies the postcondition q.

Although these two styles are different, one has the feeling that there is some connection. The connection can be intuitively seen as follows. If $\{p\}s\{q\}$ is derivable in Hoare logic, then s may be thought of as a procedure for transforming the assertion p to q. On the other hand, if $F\,;\, \{l' \triangleright q\} \ \models\ \{l \triangleright p\}$ is derivable in $\mathcal{L}_c$, then s may be thought of as a procedure for transforming the assertion $\neg q$ to $\neg p$ (because of the connection between continuations and negation). In classical logic,

$p \Rightarrow q$ is equivalent to $\neg q \Rightarrow \neg p$. Therefore, there must be a connection between the continuation-style semantics and the direct-style semantics.

Next, we define a relation $\models \{\Psi\}F\{\Psi'\}$, which extends the direct-style semantics to the case of multiple-entry and multiple-exit fragments. After that, we prove a theorem that relates the direct-style semantics to the continuation-style semantics.

Definition of $\models \{\Psi\}F\{\Psi'\}$. The partial-correctness interpretation of a Hoare triple $\{p\}s\{q\}$ is that if the state before execution of s satisfies the assertion p, then the state after execution (if there is one) satisfies q. We extend this direct-style semantics to the case of multiple-entry and multiple-exit fragments F:

$$\begin{array}{ccc} \Psi & & \Psi' \\ \{l_1 : p_1 & & \{l'_1 : p'_1 \\ \vdots & F & \vdots \\ l_n : p_n\} & & l'_m : p'_m\} \end{array}$$

We define a relation $\models \{\Psi\}F\{\Psi'\}$ to model the direct-style semantics. It means that if the state enters F through some label in Ψ with the corresponding precondition satisfied, and if F terminates, then the control of the ending state will be at one of the labels in Ψ', and the ending state will satisfy the corresponding postcondition.

To formalize this semantics, there is one technical difficulty: It needs to reason about the ending state of F. However, it is not straightforward to extract the ending state on von Neumann-style operational semantics. In some denotational semantics for Hoare Logic [57], F is modeled as a state-transformation function in the domain of $\Sigma \rightarrow \Sigma$. In this kind of semantics, given a state σ, the ending state is just $F(\sigma)$.

It is not straightforward to get the ending state for F on the operational semantics in Section 2.3.1. To see the difficulty, suppose F is loaded into the instruction memory of some state σ, whose control points to some entry in F. If we know that F takes j steps to finish, we could define the ending state to be any state σ' such that $\sigma \longmapsto^{j} \sigma'$. However, it is in general not possible to know j since F may contain loops.

Our idea is to define an ending state for F to be any state that first goes out of the address space occupied by the statement F. In other words, the first state whose control points to locations outside of F is an ending state for F.

To formalize the above idea, we introduce some definitions first.

Definition 2.7 *(Computation within the address space of F.) For a computation sequence $\sigma_0 \longmapsto_\theta \sigma_1 \longmapsto_\theta \cdots \longmapsto_\theta \sigma_j$, it is a computation within the address space of F if $\forall 0 \le i < j.\ \mathrm{control}(\sigma_i) \in \mathrm{addr}(F, \theta)$. We use the notation $\sigma_0 \overset{F,\theta}{\rightsquigarrow} \sigma_j$ to describe this.*

Note that the definition of $\sigma_0 \overset{F,\theta}{\rightsquigarrow} \sigma_j$ does not require that $\mathrm{control}(\sigma_j) \in \mathrm{addr}(F, \theta)$. If $\mathrm{control}(\sigma_j) \notin \mathrm{addr}(F, \theta)$, then σ_j is an ending state for F since it is the first one that goes out of the address space of F.

Now we define the relation $\models \{\Psi\}F\{\Psi'\}$:

Definition 2.8

$$
\begin{array}{l}
\models \{\Psi\}F\{\Psi'\} \triangleq \\
\quad \forall l \triangleright p \in \Psi,\ \sigma \in \Sigma,\ \theta \in \mathit{LMap}. \\
\quad\quad \mathrm{loaded}(F, \mathrm{i\_of}(\sigma), \theta)\ \wedge\ \mathrm{control}(\sigma) = \theta(l)\ \wedge\ \mathcal{A}\,[\![p]\!]\,\mathrm{m\_of}(\sigma) = \mathrm{tt} \\
\quad\quad\quad \Rightarrow \forall \sigma' \in \Sigma.\ \sigma \overset{F,\theta}{\rightsquigarrow} \sigma'
\end{array}
$$

$$
\begin{aligned}
&\Rightarrow (\mathrm{control}(\sigma') \in \mathrm{addr}(F,\theta) \;\Rightarrow\; \exists \sigma'' \in \Sigma.\ \sigma' \mapsto_{\theta}^{*} \sigma'') \,\wedge \\
&\quad (\mathrm{control}(\sigma') \notin \mathrm{addr}(F,\theta) \\
&\qquad\quad \Rightarrow\; \exists l' \triangleright p' \in \Psi'.\ \mathrm{control}(\sigma') = \theta(l') \;\wedge\; \mathcal{A}\,[\![p']\!]\,\mathrm{m\_of}(\sigma') = \mathrm{tt})
\end{aligned}
$$

This says that for any state σ, with F loaded, whose control is at some entry l and which satisfies the condition p, any computation sequence can either progress, when the last state is not an ending state, or the last state is an ending state, with control at some exit label and the corresponding condition satisfied.

Next, we present a theorem that relates the direct-style semantics, $\models \{\Psi\}F\{\Psi'\}$, to the continuation-style semantics, $F\,;\,\Psi' \models \Psi$. The theorem states that these two semantics are equivalent, but under certain assumptions. To present these assumptions, we first define some concepts.

Some concepts. The notation, labels(Ψ), denotes all labels mentioned in Ψ. The notation, entries(F), denotes all labels that are defined as entries in F. The notation, exits(F), denotes all labels that are possible exits of F, excluding those labels that are already in entries(F); exits are identified syntactically. Their rigorous definitions are as follows:

$$
\begin{aligned}
\mathrm{labels}(\Psi) &\triangleq \{\, l \mid \exists p.\ l \triangleright p \in \Psi \,\} \\
\mathrm{entries}(F) &\triangleq \{\, l \mid \exists t, l'.\ l : (t) : l' \in F \,\} \\
\mathrm{exits}(F) &\triangleq \{\, l' \mid \exists l, t.\ l : (t) : l' \in F \,\} \\
&\quad \cup \{\, l_1 \mid \exists l, l'.\ l : (\mathbf{goto}\ l_1) : l' \in F \,\} \\
&\quad \cup \{\, l_1 \mid \exists l, l'.\ l : (\mathbf{if}\ b\ \mathbf{goto}\ l_1) : l' \in F \,\} \\
&\quad - \mathrm{entries}(F)
\end{aligned}
$$

Some fragment sets are abnormal. For example, if a fragment set has a label that is defined twice, such as the following fragment set,

$$F = \{l : (x := 1) : l'_1, l : (x := 2) : l'_2\}.$$

Then no $\pi \in \mathit{IM}$ and $\theta \in \mathit{LMap}$ exists such that F can be loaded into π. The following concept restricts our attention to a normal subset of (F, Ψ', Ψ).

Definition 2.9

- *F is a normal fragment set if every label in* $\mathrm{entries}(F)$ *is defined exactly once in F.*
- *Ψ' is a normal exit label-continuation set relative to F if* $\mathrm{labels}(\Psi') = \mathrm{exits}(F)$*, and for each $l \in \mathrm{labels}(\Psi')$, only one $l \triangleright p$ in Ψ' exists.*
- *Ψ is a normal entry label-continuation set relative to F if* $\mathrm{labels}(\Psi) \subseteq \mathrm{entries}(F)$*, and for each $l \in \mathrm{labels}(\Psi)$, only one $l \triangleright p$ in Ψ exists.*
- *(F, Ψ', Ψ) is a normal triple if F is a normal fragment set, Ψ' is a normal exit label-continuation set and Ψ is a normal entry label-continuation set relative to F.*

Finally, the proof of the equivalence between $\models \{\Psi\}F\{\Psi'\}$ and $F\,;\Psi' \models \Psi$ depends on a notion expressing that the assertion language is *negatively testable* with respect to the statement language.

Definition 2.10

- *An assertion p is negatively testable by a sequence of statements s, if*

    1. *s terminates, when p is false before executing s,*
    2. *s diverges (in an infinite loop), when p is true before executing s.*

- *The assertion language, Assertion, is negatively testable by the statement language, Stmt, if every $p \in$ Assertion is negatively testable by a sequence of statements $s \in$ Stmt.*

- *If the assertion language is negatively testable by the statement language, let* **test**(p) *be a sequence of statements that can negatively test the assertion p.*

If the assertion language is the same as the language of boolean expressions, then it is negatively testable by the statement language: for every assertion p, the following statement satisfies the requirement in the definition.

$$l : \textbf{if } p \textbf{ goto } l$$

Another example of a negatively-testable assertion language is $\forall$-rudimentary formulas for arithmetic. A $\forall$-rudimentary formula has only one unbounded universal quantification at the front, and hence has the following form:

$$\forall x. \exists y < n_1. \forall z < n_2. \cdots \; p$$

where "$\cdots$" represents a sequence of bounded quantifiers (either universal or existential), and p is a quantifier-free formula.

For such a $\forall$-rudimentary formula, a sequence of statement that satisfies the requirement in Definition 2.10 can be a loop that enumerates x and exits if one instance of x makes the rest of the formula false.

We now present the theorem that relates the direct-style semantics to the continuation-style semantics.

Theorem 2.11 *Assume (F, Ψ', Ψ) is normal. Assume the assertion language is negatively testable by the statement language. Then, $\models \{\Psi\}F\{\Psi'\}$ is equivalent to $F\,;\,\Psi' \models \Psi$.*

The proof is in the Appendix A.2. We only point out that the proof is not constructive. First, it is a proof by contradiction. Second, for an assertion p, the proof chooses a sequence of statements, namely **test**(p), which can negatively test p; this needs the axiom of choice.

Although two styles of semantics are closely related, we prefer the continuation style for two reasons. First, the definition of $\models \{\Psi\}F\{\Psi'\}$ is technically clumsy since it needs to formally capture the state that is the first one whose control is out of the address space of F.

Second and more importantly, it is a challenge to adapt direct-style semantics to a language with first-class function pointers, which continuation-style semantics can easily accommodate. For example, to reason about the following program, the value of variable x can be modeled as a continuation—the same continuation as the one associated with l. With this modeling, the rule for **jmp** x requires x be a continuation—the same as the rule for **goto** l.

$$x := l;$$

$$\ldots$$

$$\textbf{jmp}\ x$$

We will discuss the model for indirect jumps in detail in Section 3.3.

2.4.3 Completeness

$\mathcal{L}_c$ may be incomplete due to some incompleteness of system $\mathcal{D}$. But there are other sources of incompleteness.

First of all, some fragment sets are abnormal. For example, if a fragment set has a label that is defined twice, such as the following fragment set,

$$F = \{l : (x := 1) : l'_1, l : (x := 2) : l'_2\},$$

then no $\pi \in IM$ and $\theta \in LMap$ exists such that loaded(F, π, θ) is true, since it is impossible to load two different statements into the same location $\theta(l)$. For such kind of fragment sets, because the definition of $F\,;\,\Psi' \models \Psi$ quantifies over all π and θ such that loaded(F, π, θ), the judgment $F\,;\,\Psi' \models \Psi$ is trivially true. $\mathcal{L}_c$ cannot prove $F\,;\,\Psi' \vdash \Psi$ for such kind of F.

Second, if a statement s is always in an infinite loop, then the following is always true:

$$\{l : (s) : l'\}\,;\,\emptyset \models \{l \triangleright p\}$$

However, $\mathcal{L}_c$ in general cannot derive $\{l : (s) : l'\}\,;\,\emptyset \vdash \{l \triangleright p\}$ for infinitely looping s, since $\mathcal{L}_c$ treats $l : (s) : l'$ as a one-entry and one-exit fragment, and judgments derivable in $\mathcal{L}_c$ for $\{l : (s) : l'\}$ in general assume a label continuation about l' on the left of $\vdash$. Since $\mathcal{L}_c$ is not intended to determine termination of fragments, a reasonable assumption when discussing completeness is to focus on those $F\,;\,\Psi' \models \Psi$ such that $\text{labels}(\Psi') = \text{exits}(F)$.

Therefore, when discussing the completeness, we restrict our attention to the normal set of (F, Ψ', Ψ), as defined in Definition 2.9.

Last, as pointed out by Cook [21], there is another way that a program logic can fail to be complete, and it is if the assertion language is not powerful enough to express invariants for the loops. Therefore, we must assume the assertion language is *expressive*. To formally define expressiveness, we first define a concept that expresses a *weakest precondition* of a label.

Definition 2.12 *Let $F \in \mathit{FragSet}$, $l \in \text{entries}(F)$, and $\Psi' \in \mathit{LContSet}$ such that* $\text{labels}(\Psi') = \text{exits}(F)$. *We say that p expresses the weakest precondition of the label l in the fragment set F with respect to Ψ' iff*

$$\forall m \in \mathit{DM}.\ \mathcal{A}\,[\![p]\!]\,m = \text{tt} \Leftrightarrow \left(\begin{array}{l} \forall \theta \in \mathit{LMap}, \pi \in \mathit{IM}, k \in \mathbb{N} \\ \quad \text{loaded}(F, \pi, \theta) \ \wedge\ \big((\theta(l), \pi, m); \theta \models_k \Psi'\big) \\ \qquad \Rightarrow \text{safe\_state}((\theta(l), \pi, m), k+1) \end{array}\right)$$

We write $p \simeq \text{wp}(l, F, \Psi')$ to express this.

The following lemma justifies our definition of the weakest precondition. Its proof is in Appendix A.3.

Lemma 2.13 *Let $p \in$ Assertion, $F \in$ FragSet, $l \in \mathrm{entries}(F)$, and $\Psi' \in$ LContSet. If $p \simeq \mathrm{wp}(l, F, \Psi')$, then*

(i) $F\,;\,\Psi' \models \{l \triangleright p\}$ *(i.e., p is a precondition)*

(ii) $\forall p' \in$ *Assertion.* $\big(F\,;\,\Psi' \models \{l \triangleright p'\} \;\Rightarrow\; \models p' \Rightarrow p\big)$ *(p is the weakest)*

With the concept of the weakest precondition, the expressiveness condition is defined as follows.

Definition 2.14 *(Expressiveness.) Let $\oint$ be an interpretation of the primitive relation and function symbols. We say that Assertion is expressive enough relative to FragSet and $\oint$ if for all fragments F, for all labels $l \in \mathrm{entries}(F)$, and all label invariants Ψ' such that $\mathrm{labels}(\Psi') = \mathrm{exits}(F)$, there is an assertion p such that $p \simeq \mathrm{wp}(l, F, \Psi')$.*

If the primitive symbols are such that the assertion language is the language of arithmetic, and if $\oint$ is the standard interpretation of the language of arithmetic, then, according to recursion theory, the computation from entries to exits in F are partial recursive functions in the free variables of F. A result in recursion theory is that every partial recursive function can be described by a formula in the language of arithmetic. From this we can infer that the assertion language is expressive relative to $\oint$.

Theorem 2.15 *(Completeness.)*
Assume $\mathcal{D}$ is complete and Assertion is expressive relative to $\oint$. Assume Assertion is negatively testable by the statement language. Assume (F, Ψ', Ψ) is normal. If $F\,;\,\Psi' \models \Psi$, then $F\,;\,\Psi' \vdash \Psi$.

See Appendix A.3 for the proof of the completeness theorem.

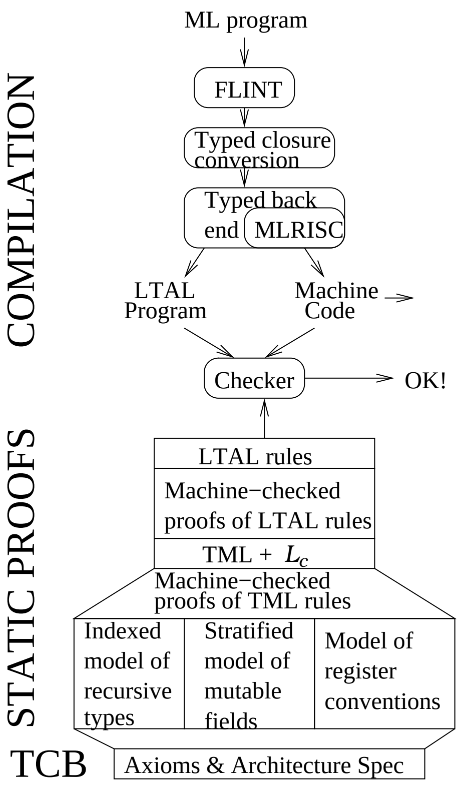
Chapter 3

From $\mathcal{L}_c$ to Typed Assembly Languages

We have introduced $\mathcal{L}_c$, which is a program logic for modularly verifying properties of machine-language programs. In this chapter, we discuss the role that $\mathcal{L}_c$ plays in the FPCC project at Princeton.

Overview of the FPCC system. Figure 3.1 shows the system setup of our FPCC system. The system translates ML programs into SPARC machine code, together with automatically generated foundational safety proof for the code.

The FPCC system follows the TAL's approach for automatic proof generation, except that we also provide a machine-checkable soundness proof for the type system. The system has three major components: a compiler that translates well-typed ML programs into a typed-assembly language, LTAL; the soundness proof of LTAL's type system; a checker that checks the soundness proof and run the type system



ML program
FLINT
Typed closure conversion
Typed back end
MLRISC
LTAL Program
Machine Code
Checker
OK!
LTAL rules
Machine−checked proofs of LTAL rules
TML + L_c
Machine−checked proofs of TML rules
Indexed model of recursive types
Stratified model of mutable fields
Model of register conventions
Axioms & Architecture Spec
COMPILATION
STATIC PROOFS
TCB

Figure 3.1: The FPCC system (figure from Chen et al. [17])

as a logic program to check the welltypedness of LTAL programs. Next, we discuss these three components in detail.

Compiler. The compiler is based on the Standard ML of New Jersey system. We call this compiler the FPCC-ML compiler. The FPCC-ML compiler transforms core ML (ML without the module system) into SPARC machine code with LTAL annotations. Interested readers should refer to Chen's thesis [16, ch7] for a detailed description of the FPCC-ML compiler.

Checker. The checker has two components. First, it includes a general purpose proof checker [10]. The proof checker can check the validity of any proof encoded in Logical Framework (LF) [34]. In our system, this proof checker is used to check the soundness proof of LTAL. After the soundness proof has been checked, the LTAL typing rules can be regarded as a set of Prolog-like clauses. Therefore, the second component of the checker is a simple Prolog interpreter [71], which runs LTAL's type system as a logic program to type check LTAL programs.

Soundness proof. The major research problem in the FPCC project is to prove the soundness of LTAL from higher-order logic and SPARC machine semantics, so that a typing derivation at the LTAL level can be mapped into a proof from the foundations of mathematics. This soundness proof is complicated and we organize it into layers for better understanding and maintenance. We next discuss the most important layer in the soundness proof.

An intermediate calculus in the soundness proof. To remove LTAL from TCB, we need to prove it is sound. On the other hand, the main design goals of

LTAL were to accommodate low-level optimizations, to have syntax-directed type checking, to deal with particulars in ML and SPARC—having a simple semantic soundness proof was not its original goal.

When proving the soundness of LTAL, we found it is easier to have an intermediate calculus. We first prove the intermediate calculus is sound from logic plus machine semantics. Then we prove LTAL is sound based on the lemmas provided by the intermediate calculus.

The intermediate calculus does not need to be decidable, since it is not used for automatic checking. It is also meant to be language-independent and machine-independent.

The intermediate calculus is required to be expressive so that features in LTAL, or other type systems, can be explained by it. It should provide orthogonal and primitive features, so that complicated features can be explained as combinations of primitive features.

The intermediate calculus in our system includes two parts. The first part is called Typed Machine Language (TML) [64], which is an expressive type theory with machine-checked soundness proofs. It has intersection types, union types, recursive types, mutable references, polymorphism, existentials, etc. Users of TML can utilize its rich set of type constructors and manipulate types using lemmas provided by TML.

The second part is $\mathcal{L}_c$. We have introduced it in Chapter 2 as a logic for modularly verifying properties of unstructured programs. In our system, we use the TML type theory as the assertion language for $\mathcal{L}_c$.

A roadmap of this chapter. The purpose of this chapter to illustrate the step between $\mathcal{L}_c$ and LTAL: We give LTAL's syntactic constructs semantics based on $\mathcal{L}_c$, and then prove LTAL is sound from the rules provided by $\mathcal{L}_c$.

For this purpose, in Section 3.2, we define a simple typed-assembly language, TAL_0, which is similar to LTAL in terms of the control-flow aspect. It has unconditional jumps, conditional jumps, and instruction blocks. We then construct a semantic model for TAL_0 from $\mathcal{L}_c$ and prove that TAL_0 is sound.

In the following sections, we will introduce a series of typed assembly languages on top of TAL_0. These typed assembly languages are used to illustrate some of the complexities in our real system. In particular, TAL_1 (Section 3.3) adds pc-relative jumps and indirect jumps. The language TAL_2 (Section 3.4) has parametric polymorphism and recursive types. The semantic modeling of TAL_2 requires us to deal with *virtual instructions*. The language TAL_3 (Section 3.5) can perform simple memory allocation and initialization. Its purpose is to illustrate an important technique in our proof—imaginary parts of states.

In the following sections, however, we will make simplifications of our real system for a clearer presentation. When it is important, we will point out the differences between our presentation and our real system.

3.1 Specifying Machine Semantics and Safety

Before we introduce a series of typed-assembly languages, we first discuss the specification of the SPARC machine and the safety policy. With this specification as the foundation, we will prove the typed-assembly languages we introduce are sound. The specification architecture in this section follows from Appel [7].

In Section 2.3.1, we have specified a machine model and proved that $\mathcal{L}_c$ is sound based on that machine model. Although that model is very low-level, in some ways it is still high-level compared to a real architecture: it has separate instruction memory and data memory, while a real architecture puts both instructions and data into the same memory; some of its instructions such as "$x := e$" are compiled into a sequence of more primitive machine instructions.

In this section, we introduce a von Neumann machine model, which is our modeling of concrete architectures. In the FPCC project, we model the SPARC architecture [42] and carry out all our proofs based on that model.

The machine model formally specifies the decoding and operational semantics for an architecture. Since the specifics of instruction encodings are not relevant to this presentation, we assume an abstract decode relation, $\mathrm{decode}(w, i)$, which decodes the word w to the machine instruction i.

A machine state σ consists of a *register bank* r and a *memory* m, both of which are modeled as functions from addresses (numbers) to contents (also numbers). Every register in the machine is assigned an index in the register bank. We assume registers 0–31 are general purpose registers. Special registers such as the program counter are assigned indexes that are greater than 31; we use pc for the index of the program counter. The notation $\mathrm{r\_of}(\sigma)$ and $\mathrm{m\_of}(\sigma)$ project σ into its register bank and memory, respectively.

A machine instruction is modeled by a relation between machine states (r, m) and (r', m'). For example, a load instruction (`ld`) is specified by

$$\begin{aligned}
&\texttt{ld}\ s, d \ \triangleq \\
&\quad \lambda(r,m),(r',m'). \\
&\qquad \big(r'(\mathrm{pc}) = r(\mathrm{pc}) + 4\big) \ \wedge\ \big(r'(d) = m(r(s))\big) \ \wedge\ \big(\forall x \notin \{\mathrm{pc}, d\}.\ r'(x) = r(x)\big) \\
&\qquad \wedge\ \mathrm{readable}(r(s)) \\
&\qquad \wedge\ \big(m' = m\big),
\end{aligned}$$

where $m' = m$ is an abbreviation for $\forall x.\ m'(x) = m(x)$. The semantics of "`ld` s, d" increments the program counter by four (on SPARC, the size of an instruction is four), loads the value at the memory address $r(s)$ into the register d, and the memory remains unchanged.

One important property of our machine semantics is that it is deliberately partial: unsafe operations are omitted from the semantics. Since the current version of the FPCC project uses *memory safety* as the safety policy, unsafe operations therefore are those reading from unreadable addresses, and those writing to unwritable addresses.

To model memory safety, we first introduce two predicates:

$$\begin{aligned}
\mathrm{readable} &: \mathbb{N} \to \mathit{Bool} \\
\mathrm{writable} &: \mathbb{N} \to \mathit{Bool}
\end{aligned}$$

The readable predicate tells which region of memory addresses is readable; the writable predicate tells which region is writable. In the FPCC project, we axiomatize[1] the set of readable and writable addresses, and the goal of FPCC is to prove that each memory-load instruction is reading from the readable addresses and each memory-write instruction is writing to the writable addresses.

The reader may have noticed that the definition of "$\texttt{ld}\ s, d$" uses the readable$(r(s))$ predicate to ensure that the address is readable. Because of this requirement, suppose in some state (r, m) the program counter points to a `ld` instruction that would, if executed, load from an address that is unreadable. Then, since our `ld` instruction requires that the address must be readable, there will not exist (r', m') such that $(\texttt{ld}\ s, d)(r, m)(r', m')$.

The machine operational semantics is modeled by a step relation, $\mapsto$, that steps from one state (r, m) to the next state (r', m'), which is the result of decoding the current machine instruction, and then executing the machine instruction:

$$(r, m) \mapsto (r', m') \triangleq \exists i.\ \text{decode}(m(r(\text{pc})), i)\ \wedge\ i(r, m)(r', m')$$

The step relation is partial; some states have no successor states and and we call them *stuck* states. A state is a stuck state if its program counter points to an integer that cannot be decoded into an instruction. Our modeling of machine instructions makes the step relation even more partial. For example, if the current instruction in a state is a load instruction that would load from an unreadable address, then the state is also a stuck state.

[1]In particular, the convention of the SML/NJ compiler specifies a heap area and a register-spilling area, using values of special registers, including an allocation pointer, a limit pointer and a stack pointer. Then, we axiomatize that both the heap area and the register-spilling area is both readable and writable.

Using this partial step relation, we can define safety; a state is safe for k steps if it will not reach a stuck state within k steps:

$$\begin{aligned}&\text{safe\_state}(\sigma, k) \triangleq \\ &\qquad \forall \sigma' \in \Sigma . \forall j < k.\ \ \sigma \mapsto^j \sigma' \ \Rightarrow\ \exists \sigma''.\ \sigma' \mapsto \sigma'',\end{aligned}$$

Safe programs. An assembly-language program P is a sequence of assembly instructions. We use the predicate loaded(P, σ) to specify that the program P is loaded into the state σ; machine integers are decoded into corresponding assembly instructions:

$$\text{loaded}(P, \sigma) \triangleq \forall i \in \text{dom}(P).\ \text{decode}(\text{m\_of}(\sigma)(4i), P(i))$$

Next, we define the safety of a program P:

$$\begin{aligned}\text{safe\_prog}(P) \ \triangleq\ & \forall \sigma.\ \text{loaded}(P, \sigma) \ \wedge\ \text{r\_of}(\sigma)(\text{pc}) = 0 \ \wedge\ \text{init\_cond}(\sigma) \\ & \Rightarrow\ \forall k.\ \text{safe\_state}(\sigma, k).\end{aligned} \tag{3.1}$$

A program P is safe if any state σ satisfying the following is a safe state: P is loaded inside σ; the program counter initially points to address zero; when the program begins executing, some initial condition holds on the state.

There are several points worth further explanation. First, our real system proves a program is safe no matter where it is loaded (that is, the program is position-independent). For simplification, this presentation assumes that the program is always loaded to address zero.

Second, the init_cond(σ) predicate axiomatizes the initial state. Among other things, it axiomatizes a return address. A program P can always jump to that address to reach a safe state.

$$\text{return\_addr}(x) \triangleq \forall r, m.\, r(\text{pc}) = x \Rightarrow \forall k.\, \text{safe\_state}((r, m), k).$$

In our implementation, we designate register 15 to store the return address and thus return_addr(r_of(σ)(15)) is part of init_cond(σ). By modeling a return address, our safe_prog(P) definition also allows a terminating program to be safe, as long as it jumps to the return address in the end.

There are other components in the init_cond(σ); one is to specify all register and memory values in the initial state are 32-bit integers. We do not go into details.

We have specified the semantics of only the load instruction so far. In the following sections, we will introduce a series of typed assembly languages so that we can add more and more complex features. Each typed assembly language has its own instruction set. Therefore, we defer the introduction of particular instruction sets and their semantics to later sections. Finally, in this chapter, we will make some simplifications and changes to the SPARC instructions. For example, instead of two arguments, the real load instruction on SPARC has three. When we discuss the implementation in the next chapter, we will present lemmas for real SPARC instructions.

3.2 TAL$_0$

In this section, we introduce a simple typed assembly language — TAL$_0$. The following few sections will cover its syntax and a type system.

The purpose of introducing TAL_0 is two-fold. First, we will use TAL_0 to illustrate the step of bridging the gap between a real typed assembly language and our logic for control flow — $\mathcal{L}_c$. The language TAL_0 has algorithmic type checking; its type checking is completely syntax-directed. On the other hand, the logic $\mathcal{L}_c$ cannot decide which labels should be kept as entries for instruction blocks. Therefore, $\mathcal{L}_c$ has declarative rules in this aspect. In Section 3.2.4, we will develop models for judgments in TAL_0 based on $\mathcal{L}_c$'s instruction judgment and then prove TAL_0's algorithmic typing rules can be justified by $\mathcal{L}_c$'s rules. Then, in Section 3.2.6, we show that our models for TAL_0 is strong enough to prove the safety theorem: If a program P has a typing derivation in TAL_0's type system, then it is safe according to the definition of $\mathrm{safe\_prog}(P)$.

Like all other typed assembly languages, TAL_0 uses types as the assertion language. The second purpose of TAL_0 is to explain some issues when types are adopted. In particular, we will briefly cover the indexed model of types in Section 3.2.5.

For a clear presentation, many simplifications are made in TAL_0. For example, a real TAL uses modular arithmetic and each register's value is bounded. TAL_0 assumes register values are unbounded and uses ordinary arithmetic.

3.2.1 Machine Instructions in TAL_0

The language TAL_0 has only a few machine instructions: The instruction "`add` s_1, s_2, d" adds values in registers s_1 and s_2, and stores the result into register d; the instruction "`ld` s, d" loads the contents of the memory address pointed by register s into register d; the branch-always instruction "`ba` l_d" unconditionally jumps to the absolute address l_d; the branch-if-zero instruction "`bz` s, l_d" jumps to l_d if the value of

`add` $s_1, s_2, d \triangleq$
$\lambda(r, m), (r', m').$
$\big(r'(\mathrm{pc}) = r(\mathrm{pc}) + 4\big) \;\wedge\; \big(r'(d) = r(s_1) + r(s_2)\big) \;\wedge\; \big(\forall x \notin \{\mathrm{pc}, d\}.\ r'(x) = r(x)\big)$
$\wedge\; \big(m' = m\big).$

`ld` $s, d \triangleq$
$\lambda(r, m), (r', m').$
$\big(r'(\mathrm{pc}) = r(\mathrm{pc}) + 4\big) \;\wedge\; \big(r'(d) = m(r(s))\big) \;\wedge\; \big(\forall x \notin \{\mathrm{pc}, d\}.\ r'(x) = r(x)\big)$
$\wedge\; \mathrm{readable}(r(s))$
$\wedge\; \big(m' = m\big).$

`ba` $l_d \triangleq$
$\lambda(r, m), (r', m').$
$\big(r'(\mathrm{pc}) = l_d\big) \;\wedge\; \big(\forall x \neq \mathrm{pc}.\ r'(x) = r(x)\big) \;\wedge\; \big(m' = m\big).$

`bz` $s, l_d \triangleq$
$\lambda(r, m), (r', m').$
$\big(r(s) = 0 \wedge r'(\mathrm{pc}) = l_d \;\vee\; r(s) \neq 0 \wedge r'(\mathrm{pc}) = r(\mathrm{pc}) + 4\big)$
$\wedge\; \big(\forall x \neq \mathrm{pc}.\ r'(x) = r(x)\big)$
$\wedge\; \big(m' = m\big).$

Figure 3.2: TAL$_0$: Semantics of machine instructions

$$\begin{array}{rrcl}
(\textit{natural numbers}) & n & ::= & 0 \mid 1 \mid 2 \mid \ldots \\
(\textit{addresses}) & l & ::= & 0 \mid 4 \mid 8 \mid \ldots \\
(\textit{register indexes}) & d, s & ::= & 0 \mid 1 \mid 2 \mid \ldots \mid 31 \\
\\
(\textit{programs}) & P & ::= & B; P \mid B \\
(\textit{instruction blocks}) & B & ::= & i; B \mid \texttt{ba}\ l_d \mid \texttt{bz}\ s, l_d \\
(\textit{instructions}) & i & ::= & \texttt{add}\ s_1, s_2, d \mid \texttt{ld}\ s, d \\
\\
(\textit{address invariants}) & \Psi & ::= & \{\, l_1 \mapsto \mathsf{codeptr}\,(\phi_1), \ldots, l_n \mapsto \mathsf{codeptr}\,(\phi_n) \,\} \\
(\textit{register-file types}) & \phi & ::= & \{\, d_1 \mapsto \tau_1, \ldots, d_n \mapsto \tau_n \,\} \\
(\textit{types}) & \tau & ::= & \mathsf{int} \mid \mathsf{int}_{=}(n) \mid \mathsf{int}_{\neq}(n) \mid \mathsf{box}\,(\tau)
\end{array}$$

Figure 3.3: TAL$_0$: Syntax

register s is zero, and falls through to the next instruction otherwise. Figure 3.2 presents the semantics of these machine instructions.

3.2.2 Syntax

Figure 3.3 presents the syntax of TAL$_0$. It has 32 registers, and addresses are multiples of four. A TAL$_0$ program consists of an assembly program P and a type annotation Ψ. An assembly program P consists of a sequence of instruction blocks, each of which is a sequence of instructions ending in a control-transfer instruction ("`ba` l_d" or "`bz` s, l_d"). A type annotation Ψ is a mapping from addresses to code-pointer types, which take register-file types as preconditions. The domain of a wellformed Ψ contains exactly those addresses that correspond to the beginning of instruction blocks. Therefore, it effectively associates a precondition with the beginning of every instruction block.

Types in TAL_0 include integer type int, immutable reference type $\mathsf{box}\,(\tau)$, singleton type $\mathsf{int}_=(n)$ (only the integer n has this type), and $\mathsf{int}_{\neq}(n)$ (integers not equal to n have this type).

A Register-file type, ϕ, specifies the type of a register bank. It is a mapping from register indexes to types.

We introduce some notation. Notation $|B|$ denotes the size of an instruction block B; we assume each instruction has size four. Notation $\phi[s \mapsto \tau]$ updates the type of the register s to τ in the register bank ϕ.

3.2.3 Type System

In Figure 3.4, we present a type system for TAL_0. One important property of the type system is that it is completely syntax-directed. That is, when given a program P and a wellformed type annotation Ψ as inputs, the type system can always pick a unique rule to proceed to have a typing derivation for P and Ψ.

Typing judgments for TAL_0 are listed as below:

- Judgment $\vdash_{\texttt{p}} P : \Psi$ means that the program P is wellformed with respect to the address invariants Ψ.

- Judgment $\Psi; l \vdash_{\texttt{f}} P$ means that the program fragment P, starting at address l, is wellformed, assuming the global address invariant Ψ. The address invariant Ψ provides preconditions of addresses to which P might jump.

- Judgment $\Psi; l; \phi \vdash_{\texttt{b}} B$ means that the instruction block B, starting at address l, is wellformed, assuming the precondition of B is ϕ and the global address invariant Ψ. The address invariant Ψ provides preconditions of addresses to which B might jump.

$\boxed{\vdash_{\mathtt{p}} P : \Psi}$

$$\frac{\Psi; 0 \vdash_{\mathtt{f}} P}{\vdash_{\mathtt{p}} P : \Psi}\ \textsf{prog}$$

$\boxed{\Psi; l \vdash_{\mathtt{f}} P}$

$$\frac{\begin{array}{c}\Psi(l) = \textsf{codeptr}\,(\phi) \quad \forall l < x < l + |B|.\ x \notin \mathrm{dom}(\Psi) \\ \Psi; l; \phi \vdash_{\mathtt{b}} B \quad \Psi; l + |B| \vdash_{\mathtt{f}} P\end{array}}{\Psi; l \vdash_{\mathtt{f}} (B; P)}\ \textsf{frag1}$$

$$\frac{\Psi(l) = \textsf{codeptr}\,(\phi) \quad \forall x > l.\ x \notin \mathrm{dom}(\Psi) \quad \Psi; l; \phi \vdash_{\mathtt{b}} B}{\Psi; l \vdash_{\mathtt{f}} B}\ \textsf{frag2}$$

$\boxed{\Psi; l; \phi \vdash_{\mathtt{b}} B}$

$$\frac{\vdash_{\mathtt{i}} \{\phi_1\} i \{\phi_2\} \quad \Psi; l + 4; \phi_2 \vdash_{\mathtt{b}} B}{\Psi; l; \phi_1 \vdash_{\mathtt{b}} (i; B)}\ \textsf{seq}$$

$$\frac{\Psi(l_d) = \textsf{codeptr}\,(\phi_d) \quad \phi <: \phi_d}{\Psi; l; \phi \vdash_{\mathtt{b}} (\texttt{ba}\ l_d)}\ \textsf{ba}$$

$$\frac{\begin{array}{cc}\Psi(l_d) = \textsf{codeptr}\,(\phi_d) & \Psi(l + 4) = \phi' \\ \phi[s \mapsto \textsf{int}_{=}(0)] <: \phi_d & \phi[s \mapsto \textsf{int}_{\neq}(0)] <: \phi'\end{array}}{\Psi; l; \phi \vdash_{\mathtt{b}} (\texttt{bz}\ s, l_d)}\ \textsf{bz}$$

$\boxed{\vdash_{\mathtt{i}} \{\phi_1\} i \{\phi_2\}}$

$$\frac{\phi <: \{\, s_1 \mapsto \textsf{int}, s_2 \mapsto \textsf{int} \,\}}{\vdash_{\mathtt{i}} \{\phi\}(\texttt{add}\ s_1, s_2, d)\{\phi[d \mapsto \textsf{int}]\}}\ \textsf{add} \qquad \frac{\phi <: \{\, s \mapsto \textsf{box}\,(\tau) \,\}}{\vdash_{\mathtt{i}} \{\phi\}(\texttt{ld}\ s, d)\{\phi[d \mapsto \tau]\}}\ \textsf{ld}$$

$\boxed{\tau_1 <: \tau_2,\ \phi_1 <: \phi_2}$

$$\frac{}{\tau <: \tau}\ \textsf{s-refl} \qquad \frac{}{\textsf{int}_{=}(n) <: \textsf{int}}\ \textsf{s-int}_{=} \qquad \frac{}{\textsf{int}_{\neq}(n) <: \textsf{int}}\ \textsf{s-int}_{\neq}$$

$$\frac{\mathrm{dom}(\phi_1) \supseteq \mathrm{dom}(\phi_2) \quad \forall d \in \mathrm{dom}(\phi_1) \cap \mathrm{dom}(\phi_2).\ \phi_1(d) <: \phi_2(d)}{\phi_1 <: \phi_2}\ \textsf{s-rfile}$$

Figure 3.4: TAL_0: Typing rules

- Judgment $\vdash_{\mathtt{i}} \{\phi_1\}i\{\phi_2\}$ means that the instruction i is wellformed with respect to the precondition ϕ_1 and the postcondition ϕ_2.
- Judgment $\phi_1 <: \phi_2$ means that ϕ_1 is a a stronger register-file type than ϕ_2. Judgment $\tau_1 <: \tau_2$ means that τ_1 is a subtype of τ_2.

To check that a program P is wellformed with respect to an address invariant Ψ, the prog rule invokes $\Psi; 0 \vdash_{\mathtt{f}} P$. Then, the type system uses the frag1 and frag2 rules to check that each instruction block in P is wellformed. The rules frag1 and frag2 look up the precondition of each block inside the global address invariant Ψ. These two rules also make sure that Ψ is wellformed by checking that Ψ associate invariants with only those addresses that correspond to the beginning of instruction blocks. (Invariants for other addresses are computed by the type system.)

When checking a single instruction block B, the type system will use the rule seq to walk through the block to check that every instruction is wellformed. The ba (branch always) and bz (branch if zero) rules check the last instruction in a block. The ba rule checks that the current precondition, ϕ, is a subtype of the precondition of the destination address (which must be in the domain of Ψ). The conditional branch instruction "bz s, l_d" has two cases. When the branch is taken, the register s has value zero; therefore the rule bz updates[2] the current precondition with s mapped to the type $\mathsf{int}_=(0)$, and checks that the new precondition is a subtype of the precondition of the destination address. Similarly, when the branch is not taken, the register s should have type $\mathsf{int}_{\neq}(0)$.

The rules for $\vdash_{\mathtt{i}} \{\phi_1\}i\{\phi_2\}$ take a precondition ϕ_1 and an instruction i as inputs and calculate a postcondition ϕ_2 as an output. For example, in the add rule, the

[2]This is over-simplified. LTAL also remembers the old type of the comparison register.

postcondition is the precondition with the destination register updated with integer type.

Finally there are the rules for subtyping; these rules are unsurprising.

3.2.4 Models of Typing Judgments in TAL_0

In this section, we show how to justify the soundness of TAL_0's type system from $\mathcal{L}_c$. We proceed in two steps. First, we develop models for judgments in TAL_0 based on $\mathcal{L}_c$'s instruction judgment. We use $C; \Psi' \models_{\mathcal{L}_c} \Psi$ for $\mathcal{L}_c$'s instruction judgment to avoid confusion. In the second step, we show how typing rules in TAL_0 can be proved from the rules in $\mathcal{L}_c$.

First we introduce some notation. In TAL_0, a program P or an instruction block B denotes a list of consecutive instructions. Therefore, we will write $P@l$ to denote that the program P starts at the address l. We will also overload the @ symbol and use it in $B@l$ (respectively, $i@l$), which means that the instruction block B (respectively, the single instruction i) starts at the address l. The logic $\mathcal{L}_c$ uses the notation $\{\, l \triangleright \phi \,\}$ to denote that l is a code pointer with precondition ϕ. In this chapter, we use $\{\, l \mapsto \mathsf{codeptr}\,(\phi) \,\}$ instead; $\{\, l \triangleright \phi \,\}$ can be thought as an abbreviation for $\{\, l \mapsto \mathsf{codeptr}\,(\phi) \,\}$.

Figure 3.5 presents models for judgments in TAL_0's type system based on $C; \Psi' \models_{\mathcal{L}_c} \Psi$. The first judgment is $\vdash_{\mathtt{p}} P : \Psi$, which we model as $P@0; \{\} \models_{\mathcal{L}_c} \Psi$. The program P has multiple entries: the beginnings of instruction blocks in P are possible entries; these beginnings correspond to the domain of Ψ. Furthermore, since P is the complete program, it does not depend on other exits and thus has no exits (except for some possible indirect exits in registers; we will discuss this in the next section). Therefore, the meaning of $\vdash_{\mathtt{p}} P : \Psi$ in Figure 3.5 says that every address in

$$\begin{array}{lcl}
\models_{\mathtt{p}} P : \Psi & \triangleq & P@0; \{\} \models_{\mathcal{L}_c} \Psi \\
\Psi; l \models_{\mathtt{f}} P & \triangleq & P@l; \Psi \models_{\mathcal{L}_c} |\Psi|_{\geq l} \\
\Psi; l; \phi \models_{\mathtt{b}} B & \triangleq & B@l; \Psi \models_{\mathcal{L}_c} \{l \mapsto \mathsf{codeptr}\,(\phi)\} \\
\models_{\mathtt{i}} \{\phi_1\}i\{\phi_2\} & \triangleq & \forall l.\ i@l; \{l + |i| \mapsto \mathsf{codeptr}\,(\phi_2)\} \models_{\mathcal{L}_c} \{l \mapsto \mathsf{codeptr}\,(\phi_1)\}
\end{array}$$

Figure 3.5: TAL_0: Models of typing judgments

the domain of Ψ is a code pointer with respect to the corresponding precondition prescribed in Ψ.

In the judgment $\Psi; l \vdash_{\mathtt{f}} P$, the P part is not a complete program, but only a partial program starting at the address l, and thus its entries include the beginnings of only those instruction blocks in P; these beginnings correspond to those addresses that are not less than l in the domain of Ψ. We use the notation $|\Psi|_{\geq l}$ to denote the restriction of Ψ to those addresses greater than or equal to l. Meanwhile, P may jump outside of P to any instruction block in the complete program. Therefore, the model of $\Psi; l \vdash_{\mathtt{f}} P$ in Figure 3.5 says that every address in the domain of $|\Psi|_{\geq l}$ is a code pointer, assuming the global address invariant Ψ.

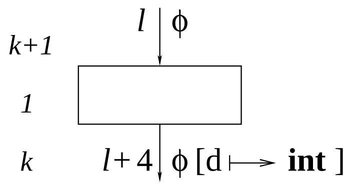
In the judgment $\Psi; l; \phi \vdash_{\mathtt{b}} B$, the block B has only one entry, namely l. But the last instruction in B may jump to any other block in the complete program. Therefore, the model of $\Psi; l \vdash_{\mathtt{f}} P$ in Figure 3.5 says that the address l is a code pointer with the precondition ϕ, assuming the global address invariant Ψ.

In the judgment $\vdash_{\mathtt{i}} \{\phi_1\}i\{\phi_2\}$, the instruction i is not a control-transfer instruction and therefore has one entry and one exit. Its model in Figure 3.5 states that the address l is a code pointer with ϕ_1, assuming the address $l + |i|$ is a code pointer with ϕ_2. ($l + |i|$ is always $l + 4$ in TAL_0).

Soundness of TAL$_0$**'s typing rules.** Having formulated the semantics for TAL$_0$'s typing judgments, we next show how TAL$_0$'s typing rules can be proved sound based on the semantics. TAL$_0$'s typing rules can be classified into two categories: rules for individual instructions, and composition rules.

Proofs of soundness of rules for individual instructions such as the `add` rule requires a model of TAL$_0$'s assertion language — types; we leave the model of types to the next section. But other than that aspect, the proofs are very similar to the ones in the last chapter (such as the proof of the assignment rule).

We next take the `add` rule as an example. Based on the model, the rule states that in a state σ if "`add` s_1, s_2, d" is at the address l, then l is a code pointer with the condition ϕ to the index $k+1$, provided that the address $l+4$ is a code pointer with the condition $\phi[d \mapsto \texttt{int}]$ to the index k:



To prove that the address l is a code pointer with the condition ϕ to approximation $k+1$ in the state σ, the major steps are as follows:

(i) Based on the definition of the code pointer (Eq. 2.2 on page 40), we start from a state σ' such that $\sigma \mapsto^* \sigma'$, the control of σ' is at l, and ϕ is true on σ'. The goal is to prove that σ' can step for $k+1$ steps.

(ii) Based on the fact that the "`add` s_1, s_2, d" instruction is at the address l[3] and the control of σ' is at l, construct a new state σ'' such that $\sigma' \mapsto \sigma''$. The construction of σ'' follows the semantics of the `add` instruction in Figure 3.2. In summary, this step shows that the state σ' can *progress* for one step.

(iii) Prove that the new state σ'' satisfies the condition $\phi[d \mapsto \mathsf{int}]$. Since σ'' was constructed following the semantics of the `add` instruction, proof of this step is also based on the semantics of the `add` instruction.

(iv) Prove that the new state σ'' is safe for k steps. This step uses the fact that the control of σ'' is at the address $l + 4$, the previous step that σ'' satisfies $\phi[d \mapsto \mathsf{int}]$, the assumption that the address $l + 4$ is a code pointer with the condition $\phi[d \mapsto \mathsf{int}]$ to approximation k in the state σ, and $\sigma \mapsto^* \sigma''$.

(v) We have proved that the state σ' can progress for one step to reach a state σ'', and σ'' is safe for k steps. On a deterministic machine[4], this is enough to show that σ' is safe for $k + 1$ steps.[5]

The above is a sketch of the major steps. In the next chapter, we will discuss the real proof in greater detail.

Next, we show how the soundness of TAL$_0$'s composition rules (prog, frag1, frag2 and seq) follows from $\mathcal{L}_c$'s rules. The two interesting cases are the prog and the seq rule, whose proofs are presented below.

[3]Actually, we only know that the `add` instruction is at the address l in the state σ, but we need the fact that the instruction is also there in the state σ'. In our real system, we also maintain the invariant that the code is in an immutable region of a state so that the `add` instruction is still at the address l in the state σ'.

[4]In our real system, we prove that our modeling of SPARC is deterministic.

[5]On nondeterministic machines, we need to show that for all σ'' such that $\sigma' \mapsto \sigma''$, σ'' is safe for k steps.

Lemma 3.16 *(The* prog *rule in* TAL_0*.) If* $\Psi; 0 \models_{\mathtt{f}} P$*, then* $\models_{\mathtt{p}} P : \Psi$*.*

Proof. From the definitions in Figure 3.5, we need to prove $P@0; \{\ \} \models_{\mathcal{L}_c} \Psi$, by assuming $P@0; \Psi \models_{\mathcal{L}_c} |\Psi|_{\geq 0}$. Since the domain of Ψ is a subset of the natural numbers, we have $|\Psi|_{\geq 0} = \Psi$, and thus

$$P@0; \Psi \models_{\mathcal{L}_c} \Psi \tag{3.16.1}$$

Note that every $\{\, l \mapsto \mathsf{codeptr}\,(\phi)\,\}$ in Ψ appears on both the left and the right in the above judgment. Because $\mathrm{dom}(\Psi)$ is finite, we use the discharge rule multiple times to remove all continuations from the left of the judgment, and then get

$$P@0; \{\ \} \models_{\mathcal{L}_c} \Psi$$

$\square$

Lemma 3.17 *(The* seq *rule.) If* $\models_{\mathtt{i}} \{\phi_1\} i \{\phi_2\}$*, and* $\Psi; l+4; \phi_2 \models_{\mathtt{b}} B$*, then* $\Psi; l; \phi_1 \models_{\mathtt{b}} i; B$*.*

Proof. From the definitions in Figure 3.5, we need to prove

$$(i;B)@l; \Psi \models_{\mathcal{L}_c} \{\, l \mapsto \mathsf{codeptr}\,(\phi_1)\,\},$$

by assuming

$$\forall l.\ i@l; \{\, l + |i| \mapsto \mathsf{codeptr}\,(\phi_2)\,\} \models_{\mathcal{L}_c} \{\, l \mapsto \mathsf{codeptr}\,(\phi_1)\,\} \tag{3.17.1}$$

$$B@(l+4); \Psi \models_{\mathcal{L}_c} \{\, l+4 \mapsto \mathsf{codeptr}\,(\phi_2)\,\} \tag{3.17.2}$$

Carrying a universal elimination on 3.17.1 using the address l, and considering that $|i| = 4$, we have

$$i@l; \{\, l + 4 \mapsto \mathsf{codeptr}\,(\phi_2)\,\} \models_{\mathcal{L}_c} \{\, l \mapsto \mathsf{codeptr}\,(\phi_1)\,\} \tag{3.17.3}$$

Use the combine rule in $\mathcal{L}_c$ on 3.17.3 and 3.17.2, and also take into account that $i@l$ together with $B@(l+4)$ is the same as $(i;B)@l$, and we get

$$(i;B)@l; \Psi \cup \{\, l + 4 \mapsto \mathsf{codeptr}\,(\phi_2)\,\} \models_{\mathcal{L}_c} \{\, l \mapsto \mathsf{codeptr}\,(\phi_1), l + 4 \mapsto \mathsf{codeptr}\,(\phi_2)\,\}. \tag{3.17.4}$$

Since $\{\, l + 4 \mapsto \mathsf{codeptr}\,(\phi_2)\,\}$ appears both on the left and on the right of the above judgment, we use the discharge rule to remove it from the left:

$$(i;B)@l; \Psi \models_{\mathcal{L}_c} \{\, l \mapsto \mathsf{codeptr}\,(\phi_1), l + 4 \mapsto \mathsf{codeptr}\,(\phi_2)\,\}. \tag{3.17.5}$$

Finally, we use the weaken rule to remove $\{\, l+4 \mapsto \mathsf{codeptr}\,(\phi_2)\,\}$ from the right of the judgment to prove our goal:

$$(i;B)@l; \Psi \models_{\mathcal{L}_c} \{\, l \mapsto \mathsf{codeptr}\,(\phi_1)\,\} \tag{3.17.6}$$

□

3.2.5 Indexed Model of Types

In this section, we present a brief description of the model of types in the FPCC project. We use the *indexed* model of types as a general mechanism for modeling types. The indexed model was introduced by Appel and McAllester [9] to model recursive types and was later adopted by Ahmed et al. to model impredicative quantified types and mutable references [4].

We model types as a series of approximations and each approximation is a set of values. On a von Neumann machine, we represent a value as a pair (σ, x), where σ is a state and x is an integer (typically representing an address). Furthermore, in the indexed model, we use a notion that states that a value (σ, x) belongs to the type τ to some approximation (index) k; we write $(\sigma, x) :_k \tau$ to express this notion.

Next, we explain the purpose of the index k in the model. If $(\sigma, x) :_k \tau$, value (σ, x) may be a real member of type τ, or it may be a "fake" member that only k-approximately belongs to τ. In general, any program taking such a "fake" member as an input cannot tell the difference within k computation steps. In some sense, the number k indicates the number of computation steps in the future.

Take the type $\mathsf{box}\,(\mathsf{box}\,(\mathsf{int}))$ as an example. Suppose $(\sigma, x) : \mathsf{box}\,(\mathsf{box}\,(\mathsf{int}))$, then x is a two-fold pointer and $\mathrm{m\_of}(\sigma)(\mathrm{m\_of}(\sigma)(x))$ is of type int. Imagine that x is only a one-fold pointer in a state s; that is, $(\sigma, x) : \mathsf{box}\,(\tau)$, where perhaps τ is disjoint from $\mathsf{box}\,(\mathsf{int})$. Then for one step (one dereference), (σ, x) safely simulates membership in $\mathsf{box}\,(\mathsf{box}\,(\mathsf{int}))$. In this case, we have $(\sigma, x) :_1 \mathsf{box}\,(\mathsf{box}\,(\mathsf{int}))$, but not $(\sigma, x) :_2 \mathsf{box}\,(\mathsf{box}\,(\mathsf{int}))$.

Having given an intuition of the indexed model, we present the indexed model of the types in TAL_0 in the first part of Figure 3.6.

Any (m, x) such that $x = n$ belongs to the type $\mathsf{int}_=(n)$, to any index. The model for $\mathsf{box}\,(\tau)$ is more interesting. It requires the content in the memory, or $\mathrm{m\_of}(\sigma)(x)$, has the type τ, but only at a lower index, $k - 1$. This is because that dereferences take one computation step; after this step, $\mathrm{m\_of}(\sigma)(x)$ is required to have the type τ only at the index $k - 1$.

$$\begin{array}{rcl}
(\sigma, x) :_k \mathsf{int} & \triangleq & \text{true} \\
(\sigma, x) :_k \mathsf{int}_=(n) & \triangleq & x = n \\
(\sigma, x) :_k \mathsf{box}\,(\tau) & \triangleq & \text{readable}(x) \;\wedge\; (\sigma, \text{m\_of}(\sigma)(x)) :_{k-1} \tau \\
\\
\sigma :_k \phi & \triangleq & \forall d \in [0, 31].\ (\sigma, \text{r\_of}(\sigma)(d)) :_k \phi(d) \\
\\
\tau_1 \;<:\; \tau_2 & \triangleq & \forall \sigma, x, k.\ (\sigma, x) :_k \tau_1 \Rightarrow (\sigma, x) :_k \tau_1 \\
\phi_1 \;<:\; \phi_2 & \triangleq & \forall \sigma, k.\ \sigma :_k \tau_1 \Rightarrow \sigma :_k \tau_1
\end{array}$$

Figure 3.6: TAL_0: Models of types, register-file types and subtyping

We write $(\sigma, x) : \tau$ to mean $(\sigma, x) :_k \tau$ is true for any k; in this case, (σ, x) is a real member of the type τ.

Figure 3.6 also presents the model of register-file types and subtyping. A state σ belongs to a register-file type ϕ to an index k, if every register value in the state belongs to the type associated with the register in ϕ to the index k. The model of the subtyping relation, $\tau_1 \;<:\; \tau_2$ and $\phi_1 \;<:\; \phi_2$, is straightforward.

Valid types. Next, we introduce one important property of types: a valid type is closed under decreasing indexes.

Definition 3.18 *A type τ is valid if $(m, x) :_k \tau$ and $j < k$ implies $(m, x) :_j \tau$.*

We can easily check that the type int and $\mathsf{int}_=(n)$ are valid types (their definitions do not mention the index). Furthermore, the type constructor box maps valid types to valid types.

Adjusting the model of $C; \Psi' \models_{\mathcal{L}_c} \Psi$. We have introduced the model of $C; \Psi' \models_{\mathcal{L}_c} \Psi$ in the last chapter. However, that definition is based on a general assertion language. Therefore, we first make some small adjustments on the model of $C; \Psi' \models_{\mathcal{L}_c}$

Ψ to use types as our assertion language. After these adjustments, all the rules in $\mathcal{L}_c$ are still provable.

First, we introduce the semantics of address invariants Ψ, which is a mapping from addresses to code pointer types. We will interpret each $\{\, l \mapsto \mathsf{codeptr}\,(\phi)\,\}$ as a continuation; that is, it is safe to jump to the address l provided that the precondition ϕ is met. To model this notion, we define a code-pointer type:

$$\begin{aligned}(\sigma, x) :_k \mathsf{codeptr}\,(\phi) \;\triangleq\; & \forall \sigma', j < k.\; \sigma \mapsto^* \sigma' \;\wedge\; \big(\mathrm{r\_of}(\sigma')(\mathrm{pc}) = x\big) \;\wedge\; \sigma' :_j \phi \\ & \Rightarrow\; \mathrm{safe\_state}(\sigma, j)\end{aligned} \tag{3.2}$$

The above definition is very similar to the notion $\sigma; \theta \models_k l \triangleright p$ (Eq. 2.2 on page 40) in the last chapter: Both are modeling continuations; they both quantify over all future states σ' that σ steps to.

What is different is that the definition of $(\sigma, x) :_k \mathsf{codeptr}\,(\phi)$ further quantifies all j that is less than k. The reason is that the version without quantification over j (as follows) does not satisfy the valid-type property (Definition 3.18) because of the contravariant appearance of the premise "$\sigma' :_k \phi$". The definition $\sigma; \theta \models_k l \triangleright p$, however, has p as the assertion, whose truth is not indexed by k, and thus does not have this problem.

Based on continuation types, we give a model to an address invariant Ψ:

$$\sigma \models_k \Psi \;\triangleq\; \forall l \in \mathrm{dom}(\Psi).\; (\sigma, l) :_k \mathsf{codeptr}\,(\Psi(l)) \tag{3.3}$$

The new definition of $C; \Psi' \models_{\mathcal{L}_c} \Psi$ is given as follows:

$$\begin{aligned} &C; \Psi' \models_{\mathcal{L}_c} \Psi \;= \\ &\quad \forall \sigma \in \Sigma.\, \text{loaded}(C, \sigma) \;\Rightarrow\; \forall k \in \mathbb{N}.\, \big(\sigma \models_k \Psi' \;\Rightarrow\; \sigma \models_{k+1} \Psi\big). \end{aligned} \tag{3.4}$$

3.2.6 Safety Theorem

We have given models to the typing judgments in TAL$_0$ in Section 3.2.4 and proved the composition rules in TAL$_0$ are sound with respect to those models. However, there remains the possibility that our models may not be strong enough to imply programs' safety. Therefore, in this section, we justify our models by proving the following safety theorem: If $\models_{\mathtt{p}} P : \Psi$, then safe_prog$(P)$.

There is one technical difficulty in proving the safety theorem. To prove safe_prog(P), we have to show that it is safe to jump to address zero, with the assumption that init_cond(σ) is true on the initial state σ (see the definition on page 65). Meanwhile, if $\Psi(0) = \mathsf{codeptr}\,(\phi_0)$, then the definition of $\models_{\mathtt{p}} P : \Psi$ gives that it is safe to jump to address zero provided that precondition ϕ_0 is true on the initial state. Both safe_prog(P) and $\models_{\mathtt{p}} P : \Psi$ mention that address zero is a code pointer, but there is a mismatch between preconditions: One is init_cond(σ), which is untyped, the other is the type ϕ_0.

Avoiding types in the definition of safe_prog(P) is actually our intention: We want our semantic model of types to be entirely in the *proof* of the safety theorem, not in the *statement* of the theorem; this way, the choice of type system is up to the compiler and prover, and is not constrained by the checker. But this makes the definition of init_cond(σ) tricky: we must not use types. In our actual implementation, ϕ_0 has two parts: simple integer types, and a return type for register 15 (that is, register 15 has a code-pointer type). These two parts can be specified easily in init_cond(σ) without using our semantic model of types; remember that one part of init_cond(σ) is return_addr(r_of(σ)(15)), which essentially states that register 15 is of a code-pointer type.

Therefore, we have proved in higher-order logic that init_cond(σ) implies the truth of ϕ_0 on σ. With that proof, we have the safety theorem for TAL$_0$:

Theorem 3.19 *(Safety Theorem)* $\dfrac{\vdash_{\mathtt{p}} P : \Psi \quad \Psi(0) = \mathsf{codeptr}\,(\phi_0)}{\mathrm{safe\_prog}(P)}$

Proof. According to the definition of safe_prog(P), for a state σ and a natural number k, we need to show that safe_state(σ, k), assuming

$$i)\ \ \mathrm{loaded}(P,\sigma) \qquad ii)\ \ \mathrm{r\_of}(\sigma)(\mathrm{pc}) = 0 \qquad iii)\ \ \mathrm{init\_cond}(\sigma)$$

On the other hand, the model of $\vdash_{\mathtt{p}} P : \Psi$ is $P@0; \{\} \models_{\mathcal{L}_c} \Psi$. The deduction steps from $P@0; \{\ \} \models_{\mathcal{L}_c} \Psi$ to the goal safe_state(σ, k) are presented by the following proof tree. (The tree has been broken into two parts for easy type-setting.)

$$\dfrac{\dfrac{P@0;\{\ \} \models_{\mathcal{L}_c} \Psi \quad \mathrm{loaded}(P,\sigma) \quad \sigma \models_k \{\ \}}{\sigma \models_{k+1} \Psi}\,(3) \qquad \Psi(0) = \mathsf{codeptr}\,(\phi_0)}{(\sigma, 0) :_{k+1} \mathsf{codeptr}\,(\phi_0)}\,(2a)$$

$$\dfrac{\dfrac{\vdots}{(\sigma, 0) :_{k+1} \mathsf{codeptr}\,(\phi_0)}\,(2a) \quad \sigma \mapsto^* \sigma \quad \mathrm{r\_of}(\sigma)\mathrm{pc} = 0 \quad \dfrac{\mathrm{init\_cond}(\sigma)}{\sigma :_k \phi_0}\,(2b)}{\mathrm{safe\_state}(\sigma, k)}\,(1)$$

Step (3) follows from the definition of $C; \Psi' \models_{\mathcal{L}_c} \Psi$. Step (2a) is by definition of $\sigma \models_{k+1} \Psi$ on page 81. Step (2b) means that the definition of init_cond(σ) is strong enough to imply that the state satisfies ϕ_0. Step (1) follows from the definition of the code-pointer type. $\square$

3.3 TAL$_1$

We have presented TAL$_0$, whose main purpose is to illustrate the step between a typed assembly language and our logic $\mathcal{L}_c$. Nevertheless, TAL$_0$ lacks several control-flow features that real architectures such as SPARC have, including pc-relative jumps and indirect jumps. We will add these features to TAL$_0$. Interestingly, accommodating these new control-flow features requires no change to our model of typing judgments.

Machine instructions. We add an indirect-jumping instruction, `jmp` d. It jumps to the value of the register d. Moreover, we change the semantics of the `ba` and `bz` instructions from absolute jumps to pc-relative jumps: Instead of taking absolute addresses, the instructions `ba` and `bz` in TAL$_1$ take offsets, which can be either positive or negative. Figure 3.7 presents the formal semantics of these new machine instructions in TAL$_1$.

Syntax. Figure 3.8 presents the differences, in framed boxes, between TAL$_1$'s syntax and TAL$_0$'s: The control-transfer instruction at the end of an instruction block can be a "`ba` o", a "`bz` s, d", or a "`jmp` d"; we also add a code-pointer type, $\mathsf{codeptr}\,(\phi)$, which models the type of the register in an indirect-jumping instruction.

Typing rules. Figure 3.9 presents the differences between TAL$_1$'s typing rules and TAL$_0$'s. The rules for `ba` o and `bz` s, o are almost the same as the versions in

$$
\begin{array}{l}
\texttt{ba}\ o\ \triangleq \\
\quad \lambda(r,m),(r',m'). \\
\qquad \big(r'(\mathrm{pc}) = r(\mathrm{pc}) + o\big)\ \wedge\ \big(\forall x \neq \mathrm{pc}.\ r'(x) = r(x)\big)\ \wedge\ \big(m' = m\big).
\end{array}
$$

$$
\begin{array}{l}
\texttt{bz}\ s, o\ \triangleq \\
\quad \lambda(r,m),(r',m'). \\
\qquad \big(r(s) = 0 \wedge r'(\mathrm{pc}) = r(\mathrm{pc}) + o\ \vee\ r(s) \neq 0 \wedge r'(\mathrm{pc}) = r(\mathrm{pc}) + 4\big) \\
\qquad \wedge\ \big(\forall x \neq \mathrm{pc}.\ r'(x) = r(x)\big) \\
\qquad \wedge\ \big(m' = m\big).
\end{array}
$$

$$
\begin{array}{l}
\texttt{jmp}\ d\ \triangleq \\
\quad \lambda(r,m),(r',m'). \\
\qquad \big(r'(\mathrm{pc}) = r(d)\big)\ \wedge\ \big(\forall x \neq \mathrm{pc}.\ r'(x) = r(x)\big)\ \wedge\ \big(m' = m\big).
\end{array}
$$

Figure 3.7: TAL$_1$: Semantics of machine instructions

$$
\begin{array}{rcll}
 & \ldots & & \\
(\textit{instruction blocks}) & B & ::= & i; B \mid \boxed{\texttt{ba}\ o \mid \texttt{bz}\ s, o \mid \texttt{jmp}\ d} \\
\boxed{(\textit{offsets})} & o & ::= & \ldots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \ldots \\
 & \ldots & & \\
(\textit{types}) & \tau & ::= & \mathsf{int} \mid \mathsf{box}\,(\tau) \mid \mathsf{int}_{=}(n) \mid \mathsf{int}_{\neq}(n) \mid \boxed{\mathsf{codeptr}\,(\phi)} \\
 & \ldots & &
\end{array}
$$

Figure 3.8: TAL$_1$: Syntax (Only the differences, in framed boxes, from TAL$_0$ are shown.)

$\boxed{\Psi; l; \phi \vdash_{\mathtt{b}} B}$

$$\frac{\phi \ <:\ \Psi(l+o)}{\Psi; l; \phi \vdash_{\mathtt{b}} (\mathtt{ba}\ o)}\ \mathsf{ba}$$

$$\frac{\phi_1[r \mapsto \mathsf{int}_{=}(0)] \ <:\ \Psi(l+o) \quad \phi_1[r \mapsto \mathsf{int}_{\neq}(0)] \ <:\ \Psi(l+4)}{\Psi; l; \phi_1 \vdash_{\mathtt{b}} (\mathtt{bz}\ r, o)}\ \mathsf{bz}$$

$$\frac{\phi \ <:\ \{\, d \mapsto \mathsf{codeptr}\,(\phi)\,\}}{\Psi; l; \phi \vdash_{\mathtt{b}} (\mathtt{jmp}\ d)}\ \mathsf{jmp}$$

$\boxed{\tau_1 \ <:\ \tau_2}$

$$\frac{\phi_2 \ <:\ \phi_1}{\mathsf{codeptr}\,(\phi_1) \ <:\ \mathsf{codeptr}\,(\phi_2)}\ \mathsf{s\text{-}codeptr}$$

Figure 3.9: TAL$_1$: Typing rules (Only the differences from TAL$_0$'s typing rules are shown).

TAL$_0$, except that the destination address is $l + o$ — the current address plus the offset.

The rule for `jmp` d is new in TAL$_1$. When jumping to the register d, the rule requires the register d be of the code-pointer type that takes the current register-file type ϕ as the precondition. As an example, suppose the instruction is "`jmp` 1", and

$$\phi = \{\, 1 \mapsto \mathsf{codeptr}\,(\{\, 2 \mapsto \mathsf{int}\,\}),\ 2 \mapsto \mathsf{int}\,\}.$$

From ϕ, register one is a code pointer with the condition on register two being an integer. In this case, we can derive $\phi \ <:\ \{1 \mapsto \mathsf{codeptr}\,(\phi)\}$ as follows, and thus `jmp` 1 is safe.

$$\dfrac{\dfrac{\dfrac{}{\{\,1 \mapsto \mathsf{codeptr}\,(\{\,2 \mapsto \mathsf{int}\,\}),\ 2 \mapsto \mathsf{int}\,\} \ <:\ \{\,2 \mapsto \mathsf{int}\,\}}\ \textsf{s-rfile}}{\mathsf{codeptr}\,(\{\,2 \mapsto \mathsf{int}\,\}) \ <:\ \mathsf{codeptr}\,(\{\,1 \mapsto \mathsf{codeptr}\,(\{\,2 \mapsto \mathsf{int}\,\}),\ 2 \mapsto \mathsf{int}\,\})}\ \textsf{s-codeptr}}{\begin{array}{l}\{\,1 \mapsto \mathsf{codeptr}\,(\{\,2 \mapsto \mathsf{int}\,\}),\ 2 \mapsto \mathsf{int}\,\} \\ \quad <:\ \{\,1 \mapsto \mathsf{codeptr}\,(\{\,1 \mapsto \mathsf{codeptr}\,(\{\,2 \mapsto \mathsf{int}\,\}),\ 2 \mapsto \mathsf{int}\,\})\,\}\end{array}}\ \textsf{s-rfile}$$

Adjustments to the models and proofs of the new rules. We have included pc-relative and indirect jumps in TAL$_1$. Interestingly, accommodating these new features requires no change to the models of the typing judgments. That is, with respect to the models in Figure 3.5, the new rules in TAL$_1$ can be proved sound.

The proofs of the ba and bz rules are similar to the ones in TAL$_0$. We start from the precondition and prove the instruction can step further to reach a safe exit.

The proof of the jmp rule is slightly different. Assuming $\phi \ <:\ \{\, d \mapsto \mathsf{codeptr}\,(\phi)\,\}$, the rule requires us to prove $\Psi; l; \phi \models_{\mathsf{b}} (\texttt{jmp}\ d)$, whose model is

$$(\texttt{jmp}\ d)@l; \Psi \models_{\mathcal{L}_c} \{\, l \mapsto \mathsf{codeptr}\,(\phi)\,\} \tag{3.5}$$

Unpack the above definition: For any state σ and any number k, assuming

$$\text{loaded}((\texttt{jmp}\ d)@l, \sigma) \tag{3.6}$$

$$\sigma \models_k \Psi, \tag{3.7}$$

the goal is

$$\sigma \models_{k+1} \{\, l \mapsto \mathsf{codeptr}\,(\phi)\,\}. \tag{3.8}$$

Unlike the cases in rules such as bz, the assumption $\sigma \models_k \Psi$ does not help us at all, since the exit of the instruction "`jmp` d" is not explicit in the domain of the global address invariant Ψ, but *implicit* as the value of the register d. However, even in this case, we are still able to prove the goal 3.8, because to prove it, we can assume the precondition ϕ is true. Together with $\phi <: \{ d \mapsto \mathsf{codeptr}(\phi) \}$, we have that the register d is a code pointer with the precondition ϕ. Therefore, although the exit in "`jmp` d" is implicit, we can still prove it to be a safe exit and the rest of the proof is the same as the ones in other rules.

3.4 TAL$_2$

In this section, we present TAL$_2$, which introduces two important features — parametric polymorphism and recursive types. To accommodate these features while still maintaining syntax-directed type checking, TAL$_2$ has virtual instructions that do not correspond to a machine instruction; these virtual instructions only manipulate types, and perform no real computation. Their purpose is to guide the type-checking process.

After presenting TAL$_2$, we will then show that with only minor changes to the models of the typing judgments, we can accommodate polymorphism, recursive types, and virtual instructions.

Syntax. Figure 3.10 presents the syntax of TAL$_2$; the differences between TAL$_2$ and TAL$_0$ are in framed boxes. We augment types with recursive types and type variables. A context Δ is a vector of type variables. An address invariant, Ψ, maps addresses to polymorphic code-pointer types. Therefore, each instruction block in TAL$_2$ is polymorphically typed. Notice that a register cannot have a polymorphic type in TAL$_2$; this is only for simplicity of presentation and our implementation has polymorphically typed registers.

$$\begin{array}{rrcl}
(\textit{natural numbers}) & n & ::= & 0 \mid 1 \mid 2 \mid \ldots \\
(\textit{addresses}) & l & ::= & 0 \mid 4 \mid 8 \mid \ldots \\
(\textit{register indexes}) & d, s & ::= & 0 \mid 1 \mid 2 \mid \ldots \mid 31 \\
\\
(\textit{programs}) & P & ::= & B; P \mid B \\
(\textit{instruction blocks}) & B & ::= & i; B \mid \boxed{\texttt{ba}\ l_d\ \texttt{with}\ \textit{inst} \mid \texttt{bz}\ s, l_d\ \texttt{with}\ (\textit{inst}_1, \textit{inst}_2)} \\
(\textit{instructions}) & i & ::= & \texttt{add}\ s_1, s_2, d \mid \texttt{ld}\ s, d \\
 & & \mid & \boxed{\texttt{fold}\ d\ \texttt{to}\ \mathsf{rec}\,\alpha.\ \tau \mid \texttt{unfold}\ d} \\
\\
(\textit{address invariants}) & \Psi & ::= & \boxed{\{\, l_1 \mapsto \forall\overrightarrow{\alpha_1}.\,\mathsf{codeptr}\,(\phi_1), \ldots, l_n \mapsto \forall\overrightarrow{\alpha_n}.\,\mathsf{codeptr}\,(\phi_n)\,\}} \\
\boxed{(\textit{type contexts})} & \Delta & ::= & \vec{\alpha} \\
(\textit{register-file types}) & \phi & ::= & \{\, d_1 \mapsto \tau_1, \ldots, d_n \mapsto \tau_n \,\} \\
(\textit{types}) & \tau & ::= & \mathsf{int} \mid \mathsf{int}_{=}(n) \mid \mathsf{int}_{\neq}(n) \mid \mathsf{box}\,(\tau) \\
 & & \mid & \boxed{\mathsf{rec}\,\alpha.\ \tau \mid \alpha} \\
\\
\boxed{(\textit{instantiations})} & \textit{inst} & ::= & [\alpha_1, \ldots, \alpha_n]
\end{array}$$

Figure 3.10: TAL_2: Syntax

In polymorphic lambda-calculus [56] (or System F [31]), an instantiation is provided when calling a polymorphic function. This is to have decidable type checking for polymorphic lambda-calculus. Similarly, for maintaining syntax-directed type checking in TAL$_2$, the "$\texttt{ba}\ l_d\ \texttt{with}\ \mathit{inst}$" instruction provides an instantiation to inform the type checker of what types should be instantiated for the polymorphic type of the destination code block. The instruction "$\texttt{bz}\ s, l_d\ \texttt{with}\ (\mathit{inst_1}, \mathit{inst_2})$" carries two instantiations, since it has two possible destinations.

In the same spirit to maintain syntax-directed type checking, TAL$_2$ has special instructions to handle folding and unfolding of recursive types: the instruction "$\texttt{fold}\ d\ \texttt{to}\ \texttt{rec}\,\alpha.\ \tau$" folds the type of the register d into the recursive type $\texttt{rec}\,\alpha.\ \tau$; the instruction "$\texttt{unfold}\ d$" unfolds the type the register d.

Instructions such as "$\texttt{unfold}\ d$" are virtual instructions and their purpose is to guide the type-checking process. These virtual instructions do not correspond to machine instructions. We define $|i|$ as follows to designate the length of the instruction i on SPARC:

$$|i| = \begin{cases} 0, \text{when } i = \text{``}\texttt{fold}\ d\ \texttt{to}\ \texttt{rec}\,\alpha.\ \tau'',\ \text{ or } \text{``}\texttt{unfold}\ d'' \\ 4, \text{otherwise} \end{cases}$$

When $|i| = 0$, it is a virtual instruction.

Type checking. Figure 3.11 presents TAL$_2$'s type system, which maintains syntax-directed type checking while accommodating polymorphism and recursive types.

The typing judgments of TAL$_2$ are almost the same as those in TAL$_0$, except some of them have an additional type context, Δ, as an input. The judgment $\Psi; \Delta; l; \phi \vdash_{\texttt{b}} B$ means that the instruction block B is wellformed with respect to the

$\boxed{\vdash_{\mathtt{p}} P : \Psi}$

$$\frac{\Psi; 0 \vdash_{\mathtt{f}} P}{\vdash_{\mathtt{p}} P : \Psi}\ \mathsf{prog}$$

$\boxed{\Psi; l \vdash_{\mathtt{f}} P}$

$$\frac{\begin{array}{c}\Psi(l) = \forall\vec{\alpha}.\,\mathsf{codeptr}\,(\phi) \qquad \vec{\alpha} \vdash \phi\ \mathrm{wf} \qquad \forall l < x < l + |B|.\ x \notin \mathrm{dom}(\Psi) \\ \Psi; \vec{\alpha}; l; \phi \vdash_{\mathtt{b}} B \qquad \Psi; l + |B| \vdash_{\mathtt{f}} P\end{array}}{\Psi; l \vdash_{\mathtt{f}} (B; P)}\ \mathsf{frag1}$$

$$\frac{\begin{array}{c}\Psi(l) = \forall\vec{\alpha}.\,\mathsf{codeptr}\,(\phi) \qquad \vec{\alpha} \vdash \phi\ \mathrm{wf} \qquad \forall x > l.\ x \notin \mathrm{dom}(\Psi) \\ \Psi; \vec{\alpha}; l; \phi \vdash_{\mathtt{b}} B\end{array}}{\Psi; l \vdash_{\mathtt{f}} B}\ \mathsf{frag2}$$

$\boxed{\Psi; \Delta; l; \phi \vdash_{\mathtt{b}} B}$

$$\frac{\Delta\ \vdash_{\mathtt{i}}\ \{\phi_1\}\, i\, \{\phi_2\} \quad \Psi; \Delta; l + |i|; \phi_2 \vdash_{\mathtt{b}} B}{\Psi; \Delta; l; \phi_1 \vdash_{\mathtt{b}} (i; B)}\ \mathsf{seq}$$

$$\frac{\begin{array}{c}\Psi(l_d) = \forall\alpha_1, \ldots, \alpha_n.\,\mathsf{codeptr}\,(\phi_d) \qquad \mathit{inst} = [\tau_1, \ldots, \tau_n] \\ \Delta \vdash \phi\ <:\ \phi_d[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]\end{array}}{\Psi; \Delta; l; \phi \vdash_{\mathtt{b}} (\mathtt{ba}\ l_d\ \mathtt{with}\ \mathit{inst})}\ \mathsf{ba}$$

$$\frac{\begin{array}{c}\Psi(l_d) = \forall\alpha_1, \ldots, \alpha_n.\,\mathsf{codeptr}\,(\phi_d) \qquad \mathit{inst_1} = [\tau_1, \ldots, \tau_n] \\ \Psi(l+4) = \forall\alpha'_1, \ldots, \alpha'_m.\,\mathsf{codeptr}\,(\phi') \qquad \mathit{inst_2} = [\tau'_1, \ldots, \tau'_m] \\ \Delta \vdash \phi[r \mapsto \mathsf{int}_{=}(0)]\ <:\ \phi_d[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n] \\ \Delta \vdash \phi[r \mapsto \mathsf{int}_{\neq}(0)]\ <:\ \phi'[\tau'_1/\alpha'_1, \ldots, \tau'_m/\alpha'_m]\end{array}}{\Psi; \Delta; l; \phi \vdash_{\mathtt{b}} (\mathtt{bz}\ s, l_d\ \mathtt{with}\ (\mathit{inst_1}, \mathit{inst_2}))}\ \mathsf{bz}$$

$\boxed{\Delta \vdash_{\mathtt{i}} \{\phi_1\}\, i\, \{\phi_2\}}$

$$\frac{\Delta \vdash \phi <: \{\, s_1 \mapsto \mathsf{int}, s_2 \mapsto \mathsf{int} \,\}}{\Delta \vdash_{\mathtt{i}} \{\phi\}\, (\mathtt{add}\ s_1, s_2, d)\, \{\phi[d \mapsto \mathsf{int}]\}}\ \mathsf{add}$$

$$\frac{\Delta \vdash \phi <: \{\, s \mapsto \mathsf{box}\,(\tau) \,\}}{\Delta \vdash_{\mathtt{i}} \{\phi\}\, (\mathtt{ld}\ s, d)\, \{\phi[d \mapsto \tau]\}}\ \mathsf{ld}$$

$$\frac{\Delta \vdash \phi <: \{\, d \mapsto \tau[\mathsf{rec}\,\alpha.\ \tau/\alpha] \,\}}{\Delta \vdash_{\mathtt{i}} \{\phi\}\, (\mathtt{fold}\ d\ \mathtt{to}\ \mathsf{rec}\,\alpha.\ \tau)\, \{\phi[d \mapsto \mathsf{rec}\,\alpha.\ \tau]\}}\ \mathsf{fold}$$

$$\frac{\Delta \vdash \phi <: \{\, d \mapsto \mathsf{rec}\,\alpha.\ \tau \,\}}{\Delta \vdash_{\mathtt{i}} \{\phi\}\, (\mathtt{unfold}\ d)\, \{\phi[d \mapsto \tau[\mathsf{rec}\,\alpha.\ \tau/\alpha]]\}}\ \mathsf{unfold}$$

$\boxed{\Delta \vdash \tau_1 <: \tau_2,\ \Delta \vdash \phi_1 <: \phi_2}$

$$\frac{\Delta \vdash \tau \text{ wf}}{\Delta \vdash \tau <: \tau}\ \mathsf{s\text{-}refl} \qquad \frac{}{\Delta \vdash \mathsf{int}_{=}(n) <: \mathsf{int}}\ \mathsf{s\text{-}int}_{=} \qquad \frac{}{\Delta \vdash \mathsf{int}_{\neq}(n) <: \mathsf{int}}\ \mathsf{s\text{-}int}_{\neq}$$

$$\frac{\mathrm{dom}(\phi_1) \supseteq \mathrm{dom}(\phi_2) \quad \forall d \in \mathrm{dom}(\phi_1) \cap \mathrm{dom}(\phi_2).\ \Delta \vdash \phi_1(d) <: \phi_2(d)}{\Delta \vdash \phi_1 <: \phi_2}\ \mathsf{s\text{-}rfile}$$

$\boxed{\Delta \vdash \tau \text{ wf},\ \Delta \vdash \phi \text{ wf}}$

$$\frac{}{\Delta \vdash \mathsf{int} \text{ wf}}\ \mathsf{wf\text{-}int} \qquad \frac{}{\Delta \vdash \mathsf{int}_{=}(n) \text{ wf}}\ \mathsf{wf\text{-}inteq} \qquad \frac{}{\Delta \vdash \mathsf{int}_{\neq}(n) \text{ wf}}\ \mathsf{wf\text{-}intneq}$$

$$\frac{\Delta \vdash \tau \text{ wf}}{\Delta \vdash \mathsf{box}\,(\tau) \text{ wf}}\ \mathsf{wf\text{-}box} \qquad \frac{\Delta, \alpha \vdash \tau \text{ wf}}{\Delta \vdash \mathsf{rec}\,\alpha.\ \tau \text{ wf}}\ \mathsf{wf\text{-}rec} \qquad \frac{1 \leq i \leq n}{\alpha_1, ..., \alpha_n \vdash \alpha_i \text{ wf}}\ \mathsf{wf\text{-}tyvar}$$

$$\frac{\forall 1 \leq i \leq n.\ \Delta \vdash \tau_i \text{ wf}}{\Delta \vdash \{\, d_1 \mapsto \tau_1, \ldots, d_n \mapsto \tau_n \,\} \text{ wf}}\ \mathsf{wf\text{-}rfile}$$

Figure 3.11: TAL$_2$: Typing rules

precondition ϕ, which is closed under the type context Δ. In the same way, in the judgment $\Delta \vdash_{\mathtt{i}} \{\phi_1\}\, i\, \{\phi_2\}$, the type context Δ restricts the type variables in ϕ_1 and ϕ_2. The type contexts in $\Delta \vdash \tau_1 <: \tau_2$ and $\Delta \vdash \phi_1 <: \phi_2$ serve the same purpose. Finally, a new judgment $\Delta \vdash \tau$ wf (respectively, $\Delta \vdash \phi$ wf) means that τ (respectively, ϕ) is closed with respect to Δ.

Most of the typing rules in Figure 3.11 are self-explanatory. We explain only the fold and ba rules in detail. The fold rule folds the type of the register d into a recursive type; it requires the register d be of the type that is the unfolding of the recursive type.

The ba rule first looks up the type of the destination address. It then checks that the size of the instantiation is the same as the number of types variables in the destination address. The reason for this check is to ensure that the type variables in $\phi_d[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ are with respect to the current type context, Δ, so that the rule can further check the precondition ϕ is a subtype of $\phi_d[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ with respect to Δ.

Adjustments to the models. To accommodate polymorphism, recursive types, and virtual instructions, the models of typing judgments has to change accordingly, but only with some minor adjustments.

First, the indexed model of types in Section 3.2.5 can model polymorphism, recursive types, and many other types. Discussing how to model these types in detail, however, is beyond the scope of this dissertation and interested readers should refer to papers by Appel and McAllester [9], Swadi [64], and Ahmed [4]. Our implementation provides the TML type theory to hide the indexed model of types.

$$
\begin{array}{lcl}
\models_{\mathtt{p}} P : \Psi & \triangleq & P@0; \{\} \models_{\mathcal{L}_c} \Psi \\
\Psi; l \models_{\mathtt{f}} P & \triangleq & P@l; \Psi \models_{\mathcal{L}_c} |\Psi|_{\geq l} \\
\Psi; \Delta; l; \phi \models_{\mathtt{b}} B & \triangleq & B@l; \Psi \models_{\mathcal{L}_c} \{l \mapsto \forall\Delta.\,\mathsf{codeptr}\,(\phi)\} \\
\Delta \models_{\mathtt{i}} \{\phi_1\}i\{\phi_2\} & \triangleq & \big(|i| > 0 \wedge \\
& & \quad \forall l.\ i@l; \{\, l + |i| \mapsto \forall\Delta.\,\mathsf{codeptr}\,(\phi_2)\,\} \models_{\mathcal{L}_c} \{\, l \mapsto \forall\Delta.\,\mathsf{codeptr}\,(\phi_1)\,\}\big) \\
& & \vee\ \big(|i| = 0\ \wedge\ \mathrm{subtype}(\phi_1, \phi_2)\big) \\
\Delta \models \tau_1\ <:\ \tau_2 & \triangleq & \mathrm{closedtype}(\tau_1, \Delta)\ \wedge\ \mathrm{closedtype}(\tau_2, \Delta)\ \wedge\ \mathrm{subtype}(\tau_1, \tau_2) \\
\Delta \models \phi_1\ <:\ \phi_2 & \triangleq & \mathrm{closedtype}(\phi_1, \Delta)\ \wedge\ \mathrm{closedtype}(\phi_2, \Delta)\ \wedge\ \mathrm{subtype}(\phi_1, \phi_2) \\
\Delta \models \tau\ \mathrm{wf} & \triangleq & \mathrm{closedtype}(\tau, \Delta) \\
\Delta \models \phi\ \mathrm{wf} & \triangleq & \mathrm{closedtype}(\phi, \Delta)
\end{array}
$$

Figure 3.12: TAL$_2$: Models of typing judgments

For our presentation purpose, we introduce some predicates in the TML type theory.

$$\mathrm{subtype}(\tau_1, \tau_2)$$

means that τ_1 is a subtype of τ_2 in TML; τ_1 and τ_2 can be open types. The predicate

$$\mathrm{closedtype}(\tau, \Delta)$$

means that τ is a closed type with respect to Δ, or all type variables in τ are in Δ.

Also, TML treats types and register-file types uniformly. Therefore, we also use the notation $\mathrm{subtype}(\phi_1, \phi_2)$ and $\mathrm{closedtype}(\phi, \Delta)$.

With these predicates from TML, Figure 3.12 presents the models of the typing judgments in TAL$_2$. Most of the definitions are straightforward and very similar to the ones in TAL$_0$. We explain only the definition of $\Delta \models_{\mathtt{i}} \{\phi_1\}i\{\phi_2\}$, since its definition needs to handle the case of virtual instructions.

To see why virtual instructions need to be specially handled, we first discuss why the old model of $\Delta \models_{\mathtt{i}} \{\phi_1\}i\{\phi_2\}$ in TAL$_0$ will not work. The old model in Figure 3.5 on page 74 is as follows:

$$\models_{\mathtt{i}} \{\phi_1\}i\{\phi_2\} \quad \triangleq \quad \forall l.\ i@l; \{l + |i| \mapsto \mathsf{codeptr}\,(\phi_2)\} \models_{\mathcal{L}_c} \{l \mapsto \mathsf{codeptr}\,(\phi_1)\}$$

Based on the meaning of $\models_{\mathcal{L}_c}$ (Eq. 3.4 on page 82), the above definition means that if address $l + |i|$ is a code pointer with precondition ϕ_2, then address l is a code pointer with precondition ϕ_1, but *to a higher approximation.*

If the instruction i is a real instruction, then it takes one step to reach $l + |i|$ from l, and thus we can prove l is a code pointer to a higher approximation. If i is a virtual instruction, however, we cannot prove l is a code pointer to a higher approximation since the virtual instruction performs no real computation.

We explain our solution next. We first look at the case that a virtual instruction i_1 is immediately followed by a real instruction i_2:

$$\begin{array}{lrl} l: & \{\,\phi_1\,\} & \\ & i_1; & \text{a virtual instruction} \\ l: & \{\,\phi_2\,\} & \\ & i_2; & \text{a real instruction} \\ l+4: & \{\,\phi_3\,\} & \end{array}$$

Notice that since the size of i_1 is zero, the address before i_1 and after are the same.

Since i_2 is a real instruction, we can prove that if $l{+}4$ is a code pointer, then l is a code pointer to a higher approximation. For i_1, since it performs type manipulation, we can safely assume that ϕ_1 is stronger than ϕ_2, or ϕ_1 is a subtype of ϕ_2. (We will give an example to this later.) This means that if l is a code pointer with precondition ϕ_2, then l is also a code pointer with precondition ϕ_1, because of the contravariance of code-pointer types. Therefore, considering i_1 and i_2 as a whole, we can still prove that if $l+4$ is a code pointer with precondition ϕ_3, then l is a code pointer with precondition ϕ_1, *even to a higher approximation.* Although i_1 performs no computation, it can *borrow* the computation from the next real instruction.

What if the next instruction to i_1 is also virtual? Then it can borrow the computation of the second next instruction, or the third, as long as eventually some real instruction turns up. In TAL_2, this is always the case. Since i_1 is in a basic block and the last instruction in a basic block is a control-transfer instruction—a real instruction.

Based on this intuition, the definition of $\Delta \models_{\mathtt{i}} \{\phi_1\}i\{\phi_2\}$ in Figure 3.12 has two cases. When the size of the instruction i is greater than zero, the instruction is a real machine instruction. In this case, the model is as before except that we close the code-pointer type with its context: Assuming the exit is a code pointer with the condition, $\forall\Delta.\,\mathsf{codeptr}\,(\phi_2)$, the entry is a code pointer with the condition, $\forall\Delta.\,\mathsf{codeptr}\,(\phi_1)$. When the size of the instruction i is zero, the model asserts that ϕ_1 is a subtype of ϕ_2.

We stated that in the case of virtual instructions, we can safely assume the precondition is a subtype of the postcondition. Let we look at the fold rule as an example. Based on the subtyping model, the **fold** rule is

$$\frac{\text{subtype}(\phi, \{\, d \mapsto \tau[\mathsf{rec}\,\alpha.\ \tau/\alpha]\,\})}{\text{subtype}(\phi, \phi[d \mapsto \mathsf{rec}\,\alpha.\ \tau])},$$

which essentially requires

$$\text{subtype}(\tau[\mathsf{rec}\,\alpha.\ \tau/\alpha], \mathsf{rec}\,\alpha.\ \tau).$$

The above rule states that the unrolled recursive type is a subtype of the recursive type. Not by coincidence, TML provides the rule as a lemma for recursive types.

Proof of the typing rules. After the minor adjustments to the models of typing judgments, both the rules in TAL_2 and the safety theorem can be proved sound. Next, we will discuss only the proof of the seq rule, since this rule involves $\Delta \vdash_{\mathtt{i}} \{\phi_1\}\, i\, \{\phi_2\}$, whose model we have changed to have two cases.

Lemma 3.20 *(The* seq *rule in* TAL_2*.) If* $\Delta \models_{\mathtt{i}} \{\phi_1\} i \{\phi_2\}$*, and* $\Psi; \Delta; l + |i|; \phi_2 \models_{\mathtt{b}} B$*, then* $\Psi; \Delta; l; \phi_1 \models_{\mathtt{b}} i; B$*.*

Proof. From the definitions in Figure 3.12, we need to prove

$$(i; B)@l; \Psi \models_{\mathcal{L}_c} \{\, l \mapsto \forall\Delta.\, \mathsf{codeptr}\,(\phi_1)\,\},$$

by assuming

$$\begin{aligned}
&\big(|i| > 0\\
&\quad \wedge\ \forall l.\ i@l; \{\, l + |i| \mapsto \forall\Delta.\, \mathsf{codeptr}\,(\phi_2)\,\} \models_{\mathcal{L}_c} \{\, l \mapsto \forall\Delta.\, \mathsf{codeptr}\,(\phi_1)\,\}\big)\\
&\vee\ \big(|i| = 0\ \wedge\ \text{subtype}(\phi_1, \phi_2)\big)
\end{aligned} \tag{3.20.1}$$

and

$$B@(l + |i|); \Psi \models_{\mathcal{L}_c} \{\, l + |i| \mapsto \forall\Delta.\, \mathsf{codeptr}\,(\phi_2)\,\}. \tag{3.20.2}$$

Do a case analysis on 3.20.1.

Case a): Suppose we have

$$|i| > 0 \tag{3.20.3}$$

$$\forall l.\ i@l; \{\, l + |i| \mapsto \forall\Delta.\, \mathsf{codeptr}\,(\phi_2)\,\} \models_{\mathcal{L}_c} \{\, l \mapsto \forall\Delta.\, \mathsf{codeptr}\,(\phi_1)\,\} \tag{3.20.4}$$

Perform a universal elimination on 3.20.4 using the address l, to get

$$i@l; \{\, l + |i| \mapsto \forall\Delta.\, \mathsf{codeptr}\,(\phi_2)\,\} \models_{\mathcal{L}_c} \{\, l \mapsto \forall\Delta.\, \mathsf{codeptr}\,(\phi_1)\,\} \tag{3.20.5}$$

Similar to the proof of Lemma 3.17, we first use the **combine** rule on 3.20.5 and 3.20.2, then eliminate $\{\, l + |i| \mapsto \forall\Delta.\, \mathsf{codeptr}\,(\phi_2)\,\}$ from both the left and the right. After these steps, we have the goal

$$(i; B)@l; \Psi \models_{\mathcal{L}_c} \{\, l \mapsto \forall\Delta.\, \mathsf{codeptr}\,(\phi_1)\,\}.$$

Case b): Suppose we have

$$|i| = 0 \tag{3.20.6}$$

$$\text{subtype}(\phi_1, \phi_2) \tag{3.20.7}$$

The TML type theory lets us to derive the following from 3.20.7:

$$\text{subtype}(\forall\Delta.\, \mathsf{codeptr}\,(\phi_2), \forall\Delta.\, \mathsf{codeptr}\,(\phi_1)) \tag{3.20.8}$$

Since $|i| = 0$, the assumption 3.20.2 becomes

$$B@l; \Psi \models_{\mathcal{L}_c} \{\, l \mapsto \forall\Delta.\, \mathsf{codeptr}\,(\phi_2)\,\}. \tag{3.20.9}$$

Based on the definition of $C; \Psi' \models_{\mathcal{L}_c} \Psi$, from 3.20.9 and 3.20.8, it is easy to prove that

$$B@l; \Psi \models_{\mathcal{L}_c} \{\, l \mapsto \forall\Delta.\, \mathsf{codeptr}\,(\phi_1)\,\}.$$

When $|i| = 0$, the instruction i is a virtual instruction and thus $(i; B)@l$ is the same as $B@l$. Therefore, we have the goal

$$(i; B)@l; \Psi \models_{\mathcal{L}_c} \{\, l \mapsto \forall\Delta.\, \mathsf{codeptr}\,(\phi_1)\,\}.$$

$\square$

Existential types. Since we already have type variables, and virtual instructions for manipulating types, adding existential types to TAL$_2$ is a fairly small step.

First, we add existential types, and then add instructions to pack and unpack exisstentials:

$$\begin{array}{llll}
(\textit{types}) & \tau & ::= & \ldots \mid \exists\alpha.\, \tau \\
(\textit{instructions}) & i & ::= & \ldots \mid \texttt{unpack}\ d\ \texttt{with}\ \alpha \mid \texttt{pack}\ d, \tau_1\ \texttt{to}\ \exists\alpha.\, \tau
\end{array}$$

Since the unpack instruction introduces a new type variable, we have to change the judgment for instructions to $\Delta_1 \vdash_{\texttt{i}} \{\phi_1\}\, i\, \{\phi_2\} \Rightarrow \Delta_2$, where Δ_2 is the new type context after the instruction i. With this new judgment, the seq rule becomes

$$\frac{\Delta_1 \vdash_{\mathtt{i}} \{\phi_1\}\, i\, \{\phi_2\} \Rightarrow \Delta_2 \quad \Psi; \Delta_2; l + |i|; \phi_2 \vdash_{\mathtt{b}} B}{\Psi; \Delta_1; l; \phi_1 \vdash_{\mathtt{b}} (i; B)}\ \textsf{seq},$$

where the remaining of the block takes the new type context.

Rules for the pack and unpack instructions are

$$\frac{\Delta \vdash \phi <: \{\, d \mapsto \exists \alpha.\, \tau \,\} \quad \alpha \notin \Delta}{\Delta \vdash_{\mathtt{i}} \{\phi\}\, (\mathtt{unpack}\ d\ \mathtt{with}\ \alpha)\, \{\phi[d \mapsto \tau]\} \Rightarrow \Delta, \alpha}\ \textsf{unpack}$$

$$\frac{\Delta \vdash \phi <: \{\, d \mapsto \tau[\tau_1/\alpha] \,\} \quad \alpha \notin \Delta}{\Delta \vdash_{\mathtt{i}} \{\phi\}\, (\mathtt{pack}\ d, \tau_1\ \mathtt{to}\ \exists \alpha.\, \tau)\, \{\phi[d \mapsto \exists \alpha.\, \tau]\} \Rightarrow \Delta}\ \textsf{pack}$$

To prove the soundness of the pack and unpack rules, the changes to the models of typing judgments are fairly small: We only need to give a model to $\Delta_1 \vdash_{\mathtt{i}} \{\phi_1\}\, i\, \{\phi_2\} \Rightarrow \Delta_2$.

$$\begin{aligned}
&\Delta_1 \models_{\mathtt{i}} \{\phi_1\}\, i\, \{\phi_2\} \Rightarrow \Delta_2 \ \triangleq \\
&\quad \big(|i| = 4 \wedge \\
&\qquad \forall l.\ i@l; \{\, l + 4 \mapsto \forall \Delta_2.\, \textsf{codeptr}\,(\phi_2) \,\} \models_{\mathcal{L}_c} \{\, l \mapsto \forall \Delta_1.\, \textsf{codeptr}\,(\phi_1) \,\}\big) \\
&\quad \vee\ \big(|i| = 0\ \wedge\ \mathrm{subtype}(\forall \Delta_2.\, \textsf{codeptr}\,(\phi_2), \forall \Delta_1.\, \textsf{codeptr}\,(\phi_1))\big)
\end{aligned}$$

Comparing to the model of $\Delta \models_{\mathtt{i}} \{\phi_1\} i \{\phi_2\}$ on page 94, the changes are that we close ϕ_2 with respect to Δ_2. Also, we cannot say subtype(ϕ_1, ϕ_2) here, because ϕ_1 and ϕ_2 have different sets of type variables. Instead, we state subtype$(\forall \Delta_2.\, \textsf{codeptr}\,(\phi_2), \forall \Delta_1.\, \textsf{codeptr}\,(\phi_1))$, which is enough for our proofs.

$$\begin{array}{rccl} & \cdots & & \\ \boxed{(\textit{register indexes})} & d, s & ::= & 1 \mid 2 \mid \ldots \mid 31 \\ & \cdots & & \\ (\textit{instructions}) & B & ::= & \ldots \mid \boxed{\texttt{alloc}\ s, d} \\ & \cdots & & \end{array}$$

Figure 3.13: TAL_3: Syntax (Only the differences, in framed boxes, from TAL_0 are shown.)

3.5 TAL_3

Up until now, our typed-assembly languages have not included memory-related features such as memory allocation, mutable references, and memory deallocation. Our FPCC project supports memory allocation, essentially by following the memory-allocation model by Appel and Felty [8]. We also implement a semantic model for mutable references, by Ahmed et al. [3] so that programs can safely update data structures in the memory. However, our implementation does not currently support either explicit deallocation (`free`) or garbage collection.

In this section, we will add memory allocation on top of TAL_0 to have TAL_3 (Figure 3.13). The main purpose of this section is to illustrate an important technique in our proofs — imaginary parts of states. Before discussing this technique, we first introduce memory allocation in TAL_3.

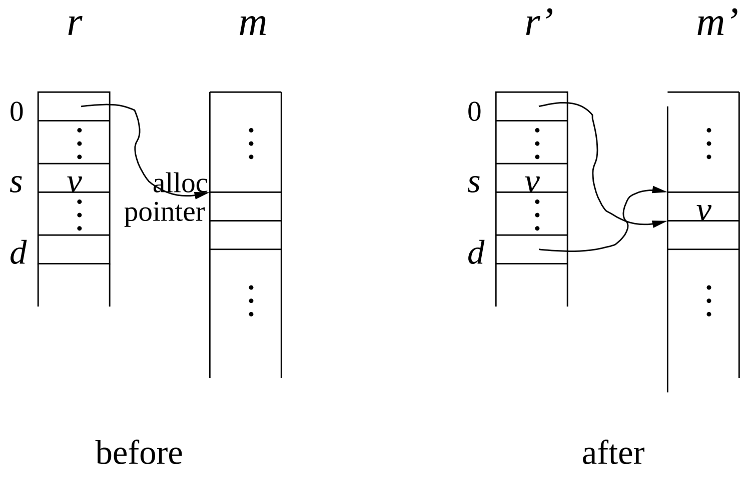
Memory allocation. TAL_3 can allocate immutable reference pointers. It assumes there is a dedicated allocation pointer, say register zero, which points to the boundary between the allocated region and the unallocated region. For simplicity, we assume that the heap is unbounded, and thus there is no limit pointer.[6]

[6]In our real system, there is a register pointing to the end of the heap.

TAL$_3$ has an allocation instruction, "`alloc` s, d", which performs an initialization at the address pointed by register zero (the allocation pointer), records the value of register zero into the register d, and then increments the register zero[7]:

$$
\begin{aligned}
&\texttt{alloc}\ s, d \triangleq \\
&\quad \lambda(r, m), (r', m'). \\
&\qquad \big(r'(\mathrm{pc}) = r(\mathrm{pc}) + 4\big) \ \wedge\ r'(d) = r(0) \ \wedge\ r'(0) = r(0) + 1 \\
&\qquad\quad \wedge\ \big(\forall x \notin \{\,\mathrm{pc}, 0, d\,\}.\ r'(x) = r(x)\big) \\
&\qquad \wedge\ \big(m'(r(0)) = r(s) \ \wedge\ \forall x \neq r(0).\ m'(x) = m(x)\big).
\end{aligned}
$$

States before and after the memory allocation instruction are depicted in the following graph.



The graph shows that if the register s is of type τ, then after "`alloc` s, d", the register d is a pointer to a value of type τ, or it has type $\mathsf{box}(\tau)$. Therefore, the

[7]In TAL$_3$, we assume "`alloc` s, d" be a single instruction. But on SPARC, the `alloc` s, d instruction is actually a macro instruction: It is a sequence of a store instruction, a move instruction, and an add-by-one instruction. Our implementation can reason about the intermediate states between these SPARC instructions: We use a virtual allocation pointer to point to the boundary between the allocated region and unallocated region, and use types to relate the value of the allocation pointer to real registers. Interested readers should refer to Chen's thesis [16, ch4.4].

typing rule in TAL$_3$ for memory allocation is as follows:

$$\frac{\phi <: \{ s \mapsto \tau \}}{\vdash_{\mathtt{i}} \{\phi\}(\mathtt{alloc}\ s, d)\{\phi[d \mapsto \mathsf{box}\,(\tau)]\}}\ \mathsf{alloc}$$

Imaginary parts. Next, we use the technique of imaginary parts to model memory allocation in TAL$_3$. Note, however, the model can handle neither mutable references nor memory deallocation. Modeling of mutable references will be sketched at the end of this section.

We first explain the difficulty of modeling memory allocation and why imaginary parts can help. In the following example, register one is of a reference type; after the allocation instruction, register one should still be of the reference type.

$$\begin{array}{l} \quad \{ 1 \mapsto \mathsf{box}\,(\mathsf{bool}), 5 \mapsto \mathsf{int} \} \\ \mathtt{alloc}\ 5, 6 \\ \quad \{ 1 \mapsto \mathsf{box}\,(\mathsf{bool}), 5 \mapsto \mathsf{int}, 6 \mapsto \mathsf{box}\,(\mathsf{int}) \} \end{array}$$

But the difficulty is, how do we know that the memory updating operation performed by "$\mathtt{alloc}\ 5, 6$" has not destroyed the memory content pointed by register one? In other words, we need a discipline to ensure old references are still valid after a memory allocation.

The solution is to remember an allocation set for each state. Each state will also maintain the invariant that every reference cell is in the allocation set. New reference cells are always allocated outside of the current allocation set and therefore will not destroy the old reference cells. Finally, the allocation set are enlarged after each memory allocation.

Allocation sets do not exist on concrete architectures and are purely imaginary. Their purpose are to aid our modeling, or to enforce our memory-allocation discipline.

To include allocation sets, we change the definition of a state to be a triple:

$$\sigma = (r, m, a),$$

where the register bank r and the memory m are the concrete part, and the allocation set a is the imaginary part. The notation $\mathrm{r\_of}(\sigma)$, $\mathrm{m\_of}(\sigma)$, and $\mathrm{a\_of}(\sigma)$ project σ into its register bank, memory, and allocation set, respectively.

We change the model of $\mathsf{box}\,(\tau)$ to enforce that every reference cell is in the allocation set:

$$(\sigma, x) :_k \mathsf{box}\,(\tau) \;\triangleq\; x \in \mathrm{a\_of}(\sigma) \;\wedge\; \mathrm{readable}(x) \;\wedge\; (\sigma, \mathrm{m\_of}(\sigma)(x)) :_{k-1} \tau,$$

where $x \in \mathrm{a\_of}(\sigma)$ makes sure that the address x is in the allocation set.

Because TAL_3 maintains register zero as the boundary between allocated addresses and unallocated ones, addresses in the allocation set of a state should be always less than the value of register zero. We define a predicate, $\mathrm{valid\_state}(\sigma)$, to capture this invariant:

$$\mathrm{valid\_state}(\sigma) \;\triangleq\; \forall x \in \mathrm{a\_of}(\sigma).\ \mathrm{r\_of}(\sigma)(0) > x.$$

Finally, we need to change the model of code-pointer types. The old model (Eq. 3.2) of $\mathsf{codeptr}\,(\phi)$ quantifies over all future states σ' such that the current state σ steps to, or $\sigma \mapsto^* \sigma'$. But this is not appropriate in the presence of memory allocation: We only care about those future states that are valid states; furthermore, in TAL$_3$'s memory model, we only care about those future states that extend the current allocation set and preserve values in the current allocation set. Therefore, we define an extend-state relation, extend_state(σ, σ'), to capture the notion of valid state extensions:

$$\begin{aligned}\text{extend\_state}(\sigma, \sigma') \;&\triangleq\; \text{valid\_state}(\sigma) \;\wedge\; \text{valid\_state}(\sigma') \\ &\qquad \wedge\; \text{a\_of}(\sigma) \subseteq \text{a\_of}(\sigma') \\ &\qquad \wedge\; \forall x \in \text{a\_of}(\sigma).\ \text{m\_of}(\sigma)(x) = \text{m\_of}(\sigma')(x)\end{aligned}$$

Then, the new model of code-pointer types quantifies only those future states that extend the current state:

$$\begin{aligned}(\sigma, x) :_k \textsf{codeptr}\,(\phi) \;&\triangleq\; \forall \sigma', j < k.\ \text{extend\_state}(\sigma, \sigma') \;\wedge\; \big(\text{r\_of}(\sigma')(\text{pc}) = x\big) \;\wedge\; \sigma' :_j \phi \\ &\qquad \Rightarrow\; \text{safe\_state}((\text{r\_of}(\sigma), \text{m\_of}(\sigma)), j)\end{aligned} \tag{3.9}$$

We have introduced the model of memory allocation in TAL$_3$, and adjusted the model of reference types and code-pointer types. With these adjustments, we can prove the soundness of the alloc rule. We do not go into detail of this proof, but only say that the important step is to find a new allocation set in the new state after the execution of `alloc` s, d: The new allocation set is not hard to find, it is just the old allocation set plus the memory cell pointed by the old allocation pointer.

Mutable references. We have used imaginary parts to model memory allocation. Ahmed et al. [3] also used imaginary parts to model mutable references. In their model, an allocation set is not just a set of addresses, but a mapping from addresses to the syntax of types. That is, the imaginary part remembers the type of each reference cell so that we can enforce the discipline of type-preserving memory updating. This model of mutable references is the model we adopted in our real system. We refer readers to the paper by Ahmed et al. for details.

Chapter 4

Implementation in FPCC

The major step of the FPCC project at Princeton is to prove the soundness of our Typed Assembly Language, LTAL, from higher-order logic and SPARC machine semantics, so that a typing derivation at the LTAL level can be mapped into a proof from the foundations of mathematics.

This step is rather difficult, since there are many operators and rules in LTAL — in fact, over a thousand. Table 4.1 shows a breakdown of these rules.

Fortunately, most of the operators in LTAL have straightforward models. Furthermore, our proofs are highly structured thanks to our two-step process. We first design an intermediate layer and show its soundness. This layer includes our logic for control flow, $\mathcal{L}_c$, and our type theory, TML. Then we encode LTAL's operators from the interface exposed by the intermediate layer, and prove soundness of LTAL's rules from lemmas provided by the intermediate layer. Because of this two-step process, type refinement rules and substitution rules in LTAL are simply lemmas (or combination of lemmas) provided by TML and therefore requires small effort to prove.

Category	Number of operators/rules
LTAL instruction constructors	51
LTAL instruction rules	54
SPARC instruction constructors	196
SPARC instruction decoding rules	263
Type constructors	27
Wellformedness of types	98
Type refinement rules	69
Structural matching heuristics for type expressions	50
Coercion operators and rules	79
Substitution operators + rules	48
Environment management	73
Arithmetic	44
Register-map, label-map rules	79
TOTAL	1131

Table 4.1: LTAL: Breakdown of operators and rules [16, ch2.5]

Rules that are difficult to prove and particularly relevant to this dissertation are the instruction rules in LTAL. To understand why they are difficult to prove, we look at an example. LTAL has its own instruction syntax and these LTAL instructions are implemented by SPARC instructions. One LTAL instruction is the addition instruction, $z = x + y$, implemented by the SPARC `add` instruction. One rule in LTAL for the addition instruction requires x and y be of integer types beforehand,

and z gets an integer type afterward:

$$\frac{\begin{array}{llll} (1) & LRT; \rho; \Phi \vdash x : \mathrm{int}_{32} & (2) & LRT; \rho; \Phi \vdash y : \mathrm{int}_{32} \\ (3) & \ell' = \ell + 4 & & \\ (4) & \mathrm{rmap}(LRT)(z) = t_z & (5) & \mathrm{rmap}(LRT)(x) = t_x \\ (6) & \mathrm{realreg}(t_z) = r_z & (7) & \mathrm{realreg}(t_x) = r_x \\ (8) & y_m = \mathrm{match\_reg\_or\_imm}(y) & & \\ (9) & \Phi' = \{z : \mathrm{int}_{32}\} \cap (\Phi \backslash z) & & \\ (10) & \mathrm{decode\_list}\ \ell\ \ell' P\ P'\ \mathrm{i\_ADD}(r_x, y_m, r_z) & & \end{array}}{LRT; \Gamma \vdash (\ell; \rho; \hbar; \Phi; cc; P)\{z = x + y\}(\ell'; \rho; \hbar; \Phi'; cc; P')} \tag{4.1}$$

The details of the rule are not important here. But as we can see, the rule is quite complicated. For the simple arithmetic-addition instruction, it involves ten premises to deal with things like register maps, label maps, immediate operators, and other issues.

LTAL's instruction rules have three categories. Table 4.2 presents a breakdown. The first category includes LTAL's composition rules. These rules combine LTAL instructions into basic blocks and combine basic blocks into a whole program.[1] The second category includes those rules for LTAL instructions that do not correspond to any SPARC instruction. Similar to the "`unfold` d" instruction in TAL_2 (page 89), these LTAL instructions perform pure type manipulation and are virtual instructions. The last category includes those rules for LTAL instructions that correspond to at least one SPARC instruction; we call these LTAL instructions real instructions. One example rule in this category is the rule (4.1) we have just seen.

[1] In TAL_0, a program is (C, Ψ), where the code C and the global address invariant Ψ are separated. LTAL's syntax, however, mixes code with the invariant. Therefore, LTAL's composition rules are also responsible for scanning the program to construct the address invariant.

Category	Number of rules
Composition rules	17
Rules for virtual instructions	5
Rules for real instructions	32

Table 4.2: LTAL: Breakdown of instructions rules

LTAL instruction rules in different categories are proved by following different schemes:

Composition rules. LTAL's composition rules are proved sound directly from $\mathcal{L}_c$'s composition rules. In Chapter 2, we have shown how to encode and prove $\mathcal{L}_c$'s composition rules on small-step machine semantics. In Chapter 3, we have shown how to justify TAL$_0$'s composition rules based on $\mathcal{L}_c$'s rules. We have also shown how to handle virtual instructions in TAL$_2$. These techniques apply to LTAL as well. The only new complexity is that a whole instruction block in LTAL can be virtual and not correspond to any machine instruction. To address this complexity, the model of block wellformedness in LTAL has two cases. One case handle blocks that have at least one SPARC instruction; the other case uses subtyping to handle virtual blocks — essentially the same trick as the one in the model of $\Delta \models_{\mathtt{i}} \{\phi_1\}i\{\phi_2\}$ (page 94) in TAL$_2$.

Rules for virtual instructions. Since the model for these rules is subtyping, we use subtyping lemmas from TML to prove these rules.

Rules for real instructions. This category is where the majority of the work is, and we will focus on it in the following discussion.

LTAL has 32 instruction rules for real instructions, which correspond to SPARC instructions such as `add`, `ld`, etc. Table 4.3 lists the categories of these rules: arithmetic instructions, branching instructions, instructions for setting condition codes,

Category	Number of rules	SPARC instructions used
Arithmetic; address arithmetic; label arithmetic	15	`add`, `sub`, `or`, `smul` (signed multiplication), `sdiv` (signed division), `wry` (write Y register), `sethi` (set high 22 bits).
Branching instructions	7	`ba`, `jmpl` (jump and link), `bgeu`, `bz`, `bne`, `blu`, `bl`, `ble`, `bg`, `bge`
Setting condition codes	4	`subcc` (sub with condition code)
Memory instructions: load, store, get tags, ...	6	`ld`, `st`.

Table 4.3: LTAL: Breakdown of rules for real instructions

and memory instructions. These rules are proved sound from higher-order logic and SPARC machine semantics. In Section 4.2, we will focus on one rule and show main structures of its proof. In Section 4.3, we will list all the typing rules we have proved for SPARC instructions. Before these, we discuss general principles in our proofs and statistics of the proof size.

One principle in our implementation is a proof-by-need principle. That is, we develop proofs only for those SPARC instructions used by our FPCC-ML compiler. Table 4.3 lists the SPARC instructions that the compiler uses. Since the compiler only uses a subset of SPARC instruction, we have developed proofs only for the subset.

Even with the proof-by-need principle , proving the soundness of those 37 rules from SPARC instruction semantics is still an enormous task. Fortunately, many parts of the proof can be factored out.

Proof factoring. To prove a LTAL real instruction rule, we find that it is more convenient to state and prove a version in $\mathcal{L}_c$ first, and then go from this version to prove the soundness of the LTAL rule. The reason for such a setup is that LTAL may use a same SPARC instruction in different scenarios. For example, the `bgeu` instruction (to branch on greater than or equal, unsigned) is used by LTAL to test if there is enough memory, to perform data-tag discrimination, and to test if a datatype constructor is in a boxed representation or not; it is used by four LTAL rules. But we only need to state one version in $\mathcal{L}_c$ for `bgeu`, and then all four rules can be proved sound from that version.

The second reason that our proofs about machine instructions are highly factored is that machine instruction sets are highly factored, both in syntax and semantics. Consider the example about ALU operations from the paper by Michael and Appel [42]. The ALU takes its input from two registers (or a register and an immediate) and produces the result in another. The only difference between ALU instructions is the operation performed. The definition of aluxcc below is reused to define 23 different ALU instructions. Argument *with_carry* specifies whether the instruction operates with a "carry", *modifies_icc* specifies whether it modifies integer condition codes, and *alufun* is the predicate describing the operation performed by the instruction.

$$
\begin{array}{l}
\text{aluxcc} \triangleq \\
\quad \lambda \mathit{with\_carry}, \mathit{modifies\_icc}, \mathit{alufun}. \\
\quad\quad \lambda s, \mathit{reg\_imm}, d. \\
\quad\quad\quad \lambda (r, m), (r', m'). \\
\quad\quad\quad\quad \ldots
\end{array}
$$

Then, the aluxcc predicate is used to define 23 ALU instructions:

`add`	$\triangleq$	aluxcc false false plus_mod_32.	
`and`	$\triangleq$	aluxcc false false and_oper.	
`andcc`	$\triangleq$	aluxcc false true and_oper.	
...	...	...	– 20 cases omitted.

Since machine-instruction syntax and semantics are highly factored, our proofs about machine instructions are also factored. For example, instead of proving only the `add` instruction takes two source registers of integer types and puts an integer value into the destination register, we prove that every ALU instruction has such a typing[2] , and the case of the `add` instruction is just one instantiation.

Proof statistics. Table 4.4 lists the size of our proofs. FPCC, as a whole, has over 140k useful lines of proofs. Out of these proofs, over 30k lines of proofs are to prove the soundness of LTAL instruction rules. These proofs have two parts. The first part is the proofs for $\mathcal{L}_c$ and typing lemmas for SPARC instructions; this step has over 27k lines of proofs. The second part is the proofs from lemmas at the $\mathcal{L}_c$'s level to LTAL instruction rules. This step has around 3k lines of proofs so far.

4.1 A Summary of Notation

In Chapter 3, we have discussed many issues that our implementation deals with, including polymorphism, existentials, and imaginary parts for memory management.

[2]More accurately, only those ALU instructions that do not change integer condition codes have such a typing. For those that do change the integer condition codes, we also need to consider the type of the condition-code register.

[3]Excluding comments and empty lines.

	Number of lines	Number of useful lines[3]
Proofs in FPCC	164220	140938
Proofs of LTAL instruction rules	35403	30740
1) Implementation of $\mathcal{L}_c$ & $\mathcal{L}_c$'s instruction rules	32058	27875
2) From $\mathcal{L}_c$'s instructions to LTAL's instruction rules	3445	2865

Table 4.4: Statistics of proof size

Along our discussion, we have also introduced notation and abstractions to handle these issues. However, they were introduced at different places. Furthermore, there are issues that our implementation handles but have not been discussed. Therefore, before we focus on the detailed proof in the next section, in Table 4.5 we summarize the notation and abstractions used in our implementation.

In Chapter 3, a register-file type, ϕ, mapped registers to types. But in our real system, we use temporaries instead. SML/NJ compiler has two phases to map local variables to registers and memory locations in the spill area. It first transforms a ML program into one that uses only 1000 local variables. Then it maps these local variables to temporaries, which are either implemented in registers or memory locations in the spill area. The first 20 temporaries are implemented in registers, including both general-purpose registers and special registers such as the condition-code register, but not including pc; the remainder of the temporaries are implemented in the spill area. For our proof purpose, we also have virtual temporaries. Virtual temporaries are not implemented in concrete parts of states, and are auxiliaries for helping us to organize the proof. In Section 3.5, we used a dedicated real register for the allocation pointer, but our implementation uses a more flexible scheme: We treat the allocation pointer (and a limit pointer) as virtual temporaries, and use types to relate values between the allocation pointer and real

Notation/Abstraction	Explanation
$t \in [0, tmax]$	A temporary is implemented in a register, or implemented in the spill area, or a virtual temporary.
$\sigma = (r, m, is) \in \Sigma$	A state consists of a register, a memory and an imaginary part.
$is \in \mathrm{IS}$	An imaginary part has two parts. The first part is an allocation set, which remembers the type of every reference cell. The second part has virtual temporaries for memory allocation.
$(\sigma, x) :_k \tau$	The value (σ, x) belongs to the type τ to the index k.
$\mathrm{temp\_vec}(\sigma)$	Construct a vector of temporaries from a state; this vector maps temporary numbers into values.
$\sigma :_k \phi$	Temporaries in σ have types in ϕ to the index k. It is defined as $\forall 0 \leq t < \mathrm{tmax}.\ (\sigma, \mathrm{temp\_vec}(\sigma)(t)) :_k \phi(t)$.
$\mathrm{valid\_state}(\sigma)$	Valid states: it roughly means that every reference cell has the type prescribed in the imaginary part.
$\mathrm{extend\_state}(\sigma, \sigma')$	The extend-state relation: the allocation set is extended and the type of every reference cell is preserved.

Table 4.5: A summary of notation and abstractions

registers; this way, there is no need for our compiler to dedicate a fixed register for the allocation pointer (and the limit pointer).

A state σ consists of a register, a memory, and an imaginary part. The register and the memory are the concrete part. The imaginary part in our real system is not just a set of addresses as in Section 3.5, it has two parts. The first part is an allocation map, which remembers the type of every reference cell. This is to enforce type-preserving store update in the presence of mutable references. The second part in an imaginary part holds virtual temporaries, including the allocation pointer and the limit pointer.

The notation $(\sigma, x) :_k \tau$ means that the value (σ, x) belongs to the type τ to approximation k. The notation $\sigma :_k \phi$ means that the state σ meets the temporary-file type, ϕ, to approximation k. The definition of $\sigma :_k \phi$ uses the predicate temp_vec(σ), which constructs a vector of temporaries from a state.

Finally, we have the valid_state(σ) and extend_state(σ, σ') predicate. In the presence of mutable references, valid_state(σ) roughly means that every reference cell has the type prescribed in the imaginary part of σ; extend_state(σ, σ') means that the allocation map is extended and the type of every reference cell is preserved.

4.2 The arithmetic-add Rule.

In this section, we discuss the proof of one particular rule, the arith-add rule. The rule is stated at the level of $\mathcal{L}_c$ and corresponds to the LTAL rule for LTAL's addition instruction (Rule 4.1).

$$\frac{\begin{array}{c}\mathrm{decode}(w, (\mathtt{add}\ s, \mathit{reg\_imm}, d)) \\ \mathrm{realreg}(t_s, s) \qquad \mathrm{realreg}(t_d, d) \\ \mathrm{subtype}(\phi, \{\, t_s \mapsto \mathsf{int32}\,\}) \qquad \mathrm{reg\_imm\_has\_ty}(\mathit{reg\_imm}, \mathsf{int32}, \phi) \\ \phi' = \phi[t_d \mapsto \mathsf{int32}]\end{array}}{w@l; \{\, l +_{32} 4 \mapsto \mathsf{codeptr}\,(\phi')\,\} \models_{\mathcal{L}_c} \{\, l \mapsto \mathsf{codeptr}\,(\phi)\,\}}\ \textsf{arith-add} \tag{4.2}$$

Essentially, the rule states that if both source operands of an add instruction have integer type, then the destination gets an integer type after the instruction.

Furthermore, this rule handles some machine-specific issues and we discuss them one by one.

First, we want to reason about machine code, not assembly code. Therefore, instead of writing ($\mathtt{add}\ s, \mathit{reg\_imm}, d$)@$l$, we write $w@l$ to denote that a word w is at the location l; and we use the decode relation to decode w into "($\mathtt{add}\ s, \mathit{reg\_imm}, d$)". The decode relation is part of our modeling of SPARC architecture.

Second, the arith-add rule (4.2) uses a predicate, $\mathrm{realreg}(t_s, s)$, to mean that the temporary t_s is implemented in the register s. (Thus, the rule would not be applicable to operands represented in spill locations.)

Third, the value in a register on SPARC is not an arbitrary integer, but an integer in $[0, 2^{32} - 1)$. Therefore, we define an int32 type and the arith-add rule requires the temporary t_s (implemented in the register s) must be of the type int32.

Fourth, the second operand of the add instruction on SPARC can be either a register or an immediate. The predicate, $\mathrm{reg\_imm\_has\_ty}(\mathit{reg\_imm}, \tau, \phi)$, makes sure in both cases $\mathit{reg\_imm}$ has the type τ in ϕ: If $\mathit{reg\_imm}$ is a register, then it must correspond to some temporary t and ϕ must state that t is of the type τ; if reg_imm is an immediate, then the immediate must be of the type τ.[4]

Finally, real architectures such as SPARC perform modulo arithmetic. We use the symbol $+_{32}$ to denote modulo-2^{32} addition. Then the exit address in the arith-add rule is $l +_{32} 4$, instead of $l + 4$.

4.2.1 Proof of the Rule

We first sketch the major steps of proving the arith-add rule. Based on the model of $C; \Psi' \models_{\mathcal{L}_c} \Psi$, the rule states that in a state σ if w is at the address l, then the entry l is a code pointer with the condition ϕ to the index $k + 1$, provided that the address $l +_{32} 4$ is a code pointer with the condition ϕ' to the index k. To prove that the address l is a code pointer with the condition ϕ to approximation $k + 1$, the steps are as follows:

(i) Start from a state σ' such that $\sigma \mapsto^* \sigma'$, the control of σ' is at l, and ϕ is true on σ'. The goal is to prove that the concrete part of σ' can step for $k+1$ steps.

(ii) Based on the fact that the control of σ' is at l, the word w is at the address l, and w decodes to the "`add` s, reg_imm, d" instruction, construct a new state σ'' such that the concrete part of σ' steps in one step to the concrete part of σ'' and the imaginary part remains the same. The construction of the concrete part of σ'' follows the semantics of the `add` instruction. In summary, this step shows that the state σ' can *progress* for one step.[5]

[4]In SPARC, when it is an immediate, we need to perform a sign extension on the 13-bit immediate to get a 32-bit signed integer. The result integer must be of the type τ in reg_imm_has_ty(reg_imm, τ, ϕ).

[5]When we say a state σ can step for k steps, we really mean its concrete part can step for k steps.

(iii) Prove that the new state σ'' satisfies the condition $\phi' = \phi[t_d \mapsto \mathsf{int32}]$. Based on the semantics of the `add` instruction, the value of the destination temporary is the modulo-2^{32} addition of two 32-bit integers and therefore is a 32-bit integer. Moreover, the `add` instruction does not change other temporaries.

(iv) Prove that the new state σ'' is safe for k steps. This step uses 1) the fact that the control of σ'' is at the address $l +_{32} 4$, and 2) the fact that σ'' satisfies ϕ', 3) the assumption that the address $l +_{32} 4$ is a code pointer with the condition ϕ' (to approximation k), and 4) $\sigma \mapsto^* \sigma''$.

(v) We have proved that the state σ' can progress for one step to reach a state σ'', and σ'' is safe for k steps. On a deterministic machine, or even if only the first step is deterministic, this is enough to show that σ' is safe for $k + 1$ steps.

Some steps above depend on the properties of the `add` instruction and some do not. Next, we will focus on the properties of the `add` instruction that are used in the above proof sketch, and then propose abstractions to capture these properties. After that, we will introduce a general theorem to combine these properties of the `add` instructions together to prove the arith-add rule. The benefit of this general theorem is that it can be readily used to prove other rules.

In the proof sketch of the arith-add rule, three properties of the `add` instruction are used:

Control. If the `add` instruction is at the address l in a state, then the control of the state after the execution of `add` is at $l +_{32} 4$.

Progress. If a state σ meets the condition ϕ, and its control points to the `add` instruction, then the state can progress for one step.

Preservation. If a state σ meets the condition ϕ, then after the execution of the `add` instruction, the new state meets the condition ϕ'.

Before we present abstractions that can capture these properties of the `add` instruction, we first introduce a general notion for proving properties of machine instructions. A machine instruction such as `add` is modeled as a relation between two concrete parts of states: (r, m) and (r', m'). To express only aspects of real machine-instruction semantics, we define an implementation relation, $\mathrm{imple}(i1, i2)$, read as "$i1$ implements $i2$":

$$\mathrm{imple}(i1, i2) \triangleq \forall r, m, r', m'.\ i1(r, m)(r', m') \Rightarrow i2(r, m)(r', m').$$

As an example, suppose a predicate, same_mem, expresses that two states have the same memory:

$$\mathrm{same\_mem} \triangleq \lambda(r, m), (r', m').\ m = m'$$

Then we can prove $\mathrm{imple}((\texttt{add}\ s, \mathit{reg\_imm}, d), \mathrm{same\_mem})$. As this example shows, we can define an abstract instruction, such as same_mem, that expresses only one aspect of the semantics of a real machine instruction, and show their connection via the implementation relation.

Abstractions. Next, we present abstractions that can capture the three properties of the `add` instruction: control, progress, and preservation.

To model that the add instruction is a nonbranch instruction, we first define an abstract instruction:

$$\text{straight\_line} \triangleq \lambda(r,m),(r',m').\ r'(\text{pc}) = r(\text{pc}) +_{32} 4$$

Then we prove a lemma that states the add instruction implements the straight_line instruction:

Lemma 4.21 $\text{imple}((\texttt{add}\ s, \mathit{reg\_imm}, d), \text{straight\_line})$.

To model that the add instruction can progress under some condition ϕ, we define a predicate, $\text{progress}(w,\phi)$, to express that when w is the next instruction word to execute in a state where ϕ is true, then the state can go forward for one step. Since this is exactly what $\textsf{codeptr}\,(\phi)$ to approximation 2 means (See Eq.3.9 on page 105), we define $\text{progress}(w,\phi)$ as follows:

$$\begin{aligned}&\text{progress}(w,\phi) \triangleq \\ &\quad \forall \sigma \in \Sigma, l \in \mathbb{N}.\ \text{m\_of}(\sigma)(l) = w \ \Rightarrow\ (\sigma,l) :_2 \textsf{codeptr}\,(\phi).\end{aligned}$$

Since the add instruction is no concern for our safety policy, memory safety, it can progress under *any* condition. Therefore, from the semantics of add, the following lemma is provable:

Lemma 4.22 $\dfrac{\text{decode}(w, (\texttt{add}\ s, \mathit{reg\_imm}, d))}{\text{progress}(w, \{\ \})}$.

Note that "{ }" makes no typing requirement on temporaries and thus behaves like a top type.

Based on the definition of $\text{progress}(w,\phi)$, we can easily prove a weakening lemma for it:

Lemma 4.23 $\dfrac{\text{subtype}(\phi_1, \phi_2) \quad \text{progress}(w, \phi_2)}{\text{progress}(w, \phi_1)}$.

To model that the `add` instruction preserves types in a certain way, we introduce a predicate, preservation(ϕ, ϕ'), which is an abstract instruction that states if ϕ is true in the first state, then ϕ' is true in the second state:

$$\begin{array}{l}
\text{preservation}(\phi, \phi') \triangleq \\
\quad \lambda(r, m), (r', m'). \\
\quad\quad \forall k \in \mathbb{N}, \mathit{is} \in \text{IS}. \\
\quad\quad\quad k \geq 1 \wedge \text{valid\_state}(r, m, \mathit{is}) \wedge (r, m, \mathit{is}) :_k \phi \Rightarrow \\
\quad\quad\quad\quad \exists \mathit{is}' \in \text{IS}. \\
\quad\quad\quad\quad\quad \text{extend\_state}((r, m, \mathit{is}), (r', m', \mathit{is}')) \wedge (r', m', \mathit{is}') :_{k-1} \phi'.
\end{array}$$

The above definition also takes care of imaginary parts and approximation indexes: for all imaginary part is such that (r, m, is) is valid, a new imaginary part is' can always be found such that states are extended and ϕ' is true on (r', m', is'); Furthermore, the definition requires that ϕ' is true only to index $k - 1$, since one step has been taken from ϕ to ϕ'.

For the `add` instruction, we can prove one preservation lemma that is useful in the proof of the arith-add rule:

Lemma 4.24 $\dfrac{\begin{array}{c}\text{realreg}(t_s, s) \qquad \text{realreg}(t_d, d) \\ \text{subtype}(\phi, \{\, t_s \mapsto \mathsf{int32}\,\}) \qquad \text{reg\_imm\_has\_ty}(\mathit{reg\_imm}, \mathsf{int32}, \phi)\end{array}}{\text{imple}((\texttt{add}\ s, \mathit{reg\_imm}, d), \text{preservation}(\phi, \phi[t_d \mapsto \mathsf{int32}]))}$

The proof of this lemma is highly structured so that many parts can be reused. But we omit a discussion of this structure here.

Finally there is a weakening lemma for the preservation predicate

Lemma 4.25 $\dfrac{\text{subtype}(\phi_1, \phi_1') \quad \text{preservation}(\phi_1', \phi_2') \quad \text{subtype}(\phi_2', \phi_2)}{\text{preservation}(\phi_1, \phi_2)}$.

Combining lemmas. We have modeled and stated lemmas about the three properties of the `add` instruction: control, progress, and preservation. The following is a general theorem to combine these lemmas together:

Theorem 4.26

$$\frac{\begin{array}{c}\text{decode}(w, i) \\ \text{imple}(i, \text{straight\_line}) \\ \text{progress}(w, \phi) \\ \text{imple}(i, \text{preservation}(\phi, \phi'))\end{array}}{w@l; \{\, l +_{32} 4 \mapsto \mathsf{codeptr}\,(\phi')\,\} \models_{\mathcal{L}_c} \{\, l \mapsto \mathsf{codeptr}\,(\phi)\,\}} \tag{4.3}$$

The theorem states if 1) w decodes to the instruction i, 2) i is a straight line (nonbranch) instruction, 3) a state can progress for one step when w is loaded in the state and ϕ is true, and 4) the instruction i will make ϕ' true provided that ϕ is true beforehand, then $w@l; \{\, l +_{32} 4 \mapsto \mathsf{codeptr}\,(\phi')\,\} \models_{\mathcal{L}_c} \{\, l \mapsto \mathsf{codeptr}\,(\phi)\,\}$ is true.

Using this theorem, together with Lemma 4.21, 4.22, 4.23, and 4.24, the arith-add rule is proved. The theorem itself follows directly from the definitions of those abstractions that we have introduced.

Proof factoring. A first and obvious proof-factoring effort is that we do not prove the arith-add rule just for the `add` instruction, we have actually proved that rule for

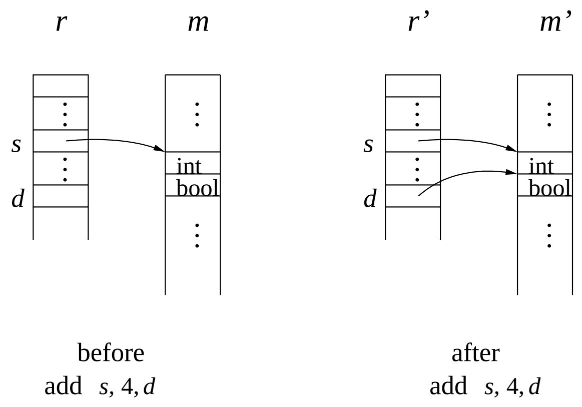


Figure 4.1: Address arithmetic

all ALU instructions that do not change condition codes (See Section 4.3.1 for the actual rule).

For a single machine instruction such as `add`, there are may be several typing rules in LTAL. For example, we have seen the arith-add rule for the `add` instruction. Another rule for `add` is an address arithmetic rule. We first explain what the rule is for.

Suppose we have a record in the memory pointed to by register s. Suppose the record consists of an integer field and a boolean field. This is depicted on the left in Figure 4.1. Now we performs an address arithmetic: add four to register s and put the result into register d, that is, execute the instruction "`add` $s, 4, d$". After the address arithmetic, the register d should point to the second field.

We can describe the scenario in types. To describe the record, we use an offset type $\mathsf{offset}(n, \tau)$. A value (σ, x) is of type $\mathsf{offset}(n, \tau)$ if $(\sigma, x +_{32} n)$ is of type τ. Then, the register s have the following type:

$$\mathsf{offset}(0, \mathsf{box}\,(\mathsf{int})) \cap \mathsf{offset}(4, \mathsf{box}\,(\mathsf{bool})),$$

which means that register s is a pointer to an integer, *and* register s plus four is a pointer to a boolean. After the execution of "`add` $s, 4, d$", register d should be of type $\mathsf{offset}(0, \mathsf{box}\,(\mathsf{bool}))$, or just $\mathsf{box}\,(\mathsf{bool})$.

Based on the above explanation, the address-add rule is as follows:

$$\frac{\begin{array}{c}\mathrm{decode}(w, (\texttt{add}\ s, \mathit{reg\_imm}, d)) \\ \mathrm{realreg}(t_s, s) \qquad \mathrm{realreg}(t_d, d) \\ \mathrm{subtype}(\phi, \{\, t_s \mapsto \mathsf{offset}(n, \tau)\,\}) \\ \mathrm{reg\_imm\_has\_ty}(\mathit{reg\_imm}, \mathsf{int}_{=}(n), \phi) \\ \phi' = \phi[t_d \mapsto \tau]\end{array}}{w@l; \{\, l +_{32} 4 \mapsto \mathsf{codeptr}\,(\phi')\,\} \models_{\mathcal{L}_c} \{\, l \mapsto \mathsf{codeptr}\,(\phi)\,\}}\ \textsf{address-add} \tag{4.4}$$

The rule states if the source temporary t_s has a value whose addition to n is of type τ, and the second operand *reg_imm* has the value n, then after the `add` instruction, the destination temporary t_d should be of type τ.

Thanks to Theorem 4.3, proving the address-add rule requires us to prove only three lemmas about the `add` instruction: control, progress, and preservation. Two of them, Lemma 4.21 and 4.22, are the same as the ones in the proof of the arith-add rule. The only new lemma we need to develop is the preservation lemma:

Lemma 4.27
$$\frac{\begin{array}{c}\mathrm{realreg}(t_s, s) \qquad \mathrm{realreg}(t_d, d) \\ \mathrm{subtype}(\phi, \{\, t_s \mapsto \mathsf{offset}(n, \tau)\,\}) \\ \mathrm{reg\_imm\_has\_ty}(\mathit{reg\_imm}, \mathsf{int}_{=}(n), \phi)\end{array}}{\mathrm{imple}((\texttt{add}\ s, \mathit{reg\_imm}, d), \mathrm{preservation}(\phi, \phi[t_d \mapsto \tau]))}$$

Even the proof of this lemma has reused many parts in the proof of the preservation lemma of the arith-add rule (Lemma 4.24). But since we have omitted the discus-

	Number of lines in specialized lemmas	Number of lines in general lemmas
arith-add	94	1,357
address-add	156	

Table 4.6: Proof size of two typing rules for the `add` instruction

sion about the structure of the proof of Lemma 4.24, we cannot go into a further discussion.

To demonstrate the effect of our proof-factoring effort, Table 4.6 shows that the proof of the arith-add rule and the arith-add rule shares 1,357 lines of common proofs.

4.3 List of Typing Lemmas

In this section, we list and explain the typing lemmas for SPARC machine instructions that we have proved. These lemmas are stated at the level of $\mathcal{L}_c$ and are used to prove the soundness of LTAL's instruction rules. We will not discuss the detailed proof of these lemmas. But since they have been proved, we will call them typing lemmas, instead of typing rules.

In the statement of these lemmas, we will use some type constructors to express types of temporaries. These type constructors are listed in Figure 4.2. Meanings of most type constructors should be clear from their explanation in the figure. Two constructors, namely $\mathsf{op}(alufun, n_1, n_2)$ and $\mathsf{cc\_cmp}(\tau_1, \tau_2)$, may not be easy to comprehend and we will explain them more in the context of related typing lemmas. The list in Figure 4.2 is not exhaustive; see Swadi [64, page 30] for an exhaustive list of the TML type theory. Some of the type constructors in Figure 4.2 are provided by the TML type theory. Others are encoded from more primitive type constructors. We do not distinguish these two categories.

$$
\begin{array}{rcll}
\tau & ::= & \mathsf{int32} & \text{32-bit integers} \\
 & \mid & \mathsf{int}_{=}(n) & \text{integer } n \\
 & \mid & \mathsf{int}_{>}(n) & \text{integers greater than } n \\
 & \mid & \mathsf{int}_{\geq}(n) & \text{integers no less than } n \\
 & \mid & \tau_1 \cap \tau_2 & \text{intersection types} \\
 & \mid & \tau_1 \cup \tau_2 & \text{union types} \\
 & \mid & \mathsf{box}\,(\tau) & \text{immutable-reference types} \\
 & \mid & \mathsf{ref}\,(\tau) & \text{mutable-reference types} \\
 & \mid & \mathsf{offset}(n, \tau) & \text{offset types} \\
 & \mid & \mathsf{op}(\mathit{alufun}, n_1, n_2) & \text{ALU-operation types} \\
 & \mid & \mathsf{aligned} & \text{aligned; a multiple of four} \\
 & \mid & \mathsf{cc\_cmp}(\tau_1, \tau_2) & \text{cond. code types} \\
 & \mid & \mathsf{cc\_z} & \text{Z cond. code is set} \\
 & \mid & \mathsf{cc\_nz} & \text{Z cond. code is not set} \\
 \\
\phi & ::= & \{\, t_1 \mapsto \tau_1, \ldots, t_n \mapsto \tau_n \,\} & \\
 & \mid & \phi_1 \cap \phi_2 & \text{conjunction} \\
 & \mid & \phi \backslash t & \text{remove } t \text{ from } \phi
\end{array}
$$

Figure 4.2: Type constructors in TML and temporary types

Also in Figure 4.2 are the syntax of temporary-file types. In our implementation, a temporary-file type is a relation, which relates temporary numbers to types.

The syntax $\phi_1 \cap \phi_2$ is somewhat ambiguous. In this presentation, a state meets $\phi_1 \cap \phi_2$ if and only if the state meets *both* ϕ_1 *and* ϕ_2. Therefore, if

$$
\begin{array}{rl}
 & \phi_1 = \{\, t_1 \mapsto \mathsf{int32}, t_2 \mapsto \mathsf{box}\,(\mathsf{int32}) \,\} \\
\text{and} & \phi_2 = \{\, t_1 \mapsto \mathsf{int}_{\geq}(10), t_3 \mapsto \mathsf{int}_{=}(3) \,\},
\end{array}
$$

then $\phi_1 \cap \phi_2$ is

$$
\{\, t_1 \mapsto \mathsf{int32}, t_1 \mapsto \mathsf{int}_{\geq}(10), t_2 \mapsto \mathsf{box}\,(\mathsf{int32}), t_3 \mapsto \mathsf{int}_{=}(3) \,\},
$$

which is the same as

$$\{\, t_1 \mapsto \mathsf{int32} \cap \mathsf{int}_{\geq}(10), t_2 \mapsto \mathsf{box}\,(\mathsf{int32}), t_3 \mapsto \mathsf{int}_{=}(3) \,\}.$$

The notation $\phi \backslash t$ removes t from the domain of ϕ. For the ϕ_1 above,

$$\phi_1 \backslash t_1 = \{\, t_2 \mapsto \mathsf{box}\,(\mathsf{int32}) \,\}.$$

As a side note, the notation $\phi_1[t \mapsto \tau]$ can be encoded as $(\phi_1 \backslash t) \cap \{\, t \mapsto \tau \,\}$.

Next, we list typing lemmas for SPARC instructions by categories.

4.3.1 Typing Lemmas for ALU Instructions

We have seen a general ALU predicate, aluxcc, on top of which other arithmetic instructions are defined. The predicate has three arguments that specific ALU instructions can use to customize: Argument *with_carry* specifies whether an ALU instruction operates with a "carry"; *modifies_icc* specifies whether it modifies the integer condition codes; *alufun* is the predicate describing the operation performed by the instruction.

The first typing lemma is a generalized version of the arith-add rule and applies to all ALU instructions that do not modify the integer condition codes. It states that if the values in two source operands are 32-bit integers, then after the ALU instruction, the destination temporary gets a 32-bit integer and all other temporaries remain the same.

$$\frac{\begin{array}{c}\mathrm{decode}(w, ((\mathrm{aluxcc}\ \mathit{with\_carry}\ \mathrm{false}\ \mathit{alufun})\ s, \mathit{reg\_imm}, d)) \\ \mathrm{realreg}(t_s, s) \qquad \mathrm{realreg}(t_d, d) \\ \mathrm{subtype}(\phi, \{\, t_s \mapsto \mathsf{int32}\,\}) \qquad \mathrm{reg\_imm\_has\_ty}(\mathit{reg\_imm}, \mathsf{int32}, \phi) \\ \phi' = \phi[t_d \mapsto \mathsf{int32}]\end{array}}{w@l; \{\, l +_{32} 4 \mapsto \mathsf{codeptr}\,(\phi')\,\} \models_{\mathcal{L}_c} \{\, l \mapsto \mathsf{codeptr}\,(\phi)\,\}}\ \textsf{arith-alu}$$

For ALU instructions, there is a more interesting typing lemma. For example, in "`add` $s, \mathit{reg\_imm}, d$", if the operand s has value n_1, and the operand $\mathit{reg\_imm}$ has value n_2, then after the instruction, the destination d will get a value $n_1 +_{32} n_2$. This rule is used in some compilers to keep track of dataflow of programs.

The dataflow-alu rule captures dataflow of ALU instructions. It applies to all ALU instructions that do not have carries and do not change the integer condition codes. The destination temporary gets the type $\mathsf{op}(\mathit{alufun}, n_1, n_2)$. This type essentially applies the *alufun* function to the two integers, n_1 and n_2. If *alufun* is $+_{32}$, then the type contains all integers equal to $n_1 +_{32} n_2$.

$$\frac{\begin{array}{c}\mathrm{decode}(w, ((\mathrm{aluxcc}\ \mathrm{false}\ \mathrm{false}\ \mathit{alufun})\ s, \mathit{reg\_imm}, d)) \\ \mathrm{realreg}(t_s, s) \qquad \mathrm{realreg}(t_d, d) \\ \mathrm{subtype}(\phi, \{\, t_s \mapsto \mathsf{int}_=(n_1)\,\}) \\ \mathrm{reg\_imm\_has\_ty}(\mathit{reg\_imm}, \mathsf{int}_=(n_2), \phi) \\ \phi' = \phi[t_d \mapsto \mathsf{op}(\mathit{alufun}, n_1, n_2)]\end{array}}{w@l; \{\, l +_{32} 4 \mapsto \mathsf{codeptr}\,(\phi')\,\} \models_{\mathcal{L}_c} \{\, l \mapsto \mathsf{codeptr}\,(\phi)\,\}}\ \textsf{dataflow-alu}$$

We have seen the address-add rule (Rule 4.4). A similar rule that subtract an integer from a type (possibly a pointer address) is listed below.

$$\frac{\begin{array}{c}\text{decode}(w, (\texttt{sub}\ s, \mathit{reg\_imm}, d)) \\ \text{realreg}(t_s, s) \qquad \text{realreg}(t_d, d) \\ \text{subtype}(\phi, \{\, t_s \mapsto \tau \,\}) \\ \text{reg\_imm\_has\_ty}(\mathit{reg\_imm}, \mathsf{int}_{=}(n), \phi) \\ \phi' = \phi[t_d \mapsto \mathsf{offset}(n, \tau)]\end{array}}{w@l; \{\, l +_{32} 4 \mapsto \mathsf{codeptr}\,(\phi') \,\} \models_{\mathcal{L}_c} \{\, l \mapsto \mathsf{codeptr}\,(\phi) \,\}}\ \textsf{address-sub}$$

4.3.2 Typing Lemmas for Branch Instructions

Next, we list typing lemmas for branch instructions. The first lemma is about the branch always instruction, "`ba` 1, *disp*". The operand *disp* is a displacement that is converted to a program-counter-relative address. We use a predicate, comp_btarget(l, *disp*, *target*), to convert the current program counter l and the displacement *disp* to the target address *target*.

The 1 in "`ba` 1, *disp*" means that the *annul* bit is one; that is, the instruction in the delay slot of `ba` will be annuled. On SPARC, control-transfer instructions such as `ba` are delayed control-transfer instructions and these instructions have delay slots [63, ch5]. The instruction following a delayed control transfer instruction is located in the delay slot. This instruction will be executed *before* the branch happens. By setting the annul bit in "`ba` 1, *disp*", the branch-always instruction annuls the instruction in the delay slot and therefore will branch to the destination immediately.

The branch-ba rule is only for "ba 1, *disp*", since our FPCC-ML compiler always sets the annual bit in a ba instruction. The rule states if the *target* address is a continuation with ϕ as the precondition, then the current address l is also a continuation with ϕ as the precondition. Note that from l to *target*, the temporary-file type does not change since the ba instruction changes only the control, which is not part of the temporaries.

$$\frac{\begin{array}{c}\mathrm{decode}(w, (\texttt{ba}\ 1, \mathit{disp})) \\ \mathrm{comp\_btarget}(l, \mathit{disp}, \mathit{target})\end{array}}{w@l; \{\ \mathit{target} \mapsto \mathsf{codeptr}\,(\phi)\ \} \models_{\mathcal{L}_c} \{\ l \mapsto \mathsf{codeptr}\,(\phi)\ \}}\ \textsf{branch-ba}$$

In TAL$_0$, there is a branch-if-zero instruction "bz s, l_d", which tests if the register s is zero, and conditionally branches accordingly. SPARC, however, uses condition codes. There are instructions, such as subcc, that set the condition codes, including N (Negative), Z (Zero), V (oVerflow), and C (Carry). There are branch instructions that jump according to the condition codes. For example, the bz instruction on SPARC jumps according to the Z condition code.

Our SPARC model models condition codes as bits in a special condition-code register. The predicate, nrBranch, is used to define all conditional-branch instructions. It takes an argument, *cnd*, as the condition on the condition-code register. The bz instruction is nrBranch applied to the condition that is true if and only if the Z bit in the condition-code register is set.

We then discuss a typing lemma for conditional branches:

$$\frac{\begin{array}{c}\mathrm{decode}(w_1, ((\mathrm{nrBranch}\ \mathit{cnd})\ 0, \mathit{disp})) \qquad \mathrm{decode}(w_2, \texttt{nop}) \\ \mathrm{comp\_btarget}(l, \mathit{disp}, \mathit{target})\end{array}}{\begin{array}{l}(w_1; w_2)@l; \{\, l +_{32} 8 \mapsto \mathsf{codeptr}\,(\phi),\ \ \mathit{target} \mapsto \mathsf{codeptr}\,(\phi) \,\} \\ \quad \models_{\mathcal{L}_c} \{\, l \mapsto \mathsf{codeptr}\,(\phi) \,\}\end{array}}\ \textsf{branch}$$

This lemma has two continuations on the left of $\models_{\mathcal{L}_c}$ since a conditional branch has two possible exits. Note that all continuations have the same precondition ϕ and therefore this lemma does not keep track of the value of condition codes in different branches. In the next section, we will discuss a lemma that does keep track of the value of condition codes. Furthermore, the above lemma puts a `nop` in the delay slot, since our FPCC-ML compiler currently does not fill in delay slots.

The last lemma in this section is about the jump-and-link instruction, "`jmpl` $d, 0$". It jumps to the value of register d and throws away the current program counter — on SPARC, when the destination register is zero, the result is thrown away and values in the register bank do not change. Since our FPCC-ML compiler uses continuation-passing style [6], the compiler produces only jump-and-link instructions whose second register is zero.

$$\frac{\begin{array}{c}\mathrm{decode}(w_1, (\texttt{jmpl}\ d, 0)) \qquad \mathrm{decode}(w_2, \texttt{nop}) \\ \mathrm{realreg}(t_d, d) \\ \mathrm{subtype}(\phi, \{\, t_d \mapsto \mathsf{aligned} \cap \mathsf{codeptr}\,(\phi) \,\})\end{array}}{(w_1; w_2)@l; \{\ \} \models_{\mathcal{L}_c} \{\, l \mapsto \mathsf{codeptr}\,(\phi) \,\}}\ \textsf{branch-jmpl}$$

The branch-jmpl rule is similar to the jmp rule in TAL$_1$ on page 86. The only difference is that we also required that the value in the register d is aligned, or a multiple of 4. If it is not, SPARC will generate an exception [63].

4.3.3 Typing Lemmas that Involve Condition Codes

We designate a temporary, t_{cc}, to correspond to the condition-code register. Then the temporary-file type, $\{ t_{cc} \mapsto \tau \}$, states that the value in the condition-code register has type τ.

As we have said, SPARC has separate instructions to set condition codes, and branch according to the condition codes. A lot of branches in a LTAL program are user branches, which do not require tracking the condition codes in types. Therefore, we fist present a typing lemma that does not keep track of how condition codes are set up. It is for all ALU instructions that set condition codes. These instructions destroy the old condition codes and therefore the following lemma removes t_{cc} from the domain of the temporary-file type. The rule is for ALU instructions whose destination register is zero and therefore (on the SPARC) the result is thrown away.

$$\frac{\begin{array}{c}\text{decode}(w, ((\text{aluxcc } \mathit{with\_carry} \text{ true } \mathit{alufun})\ s, \mathit{reg\_imm}, 0)) \\ \phi' = \phi \backslash \mathrm{t_{cc}}\end{array}}{w@l; \{ l +_{32} 4 \mapsto \mathsf{codeptr}\,(\phi') \} \models_{\mathcal{L}_c} \{ l \mapsto \mathsf{codeptr}\,(\phi) \}}\ \textsf{cc-remove}$$

For tag checking and array-bounds checking, it is necessary to track how condition codes are set up. The next typing lemma tracks condition codes for the `subcc` instruction, the only one used by our FPCC-ML compiler to set condition codes.

$$\frac{\begin{array}{c}\text{decode}(w, (\texttt{subcc}\ s, \mathit{reg\_imm}, 0)) \\ \text{realreg}(t_s, s) \\ \text{subtype}(\phi, \{\, t_s \mapsto \tau_1 \,\}) \\ \text{reg\_imm\_has\_ty}(\mathit{reg\_imm}, \tau_2, \phi) \\ \phi' = \phi[\text{t}_{\text{cc}} \mapsto \textsf{cc\_cmp}(\tau_1, \tau_2)]\end{array}}{w@l; \{\, l +_{32} 4 \mapsto \textsf{codeptr}\,(\phi') \,\} \models_{\mathcal{L}_c} \{\, l \mapsto \textsf{codeptr}\,(\phi) \,\}}\ \textsf{subcc-cmp}$$

The subcc-cmp rule states that the condition-code register gets the type $\textsf{cc\_cmp}(\tau_1, \tau_2)$, where τ_1 and τ_2 are the types of two source operands. The type $\textsf{cc\_cmp}(\tau_1, \tau_2)$ conditionally sets the condition codes (N, Z, V, and C bits) based on the values in τ_1 and τ_2. The types τ_1 and τ_2 are expected to be singleton types.

As an example, suppose before the instruction "`subcc` s_1, s_2, d", the register s_1 has the type $\tau_1 = \textsf{int}_=(n_1)$, which contains some integer n_1. Suppose the register s_2 has the type $\tau_2 = \textsf{int}_=(10)$, which has exactly the value 10.

After the instruction "`subcc` s_1, s_2, d", the subcc-cmp rule states the following type is true:

$$\{\, \text{t}_{\text{cc}} \mapsto \textsf{cc\_cmp}(\tau_1, \tau_2) \,\}$$

This type means that the N bit in the condition-code register is one if and only if n_1 is less than 10; the Z bit is one if and only if n_1 is equal to 10. It has similar components for V and C condition codes.

A `subcc` instruction is usually (but perhaps not immediately) followed by a conditional branch instruction. One rule for the `bz` instruction is as follows; it keeps track of the values of condition codes in different branches.

$$\frac{\begin{array}{c}\text{decode}(w_1, (\texttt{bz}\ 0, \mathit{disp})) \qquad \text{decode}(w_2, \texttt{nop}) \\ \text{comp\_btarget}(l, \mathit{disp}, \mathit{target}) \\ \phi_1 = \phi \cap \{\, \mathrm{t}_{\mathrm{cc}} : \mathsf{cc\_nz} \,\} \qquad \phi_2 = \phi \cap \{\, \mathrm{t}_{\mathrm{cc}} : \mathsf{cc\_z} \,\}\end{array}}{\begin{array}{l}(w_1; w_2)@l; \{\, l +_{32} 8 \mapsto \mathsf{codeptr}\,(\phi_1),\ \mathit{target} \mapsto \mathsf{codeptr}\,(\phi_2) \,\} \\ \qquad \models_{\mathcal{L}_c} \{\, l \mapsto \mathsf{codeptr}\,(\phi) \,\}\end{array}}\ \textsf{branch-bz}$$

This rule refines the type of the condition-code register depending on whether the branch is taken or not. When the branch is taken, the Z bit must be one, and we refine the type of t_{cc} to have type $\mathsf{cc\_z}$ in addition to its old type. The $\mathsf{cc\_z}$ type asserts that the Z bit is one. As a side note, remember that a state satisfies $\phi \cap \{\, \mathrm{t}_{\mathrm{cc}} : \mathsf{cc\_z} \,\}$ if the state meets both ϕ and $\{\, \mathrm{t}_{\mathrm{cc}} : \mathsf{cc\_z} \,\}$. Therefore, the temporary-file type $\phi \cap \{\, \mathrm{t}_{\mathrm{cc}} : \mathsf{cc\_z} \,\}$ refines the type of the condition-code register to have the $\mathsf{cc\_z}$ type. When the branch is not taken (the fall-through case), the $\mathsf{cc\_nz}$ type is true on the condition-code register.

For the instruction sequence

$$\begin{array}{rl} & \texttt{subcc}\ s_1, s_2, 0; \\ & \texttt{bz}\ 0, \mathit{disp};\ \texttt{nop}; \\ \mathit{next}: & \\ & \vdots \\ \mathit{target}: & \end{array}$$

we want information about the relation between registers s_1 and s_2 in different branches. For example, if s_1 is greater than or equal to 10 and s_2 is equal to 10, then at the address *target*, the register s_1 must have the value 10, and at the address *next*, the register s_1's value is greater than 10.

However, the branch-bz rule refines only the types of the condition-code register; it does not refine the types of s_1 and s_2. The trick to be able to refine the types of s_1 and s_2 is to use type variables to relate source registers and the condition-code register, that is, use a simple form of dependent types. We demonstrate this through an example.

Suppose before "`subcc` $s_1, s_2, 0$", register s_1 has type $\mathsf{int}_{\geq}(10)$ and s_2 has type $\mathsf{int}_{=}(10)$. For register s_1, we coerce its type to $\mathsf{int}_{\geq}(10) \cap \exists n_1.\, \mathsf{int}_{=}(n_1)$, which is a subtype of $\mathsf{int}_{\geq}(10)$. Then we open the existential type, so that register s_1 gets the type $\mathsf{int}_{\geq}(10) \cap \mathsf{int}_{=}(n_1)$. Similarly, we give the type $\mathsf{int}_{=}(10) \cap \mathsf{int}_{=}(n_2)$ to s_2. In the types, n_1 and n_2 are type variables of integer kinds. After these steps, the following sequence shows how we can type the "`subcc` $s_1, s_2, 0$" and "`bz` $0, \mathit{disp}$" instruction sequence using the subcc-cmp and branch-bz rules:

$$
\begin{array}{ll}
 & \{\, t_{s_1} \mapsto \mathsf{int}_{\geq}(10) \cap \mathsf{int}_{=}(n_1), t_{s_2} \mapsto \mathsf{int}_{=}(10) \cap \mathsf{int}_{=}(n_2) \,\} \\
 & \texttt{subcc}\ s_1, s_2, 0 \\
 & \{t_{s_1} \mapsto \mathsf{int}_{\geq}(10) \cap \mathsf{int}_{=}(n_1), t_{s_2} \mapsto \mathsf{int}_{=}(10) \cap \mathsf{int}_{=}(n_2), \\
 & \quad t_{cc} \mapsto \mathsf{cc\_cmp}(\mathsf{int}_{\geq}(10) \cap \mathsf{int}_{=}(n_1), \mathsf{int}_{=}(10) \cap \mathsf{int}_{=}(n_2))\} \\
 & \texttt{bz}\ 0, \mathit{disp} \\
\mathit{next}: & \{t_{s_1} \mapsto \mathsf{int}_{\geq}(10) \cap \mathsf{int}_{=}(n_1), t_{s_2} \mapsto \mathsf{int}_{=}(10) \cap \mathsf{int}_{=}(n_2), \\
 & \quad t_{cc} \mapsto \mathsf{cc\_cmp}(\mathsf{int}_{\geq}(10) \cap \mathsf{int}_{=}(n_1), \mathsf{int}_{=}(10) \cap \mathsf{int}_{=}(n_2)) \cap \mathsf{cc\_nz}\} \\
 & \vdots \\
\mathit{target}: & \{t_{s_1} \mapsto \mathsf{int}_{\geq}(10) \cap \mathsf{int}_{=}(n_1), t_{s_2} \mapsto \mathsf{int}_{=}(10) \cap \mathsf{int}_{=}(n_2), \\
 & \quad t_{cc} \mapsto \mathsf{cc\_cmp}(\mathsf{int}_{\geq}(10) \cap \mathsf{int}_{=}(n_1), \mathsf{int}_{=}(10) \cap \mathsf{int}_{=}(n_2)) \cap \mathsf{cc\_z}\}
\end{array}
$$

Notice that the type variables n_1 and n_2 connect the values of registers s_1 and s_2 with the value of the condition-code register.

Remember that part of the type $\mathsf{cc\_cmp}(\mathsf{int}_{\geq}(10) \cap \mathsf{int}_{=}(n_1), \mathsf{int}_{=}(10) \cap \mathsf{int}_{=}(n_2))$ means that the value in $\mathsf{int}_{\geq}(10) \cap \mathsf{int}_{=}(n_1)$ is equal to 10 if and only if the Z bit is one. At the address *target*, we also know that $\mathsf{cc\_z}$ is true, that is, the Z bit is one. Therefore, the value in $\mathsf{int}_{\geq}(10) \cap \mathsf{int}_{=}(n_1)$ can contain only the value 10; thus, n_1 has to be 10. This reasoning is a proof of the following subtyping lemma:

$$\begin{aligned}&\{t_{s_1} \mapsto \mathsf{int}_{\geq}(10) \cap \mathsf{int}_{=}(n_1), t_{s_2} \mapsto \mathsf{int}_{=}(10) \cap \mathsf{int}_{=}(n_2),\\ &\quad t_{cc} \mapsto \mathsf{cc\_cmp}(\mathsf{int}_{\geq}(10) \cap \mathsf{int}_{=}(n_1), \mathsf{int}_{=}(10) \cap \mathsf{int}_{=}(n_2)) \cap \mathsf{cc\_z}\}\\ &\qquad \subseteq \{\, t_{s_1} \mapsto \mathsf{int}_{=}(10)\,\}\end{aligned}$$

Consequently, at the address *target*, we have $\{\, t_{s_1} \mapsto \mathsf{int}_{=}(10)\,\}$ being true.

At the address *next*, a similar reasoning gives us that $\{\, t_{s_1} \mapsto \mathsf{int}_{>}(10)\,\}$ is true.

In our implementation, the $\mathsf{branch\text{-}bz}$ lemma is just an instantiation of a more general lemma for all conditional-branch instructions:

$$\frac{\begin{array}{c}\text{decode}(w_1, ((\text{nrBranch } \mathit{cnd})\ 0, \mathit{disp})) \qquad \text{decode}(w_2, \texttt{nop})\\ \text{cnd\_relate\_to\_type}(\mathit{cnd}, \mathit{cndTType})\\ \text{cnd\_relate\_to\_type}(\neg\mathit{cnd}, \mathit{cndFType})\\ \text{comp\_btarget}(l, \mathit{disp}, \mathit{target})\\ \phi_1 = \phi \cap \{\, \mathrm{t_{cc}} : \mathit{cndFType}\,\} \qquad \phi_2 = \phi \cap \{\, \mathrm{t_{cc}} : \mathit{cndTType}\,\}\end{array}}{\begin{array}{l}(w_1; w_2)@l; \{\, l +_{32} 8 \mapsto \mathsf{codeptr}\,(\phi_1),\ \mathit{target} \mapsto \mathsf{codeptr}\,(\phi_2)\,\}\\ \quad \models_{\mathcal{L}_c} \{\, l \mapsto \mathsf{codeptr}\,(\phi)\,\}\end{array}}\ \mathsf{branch\text{-}cnd}$$

The predicate cnd_relate_to_type(*cnd*, *cndTType*) relates a condition to a type constructor. For example, the condition for `bz` is related to the type `cc_z`, and the negation of the condition for `bz` is related to the type `cc_nz`.

4.3.4 Typing Lemmas for Memory Instructions

In this section, we list typing lemmas for memory instructions. First we discuss a lemma for the instruction "`ld` $s, \mathit{reg\_imm}, d$". This instruction loads register from memory. The memory address is the sum of the contents of register s and *reg_imm*. The destination is register d.

$$\frac{\begin{array}{c}\mathrm{decode}(w, (\texttt{ld}\ s, \mathit{reg\_imm}, d)) \\ \mathrm{realreg}(t_s, s) \qquad \mathrm{realreg}(t_d, d) \\ \mathrm{subtype}(\phi, \{\, t_s \mapsto \mathsf{offset}(n, \mathsf{box}\,(\tau)) \,\}) \\ \mathrm{reg\_imm\_has\_ty}(\mathit{reg\_imm}, \mathsf{int}_{=}(n), \phi) \\ \phi' = \phi[\mathrm{t_d} \mapsto \tau]\end{array}}{w@l; \{\, l +_{32} 4 \mapsto \mathsf{codeptr}\,(\phi') \,\} \models_{\mathcal{L}_c} \{\, l \mapsto \mathsf{codeptr}\,(\phi) \,\}}\ \mathsf{ld}$$

Register s has the type $\mathsf{offset}(n, \mathsf{box}\,(\tau))$; it means the value of register s plus n is an immutable pointer to a value of type τ. The other operand *reg_imm* has the value $\mathsf{int}_{=}(n)$. After the `ld` instruction, the destination register will have the type τ. The lemma is only for immutable-reference types; we have a similar lemma for mutable-reference types.

The next typing lemma is for type-preserving updating store. It is about the store instruction "`st` $s, d, \mathit{reg\_imm}$", which stores the value of register s into the memory address that equals to the sum of the value of register d and $\mathit{reg\_imm}$.

$$\frac{\begin{array}{c}\mathrm{decode}(w, (\texttt{st}\ s, d, \mathit{reg\_imm})) \\ \mathrm{realreg}(t_s, s) \qquad \mathrm{realreg}(t_d, d) \\ \mathrm{subtype}(\phi, \{\, t_s \mapsto \tau \,\}) \\ \mathrm{subtype}(\phi, \{\, t_d \mapsto \mathsf{offset}(n, \mathsf{ref}\ (\tau)) \,\}) \\ \mathrm{reg\_imm\_has\_ty}(\mathit{reg\_imm}, \mathsf{int}_{=}(n), \phi)\end{array}}{w@l; \{\, l +_{32} 4 \mapsto \mathsf{codeptr}\ (\phi) \,\} \models_{\mathcal{L}_c} \{\, l \mapsto \mathsf{codeptr}\ (\phi) \,\}}\ \textsf{update-st}$$

The lemma requires the type of new values to match the type of the mutable-reference cell and it is type preserving.

We also have typing lemmas for heap allocations. In her thesis, Chen [16, ch4.4] explains the heap allocation model in LTAL in great detail. The corresponding typing lemmas at the $\mathcal{L}_c$ level are different only at minor points. Therefore, we do not list those typing lemmas.

Chapter 5

Conclusion

The Foundational Proof-Carrying Code project at Princeton seeks to reduce the problem of mobile-code security to the problem of checking formal proofs in logic. Security-relevant systems are modeled in logic and security properties are proved from a small set of axioms. In their pioneering work, *Principia Mathematica*, Whitehead and Russell tried to reduce mathematics to formal logic. They demonstrated that a large portion of mathematics follows from purely logical premises and uses only concepts definable in logical terms. By founding mathematics on a small set of primitive logical notions, they hoped to get rid of paradoxes from mathematics. We believe that the same axiomatic principle should be applied to security as well. People generally believe systems with fewer assumptions—axioms—are more secure than those with more assumptions. Applying the axiomatic principle to the extreme, the FPCC project proceeds with a minimal set of axioms. Although with many theoretical and implementation obstacles, the FPCC project has progressed smoothly and shows the feasibility of reducing security to logic.

As part of the FPCC project, this dissertation proposes a principled way to verify properties of machine-language programs in the presence of unstructured control flow. We have designed and implemented a low-level control-flow logic, $\mathcal{L}_c$, for systematically combining properties of machine instructions. We have also proved many typing lemmas about SPARC machine instructions based on the machine semantics. Our proving effort is one of the few projects that develop machine-checked proofs about machine-language programs from a specification of machine semantics. The other project we know is by Boyer and Yu [14]. They formally specify a subset of the MC68020 microprocessor within the logic of the system Nqthm, a quantifier-free first-order logic with equality. Based on the specification, they have manually proved the correctness of MC68020 machine code programs for binary search, quick sort, etc. Although machine-checked proofs come with a large implementation effort, their project and ours demonstrate the feasibility of developing proofs about a real machine.

Finally, we discuss possible future work. First, we would like to add more features to our FPCC project. Currently, our FPCC-ML compiler handles features in core ML. We plan to add features such as arrays and the module system. Also, the compiler handles memory initialization and allocation, but not deallocation. We believe a region-based calculus should be able to handle memory deallocation at the LTAL level. However, developing a foundational semantics for such a calculus would be a technical challenge.

The current FPCC system is for the SPARC architecture. We would like to see our techniques applied to other architectures. For Intel x86 architectures, we believe that the major difficulty lies in the specification. On the other hand, our proof techniques should be readily applicable since they rely mostly on the notion of computation steps. RISC architectures for embedded system, such as ARM [60], are much easier to specify and verify. This should be the first step of transferring our techniques.

The safety policy we used is memory safety. There are other useful safety polices such as bounded resource usage [25, 36, 67]. Accommodating new safety policies would require a significant change to the typed-assembly language. However, we believe that our proof infrastructure for proving properties of machine-language programs can be easily factored to accommodate new safety policies.

Appendix A

Proofs of $\mathcal{L}_c$

This appendix presents detailed proofs for $\mathcal{L}_c$. As part of the FPCC system, we have developed and machine-checked the soundness proof for a version of $\mathcal{L}_c$ (on top of SPARC). The completeness proof and the proof about the connection between the direct-style semantics and the continuation-style semantics, however, have been developed only on paper and not been machine checked; they are unnecessary for the FPCC system.

A.1 Soundness Proof of $\mathcal{L}_c$

When the label map θ is clear from the context, we sometimes use $\mapsto$ as an abbreviation for $\mapsto_\theta$.

Lemma A.28 *(Determinism of the Step Relation.)*
If $(\mathrm{pc}, \pi, m) \mapsto_\theta \sigma'$,

(i) and $\pi(\mathrm{pc}) = (x := e)$, *then* $\sigma' = (\mathrm{pc}+1, \pi, m[x \mapsto \mathcal{V}\,[\![e]\!]\,m])$;

(ii) and $\pi(\mathrm{pc}) = (\textbf{goto}\ l)$, *then* $\sigma' = (\theta(l), \pi, m)$;

(iii) *and* $\pi(\mathrm{pc}) = (\textbf{if } b \textbf{ goto } l)$ *and* $\mathcal{B}\,[\![b]\!]\,m = \mathrm{tt}$*, then* $\sigma' = (\theta(l), \pi, m)$*;*

(iv) *and* $\pi(\mathrm{pc}) = (\textbf{if } b \textbf{ goto } l)$ *and* $\mathcal{B}\,[\![b]\!]\,m = \mathrm{ff}$*, then* $\sigma' = (\mathrm{pc}+1, \pi, m)$*.*

Proof. By case analysis on $(\mathrm{pc}, \pi, m) \mapsto_\theta \sigma'$. □

Lemma A.29 *For any* $\sigma, \sigma' \in \Sigma$ *such that* $\sigma \mapsto^* \sigma'$*, the instruction memory remains the same, i.e.,* $\mathrm{i\_of}(\sigma) = \mathrm{i\_of}(\sigma')$*.*

Proof. First prove $\mathrm{i\_of}(\sigma) = \mathrm{i\_of}(\sigma')$ when $\sigma \mapsto \sigma'$ by case analysis. Then by induction over the length of $\mapsto^*$. □

Lemma A.30 *(Substitution.)*

(i) $\mathcal{V}\,[\![e'[e/x]]\!]\,m = \mathcal{V}\,[\![e']\!]\,m[x \mapsto \mathcal{V}\,[\![e]\!]\,m]$

(ii) $\mathcal{B}\,[\![b[e/x]]\!]\,m = \mathcal{B}\,[\![b]\!]\,m[x \mapsto \mathcal{V}\,[\![e]\!]\,m]$

(iii) $\mathcal{A}\,[\![p[e/x]]\!]\,m = \mathcal{A}\,[\![p]\!]\,m[x \mapsto \mathcal{V}\,[\![e]\!]\,m]$

Proof. By induction over the syntax of e', b, and p, respectively. □

Lemma A.31 *For all* $\sigma \in \Sigma$ *and* $j, k \in \mathbb{N}$*, if* $\mathrm{safe\_state}(\sigma, j)$*, and* $\forall \sigma' \in \Sigma.\ \sigma \mapsto^j \sigma' \Rightarrow \mathrm{safe\_state}(\sigma', k)$*, then* $\mathrm{safe\_state}(\sigma, j+k)$*.*

Proof. Straightforward from the definition of $\mathrm{safe\_state}(\sigma, j+k)$. □

Instantiate j to 1 in Lemma A.31 and using the equivalence between $\mathrm{safe\_state}(\sigma, 1)$ and $\exists \sigma'.\ \sigma \mapsto \sigma'$, and get the following corollary:

Corollary A.32 *For all* $\sigma \in \Sigma$ *and* $k \in \mathbb{N}$*, if* $\exists \sigma'.\ \sigma \mapsto \sigma'$*, and* $\forall \sigma' \in \Sigma.\ \sigma \mapsto \sigma' \Rightarrow \mathrm{safe\_state}(\sigma', k)$*, then* $\mathrm{safe\_state}(\sigma, k+1)$*.*

Lemma A.33 *Assuming that* $\mathcal{D}$ *is sound, if* $\vdash \Psi \Rightarrow \Psi'$*, then* $\models \Psi \Rightarrow \Psi'$*.*

Proof. There are two cases to derive $\vdash \Psi \Rightarrow \Psi'$.

Case $\boxed{\dfrac{m \geq n}{\vdash \{l_1 \triangleright p_1, \ldots, l_m \triangleright p_m\} \Rightarrow \{l_1 \triangleright p_1, \ldots, l_n \triangleright p_n\}}\ \textsf{s-width}}$

The proof of this case is immediate from the definition $\models \Psi \Rightarrow \Psi'$.

Case $\boxed{\dfrac{\vdash_{\mathcal{D}} p' \Rightarrow p}{\vdash \Psi \cup \{l \triangleright p\} \Rightarrow \Psi \cup \{l \triangleright p'\}}\ \textsf{s-depth}}$

To prove $\models \Psi \cup \{l \triangleright p\} \Rightarrow \Psi \cup \{l \triangleright p'\}$, pick $\sigma \in \Sigma$, $\theta \in LMap$ and $k \in \mathbb{N}$, assume

$$\sigma; \theta \models_k \Psi \cup \{l \triangleright p\}, \tag{A.33.1}$$

and the goal is $\sigma; \theta \models_k \Psi \cup \{l \triangleright p'\}$. Result A.33.1 gives

$$\sigma; \theta \models_k \Psi, \tag{A.33.2}$$

$$\sigma; \theta \models_k l \triangleright p. \tag{A.33.3}$$

$\sigma; \theta \models_k \Psi \cup \{l \triangleright p'\}$ can be proved if $\sigma; \theta \models_k l \triangleright p'$, which is proved as follows: pick $\sigma' \in \Sigma$, assume

$$\sigma \mapsto_{\theta}^{*} \sigma' \tag{A.33.4}$$

$$\text{control}(\sigma') = \theta(l) \tag{A.33.5}$$

$$\mathcal{A}\,[\![p']\!]\,(\text{m\_of}(\sigma')) = \text{tt} \tag{A.33.6}$$

and the goal is to show safe_state(σ', k).

Since the deduction system $\mathcal{D}$ is sound, we have $\models p' \Rightarrow p$ and thus $\mathcal{A}\,[\![p' \Rightarrow p]\!]\,(\text{m\_of}(\sigma')) = \text{tt}$. Together with A.33.6, we have

$$\mathcal{A}\,[\![p]\!]\,(\text{m\_of}(\sigma')) = \text{tt} \tag{A.33.7}$$

Unpack the definition of A.33.3, perform a universal elimination using the state σ', and use A.33.4, A.33.5 and A.33.7, to get safe_state(σ', k). $\square$

Theorem A.34 *(Soundness.) Assume $\mathcal{D}$ is sound. If $F\,;\,\Psi' \vdash \Psi$, then $F\,;\,\Psi' \models \Psi$.*

Proof. Proof by induction over the derivation of $F\,;\,\Psi' \vdash \Psi$. We will cover each case by a lemma.

Case $\boxed{\dfrac{}{\{l : (x := e) : l'\}\,;\,\{l' \rhd p\} \vdash \{l \rhd p[e/x]\}}\ \textsf{assign}}$

This case is covered by the Lemma A.35.

Case $\boxed{\dfrac{}{\{l : (\mathbf{goto}\ l_1) : l'\}\,;\,\{l_1 \rhd p\} \vdash \{l \rhd p\}}\ \textsf{goto}}$

This case is covered by the Lemma A.36.

Case $\boxed{\dfrac{}{\{l : (\mathbf{if}\ b\ \mathbf{goto}\ l_1) : l'\}\,;\,\{l_1 \rhd p \wedge b, l' \rhd p \wedge \neg b\} \vdash \{l \rhd p\}}\ \textsf{if}}$

Similar to the goto case.

Case $\boxed{\dfrac{F_1\,;\,\Psi'_1 \vdash \Psi_1 \quad F_2\,;\,\Psi'_2 \vdash \Psi_2}{F_1 \cup F_2\,;\,\Psi'_1 \cup \Psi'_2 \vdash \Psi_1 \cup \Psi_2}\ \textsf{combine}}$

The induction hypothesis gives $F_1\,;\,\Psi'_1 \models \Psi_1$ and $F_2\,;\,\Psi'_2 \models \Psi_2$, and the rest is covered by Lemma A.37.

Case $\boxed{\dfrac{F\,;\,\Psi' \cup \{l \rhd p\} \vdash \Psi \cup \{l \rhd p\}}{F\,;\,\Psi' \vdash \Psi \cup \{l \rhd p\}}\ \textsf{discharge}}$

The induction hypothesis gives $F\,;\,\Psi' \cup \{l \rhd p\} \models \Psi \cup \{l \rhd p\}$, and the rest is covered by Lemma A.40.

Case $\boxed{\dfrac{\vdash \Psi'_1 \Rightarrow \Psi'_2 \quad F\,;\,\Psi'_2 \vdash \Psi_2 \quad \vdash \Psi_2 \Rightarrow \Psi_1}{F\,;\,\Psi'_1 \vdash \Psi_1}\ \textsf{weaken}}$

The induction hypothesis gives $F\,;\,\Psi'_2 \models \Psi_2$. By Lemma A.33, we have $\models \Psi'_1 \Rightarrow \Psi'_2$ and $\models \Psi_2 \Rightarrow \Psi_1$. The rest is covered by Lemma A.38.

$\square$

Lemma A.35 $\{l : (x := e) : l'\}\,;\,\{l' \rhd p\} \models \{l \rhd p[e/x]\}$

Proof. Let $F = \{l : (x := e) : l'\}$, $\Psi' = \{l' \triangleright p\}$ and $\Psi = \{l \triangleright p[e/x]\}$.

To prove $F\,;\,\Psi' \models \Psi$, pick $\sigma \in \Sigma$, $\theta \in \mathit{LMap}$, and $k \in \mathbb{N}$, and assume

$$\text{loaded}(\{l : (x := e) : l'\}, \text{i\_of}(\sigma), \theta), \tag{A.35.1}$$

$$\sigma; \theta \models_k \{l' \triangleright p\}. \tag{A.35.2}$$

The goal is to prove $\sigma; \theta \models_{k+1} \{l \triangleright p[e/x]\}$. To prove it, pick $\sigma' = (\text{pc}, \pi, m) \in \Sigma$ such that

$$\sigma \mapsto_\theta^* \sigma' \tag{A.35.3}$$

$$\text{control}(\sigma') = \text{pc} = \theta(l) \tag{A.35.4}$$

$$\mathcal{A}\,[\![p[e/x]]\!]\,m = \text{tt} \tag{A.35.5}$$

and the goal is safe_state$(\sigma', k+1)$. By Corollary A.32, it is sufficient to show the following two results:

$$\exists \sigma''.\ \sigma' \mapsto \sigma'' \tag{A.35.6}$$

$$\forall \sigma'' \in \Sigma.\ \sigma' \mapsto \sigma'' \Rightarrow \text{safe\_state}(\sigma'', k) \tag{A.35.7}$$

We prove A.35.6 first. Since the step relation does not change the instruction memory (Lemma A.29), we have the following from A.35.3:

$$\text{i\_of}(\sigma) = \text{i\_of}(\sigma') \tag{A.35.8}$$

Together with A.35.1, we get

$$\text{loaded}(\{l : (x := e) : l'\}, \text{i\_of}(\sigma'), \theta) \tag{A.35.9}$$

whose definition gives us

$$\text{i\_of}(\sigma')(\text{pc}) = x := e \tag{A.35.10}$$

$$\theta(l') = \theta(l) + 1 \tag{A.35.11}$$

Construct $\sigma'' = (\text{pc} + 1, \pi, m[x \mapsto \mathcal{V}\,[\![e]\!]\,m])$, it is easy to verify that $\sigma' \mapsto \sigma''$, which is the goal A.35.6.

Now we prove the goal A.35.7. For all $\sigma'' \in \Sigma$ such that

$$\sigma' \mapsto \sigma'', \tag{A.35.12}$$

by Lemma A.28 and the result A.35.10, we must have that

$$\sigma'' = (\text{pc} + 1, \pi, m[x \mapsto \mathcal{V}\,[\![e]\!]\,m]) \tag{A.35.13}$$

Now, use the result A.35.2: $\sigma; \theta \models_k \{l' \triangleright p\}$, which states that $l' \triangleright p$ is a continuation to approximation k. Unpack the definition $\sigma; \theta \models_k \{l' \triangleright p\}$, do a universal elimination, and use the state σ'' and the following results:

$\sigma \mapsto_\theta^* \sigma''$, from A.35.3 and A.35.12

$\text{control}(\sigma'') = \theta(l')$, from A.35.13, A.35.11 and A.35.4

$\mathcal{A}\,[\![p]\!]\,\big(\text{i\_of}(\sigma'')\big) = \text{tt}$, from the Lemma A.30 (iii) and the result A.35.5

to get $\text{safe\_state}(\sigma'', k)$. $\square$

Lemma A.36 $\{l : (\textbf{goto } l_1) : l'\}\,;\, \{l_1 \triangleright p\} \models \{l \triangleright p\}$

Proof. Let $F = \{l : (\textbf{goto } l_1) : l'\}$, $\Psi' = \{l_1 \triangleright p\}$ and $\Psi = \{l \triangleright p\}$.

To prove $F\,;\,\Psi' \models \Psi$, pick $\sigma \in \Sigma$, $\theta \in \mathit{LMap}$, and $k \in \mathbb{N}$, and assume

$$\text{loaded}(\{l : (\textbf{goto } l_1) : l'\}, \text{i\_of}(\sigma), \theta), \tag{A.36.1}$$

$$\sigma; \theta \models_k \{l_1 \triangleright p\}. \tag{A.36.2}$$

The goal is to prove $\sigma; \theta \models_{k+1} \{l \triangleright p\}$. To prove it, pick $\sigma' = (\text{pc}, \pi, m) \in \Sigma$ such that

$$\sigma \mapsto_\theta^* \sigma' \tag{A.36.3}$$

$$\text{control}(\sigma') = \text{pc} = \theta(l) \tag{A.36.4}$$

$$\mathcal{A}\,[\![p]\!]\,m = \text{tt} \tag{A.36.5}$$

and the goal is $\text{safe\_state}(\sigma', k+1)$. By Corollary A.32, it is sufficient to prove the following two results:

$$\exists \sigma''.\ \sigma' \mapsto \sigma'' \tag{A.36.6}$$

$$\forall \sigma'' \in \Sigma.\ \sigma' \mapsto \sigma'' \Rightarrow \text{safe\_state}(\sigma'', k) \tag{A.36.7}$$

We prove A.36.6 first. Since the step relation does not change the instruction memory (Lemma A.29), we have the following from A.36.3:

$$\text{i\_of}(\sigma) = \text{i\_of}(\sigma') \tag{A.36.8}$$

Together with A.36.1, we get

$$\text{loaded}(\{l : (\textbf{goto } l_1) : l'\}, \text{i\_of}(\sigma'), \theta) \tag{A.36.9}$$

whose definition gives

$$\text{i\_of}(\sigma')(\text{pc}) = \textbf{goto } l_1 \tag{A.36.10}$$

$$\theta(l') = \theta(l) + 1 \tag{A.36.11}$$

Construct $\sigma'' = (\theta(l_1), \pi, m)$, it is easy to verify that $\sigma' \mapsto \sigma''$, which is the goal A.36.6.

Now we prove the goal A.36.7. For all $\sigma'' \in \Sigma$ such that

$$\sigma' \mapsto \sigma'', \tag{A.36.12}$$

by Lemma A.28 and the result A.36.10, we must have that

$$\sigma'' = (\theta(l_1), \pi, m) \tag{A.36.13}$$

Now, use the result A.36.2: $\sigma; \theta \models_k \{l_1 \triangleright p\}$. Unpack its definition, perform a universal elimination, and use the state σ'' and the following results:

$$\sigma \mapsto_\theta^* \sigma'', \quad \text{from A.36.3 and A.36.12} \tag{A.36.14}$$

$$\text{control}(\sigma'') = \theta(l_1), \quad \text{from A.36.13} \tag{A.36.15}$$

$$\mathcal{A}\,[\![p]\!]\,\big(\text{i\_of}(\sigma'')\big) = \text{tt}, \quad \text{from A.36.5 and i\_of}(\sigma'') = m. \tag{A.36.16}$$

to get safe_state(σ'', k). $\square$

Lemma A.37 *If* $F_1\,;\,\Psi_1' \models \Psi_1$ *and* $F_2\,;\,\Psi_2' \models \Psi_2$*, then* $F_1 \cup F_2\,;\,\Psi_1' \cup \Psi_2' \models \Psi_1 \cup \Psi_2$.

Proof. To prove $F_1 \cup F_2\,;\,\Psi_1' \cup \Psi_2' \models \Psi_1 \cup \Psi_2$, pick $\sigma \in \Sigma$, $\theta \in \mathit{LMap}$, and $k \in \mathbb{N}$, and assume

$$\text{loaded}(F_1 \cup F_2, \text{i\_of}(\sigma), \theta) \tag{A.37.1}$$

$$\sigma; \theta \models_k \Psi_1' \cup \Psi_2' \tag{A.37.2}$$

We need to prove $\sigma; \theta \models_{k+1} \Psi_1 \cup \Psi_2$.

From A.37.1, it is easy to prove

$$\text{loaded}(F_1, \text{i\_of}(\sigma), \theta) \tag{A.37.3}$$

$$\text{loaded}(F_2, \text{i\_of}(\sigma), \theta) \tag{A.37.4}$$

From A.37.2, we get

$$\sigma; \theta \models_k \Psi_1' \tag{A.37.5}$$

$$\sigma; \theta \models_k \Psi_2' \tag{A.37.6}$$

From the assumption $F_1\,;\,\Psi_1' \models \Psi_1$, A.37.3 and A.37.5, derive

$$\sigma; \theta \models_{k+1} \Psi_1 \tag{A.37.7}$$

Similarly, from $F_2 \,;\, \Psi_2' \;\models\; \Psi_2$, A.37.4 and A.37.6, derive

$$\sigma; \theta \models_{k+1} \Psi_2 \tag{A.37.8}$$

These two results give us $\sigma; \theta \models_{k+1} \Psi_1 \cup \Psi_2$, which is the goal. □

Lemma A.38 *If* $\models \Psi_1' \Rightarrow \Psi_2'$, $F \,;\, \Psi_2' \;\models\; \Psi_2$ *and* $\models \Psi_2 \Rightarrow \Psi_1$, *then* $F \,;\, \Psi_1' \;\models\; \Psi_1$.

Proof. To prove $F \,;\, \Psi_1' \;\models\; \Psi_1$, pick $\sigma \in \Sigma$, $\theta \in \mathit{LMap}$, and $k \in \mathbb{N}$, and assume

$$\text{loaded}(F, \text{i\_of}(\sigma), \theta) \tag{A.38.1}$$

$$\sigma; \theta \models_k \Psi_1'. \tag{A.38.2}$$

We need to prove $\sigma; \theta \models_{k+1} \Psi_1$.

From the assumption $\models \Psi_1' \Rightarrow \Psi_2'$, and A.38.2, we have

$$\sigma; \theta \models_k \Psi_2' \tag{A.38.3}$$

From the assumption $F \,;\, \Psi_2' \;\models\; \Psi_2$, the result A.38.1 and A.38.3, we get

$$\sigma; \theta \models_{k+1} \Psi_2 \tag{A.38.4}$$

From the assumption $\models \Psi_2 \Rightarrow \Psi_1$, and the result A.38.4, we have $\sigma; \theta \models_{k+1} \Psi_1$, which is the goal. □

Lemma A.39 *Let* $\sigma \in \Sigma$, $\theta \in \mathit{LMap}$, $k \in \mathbb{N}$ *and* $\Psi \in \mathit{FragSet}$.

(i) $\sigma; \theta \models_0 \Psi$. *(true at index 0)*

(ii) if $\sigma; \theta \models_k \Psi$, *then* $\sigma; \theta \models_j \Psi$, *for all* $j \leq k$. *(downward closed)*

(iii) if $\sigma;\theta \models_k \Psi$ and $\sigma \mapsto^*_\theta \sigma'$, then $\sigma';\theta \models_k \Psi$. (monotonicity with respect to the $\mapsto^*_\theta$)

Proof. Straightforward from the definition of $\sigma;\theta \models_k \Psi$. $\square$

Lemma A.40 *(Soundness of the discharge rule.) If $F\,;\,\Psi' \cup \{l \triangleright p\} \models \Psi \cup \{l \triangleright p\}$, then $F\,;\,\Psi' \models \Psi \cup \{l \triangleright p\}$.*

Proof. To prove $F\,;\,\Psi' \models \Psi \cup \{l \triangleright p\}$, pick $\sigma \in \Sigma$ and $\theta \in \mathit{LMap}$. Assume

$$\text{loaded}(F, \text{i\_of}(\sigma), \theta). \tag{A.40.1}$$

The goal is $\forall k \in \mathbb{N}.\ \big(\sigma;\theta \models_k \Psi' \Rightarrow \sigma;\theta \models_{k+1} \Psi \cup \{l \triangleright p\}\big)$. We prove it by induction over the natural number k.

For the base case, assume

$$\sigma;\theta \models_0 \Psi', \tag{A.40.2}$$

and prove that $\sigma;\theta \models_1 \Psi \cup \{l \triangleright p\}$.

Lemma A.39 (i) gives

$$\sigma;\theta \models_0 \{l \triangleright p\}. \tag{A.40.3}$$

Together with A.40.2, we have

$$\sigma;\theta \models_0 \Psi' \cup \{l \triangleright p\}. \tag{A.40.4}$$

Now from the assumption, $F\,;\,\Psi' \cup \{l \triangleright p\} \models \Psi \cup \{l \triangleright p\}$, the result A.40.1, and the result A.40.4, we get $\sigma;\theta \models_1 \Psi \cup \{l \triangleright p\}$, which is the goal for the base case.

For the inductive case, assume the induction hypothesis is true for k:

$$\sigma; \theta \models_k \Psi' \ \Rightarrow \sigma; \theta \models_{k+1} \Psi \cup \{l \triangleright p\}. \tag{A.40.5}$$

The goal is to prove that the induction hypothesis is true for $k + 1$:

$$\sigma; \theta \models_{k+1} \Psi' \ \Rightarrow \sigma; \theta \models_{k+2} \Psi \cup \{l \triangleright p\}.$$

Thus, assume

$$\sigma; \theta \models_{k+1} \Psi'. \tag{A.40.6}$$

Lemma A.39 (ii) gives

$$\sigma; \theta \models_k \Psi'. \tag{A.40.7}$$

From the induction hypothesis A.40.5 and the result A.40.7, we have

$$\sigma; \theta \models_{k+1} \Psi \cup \{l \triangleright p\}, \tag{A.40.8}$$

from which the following is derivable:

$$\sigma; \theta \models_{k+1} \{l \triangleright p\}. \tag{A.40.9}$$

Together with A.40.6, we have

$$\sigma; \theta \models_{k+1} \Psi' \cup \{l \triangleright p\}. \tag{A.40.10}$$

Now, use the assumption $F\,;\,\Psi' \cup \{l \triangleright p\} \models \Psi \cup \{l \triangleright p\}$, together with A.40.1 and A.40.10, to derive

$$\sigma;\theta \models_{k+2} \Psi \cup \{l \triangleright p\},$$

which is the goal for the inductive case. $\square$

A.2 Connection Between the Direct-Style Semantics and the Continuation-Style Semantics

Lemma A.41 *If* labels$(\Psi) \subseteq$ entries(F)*, then* $\models \{\Psi\}F\{\Psi'\}$ *implies* $F\,;\,\Psi' \models \Psi$*.*

Proof. To prove $F\,;\,\Psi' \models \Psi$, pick $\theta \in LMap$, $\sigma \in \Sigma$ and $k \in \mathbb{N}$, and assume

$$\text{loaded}(F, \text{i\_of}(\sigma), \theta), \tag{A.41.1}$$

$$\sigma;\theta \models_k \Psi'. \tag{A.41.2}$$

The goal is to prove $\sigma;\theta \models_{k+1} \Psi$. To prove it, pick $l \triangleright p \in \Psi$, $\sigma' \in \Sigma$, and assume

$$\sigma \mapsto_\theta^* \sigma' \tag{A.41.3}$$

$$\text{control}(\sigma') = \theta(l) \tag{A.41.4}$$

$$\mathcal{A}\,[\![p]\!]\,(\text{m\_of}(\sigma')) = \text{tt} \tag{A.41.5}$$

The new goal is to prove safe_state$(\sigma', k+1)$. To prove it, choose $j < k+1$, and a sequence of states $\sigma_0, \sigma_1, \ldots, \sigma_j$ such that

$$\sigma' = \sigma_0 \mapsto \sigma_1 \mapsto \ldots \mapsto \sigma_j \tag{A.41.6}$$

The new goal is to prove the existence of some σ_{j+1} such that $\sigma_j \mapsto \sigma_{j+1}$. The proof proceeds in two cases.

Case $\forall 0 \leq i \leq j$. $\text{control}(\sigma_i) \in \text{addr}(F, \theta)$. That is, the control never leaves the address space of F. Then, we have both $\sigma_0 \overset{F,\theta}{\rightsquigarrow} \sigma_j$ and $\text{control}(\sigma_j) \in \text{addr}(F, \theta)$. By the definition of $\models \{\Psi\}F\{\Psi'\}$, we have $\exists \sigma_{j+1} \in \Sigma.\sigma_j \mapsto_\theta \sigma_{j+1}$.

Case $\exists 0 \leq i < j$. $\text{control}(\sigma_i) \notin \text{addr}(F, \theta)$. Let i be the least such index, then $\sigma_0 \overset{F,\theta}{\rightsquigarrow} \sigma_i$.

By the definition of $\models \{\Psi\}F\{\Psi'\}$, there exists some $l' \triangleright p' \in \Psi'$ such that

$$\text{control}(\sigma_i) = \theta(l') \tag{A.41.7}$$

$$\mathcal{A}\,[\![p']\!]\,\sigma_i = \text{tt} \tag{A.41.8}$$

By A.41.2, together with the above two results, get $\text{safe\_state}(\sigma_i, k)$.

On the other hand, we have $i > 0$, because $\text{control}(\sigma_0) = \text{control}(\sigma') = \theta(l) \in \text{addr}(F, \Psi)$. (The last step is because that $l \in \text{labels}(\Psi) \subseteq \text{entries}(F)$.) Therefore, the length of the computation sequence from σ_i to σ_j is less than k. Hence, by $\text{safe\_state}(\sigma_i, k)$, we have that $\exists \sigma_{j+1} \in \Sigma.\sigma_j \mapsto_\theta \sigma_{j+1}$. $\square$

Lemma A.42 *Assume* (F, Ψ', Ψ) *is normal. Assume Assertion is negatively testable by the statement language. If* $F\,;\,\Psi' \models \Psi$*, then* $\models \{\Psi\}F\{\Psi'\}$.

To prove the lemma, we need to introduce auxiliary concepts and lemmas.

The set $\text{exits}(F)$ statically characterizes the set of possible exits of F. The following lemma states that dynamically any "exit" from F is indeed an element in $\text{exits}(F)$.

Lemma A.43 *If* $\text{loaded}(F, \text{i\_of}(\sigma), \theta)$*,* $\sigma \mapsto_\theta \sigma'$*, and* $\text{control}(\sigma) \in \text{addr}(F, \theta)$*, but* $\text{control}(\sigma') \notin \text{addr}(F, \theta)$*, then exists some* $l \in \text{exits}(F)$ *such that* $\text{control}(\sigma') = \theta(l)$*.*

Proof. The proof is by case analysis on $\sigma \mapsto_\theta \sigma'$. Suppose $\sigma = (\text{pc}, \pi, m)$.

Case $\pi(\text{pc}) = (x := e)$.

Then $\sigma' = (\text{pc}+1, \pi, m[x \mapsto \mathcal{V}\,[\![e]\!]\, m])$. Therefore,

$$\text{pc} = \text{control}(\sigma) \in \text{addr}(F, \theta) \tag{A.43.1}$$

$$\text{pc}+1 = \text{control}(\sigma') \notin \text{addr}(F, \theta) \tag{A.43.2}$$

By the definition of $\text{addr}(F, \theta)$, there exists some $(l : (x := e) : l') \in F$ such that

$$\text{pc} = \theta(l) \tag{A.43.3}$$

$\text{loaded}(F, \text{i\_of}(\sigma), \theta)$ gives that $\theta(l') = \theta(l) + 1$, hence

$$\text{control}(\sigma') = \text{pc} + 1 = \theta(l) + 1 = \theta(l').$$

We now prove that $l' \in \text{exits}(F)$: otherwise, there is some fragment $l' : (t) : l''$ in F; then $\text{control}(\sigma') = \theta(l') \in \text{addr}(F, \theta)$, which contradicts with the assumption that $\text{control}(\sigma') \notin \text{addr}(F, \theta)$.

The proofs of other cases are similar. $\square$

Definition A.44 *(Address mapping.) Let* $\pi, \pi' \in IM$*,* $F \in FragSet$ *and* F *is normal,* $\theta, \theta' \in LMap$*. Suppose* $\text{loaded}(F, \pi, \theta)$ *and* $\text{loaded}(F, \pi', \theta')$*. An address mapping* $\epsilon \in \mathbb{N} \to \mathbb{N}$ *maps* n *to* n' *if*

(i) there is some $(l : (t) : l') \in F$*, such that* $n = \theta(l)$ *and* $n' = \theta'(l)$*,*

(ii) or there is some $l' \in \text{exits}(F)$ such that $n = \theta(l')$ and $n' = \theta'(l')$.

The definition of ϵ is well-defined since F is normal and the restrictions placed on θ and θ' in the definitions of $\text{loaded}(F, \pi, \theta)$ and $\text{loaded}(F, \pi', \theta')$.

Definition A.45 *Let $F \in \mathit{FragSet}$, $\theta, \theta' \in \mathit{LMap}$. We call two states σ and σ' are related if there is an address mapping ϵ such that*

(i) $\text{loaded}(F, \text{i\_of}(\sigma), \theta)$,

(ii) $\text{loaded}(F, \text{i\_of}(\sigma'), \theta')$,

(iii) $\epsilon(\text{control}(\sigma)) = \text{control}(\sigma')$,

(iv) $\text{m\_of}(\sigma) = \text{m\_of}(\sigma')$,

Where ϵ is as defined in Definition A.44. We describe the above as $\sigma \overset{F,\theta,\theta'}{\leftrightarrows} \sigma'$.

Lemma A.46 *Let $\sigma, \sigma' \in \Sigma$, $F \in \mathit{FragSet}$, $\theta, \theta' \in \mathit{LMap}$. If*

(i) $\sigma \overset{F,\theta,\theta'}{\leftrightarrows} \sigma'$

(ii) $\text{control}(\sigma) \in \text{addr}(F, \theta)$,

(iii) $\exists \sigma_1 \in \Sigma.\ \sigma \mapsto_\theta \sigma_1$,

then there exists $\sigma_1' \in \Sigma$ such that $\sigma' \mapsto_{\theta'} \sigma_1'$. Furthermore, σ_1 and σ_1' are also related: $\sigma_1 \overset{F,\theta,\theta'}{\leftrightarrows} \sigma_1'$.

Proof. By case analysis on $\sigma \mapsto_\theta \sigma_1$. □

Corollary A.47 *If $\sigma_0 \overset{F,\theta}{\rightsquigarrow} \sigma_1$ and $\sigma_0 \overset{F,\theta,\theta'}{\leftrightarrows} \sigma_0'$, then there is $\sigma_1' \in \Sigma$ such that $\sigma_0' \overset{F,\theta'}{\rightsquigarrow} \sigma_1'$, and $\sigma_1 \overset{F,\theta,\theta'}{\leftrightarrows} \sigma_1'$.*

Proof of Lemma A.42. To prove $\models \{\Psi\}F\{\Psi'\}$, choose $l \triangleright p \in \Psi$, $\sigma \in \Sigma$, $\theta \in LMap$, and $\sigma_1 \in \Sigma$, and assume

$$\text{loaded}(F, \text{i\_of}(\sigma), \theta) \tag{A.47.1}$$

$$\text{control}(\sigma) = \theta(l) \tag{A.47.2}$$

$$\mathcal{A}\,[\![p]\!]\,\text{m\_of}(\sigma) = \text{tt} \tag{A.47.3}$$

$$\sigma \overset{F,\theta}{\rightsquigarrow} \sigma_1 \tag{A.47.4}$$

The goal is to prove that

$$\begin{aligned}
&(\text{control}(\sigma_1) \in \text{addr}(F,\theta) \;\Rightarrow\; \exists \sigma_2 \in \Sigma.\ \sigma_1 \mapsto_\theta^* \sigma_2) \\
&(\text{control}(\sigma_1) \notin \text{addr}(F,\theta) \\
&\quad \Rightarrow\; \exists l' \triangleright p' \in \Psi'.\ \text{control}(\sigma_1) = \theta(l') \;\wedge\; \mathcal{A}\,[\![p']\!]\,\text{m\_of}(\sigma_1) = \text{tt})
\end{aligned} \tag{A.47.5}$$

To prove the goal, we construct a new instruction memory π' and a new label mapping θ', whose layout is

(i) The instruction memory π' contains F so that $\text{loaded}(F, \pi', \theta')$ is true.

(ii) For each $l' \triangleright p' \in \Psi'$, put the statement sequence $\textbf{test}(p')$ at the address $\theta'(l')$; $\textbf{test}(p')$ exists for p' since *Assertion* is negatively testable. After $\textbf{test}(p')$, put an **illegal** statement. This arrangement makes sure that $(\theta'(l), \pi', \text{m\_of}(\sigma)); \theta \models_k l' \triangleright p'$ is true for all k, because if p' is true before executing $\textbf{test}(p')$, $\textbf{test}(p')$ will be in an infinite loop (see Definition 2.10). Furthermore, the arrangement also implies that if p' is false before executing $\textbf{test}(p')$, then the state will get stuck eventually, since $\textbf{test}(p')$ will terminates and the next instruction is the **illegal** statement.

Let $\sigma' = (\theta'(l), \pi', \mathrm{m\_of}(\sigma))$. By the definition of $F\,;\,\Psi' \models \Psi$ and the special arrangement of σ', we get

$$\forall k \in \mathbb{N}.\ \sigma'; \theta \models_{k+1} \Psi \tag{A.47.6}$$

By the definition of $\sigma'; \theta \models_{k+1} \Psi$, together with the fact $\mathrm{control}(\sigma') = \theta'(l)$, the result A.47.3, and the fact $\mathrm{m\_of}(\sigma) = \mathrm{m\_of}(\sigma')$, the state σ' can run for any number of steps:

$$\forall k \in \mathbb{N}.\ \mathrm{safe\_state}(\sigma', k). \tag{A.47.7}$$

The state σ' has been constructed in such a way that it is straightforward to verify that

$$\sigma \overset{F,\theta,\theta'}{\leftrightarrows} \sigma'. \tag{A.47.8}$$

By Corollary A.47 together with the result A.47.4, there is some $\sigma_1' \in \Sigma$ and

$$\sigma' \overset{F,\theta'}{\rightsquigarrow} \sigma_1' \tag{A.47.9}$$

$$\sigma_1 \overset{F,\theta,\theta'}{\leftrightarrows} \sigma_1'. \tag{A.47.10}$$

From the result A.47.7 and A.47.9, the following can be derived:

$$\forall k \in \mathbb{N}.\ \mathrm{safe\_state}(\sigma_1', k). \tag{A.47.11}$$

The goal A.47.5 has two cases.

Case: when $\mathrm{control}(\sigma_1) \in \mathrm{addr}(F, \theta)$. Then $\mathrm{control}(\sigma_1') \in \mathrm{addr}(F, \theta')$. By Lemma A.46, the result A.47.10, A.47.11, there is a next state for σ_1.

Case: when $\mathrm{control}(\sigma_1) \notin \mathrm{addr}(F, \theta)$. Applying the Lemma A.43, there exists some $l' \in \mathrm{exits}(F)$ such that

$$\mathrm{control}(\sigma_1) = \theta(l') \tag{A.47.12}$$

Then,

$$\mathrm{control}(\sigma_1') = \theta'(l') \tag{A.47.13}$$

Since Ψ' is a normal label-continuation set relative to F, there is some $l' \triangleright p' \in \Psi'$ such that $\mathrm{control}(\sigma_1) = \theta(l')$. Furthermore,

$$\mathcal{A}\,[\![p']\!]\,\mathrm{m\_of}(\sigma_1) = \mathcal{A}\,[\![p']\!]\,\mathrm{m\_of}(\sigma_1') = \mathrm{tt} \tag{A.47.14}$$

Otherwise, because of the arrangement to the instruction memory in σ_1', the state σ_1' will eventually get stuck — contradiction with A.47.11. □

With Lemma A.41 and Lemma A.42, we have that the continuation-style semantics and direct-style semantics are equivalent.

Theorem A.48 *Assume (F, Ψ', Ψ) is normal. Assume the assertion language is negatively testable by the statement language. Then, $\models \{\Psi\}F\{\Psi'\}$ is equivalent to $F\,;\,\Psi' \models \Psi$.*

A.3 Completeness Proof of $\mathcal{L}_c$

Lemma A.49 *Let $p \in$ Assertion, $F \in$ FragSet, $l \in \mathrm{entries}(F)$, and $\Psi' \in$ LContSet. If $p \simeq \mathrm{wp}(l, F, \Psi')$, then*

(i) $F\,;\,\Psi' \models \{l \triangleright p\}$ (i.e., p is a precondition)

(ii) $\forall p' \in \mathit{Assertion}.\ \big(F\,;\,\Psi' \models \{l \rhd p'\} \;\Rightarrow\; \models p' \Rightarrow p\big)$ *(p is the weakest)*

Proof. (i). Choose $\theta \in \mathit{LMap}$, $\sigma \in \Sigma$, $k \in \mathbb{N}$. Assume

$$\text{loaded}(F, \text{i\_of}(\sigma), \theta), \tag{A.49.1}$$

$$\sigma;\theta \models_k \Psi'. \tag{A.49.2}$$

The goal is to prove $\sigma;\theta \models_{k+1} \{l \rhd p\}$. From its definition, choose $\sigma' \in \Sigma$ and assume

$$\sigma \mapsto_\theta^* \sigma', \tag{A.49.3}$$

$$\text{control}(\sigma') = \theta(l), \tag{A.49.4}$$

$$\mathcal{A}\,[\![p]\!]\,(\text{m\_of}(\sigma')) = \text{tt}. \tag{A.49.5}$$

We need to prove that $\text{safe\_state}(\sigma', k+1)$.

From the Lemma A.29, the result A.49.3 and A.49.1, we have

$$\text{loaded}(F, \text{i\_of}(\sigma'), \theta) \tag{A.49.6}$$

From the Lemma A.39 (iii), and the result A.49.3 and A.49.2, we have

$$\sigma';\theta \models_k \Psi' \tag{A.49.7}$$

Now use the definition of $p \simeq \text{wp}(l, F, \Psi')$, together with A.49.5, A.49.6, A.49.7 and A.49.4, to get the conclusion

$$\text{safe\_state}(\sigma', k+1)$$

(ii). To prove $\models p' \Rightarrow p$, assume

$$\mathcal{A}\,[\![p']\!]\,m = \mathrm{tt}, \tag{A.49.8}$$

and the goal is $\mathcal{A}\,[\![p]\!]\,m = \mathrm{tt}$. By the definition of $p \simeq \mathrm{wp}(l, F, \Psi')$, it is sufficient to prove the following:

$$\begin{aligned} &\forall \theta \in \mathit{LMap}, \pi \in \mathit{IM}, k \in \mathbb{N} \\ &\quad \mathrm{loaded}(F, \pi, \theta) \ \wedge \ \Big((\theta(l), \pi, m); \theta \models_k \Psi'\Big) \\ &\qquad \Rightarrow \mathrm{safe\_state}((\theta(l), \pi, m), k+1) \end{aligned} \tag{A.49.9}$$

To prove it, pick $\theta \in \mathit{LMap}$, $\pi \in \mathit{IM}$, $k \in \mathbb{N}$, assume

$$\mathrm{loaded}(F, \pi, \theta), \tag{A.49.10}$$

$$(\theta(l), \pi, m); \theta \models_k \Psi'. \tag{A.49.11}$$

Now, use the assumption $F\,;\Psi' \models \{l \triangleright p'\}$. Unpack its definition, use the result A.49.10, A.49.11, $(\theta(l), \pi, m) \longmapsto_\theta^* (\theta(l), \pi, m)$, A.49.8, to get $\mathrm{safe\_state}((\theta(l), \pi, m), k+1)$. □

Theorem A.50 *Assume $\mathcal{D}$ is complete. Assume Assertion is negatively testable by the statement language. Let $F = \{l : (t) : l'\}$. Let $\Psi' \in \mathit{LContSet}$, and assume Ψ' is a normal exit label-continuation set relative to F. Then we can find an assertion p such that $p \simeq \mathrm{wp}(l, \{l : (t) : l'\}, \Psi')$ and $\{l : (t) : l'\}\,;\Psi' \vdash \{l \triangleright p\}$ is derivable in $\mathcal{L}_c$.*

Proof. Proof by case analysis on t.

Case $\boxed{t = (x := e)}$

Since $\text{exits}(F) = \{\, l' \,\}$ and Ψ' is normal, assume $\Psi' = \{l' \triangleright p'\}$ for some assertion p'.

Let $p = p'[e/x]$. Then,

$$\{l : (x := e) : l'\}\,;\, \{l' \triangleright p'\} \;\vdash\; \{l \triangleright p'[e/x]\} \tag{A.50.1}$$

is derivable by the assign rule.

Then, we prove that $p'[e/x]$ is a weakest precondition, i.e.,

$$p'[e/x] \simeq \text{wp}(l, \{l : (x := e) : l'\}, \{l' \triangleright p'\}) \tag{A.50.2}$$

By definition 2.12, we need to prove that

$$\begin{aligned}&\forall m \in DM.\, \mathcal{A}\,[\![p'[e/x]]\!]\, m = \text{tt}\\ &\Leftrightarrow \left(\begin{array}{l}\forall \theta \in LMap, \pi \in IM, k \in \mathbb{N}\\ \quad \text{loaded}(\{l : (x := e) : l'\}, \pi, \theta) \;\wedge\; \Big((\theta(l), \pi, m); \theta \models_k \{l' \triangleright p'\}\Big)\\ \qquad \Rightarrow \text{safe\_state}((\theta(l), \pi, m), k+1)\end{array}\right)\end{aligned} \tag{A.50.3}$$

From the soundness theorem (Theorem A.34) , and the result A.50.1, we have

$$\{l : (x := e) : l'\}\,;\, \{l' \triangleright p'\} \;\models\; \{l \triangleright p'[e/x]\},$$

by which we can easily prove the forward direction in the goal A.50.3.

For the backward direction, assume

$$\begin{aligned}
&\forall \theta \in \mathit{LMap}, \pi \in \mathit{IM}, k \in \mathbb{N} \\
&\quad \text{loaded}(\{l : (x := e) : l'\}, \pi, \theta) \ \wedge \ \Big((\theta(l), \pi, m); \theta \models_k \{l' \triangleright p'\}\Big) \\
&\qquad \Rightarrow \text{safe\_state}((\theta(l), \pi, m), k+1)
\end{aligned} \tag{A.50.4}$$

We prove $\mathcal{A} [\![p'[e/x]]\!] \, m = \text{tt}$ by contradiction.

Suppose $\mathcal{A} [\![p'[e/x]]\!] \, m = \text{ff}$. Construct $\theta \in \mathit{LMap}$, $\pi \in \mathit{IM}$ so that π is arranged like the following:

- At $\theta(l)$, put the statement $x := e$.
- After $x := e$, put **test**(p'), and then **illegal**.

Construct a state $(\theta(l), \pi, m)$. Because of the arrangement of of π and θ, the following two results can be verified:

$$\text{loaded}(\{l : (x := e) : l'\}, \pi, \theta) \tag{A.50.5}$$

$$(\theta(l), \pi, m); \theta \models_k \{l' \triangleright p'\} \tag{A.50.6}$$

Therefore, from A.50.4, we have safe_state$((\theta(l), \pi, m), k+1)$ for any k.

If $\mathcal{A} [\![p'[e/x]]\!] \, m = \text{ff}$, then by the Lemma A.30 (iii), we have that $\mathcal{A} [\![p']\!] \, m[x \mapsto \mathcal{V} [\![e]\!] \, m] = \text{ff}$. This means that the state $(\theta(l), \pi, m)$ will reach an **illegal** statement eventually, which contradicts with the result safe_state$((\theta(l), \pi, m), k+1)$ for any k.

Case $\boxed{t = (\textbf{goto } l_1)}$

Since exits$(\{l : (\textbf{goto } l_1) : l'\}) = \{ l_1, l' \}$, assume $\Psi' = \{l_1 \triangleright p_1, l' \triangleright p'\}$ for some assertion p_1 and p'.

Then,

$$\{l : (\textbf{goto}\ l_1) : l'\}\,;\ \{l_1 \triangleright p_1, l' \triangleright p'\} \ \vdash\ \{l \triangleright p_1\} \tag{A.50.7}$$

is derivable by the goto rule and the weaken rule.

By a proof similar to the case of $t = (x := e)$, we can prove that p_1 is a weakest precondition of l, or

$$p_1 \simeq \mathrm{wp}(l, \{l : (\textbf{goto}\ l_1) : l'\}, \{l_1 \triangleright p_1, l' \triangleright p'\}) \tag{A.50.8}$$

Case $\boxed{t = (\textbf{if}\ b\ \textbf{goto}\ l_1)}$

Since $\mathrm{exits}(\{l : (\textbf{if}\ b\ \textbf{goto}\ l_1) : l'\}) = \{\, l_1, l' \,\}$, assume $\Psi' = \{l_1 \triangleright p_1, l' \triangleright p'\}$ for some assertion p_1 and p'.

Let $p = (b \Rightarrow p_1) \wedge (\neg b \Rightarrow p')$. By the if rule, $\mathcal{L}_c$ can derive

$$\{l : (\textbf{if}\ b\ \textbf{goto}\ l_1) : l'\}\,;\ \{l_1 \triangleright p \wedge b, l' \triangleright p \wedge \neg b\} \ \vdash\ \{l \triangleright p\} \tag{A.50.9}$$

Since $\mathcal{D}$ is complete, the following two are derivable:

$$\vdash_{\mathcal{D}} p \wedge b \Rightarrow p_1 \tag{A.50.10}$$

$$\vdash_{\mathcal{D}} p \wedge \neg b \Rightarrow p' \tag{A.50.11}$$

Use the weaken rule twice on A.50.9 to get

$$\{l : (\textbf{if}\ b\ \textbf{goto}\ l_1) : l'\}\,;\ \{l_1 \triangleright p_1, l' \triangleright p'\} \ \vdash\ \{l \triangleright p\} \tag{A.50.12}$$

At the same time, by a proof similar to the case of $t = (x := e)$, we can prove that p is a weakest precondition of l, or

$$p \simeq \mathrm{wp}(l, \{l : (\textbf{if } b \textbf{ goto } l_1) : l'\}, \{l_1 \triangleright p_1, l' \triangleright p'\}) \tag{A.50.13}$$

$\square$

Theorem A.51 *Assume $\mathcal{D}$ is complete and Assertion is expressive relative to $\oint$. Assume Assertion is negatively testable by the statement language. Let F be a normal fragment set, and Ψ' be a normal exit label-continuation set relative to F. Then for all $l \in \mathrm{entries}(F)$, we can find an assertion p such that $p \simeq \mathrm{wp}(l, F, \Psi')$ and $F\,;\,\Psi' \vdash \{l \triangleright p\}$ is derivable in $\mathcal{L}_c$.*

Before proving this theorem, we first prove an auxiliary lemma.

Lemma A.52 *Assume Assertion is negatively testable by the statement language. Let $F_1, F_2 \in$ FragSet and $F_1 \cup F_2$ is a normal fragment set. Let Ψ be a normal exit label-continuation set relative to $F_1 \cup F_2$. Let $L_2 = \mathrm{exits}(F_1) \cap \mathrm{entries}(F_2)$. Let Ψ_2 be a label-continuation set for labels in L_2 such that for each $l_2 \in L_2$, there is one assertion p_2 such that $l_2 \triangleright p_2 \in \Psi_2$ and $p_2 \simeq \mathrm{wp}(l_2, F_1 \cup F_2, \Psi)$. Then for any assertion p, $p \simeq \mathrm{wp}(l, F_1 \cup F_2, \Psi) \Leftrightarrow p \simeq \mathrm{wp}(l, F_1, \Psi \cup \Psi_2)$.*

Proof. The conclusion can be proved if the following equivalence is true:

$$\begin{array}{l}
\forall m \in DM. \\
\left(\begin{array}{l}
\forall \theta \in LMap, \pi \in IM, k \in \mathbb{N} \\
\quad \text{loaded}(F_1 \cup F_2, \pi, \theta) \ \wedge \ \big((\theta(l), \pi, m); \theta \models_k \Psi\big) \\
\quad\quad \Rightarrow \text{safe\_state}((\theta(l), \pi, m), k+1)
\end{array}\right) \\
\Leftrightarrow \\
\left(\begin{array}{l}
\forall \theta \in LMap, \pi \in IM, k \in \mathbb{N} \\
\quad \text{loaded}(F_1, \pi, \theta) \ \wedge \ \big((\theta(l), \pi, m); \theta \models_k \Psi \cup \Psi_2\big) \\
\quad\quad \Rightarrow \text{safe\_state}((\theta(l), \pi, m), k+1)
\end{array}\right)
\end{array}$$

"$\Rightarrow$". Assume

$$\begin{array}{l}
\forall \theta \in LMap, \pi \in IM, k \in \mathbb{N} \\
\quad \text{loaded}(F_1 \cup F_2, \pi, \theta) \ \wedge \ \big((\theta(l), \pi, m); \theta \models_k \Psi\big) \\
\quad\quad \Rightarrow \text{safe\_state}((\theta(l), \pi, m), k+1)
\end{array} \tag{A.52.1}$$

Pick $\theta \in LMap$, $\pi \in IM$ and $k \in \mathbb{N}$ and assume

$$\text{loaded}(F_1, \pi, \theta) \tag{A.52.2}$$

$$(\theta(l), \pi, m); \theta \models_k \Psi \cup \Psi_2 \tag{A.52.3}$$

and safe_state$((\theta(l), \pi, m), k+1)$ needs to be proved. To prove it, choose $j < k+1$, and a sequence of states $\sigma_0, \sigma_1, \ldots, \sigma_j$ such that

$$(\theta(l), \pi, m) = \sigma_0 \mapsto \sigma_1 \mapsto \ldots \mapsto \sigma_j \tag{A.52.4}$$

The new goal is to show the existence of σ_{j+1} such that $\sigma_j \mapsto \sigma_{j+1}$.

The proof uses a new π' and θ', whose arrangement is:

(i) The instruction memory π' contains both F_1 and F_2 such that $\mathrm{loaded}(F_1 \cup F_2, \pi', \theta')$ is true.

(ii) For each $l' \triangleright p' \in \Psi$, put $\mathbf{test}(p')$ at the address $\theta'(l')$ in π'; hence, $(\theta'(l), \pi', m); \theta' \models_k \Psi$ is true. Furthermore, after $\mathbf{test}(p')$, put an **illegal** statement.

Therefore, by A.52.1, we get

$$\forall k \in \mathbb{N}.\ \mathrm{safe\_state}((\theta'(l), \pi', m), k+1) \tag{A.52.5}$$

That is, the state $(\theta'(l), \pi', m)$ can run for any number of steps.

The state $(\theta(l), \pi, m)$ and $(\theta'(l), \pi', m)$ are related:

$$(\theta(l), \pi, m) \overset{F_1,\theta,\theta'}{\leftrightarrows} (\theta'(l), \pi', m) \tag{A.52.6}$$

The proof proceeds in two cases.

Case $\forall 0 \le i \le j.\ \mathrm{control}(\sigma_i) \in \mathrm{addr}(F_1, \theta)$. That is, the control never leaves the address space of F_1. Thus, $\sigma_0 \overset{F_1,\theta}{\leadsto} \sigma_j$ is true. By Corollary A.47, there is some $\sigma'_j \in \Sigma$ such that $(\theta'(l), \pi', m) \overset{F_1,\theta'}{\leadsto} \sigma'_j$ and $\sigma_j \overset{F_1,\theta,\theta'}{\leftrightarrows} \sigma'_j$. Use Lemma A.46, and the fact that σ'_j has a next state (by A.52.5), to get a $\sigma_{j+1} \in \Sigma$ such that $\sigma_j \mapsto \sigma_{j+1}$.

Case $\exists i \le j.\ \mathrm{control}(\sigma_i) \notin \mathrm{addr}(F_1, \theta)$. Let i be the least such number. Since $\mathrm{control}(\sigma_0) \in \mathrm{addr}(F_1, \theta)$, we must have that $i > 0$. Thus, the length of computation from σ_i to σ_j is less than k.

Therefore, if we can show $\mathrm{safe\_state}(\sigma_i, k)$, the goal $\exists \sigma_{j+1}.\ \sigma_j \mapsto \sigma_{j+1}$ follows from the definition of $\mathrm{safe\_state}(\sigma_i, k)$.

To prove safe_state(σ_i, k), the first observation is that the control of σ_i is at some label in exits(F_1), by Lemma A.43. We then divide the proof of safe_state(σ_i, k) into two subcases.

Subcase $\exists l'.\ \text{control}(\sigma_i) = \theta(l') \ \wedge\ l' \in \text{exits}(F_1) \ \wedge\ l' \notin \text{entries}(F_2)$. Therefore, $l' \in \text{exits}(F_1 \cup F_2)$. Because Ψ is a normal exit label-continuation set relative to $F_1 \cup F_2$, there is some assertion p' such that $l' \triangleright p' \in \Psi$.

Because i is the least index such that control$(\sigma_i) \notin \text{addr}(F_1, \theta)$, we can prove that $\sigma_0 \overset{F_1,\theta}{\rightsquigarrow} \sigma_i$. By Corollary A.47, there is some $\sigma_i' \in \Sigma$ such that $(\theta'(l), \pi', m) \overset{F_1,\theta'}{\rightsquigarrow} \sigma_i'$ and $\sigma_i \overset{F_1,\theta,\theta'}{\leftrightarrows} \sigma_i'$.

Now we prove that $\mathcal{A}\,[\![p']\!]\,(\text{m\_of}(\sigma_i)) = \text{tt}$. Otherwise, because of $\sigma_i \overset{F_1,\theta,\theta'}{\leftrightarrows} \sigma_i'$, $\mathcal{A}\,[\![p']\!]\,(\text{m\_of}(\sigma_i')) = \mathcal{A}\,[\![p']\!]\,(\text{m\_of}(\sigma_i)) = \text{ff}$, and the next sequence of statements to execute in the state σ_i' are **test**(p') and then **illegal**. By the definition of **test**(p'), σ_i' will reach the **illegal** statement, and thus will get stuck. This contradicts with the result A.52.5.

With $\mathcal{A}\,[\![p']\!]\,(\text{m\_of}(\sigma_i)) = \text{tt}$, we can use A.52.3 to get that safe_state(σ_i, k).

Subcase $\exists l_2.\ \text{control}(\sigma_i) = \theta(l_2) \ \wedge\ l_2 \in \text{exits}(F_1) \cap \text{entries}(F_2)$. In this case, the following statement is true

$$\begin{aligned}
&\forall \theta'' \in \mathit{LMap}, \pi'' \in \mathit{IM}, k \in \mathbb{N}\\
&\quad \text{loaded}(F_1 \cup F_2, \pi'', \theta'') \ \wedge\ \big((\theta''(l_2), \pi'', m); \theta'' \models_k \Psi\big)\\
&\qquad \Rightarrow \text{safe\_state}((\theta''(l_2), \pi'', \text{m\_of}(\sigma_i)), k+1).
\end{aligned} \tag{A.52.7}$$

The proof is similar to the proof of the first subcase. First, the state σ_i' and $(\theta''(l_2), \pi'', \text{m\_of}(\sigma_i))$ are related:

$$\sigma_i' \overset{F_1 \cup F_2,\theta',\theta''}{\leftrightarrows} (\theta''(l_2), \pi'', \text{m\_of}(\sigma_i)).$$

Then by Corollary A.47, and the fact that σ_i' never gets stuck to get that $\text{safe\_state}((\theta''(l_2), \pi'', \text{m\_of}(\sigma_i)), k+1)$.

Suppose p_2 is the assertion such that $l_2 \triangleright p_2 \in \Psi_2$. By the definition $p_2 \simeq \text{wp}(l_2, F_1 \cup F_2, \Psi)$ and the result A.52.7, we get $\mathcal{A}\,[\![p_2]\!]\,\text{m\_of}(\sigma_i) = \text{tt}$. Then by A.52.3, we get $\text{safe\_state}(\sigma_i, k)$.

"$\Leftarrow$". Assume

$$\begin{aligned} &\forall \theta \in \mathit{LMap}, \pi \in \mathit{IM}, k \in \mathbb{N} \\ &\quad \text{loaded}(F_1, \pi, \theta) \ \wedge \ \big((\theta(l), \pi, m); \theta \models_k \Psi \cup \Psi_2\big) \\ &\qquad \Rightarrow \text{safe\_state}((\theta(l), \pi, m), k+1) \end{aligned} \tag{A.52.8}$$

Choose $\theta \in \mathit{LMap}$, $\pi \in \mathit{IM}$ and $k \in \mathbb{N}$, assume

$$\text{loaded}(F_1 \cup F_2, \pi, \theta), \tag{A.52.9}$$

$$(\theta(l), \pi, m); \theta \models_k \Psi, \tag{A.52.10}$$

and the goal is $\text{safe\_state}((\theta(l), \pi, m), k+1)$. To prove it, choose $j < k+1$, and a sequence of states $\sigma_0, \sigma_1, \ldots, \sigma_j$ such that

$$(\theta(l), \pi, m) = \sigma_0 \mapsto \sigma_1 \mapsto \ldots \mapsto \sigma_j, \tag{A.52.11}$$

and the new goal is to show the existence of some σ_{j+1} such that $\sigma_j \mapsto \sigma_{j+1}$.

Construct π' and θ' whose arrangement is like the following:

(i) The instruction memory π' contains F_1 such that $\text{loaded}(F_1, \pi', \theta')$.

(ii) For each $l' \triangleright p' \in \Psi \cup \Psi_2$, put $\mathbf{test}(p')$ at the address $\theta'(l')$ in π'; hence, $(\theta'(l), \pi', m); \theta' \models_k \Psi \cup \Psi_2$ is true. Furthermore, after the statement $\mathbf{test}(p')$, put an **illegal** statement.

Therefore, A.52.8 gives

$$\forall k \in \mathbb{N}.\ \mathrm{safe\_state}((\theta'(l), \pi', m), k) \tag{A.52.12}$$

Also, it is easy to verify

$$(\theta(l), \pi, m) \overset{F_1,\theta,\theta'}{\leftrightarrows} (\theta'(l), \pi', m) \tag{A.52.13}$$

If $\forall 0 \le i \le j.\ \mathrm{control}(\sigma_i) \in \mathrm{addr}(F_1, \theta)$, or $\exists i < j.\ \exists l'.\ \mathrm{control}(\sigma_i) = \theta(l') \ \wedge\ l' \in \mathrm{exits}(F_1 \cup F_2)$, a proof which is the similar to the one in the forward direction of this lemma will prove $\exists \sigma_{j+1} \in \Sigma.\ \sigma_j \mapsto \sigma_{j+1}$.

The interesting case is when there is some σ_i such that $\exists l_2.\ \mathrm{control}(\sigma_i) = \theta(l_2) \ \wedge\ l_2 \in \mathrm{exits}(F_1) \cap \mathrm{entries}(F_2)$. Assume i is the least index with the property.

Then by Corollary A.47, there is some $\sigma_i' \in \Sigma$ such that

$$\sigma_i \overset{F_1,\theta,\theta'}{\leftrightarrows} \sigma_i' \tag{A.52.14}$$

$$(\theta'(l), \pi', m) \overset{F_1,\theta'}{\rightsquigarrow} \sigma_i' \tag{A.52.15}$$

Therefore, $\mathrm{control}(\sigma_i') = \theta'(l_2)$. Now derive $\mathcal{A}\llbracket p_2 \rrbracket (\mathrm{m\_of}(\sigma_i)) = \mathcal{A}\llbracket p_2 \rrbracket (\mathrm{m\_of}(\sigma_i')) = \mathrm{tt}$. Otherwise, the state σ_i' will get stuck eventually.

Use the definition of $p_2 \simeq \mathrm{wp}(l_2, F_1 \cup F_2, \Psi)$, with $\mathcal{A}\llbracket p_2 \rrbracket (\mathrm{m\_of}(\sigma_i)) = \mathrm{tt}$, to derive $\mathrm{safe\_state}(\sigma_i, k+1)$.

The computation from σ_i to σ_j has length less than $k+1$. Therefore, by the definition of safe_state$(\sigma_i, k+1)$, there is some σ_{j+1} such that $\sigma_j \mapsto_\theta \sigma_{j+1}$. $\square$

Proof of Theorem A.51. By induction over the number of fragments in F. When the number is exactly one, use Theorem A.50. When the number is greater than one, divide F into two disjoint sets, F_1 and F_2, so that each one has at least one fragment.

Define label sets L_1 and L_2 to be

$$L_1 = \mathrm{entries}(F_1) \cap \mathrm{exits}(F_2)$$
$$L_2 = \mathrm{entries}(F_2) \cap \mathrm{exits}(F_1)$$

That is, L_1 is the set of labels that are defined in F_1 and are exits of F_2. L_2 is the set of labels that are defined in F_2 and are exits of F_1.

Because that the assertion language is expressive, for each $l_1 \in L_1$, there is some $p_1 \in \mathit{Assertion}$ such that $p_1 \simeq \mathrm{wp}(l_1, F_1 \cup F_2, \Psi')$. Construct $\Psi_1 \in \mathit{LContSet}$ to be a set of such $l_1 \triangleright p_1$.

Similarly, for each $l_2 \in L_2$, there is some $p_2 \in \mathit{Assertion}$ such that $p_2 \simeq \mathrm{wp}(l_2, F_1 \cup F_2, \Psi')$. Construct $\Psi_2 \in \mathit{LContSet}$ to be a set of such $l_2 \triangleright p_2$.

For each $l_1 \triangleright p_1 \in \Psi_1$, use the induction hypothesis on F_1 and l_1. Hence, there is some p'_1 such that

$$p'_1 \simeq \mathrm{wp}(l_1, F_1, \Psi' \cup \Psi_2) \tag{A.52.1}$$

$$F_1\,;\, \Psi' \cup \Psi_2 \vdash \{l_1 \triangleright p'_1\} \tag{A.52.2}$$

Use Lemma A.52, together with the result A.52.1, to get

$$p_1' \simeq \mathrm{wp}(l_1, F_1 \cup F_2, \Psi') \tag{A.52.3}$$

Since $p_1 \simeq \mathrm{wp}(l_1, \{l : (s_1; s_2) : l'\}, \Psi')$, we can derive that $\models p_1 \Leftrightarrow p_1'$. Because $\mathcal{D}$ is complete, $\vdash_{\mathcal{D}} p_1 \Leftrightarrow p_1'$ is derivable.

Use the weaken rule on the judgment A.52.2 to get

$$F_1 \,;\, \Psi' \cup \Psi_2 \;\vdash\; \{l_1 \triangleright p_1\} \tag{A.52.4}$$

Since the above judgment is derivable for each $l_1 \triangleright p_1 \in \Psi_1$, the $\mathsf{combine}$ rule can derive

$$F_1 \,;\, \Psi' \cup \Psi_2 \;\vdash\; \Psi_1 \tag{A.52.5}$$

Use a similar reasoning on Ψ_2, the following judgment is also derivable:

$$F_2 \,;\, \Psi' \cup \Psi_1 \;\vdash\; \Psi_2. \tag{A.52.6}$$

Now discuss the label l in the statement of theorem. Without loss of generality, assume $l \in \mathrm{entries}(F_1)$. Use the induction hypothesis on F_1 and l, we have some $p \in \mathit{Assertion}$ such that

$$p \simeq \mathrm{wp}(l, F_1, \Psi' \cup \Psi_2) \tag{A.52.7}$$

$$F_1 \,;\, \Psi' \cup \Psi_2 \;\vdash\; \{l \triangleright p\} \tag{A.52.8}$$

By Lemma A.52, together with the result A.52.7, we have

$$p \simeq \mathrm{wp}(l, F_1 \cup F_2, \Psi') \tag{A.52.9}$$

Use the combine rule to combine judgments A.52.5, A.52.6 and A.52.8, to get

$$F_1 \cup F_2\,;\, \Psi' \cup \Psi_1 \cup \Psi_2 \;\vdash\; \Psi_1 \cup \Psi_2 \cup \{l \triangleright p\} \tag{A.52.10}$$

Use the discharge rule multiple times to discharge $\Psi_1 \cup \Psi_2$ on the left of $\vdash$:

$$F_1 \cup F_2\,;\, \Psi' \;\vdash\; \Psi_1 \cup \Psi_2 \cup \{l \triangleright p\} \tag{A.52.11}$$

Use the weaken rule, together with $\vdash \Psi_1 \cup \Psi_2 \cup \{l \triangleright p\} \Rightarrow \{l \triangleright p\}$, to get

$$F_1 \cup F_2\,;\, \Psi' \;\vdash\; \{l \triangleright p\} \tag{A.52.12}$$

□

Theorem A.53 *(Completeness.)*
Assume $\mathcal{D}$ is complete and Assertion is expressive relative to $\oint$. Assume Assertion is negatively testable by the statement language. Let Assume (F, Ψ', Ψ) is normal. If $F\,;\, \Psi' \models \Psi$, then $F\,;\, \Psi' \vdash \Psi$.

Proof. For each $l \triangleright p \in \Psi$, $F\,;\, \Psi' \models \{l \triangleright p\}$, because $F\,;\, \Psi' \models \Psi$.

Since Ψ is a normal entry label-continuation set relative to F, we have $l \in$ entries(F). Use Theorem A.51, there is some $p' \in \mathit{Assertion}$, such that

$$p' \simeq \mathrm{wp}(l, F, \Psi') \tag{A.53.1}$$

$$F\,;\,\Psi' \vdash \{l \triangleright p'\} \tag{A.53.2}$$

By Lemma A.49 (ii), together with $F\,;\,\Psi' \models \{l \triangleright p\}$ and the result A.53.1, get

$$\models p \Rightarrow p'$$

By the completeness of $\mathcal{D}$, $\vdash_{\mathcal{D}} p \Rightarrow p'$ is derivable. By the weaken rule, together with the judgment A.53.2, gives that

$$F\,;\,\Psi' \vdash \{l \triangleright p\}$$

The above judgment is derivable for each $l \triangleright p \in \Psi$. Then the combine rule gives

$$F\,;\,\Psi' \vdash \Psi$$

$\square$

Bibliography

[1] CERT advisory CA-1999-04 Melissa macro virus. http://www.cert.org/advisories/CA-1999-04.html, March 1999.

[2] Microsoft security bulletin MS02-065. http://www.microsoft.com/technet/security/bulletin/MS02-065.mspx, November 2002.

[3] A. Ahmed, A. W. Appel, and R. Virga. A stratified semantics of general references embeddable in higher-order logic. In *17th Annual IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 75–86, July 2002.

[4] A. J. Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, Nov. 2004.

[5] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[6] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, Cambridge, England, 1992.

[7] A. W. Appel. Foundational proof-carrying code. In *Symposium on Logic in Computer Science (LICS '01)*, pages 247–258. IEEE, 2001.

[8] A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *POPL '00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 243–253. ACM Press, Jan. 2000.

[9] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. on Programming Languages and Systems*, 23(5):657–683, Sept. 2001.

[10] A. W. Appel, N. G. Michael, A. Stump, and R. Virga. A trustworthy proof checker. *Journal of Automated Reasoning*, 31:231–260, 2003.

[11] M. Arbib and S. Alagic. Proof rules for gotos. *Acta Informatica*, 11:139–148, 1979.

[12] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language, Third Edition*. Addison Wesley, Reading, MA, 2000.

[13] A. Bernard and P. Lee. Temporal logic for proof-carrying code. In *18th International Conference on Automated Deduction (CADE '02)*, pages 31–46, 2002.

[14] R. S. Boyer and Y. Yu. Automated proofs of object code for a widely used microprocessor. *Journal of the Association for Computing Machinery*, 43(1):166–192, 1996.

[15] L. Cardelli. Type systems. In *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC press, 1997.

[16] J. Chen. *A Low-Level Typed Assembly Language with a Machine-Checkable Soundness Proof*. PhD thesis, Princeton University, Feb. 2004.

[17] J. Chen, D. Wu, A. W. Appel, and H. Fang. A provably sound TAL for back-end optimization. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '03)*, pages 208–219, 2003.

[18] *Common Language Infrastructure (CLI), 2nd Edition.* ECMA International, Geneva, Switzerland, 2002.

[19] M. Clint and C. A. R. Hoare. Program proving: Jumps and functions. *Acta Informatica*, pages 214–224, 1972.

[20] C. Colby, P. Lee, G. C. Necula, F. Blau, K. Cline, and M. Plesko. A certifying compiler for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*, New York, June 2000. ACM Press.

[21] S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90, Feb. 1978.

[22] K. Crary. Toward a foundational typed assembly language. In *The 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 198–212. ACM Press, Jan. 2003.

[23] K. Crary and S. Sarkar. Foundational certified code in a metalogical framework. In *19th International Conference on Automated Deduction (CADE '03)*, pages 106–120, 2003.

[24] K. Crary, D. Walker, and J. G. Morrisett. Typed memory management in a calculus of capabilities. In *POPL'99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 262–275, 1999.

[25] K. Crary and S. Weirich. Resource bound certification. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–198, New York, NY, USA, 2000. ACM Press.

[26] A. de Bruin. Goto statements: Semantics and deduction systems. *Acta Informatica*, 15:385–424, 1981.

[27] R. A. DeMillo, R. J. Lipton, and A. J. Perlis. Social processes and proofs of theorems and programs. In *POPL '77: The Fourth ACM Symposium on Principles of Programming Languages*, pages 206–214, 1977.

[28] S. Drossopoulou and S. Eisenbach. Java is type safe - probably. In *ECOOP '97: European Conference on Object-Oriented Programming*, pages 389–418, 1997.

[29] D. Evans and A. Twyman. Flexible policy-directed code safety. In *IEEE Symposium on Security and Privacy*, pages 32–45, 1999.

[30] R. W. Floyd. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967.

[31] J.-Y. Girard. *Interprétation Fonctionnelle et Elimination des Coupures dans l'Arithmétique d'Ordre Supérieur.* PhD thesis, University of Paris VII, 1972.

[32] M. J. C. Gordon and T. F. Melham. *Introduction to HOL—A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[33] N. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. In *Proc. 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*, pages 89–100, July 2002.

[34] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, Jan. 1993.

[35] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the Association for Computing Machinery*, 12(10):578–580, October 1969.

[36] M. Hofmann. Linear types and non-size-increasing polynomial time computation. *Information and Computation*, 183(1):57–85, 2003.

[37] G. Hoglund and G. McGraw. *Exploiting Software: How to Break Code.* Addison-Wesley Professional, 2004.

[38] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 275–288. USENIX Association, 2002.

[39] T. Kowaltowski. Axiomatic approach to side effects and general jumps. *Acta Informatica*, 7(4):357–360, 1977.

[40] C. League, Z. Shao, and V. Trifonov. Precision in practice: A type-preserving Java compiler. In *Proc. Int'l. Conf. on Compiler Construction*, April 2003.

[41] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition.* Addison Wesley, Reading, MA, 1999.

[42] N. G. Michael and A. W. Appel. Machine instruction syntax and semantics in higher-order logic. In *17th International Conference on Automated Deduction*, pages 7–24, Berlin, June 2000. Springer-Verlag. LNAI 1831.

[43] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, 1997.

[44] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A realistic typed assembly language. In *Second ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, 1999. INRIA Technical Report 0288, March 1999.

[45] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Trans. on Programming Languages and Systems*, 21(3):527–568, May 1999.

[46] G. Necula. Proof-carrying code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, New York, Jan. 1997. ACM Press.

[47] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Symposium On Operating System Design and Implementation*, pages 229–243, 1996.

[48] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '98)*, pages 333–344, 1998.

[49] G. C. Necula, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 128–139, 2002.

[50] G. Nelson, editor. *Systems Programming with Modula-3*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[51] T. Nipkow and D. von Oheimb. Java-light is type-safe - definitely. In *ACM Symposium on Principles of Programming Languages*, pages 161–170, 1998.

[52] The economic impacts of inadequate infrastructure for software testing. `http://www.nist.gov/director/prog-ofc/report02-3.pdf`, May 2002.

[53] M. J. O'Donnell. A critique of the foundations of hoare style programming logics. *Communications of the Association for Computing Machinery*, 25(12):927–935, 1982.

[54] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[55] F. Pfenning and C. Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *The 16th International Conference on Automated Deduction*, Berlin, July 1999. Springer-Verlag.

[56] J. C. Reynolds. Towards a theory of type structure. In *Proc. Paris Symp. on Programming*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, Berlin, 1974. Springer.

[57] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Boston, 1986.

[58] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, Feb. 2000.

[59] F. B. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. In *Informatics: 10 Years Back, 10 Years Ahead*, pages 86–101, 2001.

[60] D. Seal. *ARM Architecture Reference Manual, Second Edition*. Addison-Wesley Professional, Reading, MA, 2000.

[61] Z. Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97)*, Amsterdam, The Netherlands, June 1997.

[62] M. H. Sørensen and P. Urzyczyn. Lectures on the Curry-Howard isomorphism. Available as DIKU Rapport 98/14, 1998.

[63] *The SPARC Architecture Manual, Version 8*. SPARC International Inc., Menlo Park, CA, 1991.

[64] K. N. Swadi. *Typed Machine Language*. PhD thesis, Princeton University, Nov. 2003.

[65] G. Tan, A. W. Appel, K. N. Swadi, and D. Wu. Construction of a semantic model for a typed assembly language. In *Fifth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 04)*, volume 2937 of *Lecture Notes in Computer Science*, pages 30–43. Springer, 2004.

[66] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ml. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*, pages 181–192, 1996.

[67] J. C. Vanderwaart and K. Crary. Automated and certified conformance to responsiveness policies. In *TLDI '05: Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 79–90, New York, NY, USA, 2005.

[68] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 203–216, 1993.

[69] D. Walker. A type system for expressive security policies. In *POPL '00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 254–267. ACM Press, Jan. 2000.

[70] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[71] D. Wu, A. W. Appel, and A. Stump. Foundational proof checkers with small witnesses. In *5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 264–274, Aug. 2003.

[72] H. Xi and R. Harper. A dependently typed assembly language. In *2001 ACM SIGPLAN International Conference on Function Programming*, pages 169–180. ACM Press, Sept. 2001.

Index of Notation/Definitions