INTERFACING COMPILERS, PROOF CHECKERS, AND PROOFS FOR FOUNDATIONAL PROOF-CARRYING CODE

Dinghao Wu

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 general framework for the mechanical verification of safety properties of machine-language programs. It allows a code producer to provide an executable program to a code consumer, along with a machine-checkable proof of safety such that the code consumer can check the proof before running the program. PCC has the advantage of small Trusted Computing Base (TCB), since the proof checking can be a simple mechanical procedure. A weakness of previous PCC systems is that the proof-checking infrastructure is based on some complicated logic or type system that is not necessarily sound.

Foundational Proof-Carrying Code (FPCC) aims to further reduce the TCB size by an order of magnitude by building the safety proof based on the simple and trustworthy foundations of mathematical logic. There are three major components in an FPCC system: a compiler, a proof checker, and the safety proof of an input machine-language program. The compiler produces machine code accompanied by a proof of safety. The proof checker verifies, sometimes also reconstructs, the safety proof before the program gets executed.

We have built a prototype system. Our prototype is the first end-to-end FPCC system, including a type-preserving compiler from Core ML to SPARC (based on SML/NJ), a low-level typed assembly language LTAL, a foundational proof-checker Flit, and a nearly complete machine-checkable soundness proof. The system compiles Core ML programs to SPARC code, accompanied with programs in a low-level typed assembly language; these typed assembly programs serve as the proof witnesses of the safety of the corresponding SPARC machine code.

In this thesis, I'll explain the design of interfaces between these components and show how to build an end-to-end FPCC system. We have concluded that a type

system (a low-level typed assembly language) should be designed to check machine code, and that the proof-checking should be factored into two stages, namely type-checking of the input machine code and verification of soundness of the type system. Since a type checker can be efficiently interpreted as a logic program, Flit builds in a simple logic programming engine which enables efficient proof-checking.

Acknowledgements

First of all I would like to thank my advisor Andrew Appel for his kindness, encouragement, and guidance. During the last five years Andrew has been extremely helpful on my research and life at Princeton. I am always impressed by his smartness, quick thinking and hacking, and clear explanation of things. It is of great fun working with Andrew! I also thank him for improving my English grammar and speaking; I am impressed by his fun explanation of the tenses using the time axis, a programming language researcher's rigorous way of interpreting the natural language English.

I would also like to thank other members of my thesis committee. Thank Aaron Stump and Dave Walker for reading my thesis and giving helpful comments. Thank Ed Felten and Bob Tarjan for serving on my PhD thesis committee.

I have benefited greatly from working with the fellow students and colleagues. Especially I would like to thank Amal Ahmed, Juan Chen, Hai Fang, Eun-Young Lee, Neophytos Michael, Xinming Ou, Chris Richards, Kedar Swadi, Gang Tan, Roberto Virga, and Dan Wang. I have also had a lot fun with other students including, not exclusively, Sudhakar Govindavajhala, Aquinas Hobor, Limin Jia, Junwen Lai, Jay Ligatti, Qin Lv, Yitzhak Mandelbaum, Ruoming Pang, Yaoping Ruan, Frances Spalding, Ming Zhang, Fengzhou Zheng, and Wen Xu.

Office 215 has always been a friendly and fun place. Thanks to Benedict Brown, Edith Elkind, Allison Klein, Steven Kleinstein, Gang Tan, Stephen Soltesz, and Zhe Wang for being my wonderful office mates.

Many thanks to Melissa Lawson for taking care of numerous things and making my life at Princeton a lot easier. Many people outside of the campus, some of whom have been my friends for years, have also helped me during my study at Princeton.

I am grateful to John Desai, Dick & Marie Gons, David Kim, Miriam Miller, and Jeff Olesnevich for their help with my English speaking and for many interesting discussions and activities.

I would like to give my special thanks to my family. The memory of my peaceful and happy childhood with Tài Pó (great grandmother) is always a source of peace and joy when I am anxious and restless. Mom's story of struggling to get her own education due to poverty, her long-lasting determination and faith in my education, her persevering fight for a better life, and her unfailing love are the greatest sources of strength that drives me forward. I could not express in any language that how much I owe to my mom; two college graduates and one PhD come out of a family once even struggled for food and clothes! My heartfelt thanks also go to my dad and brothers.

Finally, I would like to thank my wife Li Li for her love, care, support, and belief in me. Time at Princeton with Li Li has been one of the most joyful in my life so far. I am so lucky to have her accompanied me through high school, college, and graduate schools... I couldn't imagine how I could go through all these tough days without her support and love.

This work is partially funded by DARPA under grant F30602-99-1-0519, NSF under grant CCR-0208601, and ARDA under grant NBCHC030106.

TO LILI

Contents

List of Figures

List of Tables

Chapter 1

Introduction

During 1960s, with the rapid advance in the hardware industry, the so-called "software crisis" emerged. The software industry were not able to keep pace with the rapid advance of hardware. Software projects were notoriously behind schedule and over budget, and software products were full of defects and unreliable. While there have been lots of improvement with programming language and software engineering technology since then, software is still extremely fragile: unreliable, insecure, and full of bugs. Frederick Brooks explains "why programming is hard to manage" in his book *The Mythical Man-Month* [Brooks, 1975], and many principles and observations still apply today.

1.1 Software Security: A Growing Problem

On the other hand, the extensive use of computers and the accelerating trends of interconnectedness, complexity, and extensibility pose an increasing demand on the security of software. While interconnected computers on the Internet make our life

easier, malicious code such as viruses and worms can exploit the vulnerability of software and spread over the world in a minute. The complexity of software systems is rising. Large and complex systems tend to have more bugs and are more vulnerable to malicious code. Many of today's software systems support extensibility through a number of ways such as scripting, macros, and applets. The infamous Melissa and Love Bug viruses took advantage of the Internet and the macro and scripting extensions of the Microsoft Word document processing program and Outlook e-mail client [McGraw and Morrisett, 2000; Martin, 2000; Slade, 1999].

As our society becomes increasingly dependent on information technology, we must be able to produce software systems that are more secure, reliable, and dependable. In this thesis, we describe a promising approach to addressing the program safety problem and show how to build and verify secure software from the minimum trusted computing base. Part of this thesis work has been published in several conferences. Chapter 3 is the extended version of a PLDI paper [Chen et al., 2003]. Chapter 4 is based on the techniques described in Appel and McAllester [2001], Wu et al. [2003] and Tan et al. [2004]. Chapter 5 is the extended version of a PPDP paper [Wu et al., 2003].

1.2 Classical Security Principles

The price of reliability is the pursuit of utmost simplicity.

C.A.R. Hoare

To design secure systems, it is important to follow well-known design principles. In this section, we review two classical security principles, namely the principle of

least privilege and the principle of minimum Trusted Computing Base (TCB). The principle of minimum TCB is one of the important criteria used to measure the trustworthiness of our system.

1.2.1 Principle of least privilege

The principle of least privilege is an important concept in computer security. It was first described by Saltzer and Schroeder [1975]:

> Every program and every user of the system should operate using the least set of privileges necessary to complete the job. Primarily, this principle limits the damage that can result from an accident or error. It also reduces the number of potential interactions among privileged programs to the minimum for correct operation, so that unintentional, unwanted, or improper uses of privilege are less likely to occur.

The principle of least privilege states that a user, a system, or a program should be given no more privilege than necessary to perform a task. This can minimize the damage that can occur should your code be exploited by a malicious user since the minimum privilege is granted for the code.

The principle should be used in every system that is applicable. A good real-world example of this principle is the US government "need to know" policy in the security clearance system. People are only allowed to access documents that are relevant to their tasks.

Many programs run under UNIX systems (and other operating systems too) violate the principle of least privilege. The Sendmail program is a classical example. Sendmail runs with root permissions since it requires root privileges to set up a

service on port 25—the SMTP port. After the set up, Sendmail never gives up its root privileges. Therefore, if a malicious attack can buffer overflow in Sendmail, the attack can trick Sendmail to run arbitrary code with root permissions.

1.2.2 Principle of minimum trusted computing base

The Trusted Computing Base (TCB) is the set of hardware and software that needs to be trusted for the security of a task. Since nowadays hardware is quite reliable, trusted software systems tend to be the most significant component of TCB.

It is important to keep the TCB small and simple because, in general, large and complex systems tend to have more defects. In an investigation of Java-enabled browsers conducted by Dean et al. [1997], they found that there is one security-relevant bug per 3,000 lines of source code in average in the first-generation implementations. The TCBs of various Java Virtual Machines are at between 50,000 and 200,000 lines of code [Appel and Wang, 2002]. The SpecialJ JVM [Colby et al., 2000] reduces the TCB to 36,000 lines by using proof-carrying code. In this work, we will show how to reduce the size of the TCB to under 3,000 lines and make the proof checker small and simple enough to be manually verifiable.

1.3 Existing Practices

Traditional language-based techniques have been focused on the high-level code safety. Recent researches on Typed Assembly Languages (TAL) [Morrisett et al., 1998, 1999a,b], Proof-Carrying Code (PCC) [Necula and Lee, 1996; Necula, 1997], security types and information flow security [Sabelfeld and Myers, 2003], software fault isolation [Wahbe et al., 1993], virtual machines [Lindholm and Yellin, 1996;

Platt, 2001], typed intermediate languages [Tarditi et al., 1996; Shao, 1997; Shao and Appel, 1995; Chen et al., 2003], and certifying compilers [Colby et al., 2000; League et al., 2003; Chen et al., 2003] have generated exciting results on low-level code safety, demonstrating that *language-based security* is a promising technique for many security problems, such as buffer overflow and format string attacks, information leaks, etc., and for building trustworthy and high-assurance systems.

In the following, we review some of the existing techniques for ensuring the reliability and safety of running untrusted code. One of the important criteria we used to compare these different approaches is the size of TCB.

1.3.1 Authentication

Users may install and run untrusted programs based on authentication from some known and trusted party. A typical example is the dynamic software patch update system for Microsoft Windows, for example. Users download patches and install them after checking the authentication (electronically signed by Microsoft). Strictly speaking, authentication does not guarantee any property of the authenticated code. It only guarantees that the code is from some known party based on cryptography. Authentication does not reduce the TCB size since it does not guarantee any program property.

1.3.2 Virtual memory protection

Modern computer systems use virtual memory to protect a process from other processes by checking the memory boundary. Hardware and operating system are coordinated to make sure that application programs do not bypass the virtual mem-

ory API, which is usually implemented as OS system calls. Although the virtual memory is a very successful technique used in the modern operating systems, it is clumsy to implement and not flexible enough for situations other than memory safety.

1.3.3 Software fault isolation

Software Fault Isolation (SFI) [Wahbe et al., 1993] instruments machine code with additional runtime checking to ensure some safety property. It allows cooperating software modules to exist in the same address space and make sure that they don't trash each other by additional runtime checking on jump and store to ensure safety. Applications such as extensible kernels and databases can benefit from SFI because SFI provides an efficient way to run external programs safely via application isolation in the same address space without context switch overhead. One disadvantage of SFI is that it has some runtime overhead, and is not very flexible for ensuring properties other than memory safety.

1.3.4 Java bytecode verification

The Java Virtual Machine (JVM) [Lindholm and Yellin, 1996] provides additional safety check at the *bytecode* level via a mechanism called Java bytecode verification. In this framework, the bytecode, compiled from Java source code, is checked for safety before execution. Then users do not need to trust the Java compiler, which translates Java source programs into bytecode, since the safety of bytecode is verified. So the Java compilers are not in the TCB. However, in practice, bytecode is not interpreted due to inefficiency. Usually bytecode is compiled *just in time* to

machine code before execution. The Java Just-In-Time (JIT) compilers must be trusted for correctness and safety. Note that production-quality JIT compilers are usually large and complex.

1.3.5 Typed assembly languages

In the Typed Assembly Language (TAL) [Morrisett et al., 1998, 1999a,b] framework, a source program is compiled into a *typed* assembly program, which can be type checked. Since the assembly code is type checked, users do not need to trust the whole complicated (and maybe buggy) compiler anymore. Only the TAL type checker, assembler and linker are in the TCB, which is much smaller than traditional compilers. A weakness of the most existing TAL systems is that their soundness is not formally verified. In our foundational proof-carrying code project, we address this problem by designing a low-level typed assembly language with fully machine-checkable soundness proof.

1.3.6 Proof-carrying code

Proof-Carrying Code (PCC) [Necula and Lee, 1996; Necula, 1997] is a general framework for the mechanical verification of safety properties of machine-language programs. It allows a code producer to provide an executable program to a host (code consumer), along with a machine-checkable proof of safety such that the code consumer can check the proof before running the program. PCC has the advantage of small TCB, since the proof checking can be a simple mechanical procedure. A weakness of previous PCC systems is that the proof-checking infrastructure is too complex to prove sound using conventional techniques.

1.4 Foundational Proof-Carrying Code

Foundational Proof-Carrying Code (FPCC) [Appel, 2001] aims to further reduce the TCB size by an order of magnitude and to build the soundness proof based on the foundation of mathematical logic. There are three main components in a foundational proof-carrying code system: a compiler, a proof checker, and a safety proof of the machine-language program compiled from a source program. The compiler should produce machine code accompanied by a proof (hint) of safety. The proof checker verifies, sometimes also reconstructs, the safety proof before the program gets executed.

It is crucial to design appropriate interfaces between these components. This thesis is on how to design interfaces between type-preserving compilers, foundational proof checkers, and machine-checkable proofs, and on how to build an end-to-end FPCC system. We have come to the conclusion that: (1) a typed assembly language should serve as an interface between the proof-generating compiler and other components; (2) logic programming (in some restricted way) is a good mechanism for efficient proof-checking; and (3) an unified logical framework is convenient for representing proofs, as well as specifying the safety theorem and machine semantics. By using an unified logical framework, such as LF [Harper et al., 1993] and its implementation Twelf [Pfenning and Schürmann, 1999, 2002], we can manipulate proofs and specification in the same language.

Our prototype system is the first end-to-end FPCC system, including a type-preserving compiler from core ML to SPARC [Chen, 2004] (based on SML/NJ [Appel and MacQueen, 1987, 1991]), a low-level typed assembly language LTAL [Chen et al., 2003], a foundational proof-checker Flit [Appel et al., 2002; Wu et al., 2003],

and a nearly complete machine-checkable soundness proof [Tan et al., 2004]. In the following, we briefly explain some of the design choices and implementation of our system, as well as the connections and interfaces between the compiler, the proof checker, and the soundness proof.

1.4.1 Typed assembly language interface

Typed assembly languages provide a way to generate machine-checkable safety proofs for machine-language programs. But most existing typed assembly languages [Morrisett et al., 1999b,a] either don't have soundness proofs, or have proofs that are hand-written and cannot be machine-checked, which is worrisome for such large calculi. We have designed and implemented a low-level typed assembly language (LTAL) with a semantic model and established its soundness from the model. LTAL serves as the interface between the proof-generating (type-preserving) compiler and the proof-checking components. It is the language for the compiler to express proof hints of the safety of input user programs. Compared to existing typed assembly languages, LTAL is more scalable and more secure; it has no macro instructions that hinder low-level optimizations such as instruction scheduling; its type constructors are expressive enough to capture dataflow information, support the compiler's choice of data representations and permit typed position-independent code; and its type-checking algorithm is completely syntax-directed. We encode the LTAL type checker as a logic program that does not need to backtrack since the type checking is completely syntax-directed; this has important implication of efficient proof-checking that we will present later in Chapter 5. The details of LTAL is presented in Chapter 3.

1.4.2 Logic programming interface

The interface and mechanism for proof-checking are as important as proof-generating. In a naively designed FPCC system, the soundness proof could be 100 times larger than the machine program proved. Take the LTAL type system for example: It has about 1000 type checking rules, among which about 50 are type checking rules for instructions. And in average, each instruction type checking rule has about 10 premises, with about 20 implicit variable bindings. Assume that we need 10 application nodes and variable bindings to check each premise. Encoded in the Edinburgh Logical Framework (LF) [Harper et al., 1993], there are 120 application nodes per machine instruction. For each instruction, there are also about 20 application nodes for machine instruction decoding. If we encode each application node of LF with 3 words, it is 420 times of the size of the machine code. This is the size of type derivation tree, not including the size of the proof for each type checking rule. The size of the type checking rules is a constant, though it could be large; this size can be amortized away because the proof is checked once and for all.

In addition to the huge proof witness problem, untrustworthy proof rules are another problem in the previous PCC systems. In both Necula's PCC and Morrisett's TAL systems, type checking rules are trusted as axioms; the type systems used in their systems do not have a machine-checked soundness proof. Any misunderstanding of the semantics of type checking or proof rules could lead to errors in the type system. League et al. [2003] found an unsound proof rule in the SpecialJ [Colby et al., 2000] type system. In the process of refining our own TAL [Chen et al., 2003], we routinely find and fix bugs that can lead to unsoundness.

No previous design has addressed both of these problems simultaneously. We

show the theory, design, and implementation of a proof-checker that permits small proof witnesses and machine-checkable proofs of the soundness of the system. The general approach is to write a logic program that has a machine-checked semantic correctness proof. The logic program encodes the type checking rules of the type system (typed assembly language) for checking machine code. Fist, the correctness proof of the logic program is checked by a proof-checker; in our system, it is the LF proof-checker component of Flit. Then, the logic program is interpreted by a logic programming engine to check the input machine code. Our checker Flit has a simple logic programming engine that can efficiently interpret LTAL type checker as a logic program. This technique can be used in other domains (besides "proof-carrying") to write logic programs with machine-checked guarantees of correctness. The details of the efficient and foundational proof checking techniques is presented in Chapter 5.

1.4.3 Logical framework

To develop machine-checkable proofs, one must first choose a logic and a logical framework in which we encode and manipulate objects of the logic chosen. Our proof and specification of machine semantics and safety properties are based on higher-order logic. It is convenient to use the same representation for logics, theorems, and proofs. We choose the LF logical framework [Harper et al., 1993] to encode and manipulate higher-order logic.

LF is a dependently-typed λ-calculus with type families and $\beta\eta$-equality. It has three levels of terms: objects, types, and kinds. Types classify objects and kinds classify type families. LF provides a convenient tool for defining logics, with the support of higher-order abstract syntax. The framework is general enough to

represent logics of interest; we use it to encode higher-order logic. For development, we use Twelf [Pfenning and Schürmann, 1999, 2002], an implementation of LF. Twelf has many useful features, such as type reconstructing and mode analysis, which make it a convenient tool for us to develop machine-checkable proofs in LF.

While Twelf is very useful for developing proofs in LF, it is not minimal in terms of system size and features. Many advanced features, such as type inference and Emacs interface, are not needed for the proof checking at the user site, though these features are very useful for development. Thus if we use Twelf as the ultimate proof checker, it will violates the "pay as you go" principle and users have to trusted components that are not actually needed for the proof checking task. For efficient and trustworthy proof checking in LF, we have developed our own LF proof checker called Flit, which is presented in Chapter 5.

1.5 Thesis Outline

The remainder of the thesis is organized as follows. Chapter 2 gives an overview of our foundational proof-carrying code system. In Chapter 3, we introduce the LTAL interface, including its syntax, semantics, type checking rules, and measurements. In Chapter 4, LTAL is given a semantic model, and thus the machine-checkable soundness proof of the LTAL calculus is presented. In Chapter 5, we present the proof-checking mechanism; we use a simple system to illustrate the interfaces between different components including the proof checker. Finally, we summarize and give an outlook of future work in Chapter 6.

Chapter 2

Foundational Proof-Carrying Code

Everything should be made as simple as possible, but no simpler.

Albert Einstein

Necula's PCC system [Necula, 1997] constructs for untrusted code a verification condition (VC), which has the property that if VC holds with regard to the logic axioms and the typing rules, the program is safe. A VC generator (VCGen) is used by both the code producer and the code consumer to construct VCs. VCGen examines a machine-code program instruction by instruction and calculates the weakest preconditions for each instruction in Hoare-logic style. This VC-based verification builds the type system and machine instruction semantics into the algorithm for formulating the safety predicate. VCGen must be trusted to generate the right formula, but it is a large program (23,000 lines of C code [Appel and Wang, 2002]), thus difficult to guarantee bug-free.

2.1 FPCC

The motivation of Foundational PCC is to make the TCB as small as possible, without committing to any specific type system. We believe that the smaller the TCB, the more confidence PCC users can have. Our TCB consists of the specification of the safety policy, machine instruction semantics, and the proof checker. In the current implementation, it is about 3,000 lines of code [Appel et al., 2002; Wu et al., 2003], of which about half is the specification of the SPARC instruction set architecture. To make the TCB minimal, we choose Church's higher-order logic with a few axioms of arithmetic, give types a semantic model to move the type system out of the TCB, and model machine instructions by a step relation between machine states; we avoid VCGen entirely [Appel and Felty, 2000].

In order to support contravariant recursive datatypes and mutable fields, we model types as predicates on states, approximation indices [Appel and McAllester, 2001], and type levels [Ahmed et al., 2002]. We have an abstraction layer, Typed Machine Language (TML) [Swadi and Appel, 2001; Swadi, 2003], to hide the complex semantic models for types. TML provides a rich set of constructors for types, type maps, and instructions, and an orthogonal set of primitive type constructors such as union, intersection, existential and universal quantification, and so on. TML is so expressive that its type-checking is undecidable; it is more a logic than a type system. However, it is very useful for building semantic models of higher-level, application-specific type systems such as LTAL: We give LTAL constructors a semantic model in terms of TML.

The FPCC framework is shown in Figure 2.1. A source program is compiled into a machine-code program and an LTAL program. The code consumer receives the

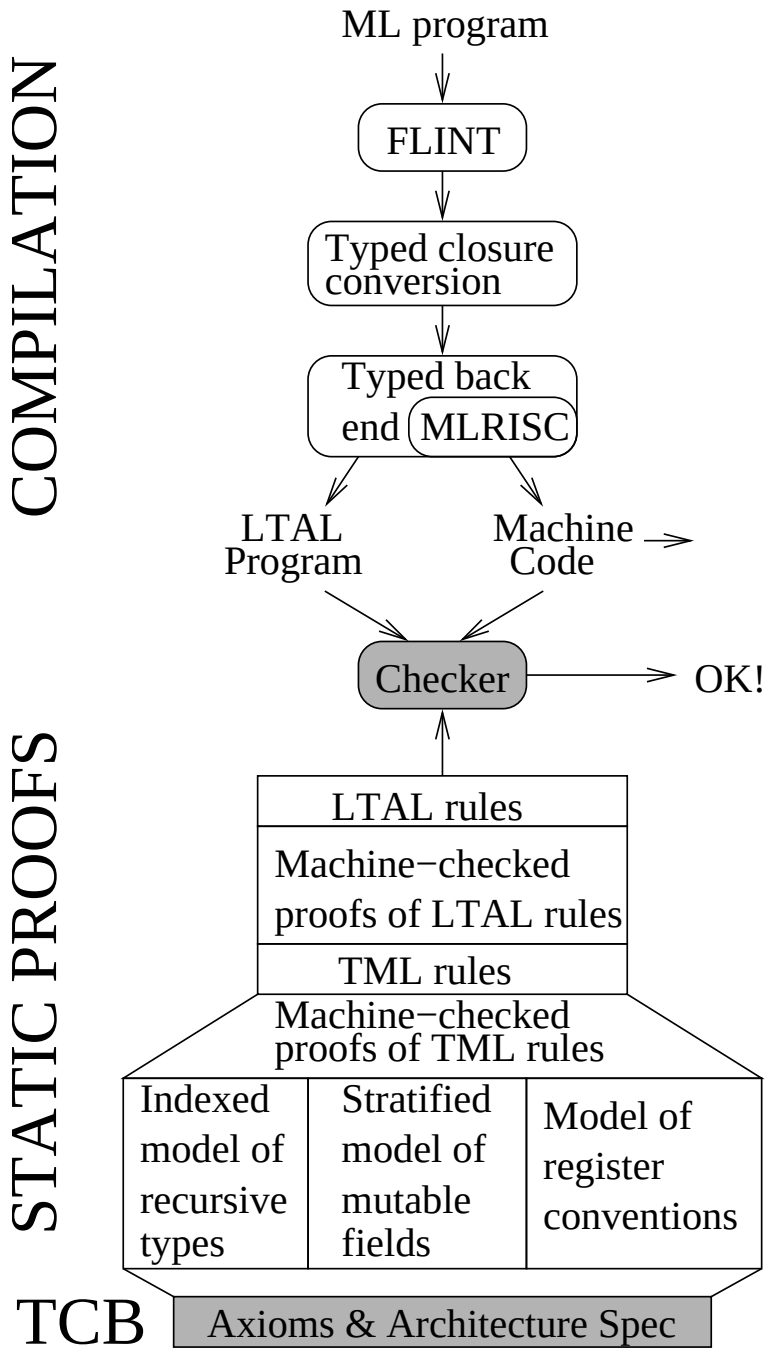


Figure 2.1: Foundational PCC framework. Trusted components are shaded.

LTAL rules, along with their soundness proof; checks the soundness proof [Appel et al., 2002; Wu et al., 2003]; and then runs the LTAL checker, which is a simple computation (like Prolog but without backtracking).

2.2 The LTAL Interface

The idea of Proof-Carrying Code [Necula, 1997] is that the compiler should produce machine code accompanied by a proof of safety. A weakness of previous PCC systems is that the proof-checking infrastructure is too complex to prove sound by conventional techniques. We have built the first compiler that produces machine code accompanied by safety proofs that are machine-checkable in a simple logic from minimal axioms.

Most PCC compilers, including ours, are based on typed intermediate languages or typed assembly language [Morrisett et al., 1998, 1999b], which provide a way to generate safety proofs automatically. TAL has a soundness guarantee: If a TAL program type-checks and there is no bug in the assembler, the machine code is safe to execute. Soundness is proved as a metatheorem outside of the proving system; the proof is hand-written and not machine-checkable. The typing rules and the type checker are in the trusted computing base, that is, bugs in these components can let unsafe code slip past the checker. There have been many variants of TAL [Morrisett et al., 1999a; Xi and Harper, 2001; Morrisett et al., 2002], which rely on similar soundness metatheorems. A recent variant TALT [Crary, 2003] has a machine-checkable metatheorem, which moves the typing rules and the type checker out of TCB. The metatheorem proof checker, such as Twelf [Pfenning and Schürmann, 1999, 2002] used by TALT, is usually a quite big program and has to be trusted.

It is hard to manage the soundness proofs and avoid errors when scaling up to realistic type systems for real compilers. The goal of our Foundational Proof-Carrying Code (FPCC) [Appel, 2001] project is to build machine-checkable safety proofs for machine-code programs from the minimal set of axioms. We have designed a low-level typed assembly language (LTAL) to be the interface between the compiler and the checker: The compiler compiles a source program to machine code annotated by an LTAL program. LTAL annotations are not in the machine code, so they don't increase machine code size or execution time.

The soundness of LTAL typing rules is proved not by a metatheorem as in TAL, but by their semantic model [Tan et al., 2004], bottom up: First we use higher-order logic with axioms for arithmetic to prove lemmas about machine instructions and types, then we prove the TML typing rules based on these lemmas, then we prove the soundness of LTAL typing rules in the TML model. Each typing rule is represented as a derived lemma in our logic.

LTAL benefits from its semantic model in many aspects: First, it is more scalable. Adding new rules that can be described in our semantic model generally does not affect the soundness of existing rules, which we found very useful in evolving the design. Second, it is more secure because the typing rules are moved out of the TCB. Third, TML connects LTAL to real machine instruction semantics, thus bridges the gap between typed assembly language and machine language.

LTAL is not intended as a universal TAL. Instead, it is extensible. Our semantic modeling technique is very modular. New operators can be added to LTAL (and proved sound) without disturbing the soundness proofs for existing operators, as long as the new operators conform to the assumptions in the semantic model. We started with a very simple model [Appel and Felty, 2000], and when we added

contravariant recursive types [Appel and McAllester, 2001] and mutable record fields [Ahmed et al., 2002] these changes did violate previous assumptions and require nonmodular rewrites. But now our model is very powerful and general: None of the existing LTAL soundness proofs will need to be touched when we add operators to handle extensible sums, various kinds of exception handling mechanisms, various kinds of multidimensional arrays (with or without pointer indirections), or arbitrary predicates on scalar values.

2.3 FPCC/ML Compiler

The FPCC/ML compiler, built by Chen and Fang [Chen et al., 2003], transforms core ML (ML without the module system) into SPARC code with LTAL annotations. At present our prototype omits exceptions and strings. The compiler is based on the Standard ML of New Jersey (SML/NJ) system [Appel and MacQueen, 1987, 1991].

There are several stages: the front end of SML/NJ translates source ML programs to FLINT (a typed intermediate language based on F_ω) [Shao, 1997]; we have reused the FLINT front end. The newly built typed CPS-conversion and closure conversion phases, built by Hai Fang, generate NFLINT (a typed intermediate language like Morrisett's λ_C [Morrisett et al., 1998, 1999b]). The next few phases, built by Juan Chen, break down complex instructions, build basic blocks, and insert coercions to get machine-independent LTAL programs. The back end, also by Juan Chen, takes machine-independent LTAL, and produces machine code with machine-specific LTAL annotations and some auxiliary information, such as mapping from labels to their addresses.

SML/NJ's back end uses the untyped MLRISC retargetable instruction selection, register allocation, and low-level optimization software [George, 1997]. The difficulty is to make MLRISC preserve and manipulate type information, without rewriting the MLRISC or making it dependent on our particular type system. Fortunately, MLRISC already had some support for an annotation mechanism [Leung and George] that permits "comments" on the instructions; we have generalized this mechanism and used it to propagate types.

2.4 Checker

Our checker has two main components. First, it uses a simple LF type-checker to check a proof, in higher-order logic, of the soundness of the LTAL typing rules [Appel et al., 2002; Wu et al., 2003]. We can view these LTAL rules as a set of lemmas.

On the other hand, the LTAL rules can be regarded as a set of Prolog-like clauses. Then, because these rules are syntax-directed, the checker can run a very simple subset Prolog interpreter (without backtracking) on these rules to type-check the machine-language program [Wu et al., 2003].

The LTAL program is only an untrusted hint so that the checker can take advantage of type and dataflow information from the compiler in proving the safety of the machine code. The process of running the checker on a machine code and the corresponding LTAL program is like type-checking the machine code according to the structural information from the LTAL program. The overall goal of the checker is judge_prog(H, P) where P is the binary code (a sequence of instruction words) and H is the corresponding LTAL program. The predicate judge_prog characterizes

well-typedness. The checker solves this goal according to the structure of H. In the underlying semantic model, we can prove that well-typedness implies safety:

$$\text{judge\_prog}(H, P) \rightarrow \text{safe}(P).$$

The predicate safe is the machine-level safety policy. When the checker succeeds on the goal judge_prog(H, P), we apply this lemma to get a proof of safe(P).

2.5 Safety Proof

The safety proof for input machine code consists of two parts: the static proof of the LTAL type checking rules and the syntactic type derivation. We have built an semantic model for the LTAL type system based on mathematical logic and the machine semantics, and proved each LTAL type checking rule as a lemma in our system. This static proof of the soundness of LTAL is checked once and for all. The syntactic LTAL type checking rules are interpreted by the checker to verify that a type derivation exists, that is, the input program is typeable in LTAL. The checker does not actually build the *huge* proof, *i.e.* the type derivation; it just makes sure that there exists one.

We encode the LTAL type checking rules in LF, and prove them as lemmas. After checking the validity of the proofs of lemmas, we strip off the proofs and interpret the set of rules as a logic program in the Flit checker. If the logic program terminates with a positive result, a type derivation exists. That is, there exists a machine-checkable safety proof of the input program, although we actually didn't fully build it.

Chapter 3

Low-Level Typed Assembly Language

We have designed our own typed assembly language LTAL because we want to generate safety proofs of machine code, with as much flexibility as possible for an optimizing compiler. Thus, even part-way through a sequence of instructions that allocates on the heap or that does datatype-tag discrimination, the LTAL type system must be able to describe the machine state. That is, LTAL has no "macro" instructions: Each LTAL instruction corresponds to one SPARC instruction or is a coercion with no runtime effect. Because no sequence of instructions is unbreakable, low-level optimizations such as instruction scheduling are permissible (however, at present our LTAL does not accommodate the filling of branch-delay slots on SPARC). Macro instructions in other TALs (such as *malloc* and *test-and-branch*) that expand to a fixed sequence of machine instructions, interfere with low-level optimization.

TAL Systems	1	2	3	4	5	6	7	8	9	10	11	12
SpecialJ [Colby et al., 2000]	●	●				○		●				
TALx86 [Morrisett et al., 1999b]	⊙	●				○	●	○		○	○	
DTAL [Xi and Harper, 2001]								●				
FTAL [Hamid et al., 2002]			⊙	●							●	
TALT [Crary, 2003]		●	⊙	●			●	●	○			
Open Verifier [Chang et al., 2005]	⊙	●			○	○	●	●				●
Our LTAL [Chen et al., 2003]	⊙	●	●	⊙	●	⊙	●	●	●	●	●	⊙

TAL Features:
1 Compiles "real" source language
2 Compiles to real target machine
3 Foundational specification
4 Machine-checked soundness proof
5 Minimal checker
6 Atomicity
7 Compiler can choose data representations
8 Dataflow analysis
9 Position-independent code
10 Basic blocks
11 Syntax-directed checking
12 Flexibility

Keys:
○ partially
⊙ nearly
● completely

Figure 3.1: Comparison of TAL and PCC systems. (The table is based on Chen et al. [2003]. The status of TALT, the last column, and the entry for the Open Verifier are new.)

3.1 LTAL Features

Our design and implementation has the following desirable properties, some of which are shared by some other TAL and PCC systems (see Figure 3.1):

- **Compiles a "real" source language.** We have built a compiler for almost all of core ML—a full-scale source language with polymorphic higher-order functions, disjoint-sum recursive datatypes, and so on.

- **Compiles to a real target machine.** We generate high-quality SPARC

code. Our type-preserving compiler is based on the SML/NJ system [Appel and MacQueen, 1987, 1991].

- **Foundational specification.** We have a concise logical specification, independent of any type system, of the safety property guaranteed by our system: In our prototype we guarantee memory safety and that only a certain subset of SPARC instructions will be executed [Appel, 2001]. Furthermore, our specification relates to the actual *machine language* to be executed—not assembly language—we model and check instruction encodings explicitly.

- **Machine-checked proof.** We have a machine-checked proof (mostly finished) of the soundness of our system—that is, if the LTAL type-checks, the machine code is safe. Unlike any other TAL or PCC system, our proof is with respect to a minimal set of axioms, the largest part of which is a logical specification of the instruction set architecture of the SPARC processor.

- **Minimal checker.** Just in case you are worried about bugs (or Trojan horses) in proof checkers, our soundness proof is checkable in a very minimal logic: The trusted base of our system (including axioms, machine specification, and a C program implementing the LF checking and a simple logic programming engine) is about 3034 lines of code [Appel et al., 2002; Wu et al., 2003], an order of magnitude smaller than other systems.

- **Atomicity.** Some other TALs have "macro" instruction sequences (or even worse, calls to the runtime system) for compare-and-branch, or datatype tag-checking, or memory allocation. This inhibits optimizations such as hoisting and scheduling.[1] Each of our LTAL instructions corresponds to at most one

[1]These optimizations can be done in the assembler, but need to be trusted bug-free, whereas

machine instruction. Some LTAL instructions are only for type coercion, and do not correspond to any real machine instructions. Because no sequence of instructions is unbreakable, low-level optimizations such as instruction scheduling are permissible.

- **Compiler can choose data representations.** For data structures such as tagged disjoint sums, a compiler may want to exercise discretion in choosing data layouts, unhampered by assumptions built into a typed assembly language. LTAL permits this flexibility; some other TALs do not.

- **Dataflow & induction analysis.** LTAL includes existential and singleton types that are powerful enough to permit dataflow-based safety proofs of optimized machine code (though our prototype compiler does not exploit all of this power yet).

- **Position-independent code.** To avoid the need to trust a linker, we show how to check typed position-independent code—even in the presence of long jumps and of operations that move code addresses into pointer variables and closures.

- **Basic blocks.** LTAL groups instructions into basic blocks, making it easy for an optimizing compiler to reorder blocks to optimize cache placement or shorten span-dependent instructions.

- **Syntax-directed.** Typechecking LTAL is completely syntax-directed. As we will describe later, the LTAL type checker can be encoded as a simple logic

our system does not need to trust them.

κ	::=	Numeric | Scalar | Ω	**Kinds**
τ	::=	(See Figure 3.4)	**Types**
cc	::=	$\mathsf{cc\_cmp}(\tau_1, \tau_2)$ | $\mathsf{cc\_testbox}(\tau)$	**Condition Codes**
	|	$\mathsf{cc\_testmem}(m)$ | $\mathsf{cc\_none}$	
v	::=	$x \mid i \mid l \mid c(v) \mid \mathit{vdiff}\,(l_1, l_2)$	**Values**
c	::=	(See Figure 3.7)	**Coercions**
op	::=	$+ \mid - \mid * \mid /$	**Arithmetic Operators**
π	::=	$= \mid \neq \mid > \mid \geq \mid < \mid \leq$	**Arithmetic Compares**
ι	::=	(See Figure 3.16)	**Instructions**
B	::=	$l[\vec{\alpha} : \vec{\kappa}](m, cc, v_1 : \tau_1, \ldots, v_n : \tau_n) = \iota_1; \ldots; \iota_k$	**Basic blocks**
LRT	::=	(L, R, T)	**Environments**
L	::=	$\{l_1 \mapsto a_1, \ldots, l_n \mapsto a_n\}$	*label map*
R	::=	$\{x_1 \mapsto r_1, \ldots, x_n \mapsto r_n\}$	*register map*
T	::=	$\{\mathscr{D}_1 \mapsto \tau_1, \ldots, \mathscr{D}_n \mapsto \tau_n\}$	*type abbreviation map*
P	::=	$(LRT, \vec{B})$	**Program**

Figure 3.2: LTAL syntax: Overview.

program that does not need to backtrack, which has important implication for efficient proof checking.

- **Flexibility.** Our framework is very flexible. Many of the LTAL features are orthogonal, which makes LTAL easily extensible. We believe LTAL can be extended to compile other source languages on different architectures without much difficulty.

3.2 Syntax Overview

LTAL is a calculus with conventional features such as variable names and scoping rules. This is unlike other TALs, which use registers and memory locations directly instead of variables. The LTAL syntax is shown in Figure 3.2, 3.4, 3.7, and 3.16.

LTAL supports first-order kinds; it has only limited support for higher-order

kinds, since TML does not model higher-order kinds in full generality. For core ML, this is enough. The kind *Numeric* classifies singleton numeric types. The kind *Scalar* classifies types that are scalar under the TML semantic model. Most types presented in Section 3.4 are of kind *Scalar*. However, in our semantic model, it is convenient to describe the register bank or typing environment ϕ as a type. The register bank type and environment types are not scalar; the scalar types only care about the first element of the vector in the semantic model. The kind *Numeric* is a sub-kind of *Scalar*. All other types are of kind Ω.

LTAL has a set of standard types: type variables,[2] top and bottom types, integer types, existential types, and recursive types (See Figure 3.4 for the detailed description of the LTAL types). There are low-level constructors to model high-level abstractions, such as singleton integer type $\overline{n}$ and refined integer type $\mathsf{int}_\pi(\overline{n})$ for integers (i has type $\mathsf{int}_\pi(\overline{n})$ means $i\,\pi\,n$ is true, where π is a predicate on integers such as $=$ or $\leq$), field types, intersection types and union types for records and user-defined datatypes.

To model basic blocks (with their live variables) and functions (with their formal parameters) we have polymorphic "code pointer" types $\mathsf{codeptr}[\vec{\alpha} : \vec{\kappa}](m, cc, v_1 : \tau_1, \ldots, v_n : \tau_n)$, where $\vec{\alpha} : \vec{\kappa}$ is a list of type variables with kinds, m is the available memory size known at this point, cc is the condition code requirement, and $v_i : \tau_i$ are the input arguments and types.

For label arithmetic and position-independent code type checking, we have type constructors addr and diff, which will be further explained in Section 3.9 and 3.10.

Type def refers to a type expression by a name $\mathscr{D}$; in our implementation, names

[2]In our implementation we use de Bruijn indices, but for presentation purpose, we sometimes show named variables.

are just integers. Each program can have a sequence of type abbreviations that give names to type expressions. This mechanism makes LTAL programs concise, and saves the checker some work. The checker expands a name to the type expression it stands for only when such expansion is needed. Otherwise, the checker simply passes the name around, which is more efficient than passing the type expression. The body of a type definition could be an open type expression with free variables. Type variables cannot be used for this purpose since they usually stand for closed type expressions.

We have a special category *cc* to capture the condition code status (on machines with condition codes), which includes cc_cmp for comparison, cc_testbox for testing whether or not a type is boxed, cc_testmem for memory availability testing, and cc_none for arbitrary status.

A value can be a variable x, an immediate integer i, a label l, a coerced value $c(v)$ (where c is a coercion), or a *vdiff* value. Values and their typing are further explained in Section 3.5.

Coercions are used to change the type of values; all coercions are free of runtime effect, as they follow subtyping relations in the underlying model. Many of these coercions are conventional, such as identity, composition, pack, fold/unfold, inject, and project. Coercion rules are further discussed in Section 3.6.

LTAL has a machine-independent core, which includes: move and ALU instructions, *sethi* for loading large integers, store and load instructions, *addradd* for address arithmetic, *select* for loading a record field, *gettag* for loading the tag field of a sum type value, *init*, *record*, and *inc_allocptr* for heap allocation, *call* for jumping to some label, and *calln* for "call by fall-through," (which generates no code). Each target machine requires the addition of machine-specific operators and rules. The

instructions in $\text{LTAL}_{\text{SPARC}}$ that are specific to machines with condition codes are: *cmpcc* compares two integers and sets condition codes; *cmpcci* compares a value with a compile-time-known integer, sets condition codes and refines the type of the value; *testbox* tests if a value is boxed; *testmem* tests for out-of-heap; *if* is normal conditional branch without type refinement; *ifr* is conditional branch with type refinement in both branches, *ifboxed* refines types for boxedness of the value (of a sum type), and *iffull* and *iftag* specialize type refinement for memory allocation and datatype tag discrimination, respectively. The LTAL instructions and their typing rules will be further discussed in Section 3.9.

Function declaration $l[\vec{\alpha} : \vec{\kappa}](m, cc, v_1 : \tau_1, \ldots, v_n : \tau_n) = \iota_1; \ldots; \iota_k$ defines a function (basic block) with label l, type parameters $\vec{\alpha} : \vec{\kappa}$, formal parameters $v_1 : \tau_1, \ldots, v_n : \tau_n$, and function body $\iota_1 \ldots \iota_k$ which is a sequence of LTAL instructions. The number m specifies how much memory is guaranteed to be available when the function is called. It is a compile-time known constant. If a function specifies 16 words and allocates no more than 16 words, for example, there is no need to test the memory availability. Otherwise, it has to check explicitly if there is enough memory. The condition-code requirement cc specifies the status of condition codes when the function is called. The function label l is assigned a code pointer type $\mathsf{codeptr}[\vec{\alpha} : \vec{\kappa}](m, cc, v_1 : \tau_1, \ldots, v_n : \tau_n)$. Each function is closed in the sense that there are no free type variables or value variables.

The triple LRT represents three environments that keep auxiliary information for type checking. The label environment L is a map from program labels to their addresses (offset from the beginning of the program). The register-allocation environment R maps variables to temporaries (registers or spill locations).[3] The type

[3]To model the fact that a value in a register can belong to two different types at once (if their

abbreviation environment T maps type abbreviations to their expansions. Type abbreviations are used to gain concise type expressions and the type checker opens a type abbreviation only when needed.

An LTAL program consists of the above environments and a list of basic blocks, which can be viewed as a set of function declarations.

3.3 Static Semantics Overview

The low-level type and term constructors in LTAL make the typing system expressive. Yet we need a decidable and simple type-checking algorithm so that proof generation can be done without a complicated decision procedure or constraint solver. To this end, we have made LTAL completely syntax-directed. There are no subtyping rules; instead, we use coercions to avoid nondeterministic choices during type checking. We explain various typing judgements, and then show some typing rules in this section.

The typing judgement for values $LRT; \rho; \phi \vdash v : \tau$ means value v has type τ under environment $LRT; \rho; \phi$. The triple LRT is part of the program. The kind environment ρ is a list of kinds for type variables bound so far. In our implementation we use de Bruijn numbers to represent type variables; the ith (starting from 0) element of the kind list ρ is the kind for the type variable of de Bruijn index i. The value environment ϕ maps variables to their types.

The judgement $LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{\iota\}\ (\rho'; \mathscr{H}'; \phi'; cc')$ means after instruction ι is executed, environment $(\rho; \mathscr{H}; \phi; cc)$ becomes $(\rho'; \mathscr{H}'; \phi'; cc')$. The construction

intersection is nonempty), we choose not to use intersection types. Instead, we say that each *variable* can have only one type (globally), and in the register-allocation environment more than one variable at a time can mapped to the same register.

$\phi, v : \tau$ augments ϕ. The construction $(\phi \backslash v), v : \tau$ kills dead bindings before adding the new binding $v : \tau$; it keeps the alive bindings unchanged. To be more specific, for a binding $u : \tau_u$ in ϕ, if variables u and v are assigned to the same register and u is not alive after the execution of instruction ι, the binding $u : \tau_u$ is killed; otherwise, it remains the same. When there is no ambiguity, we use $\phi, v : \tau$ for both purposes. The heap-allocation environment $\mathscr{H}$ is explained in Section 3.8. The environment cc specifies the current status of condition codes.

As an example we will show a simplified rule for an LTAL *add* instruction. In Section 3.10 we will show a different typed version of *add*. These two different typed versions of *add* expand to the same SPARC machine instruction. The first rule we show here is useful for compiling a source-language *add* for which no dataflow tracking is needed to prove safety; the second is useful for compiling address arithmetic. Having multiple LTAL instructions for the same machine instruction simplifies typechecking.

$$\frac{LRT; \rho; \phi \vdash x : \mathsf{int} \quad LRT; \rho; \phi \vdash y : \mathsf{int}}{LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{z = x + y\}\ (\rho; \mathscr{H}; (\phi \backslash z), z : \mathsf{int}; cc)}$$

In fact, this rule is dramatically simplified for clarity. The full version, which is shown in Figure 3.3, has ten premises and one complicated conclusion.

The first and second premises state that both x and y have type int_{32}, the 32-bit integer type. The environment LRT is label, register allocation, and type abbreviation maps. Address ℓ is the location of current instruction $z = x + y$; ℓ' is the location of the next instruction. Premise (3) specifies that the length of the *add* instruction is 4 bytes.

Premises (4) and (5) relate variables z and x to their temporary numbers, and

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

Figure 3.3: A sophisticated LTAL type checking rule.

premises (6) and (7) map temporaries to registers; this rule would not be applicable to operands represented in spill locations (but of course that's true of the actual SPARC *add* instruction too). There are about 1000 temporaries (after register allocation); the first 64 are registers (including 32 floating-point registers), and the remainder are in the spill area. The per-program *rmap*—the R component of LRT—maps variables to temporaries; the program-independent relations *realreg* and *memtemp* relate temporaries to their machine representation.

Since value y can be either a register or an immediate, we use *match_reg_or_imm* in premise (8) to match either a register or an immediate. So y_m can be either (rmode r_y) for some register r_y or (imode i) for some immediate i. Premise (8) matches a particular SPARC addressing mode.

Premise (9) states the relation between the value typing context before and after execution of the current instruction. Before we add the type of variable z into the context, all aliases of z should be killed since they are not live anymore, which is what $\phi \backslash z$ does. We use intersection type to extend the old typing environment ϕ with the new binding $z : \mathsf{int_{32}}$. The ϕ context is small (it just maps currently live

local variables) and is represented as a list, not with dynamic atomic clauses as will be described in Section 5.3.1.

The *decode_list* relation in premise (10) maps an instruction encoding (i.e., an integer) to its semantics. Specifically, it says that the instruction word at the beginning of machine code P with length $\ell' - \ell$ is an add instruction i_ADD(r_x, y_m, r_v). Machine code P is a sequence of integers (instruction words); the pair (P, P') is a conventional Prolog difference list [Sethi, 1989, §8.4]. Premise (10) will also be explained in the next subsection.

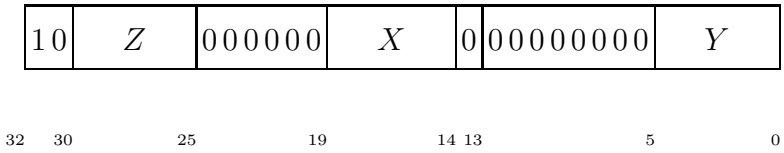
The conclusion is a Hoare-logic style judgement. Under environment LRT, the instruction $z = x + y$ is at location ℓ; the length of the instruction is $\ell' - \ell$; this instruction does not affect type contexts ρ or heap allocation environment $\mathscr{H}$; value context ϕ becomes ϕ' after execution; the machine code at location ℓ' is P'.

For a real-life program, the generated maps L, R, and T can be very large: The sizes of L and R are approximately linear in the size of the program, and we intend to be able to type-check programs with millions of instructions. In this typing rule, premises (4) and (5) look up the temporaries of variables v and x in map R; premise (8) looks up the temporary of y if it is not an immediate. Therefore, an efficient environment management scheme is necessary; in our implementation, we use dynamic clauses to efficiently maintain various environments. We will further discuss this issue in Section 5.3.

3.3.1 Instruction decoding

The *decode_list* relation in the premise (10) of the sophisticated rule shown in Figure 3.3 maps an instruction word to a higher-level instruction with semantic meaning. Specifically, it says that the instruction word at the beginning of the machine

code P with length $\ell' - \ell$ is an add instruction i_ADD(r_x, y_m, r_z). We check for proper instruction encoding with rules such as the following:



$$\frac{\begin{array}{ll} 32 \cdot 2 + Z = X_9 & 64 \cdot X_9 + 0 = X_7 \\ 32 \cdot X_7 + X = X_6 & 2 \cdot X_6 + 0 = X_4 \\ 256 \cdot X_4 + 0 = X_1 & 32 \cdot X_1 + Y = W \end{array}}{\text{decode}(\text{i\_ADD}(X, \text{rmode}(Y), Z),\ W)}$$

This rule is not an axiom of our system, it is a lemma derived from a more concise and readable definition of instruction encodings [Michael and Appel, 2000]. The predicate $A \cdot B + C = D$ shown here is a simplification of an actual predicate that also checks that $C < A$ and that A, B, C, D are natural numbers.

3.4 Types

The LTAL type system is very expressive, with support for many advanced features such as position-independent code, type definitions, singleton types, and polymorphic function types. The LTAL types are shown in Figure 3.4, and we give a brief introduction below:

Type variables: We use named variables for presentation purposes; however, in our actual implementation, we use de Bruijn indices such as $\underline{0}$. The de Bruijn indices enable convenient manipulation of (open) type terms.

Types

τ	::=	α	type variables
	|	$\top$	top type
	|	$\bot$	bottom type
	|	int	integer type
	|	$\overline{n}$	singleton integer type
	|	$\mathsf{int}_\pi(\tau)$	refined integer type
	|	$\mathsf{range}(\tau_1, \tau_2)$	range type
	|	$\mathsf{def}(\mathscr{D})$	type definition
	|	$\tau_1 \cap \tau_2$	intersection type
	|	$\tau_1 \cup \tau_2$	union type
	|	$\mathsf{array}(\tau_1, \tau_2)$	array type
	|	$\mathsf{codeptr}[\vec{\alpha} : \vec{\kappa}](m, cc, v_1 : \tau_1, \ldots, v_n : \tau_n)$	polymorphic code pointer type
	|	$\mathsf{offset}(\tau_1, \tau_2)$	offset type
	|	$\mathsf{field}(i, \tau)$	field type
	|	$\mathsf{sum}(\tau_1, \tau_2)$	sum type
	|	$\mathsf{hastag}(\tau_1, \tau_2)$	hastag type
	|	$\exists\alpha.\tau$	existential type
	|	$\mu\alpha.\tau$	recursive type
	|	box	immutable reference type
	|	ref	mutable reference type
	|	$\mathsf{addr}(l)$	label type
	|	$\mathsf{diff}(l_1, l_2)$	label-difference type

Figure 3.4: LTAL syntax: Types.

Top type: Any term can have the top type $\top$.

Bottom type: No term can have the bottom type $\bot$. It is useful, however, in some situations. For example, we can use $\mathsf{sum}(\tau, \bot)$ to represent a data type with no boxed cases.

Integer types: The integer types are bounded. A term of int_{32} type has a 32-bit integer value.

Singleton integer types: The singleton integer type is written as $\overline{n}$. A term of

type $\overline{n}$ has an integer value n. Singleton integer types are used in many places such as dataflow analysis in our system.

Refined integer types: LTAL has a set of refined integer types $\mathsf{int}_\pi(\tau)$ to express relational constraints, where π is a binary operator. For example, $\mathsf{int}_>(\overline{9})$ is a refined type and it classifies integers that are greater than 9.

Range types: Range type is a syntactic sugar. Range type $\mathsf{range}(\tau_1, \tau_2)$ is an abbreviation for $\mathsf{int}_\geq(\tau_1) \cap \mathsf{int}_<(\tau_1)$.

Type definitions: Type definition def is used for concise type representation and efficient type checking. The type checker opens a type definition only when it is necessary to do so.

Intersection types: A term v has an intersection type $\tau_1 \cap \tau_2$ means v has both type τ_1 and type τ_2.

Union types: A term v has a union type $\tau_1 \cup \tau_2$ means that v has either type τ_1 or type τ_2.

Array types: Type $\mathsf{array}(\tau_1, \tau_2)$ describes an array whose size is of type τ_1 and whose elements are of type τ_2. Type τ_1 is expected to be a singleton integer type such as $\overline{100}$.

Polymorphic code pointer types: The type $\mathsf{codeptr}[\vec{\alpha} : \vec{\kappa}](m, cc, v_1 : \tau_1, \ldots, v_n : \tau_n)$ is for polymorphic code pointers. It takes a list of type variables $\vec{\alpha} : \vec{\kappa}$ (with their kinds), a type m describing the available memory slots without further availability testing, the required condition code type, and a list of arguments and their types.

Offset types: A term v has type $\mathsf{offset}(i, \tau)$ means that value $v + i$ has type τ. Offset types are used in address arithmetic.

Field types: Field types are used for constructing types of records. A record type is the intersection of field types describing the record fields.

Sum types: Sum types are used to describe (user-defined) data types. Sum types have the form $\mathsf{sum}(\tau_r, \tau_u)$, where τ_r is a range type and τ_u is a union type of record types. That is, the τ_r cases of the data type described by the sum type are not boxed, and the τ_u cases are boxed. See Section 3.8 for detailed descriptions of the use of sum types.

Hastag types: The type $\mathsf{hastag}(\tau_1, \tau_2)$ refines a sum type by requiring that the value has a tag at the first field of the record.

Existential types: The existential type $\exists\alpha.\tau$ is useful for data abstraction and information hiding. In LTAL, we use it for typing closures, tagged sum values, position-independent code, etc.

Recursive types: Recursive types $\mu\alpha.\tau$ model inductively defined recursive types.

Pointer types: Type boxed is for pointer values.

Immutable reference types: The immutable reference type $\mathsf{box}(\tau)$ describes a pointer that points to some memory slot whose content is of type τ. The memory slot to which the pointer points is not allowed to be written after initialization.

Mutable reference types: The ref type is the mutable version of the above reference type box.

Values

v	::=	x	variables
	|	i	known integer value
	|	l	label value
	|	$c(v)$	coercion value
	|	$\mathit{vdiff}(l_1, l_2)$	label-difference value

Figure 3.5: LTAL syntax: Values.

Label types: Label type $\mathsf{addr}(l)$ describes a label. The value of a label is an address at which some code pointer resides.

Label-difference types: The type $\mathsf{diff}(l_1, l_2)$ describes a (compile-time known) value $l_1 - l_2$. This type is used to check address arithmetic and position-independent code which we will discuss in Section 3.9 and 3.10.

3.5 Values

Values are shown in Figure 3.5. A value can be a variable x, an immediate integer i, a label l, a coerced value $c(v)$ (where c is a coercion), or a *vdiff* value. We use variables to track aliases of registers. Different variables with different types can be assigned the same register, indicating different views of the same register to the type-checker. The value constructor *vdiff* and type constructors addr and diff are used for address arithmetic and typed position-independent code. Their meanings are explained in Section 3.9 and 3.10.

Unlike λ-calculus, function values are not classified as values in LTAL. LTAL is a typed calculus for low-level code, which has labels and code blocks. A label value in LTAL stands for the address at which a code block resides.

The typing rules for values are quite straightforward, as shown in Figure 3.6.

$$\frac{\phi(x) = \tau}{LRT; \rho; \phi \vdash x : \tau} \; \textit{ValVar}$$

$$\frac{}{LRT; \rho; \phi \vdash i : \mathsf{int}_{=}(\bar{i})} \; \textit{ValConstant}$$

$$\frac{l[\vec{\alpha} : \vec{\kappa}](m, cc, v_1 : \tau_1, \ldots, v_n : \tau_n) = \ldots \text{ is a code block}}{LRT; \rho; \phi \vdash l : \mathsf{codeptr}[\vec{\alpha} : \vec{\kappa}](m, cc, v_1 : \tau_1, \ldots, v_n : \tau_n)} \; \textit{ValLab}$$

$$\frac{}{LRT; \rho; \phi \vdash \textit{vdiff}\,(l_1, l_2) : \mathsf{diff}(l_1, l_2)} \; \textit{ValDiff}$$

$$\frac{LRT; \rho; \phi \vdash v : \tau_v \quad \rho; LRT \vdash_c \tau_v \overset{c}{\hookrightarrow} \tau}{LRT; \rho; \phi \vdash c(v) : \tau} \; \textit{ValCoerce}$$

Figure 3.6: Value typing rules.

What is worth mentioning is *ValLab*, the value typing rule of a label value. We put in the premise informally that label l is declared as the label of some code block. In our actual implementation, we use local assumptions (or dynamic clauses in Prolog) to check that label l is declared. The use of dynamic clauses enables efficient and concise proof checking; we will explain this in detail in Section 5.3.

3.6 Coercions

A coercion only changes the static type of a value; it has no runtime effect. A coercion c defines a type transformation function f_c. If c is applied to value v of type τ, we get another value $c(v)$ of type $f_c(\tau)$. Type τ and $f_c(\tau)$ should be compatible; more accurately, it should be provable that τ is a subtype of $f_c(\tau)$. Coercions simplify type-checking by telling the checker, in effect, where to apply subtyping. However, this can significantly increase the size of the LTAL code.

Coercions

c	::=	cid	identity coercion
	|	$\mathrm{c}_1 \circ \mathrm{c}_2$	composition coercion
	|	cfold[τ]	fold
	|	cunfold	unfold
	|	cpack(τ_1, τ_2)	pack
	|	cinjt[$\mathsf{sum}(\tau_r, \tau_u)$]	Injection
	|	csum2range	sum to range
	|	csum2boxedone	sum to boxedone
	|	csum2hastag	sum to hastag
	|	cunhastag	hastag elimination
	|	c2int32	int to int_{32}
	|	ci2nz	singleton refinement
	|	crange[n_1, n_2]	singleton to range
	|	cinj1 (τ)	left injection
	|	cinj2 (τ)	right injection
	|	cproj1	left projection
	|	cproj2	right projection
	|	cdef $\mathscr{D}$	definition introduction
	|	cname	definition expansion
	|	cinters(c_1, c_2)	intersection
	|	cunion(c_1, c_2)	union
	|	c2inters	simultaneous coercion
	|	cfield c	field
	|	caddr2code	label to code pointer
	|	coffset0	offset 0 introduction
	|	coffset0elim	offset 0 elimination
	|	cptapp[τ]	partial instantiation of polymorphic functions

Figure 3.7: LTAL syntax: Coercions.

LTAL are shown in Figure 3.7. We briefly explain them below:

Identity: The identity coercion cid coerces a type to itself.

Composition: The composition coercions $\mathrm{c}_1 \circ \mathrm{c}_2$ applies c_2 first, and then applies c_1 to that result.

Fold: The coercion cfold[τ] transforms a type τ_0 into a recursive type τ. The recursive type τ is of the form $\mu\alpha.\tau_1$, and $\tau_0 = \tau_1[\tau/\alpha]$.

Unfold: The unfold coercion is the opposite to the fold coercion; it unfolds a recursive type $\tau_0 = \mu\alpha.\tau_1$ into type $\tau = \tau_1[\tau/\alpha]$.

Pack: The coercion cpack(τ_1, τ_2) coerces a type τ into an existential type τ_2. Here τ_2 is of the form $\exists\alpha.\tau_3$, and $\tau = \tau_3[\tau_1/\alpha]$.[4] The opposite of pack coercion is the instruction *open* (See Section 3.9), which has no runtime effect, but opens an existential type at type-checking time. Because *open* must bind a fresh type variable, it is not convenient to design it as a coercion. In our implementation, $A = open(B)$ is a virtual instruction with no runtime effect. At the type-checking time, B must be of some existential type $\exists.\tau$. Assume the current typing environment is ϕ. We must shift ϕ before adding the new binding $A : \tau$ into the typing environment. The result typing environment is $\phi^{[\uparrow]}, A : \tau$, where $^{[\uparrow]}$ is the shift operator in the explicit substitution calculus [Abadi et al., 1990].

Injection: The injection coercion cinjt[$\mathsf{sum}(\tau_r, \tau_u)$] coerces a type τ_u into a sum type $\mathsf{sum}(\tau_r, \tau_u)$.

From sum to range: The coercion csum2range coerces a sum type $\mathsf{sum}(\tau_r, \bot)$ to a range type τ_r. Note that the second argument of the sum type must be the bottom type $\bot$.

From sum to boxedone: The coercion csum2boxedone coerces a sum type $\mathsf{sum}(\bot, \tau)$ to type τ and makes sure that τ is not a union type; that is, there is only

[4] In our actual implementation, because we use de Bruijn index representation of variable bindings, τ_2 here is an open type and the result type of the pack coercion is $\exists.\tau_2$.

one case in the boxed part of the sum type. The compiler uses this one-boxed fact to optimize away the extra boxing. Note that the first argument of the sum type must be the bottom type $\bot$, stating that it is boxed and there is no unboxed case.

From sum to hastag: The coercion csum2hastag coerces a sum type $\mathsf{sum}(\bot, \tau)$ to type $\exists\alpha.\mathsf{hastag}(\alpha, \tau)$, or $\exists.\mathsf{hastag}(\underline{0}, \tau^{[\uparrow]})$ in the de Bruijn index representation. Note that the first argument of the sum type must be the bottom type $\bot$. The type $\tau^{[\uparrow]}$ is resulted from shifting type τ one step in the explicit substitution calculus [Abadi et al., 1990].

hastag elimination: The coercion cunhastag coerces a hastag type $\mathsf{hastag}(\tau_{tag}, \tau)$ into τ if τ is not a union type and $\tau \neq \bot$.

From int to int_{32}: The coercion cint2int32 coerces some refined integer type (such as $\mathsf{int}_{=}(\overline{1})$) or range type (such as $\mathsf{range}(\overline{0}, \overline{2})$) into an int_{32} type.

From singleton type to nonzero integer type: The coercion ci2nz coerces a singleton integer type to a refined integer type $\mathsf{int}_{\neq}(\overline{0})$ if values of the original type are not equal to zero.

From a singleton type to a range type: The coercion crange$[n_1, n_2]$ coerces a singleton type $\overline{i}$ to a range type $\mathsf{range}(n_1, n_2)$. The coerce rule checks that $n_1 \leq i < n_2$ holds.

Injection left: The injection coercion cinj1 (τ_1, τ_2) injects a type τ_1 into a union type $\tau_1 \cup \tau_2$.

Injection right: The injection coercion cinj2 (τ_1, τ_2) injects a type τ_2 into a union type $\tau_1 \cup \tau_2$.

Projection left: The projection coercion cproj1 transforms an intersection type $\tau_1 \cap \tau_2$ into its first component, that is, the type τ_1.

Projection right: The projection coercion cproj2 transforms an intersection type $\tau_1 \cap \tau_2$ into its second component, that is, the type τ_2.

Definition introduction: The cdef $\mathscr{D}$ coerces a type τ into a type definition $\mathsf{def}(\mathscr{D})$ if $\mathscr{D}$ is defined as τ.

Definition expansion: The coercion cname expands a def type $\mathsf{def}(\mathscr{D})$ into its definition if type definition $\mathscr{D}$ is defined. The LTAL type checker only expands a type definition when necessary.

Intersection coercion: The coercion cinters(c_1, c_2) coerces a type of the form $\tau_1 \cap \tau_2$ into $\tau_1' \cap \tau_2'$ if c_1 coerces τ_1 into τ_1' and c_2 coerces τ_2 into τ_2'.

Union coercion: The coercion cunion(c_1, c_2) is similar to cinters(c_1, c_2) except that it coerces a union type instead of an intersection type.

From the same type to intersection type: The coercion c2inters coerces a type τ into $\tau_1 \cap \tau_2$ if c_1 coerces τ into τ_1 and c_2 coerces τ into τ_2.

Field coercion: The coercion cfield c coerces a field type $\mathsf{field}(\tau_i, \tau)$ into another field type $\mathsf{field}(\tau_i, \tau')$ if c coerces τ into τ'.

From a label type addr to a $\mathsf{codeptr}$ type: The coercion caddr2code coerces a label type $\mathsf{addr}(l)$ into the label's code pointer type if the label is declared in the actual code.

Introduction of offset type with offset zero: The coercion coffset0 adds prefix offset 0 to a type.

Elimination of offset type with offset zero: This coercion removes the prefix offset 0 from a type. It is opposite to the previous one.

Partial instantiation of polymorphic functions: The coercion cptapp[τ] partially instantiates a polymorphic function, with the first type variable substituted with type τ.

Some of the coercion rules are shown in Figure 3.8. See Appendix A.1 for the complete set of coercion rules. The coercion typing judgement $\rho; LRT \vdash_c \tau \overset{c}{\hookrightarrow} \tau'$ means that under the kind environment ρ and maps LRT, coercion c changes type τ to τ' and τ' must be a subtype of τ.

If value v of type τ_1 is used in a place requiring type τ_2, the compiler has to insert a coercion c_{τ_1,τ_2} that transforms τ_1 to τ_2 explicitly. Thus the choice of subtyping rules is made explicit and LTAL needs no subtyping rules. Also, coercions make type equivalence rules unnecessary because two equivalent types can also be coerced to each other. Two types in LTAL are equivalent if and only if they are exactly the same.

Sometimes after applying a coercion we need to use the value both at its old type and its new type. This has been a difficulty in some previous TALs, which assign types to registers: They have to emit a `mov` instruction to handle this case.

We solve this problem by assigning types to variables, not to registers: A variable has only one type, but different variables can be assigned the same register. A move-with-coercion creates a new variable (in the same register) without executing

$$\frac{}{\rho; LRT \vdash_c \tau_1 \overset{\text{cinj1}\,[\tau_1 \cup \tau_2]}{\hookrightarrow} \tau_1 \cup \tau_2}\ \textit{CoerceInjectionLeft}$$

$$\frac{}{\rho; LRT \vdash_c \tau_2 \overset{\text{cinj2}\,[\tau_1 \cup \tau_2]}{\hookrightarrow} \tau_1 \cup \tau_2}\ \textit{CoerceInjectionRight}$$

$$\frac{\tau' = \tau[\mu\alpha.\tau/\alpha]}{\rho; LRT \vdash_c \tau' \overset{\text{cfold}[\mu\alpha.\tau]}{\hookrightarrow} \mu\alpha.\tau}\ \textit{CoerceFold}$$

$$\frac{}{\rho; LRT \vdash_c \mu\alpha : \kappa.\tau \overset{\text{cunfold}}{\hookrightarrow} \tau[\mu\alpha : \kappa.\tau/\alpha]}\ \textit{CoerceUnfold}$$

$$\frac{\tau_1 : \kappa}{\rho; LRT \vdash_c \tau_2[\tau_1/\alpha] \overset{\text{cpack}[\tau_1, \exists\alpha:\kappa.\tau_2]}{\hookrightarrow} \exists\alpha : \kappa.\tau_2}\ \textit{CoercePack}$$

$$\frac{}{\rho; LRT \vdash_c \tau_u \overset{\text{cinjection}(\mathsf{sum}(\tau_r, \tau_u))}{\hookrightarrow} \mathsf{sum}(\tau_r, \tau_u)}\ \textit{CoerceSumInjection}$$

$$\frac{\rho; LRT \vdash_c \tau \overset{c_2}{\hookrightarrow} \tau' \qquad \rho; LRT \vdash_c \tau' \overset{c_1}{\hookrightarrow} \tau''}{\rho; LRT \vdash_c \tau \overset{c_1 \circ c_2}{\hookrightarrow} \tau''}\ \textit{CoerceComposition}$$

$$\frac{\rho; LRT \vdash_c \tau_1 \overset{c_1}{\hookrightarrow} \tau_1' \qquad \rho; LRT \vdash_c \tau_2 \overset{c_2}{\hookrightarrow} \tau_2'}{\rho; LRT \vdash_c \tau_1 \cup \tau_2 \overset{\text{cunion}(c_1, c_2)}{\hookrightarrow} \tau_1' \cup \tau_2'}\ \textit{CoerceUnion}$$

Figure 3.8: Selected LTAL coercion rules.

an instruction. In effect, the variable name in an LTAL instruction tells the checker which type to use.

This means that when we "kill" a variable (by assigning a new value to its underlying register), we must also kill all the other variables bound to that register. When adding a new type binding $v : \tau$, we examine each binding $v' : \tau'$ in ϕ and remove it from ϕ if v' is assigned the same register as v, which means v' should be no longer live. We use $(\phi \backslash v), v : \tau$ to represent this operation; it can be seen in the premise (9) of the big rule in Section 3.3. When there is no ambiguity, it is

abbreviated to $\phi, v : \tau$. On the other hand, a move-with-coercions such as $v = c(v')$ does not require the application of the $\backslash v$ operator; other aliases of v continue to be active.

3.7 User-Defined Datatypes

LTAL's low-level type constructors provide support for various data representations, and extracting and checking tags. The type-checker can check the connection between a sum value and its tag, and refine the type of sum values after tag-checking. We provide flexibility for the compiler writer to choose her preferred style of datatype representation; the representations we describe in this section are not new, but the point is that we can type each aspect of their construction and deconstruction. Chen [2004, Chapter 5] presents a more detailed discussion of data type representations and their type checking in LTAL and some other TAL variants.

For simplicity, we use the notation $[\tau_0, \tau_1, \ldots, \tau_{n-1}]$ for tuple types and use the following two type macros:

- Type $\mathsf{range}(n_1, n_2)$ for type $(\mathsf{int}_{\geq}(\overline{n_1})) \cap (\mathsf{int}_{<}(\overline{n_2}))$. A sum type is often represented as $\mathsf{range}(0, n) \cup t$. The number n indicates the number of constant constructors, which are represented as integer $0, 1, \ldots, n-1$. Type t is the union of types for the boxed constructors.

- Type $\mathsf{hastag}(\tau_{tag}, \tau)$ for $(\mathsf{field}(0, \tau_{tag})) \cap \tau$. It means that the tag of a sum value has type τ_{tag}, and the sum value is of type τ.

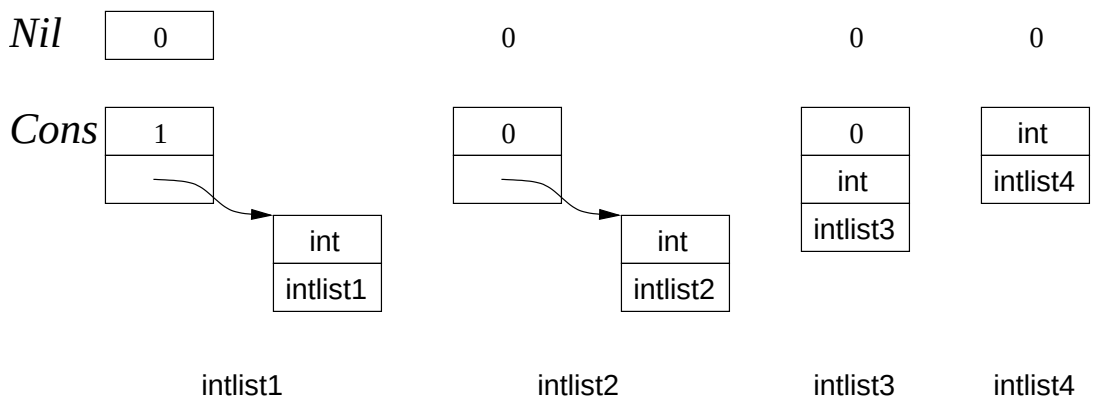


Figure 3.9: Datatype representations in LTAL. The boxed values are represented as $\boxed{\cdot}$.

3.7.1 Datatype representation

The compiler can choose from different data representations for user-defined datatypes such as intlist:

$$\text{datatype } \mathsf{intlist} = \mathit{Nil} \quad | \ \mathit{Cons} \ \text{ of } \mathsf{int} * \mathsf{intlist}$$

Figure 3.9 shows four kinds of data representations of the above intlist datatype:

1. The most straightforward representation is to tag each constructor with a small integer: *Nil* is tagged 0, and *Cons* tagged 1. In LTAL, this representation is expressed as the following type:

$$\mathsf{intlist}_1 = \mu\alpha.([\mathsf{int}_=(\overline{0})] \cup [\mathsf{int}_=(\overline{1}), [\mathsf{int}, \alpha]])$$

   This is a recursive type, whose body is a sum type. A sum type is represented as union of a range type and a tuple type. The range type represents the unboxed cases and the tuple type represents the boxed cases.

2. We assume that small integers can be distinguished from pointers, thus constant data constructors can be represented as small integers: *Nil* is represented as integer 0; *Cons* is a boxed record with tag 0. In LTAL, this representation is expressed as the following type:

$$\mathsf{intlist}_2 = \mu\alpha.(\mathsf{range}(0,1) \cup [\mathsf{int}_=(\overline{0}), [\mathsf{int}, \alpha]])$$

3. In the data representation of the *Cons* case in $\mathsf{intlist}_1$ and $\mathsf{intlist}_2$, there are two layers of boxing, one for tags and one for actual user data. We can optimize the representation so that only one boxing is need. In LTAL, this representation is expressed as the following type:

$$\mathsf{intlist}_3 = \mu\alpha.(\mathsf{range}(0,1) \cup [\mathsf{int}_=(\overline{0}), \mathsf{int}, \alpha])$$

4. A datatype with only one value-carrying constructor can be optimized further. It need not be tagged since there is only one boxed case. In LTAL, this representation is expressed as the following type:

$$\mathsf{intlist}_4 = \mu\alpha.(\mathsf{range}(0,1) \cup [\mathsf{int}, \alpha])$$

The $\mathsf{intlist}_4$ representation is specially optimized for the data types with only one boxed case. If there are multiple boxed cases, the tag field cannot be omitted and the $\mathsf{intlist}_3$ representation could be used, as the representation of the datatype example in Section 3.7.3 shows.

3.7.2 Creating sum values

We create an empty list of $\mathsf{intlist}_1$ by building a 1-element record $v_0 = [0]$, then coercing it to type $\mathsf{intlist}_1$:

LTAL	SPARC
(assume $v_0 : [\mathsf{int}_=(\overline{0})]$)	
$v_1 = \text{cinj1}\ ([\mathsf{int}_=(\overline{0})] \cup [\mathsf{int}_=(\overline{1}), [\mathsf{int}, \mathsf{intlist}_1]])(v_0)$	
$v_2 = \text{cfold}[\mathsf{intlist}_1](v_1)$	

The only difference between v_0, v_1, and v_2 is types. They have different types created by coercions, but they are assigned the same register, so no SPARC instruction is emitted for the above LTAL instructions.

By inserting coercions, the type-checker can easily tell that value v_0 can be coerced to be of type $\mathsf{intlist}_1$. In the first step, it simply checks if the type of v_0 is the first part of union type $[\mathsf{int}_=(\overline{0})] \cup [\mathsf{int}_=(\overline{1}), [\mathsf{int}, \mathsf{intlist}_1]]$ (by the rule of coercion *cinj1*). After this step, the type of v_1 is $[\mathsf{int}_=(\overline{0})] \cup [\mathsf{int}_=(\overline{1}), [\mathsf{int}, \mathsf{intlist}_1]]$. In the second step, if the type of v_1 is exactly the same as $\mathsf{intlist}_1$ with type variable α replaced with $\mathsf{intlist}_1$ (coercion *cfold*), the type of v_1 is coerced into $\mathsf{intlist}_1$ (the result type of v_2).

The following two LTAL instructions create an empty list of $\mathsf{intlist}_4$ by coercing integer 0 to be of type $\mathsf{intlist}_4$.

$v_1 = \text{crange}[0, 1](0)$	`mov` $0, d_1$
$v_2 = \text{cinj1}\ (\mathsf{range}(0, 1) \cup [\mathsf{int}, \mathsf{intlist}_4])(v_1)$	
$v_3 = \text{cfold}[\mathsf{intlist}_4](v_2)$	

Coercion $\text{crange}[n_1, n_2]$ changes a value of type $\mathsf{int}_=(\overline{n})$ to type $\mathsf{range}(n_1, n_2)$

if $n_1 \leq n < n_2$. In the first instruction the type-checker only needs to check if $0 \leq 0 < 1$ holds.

3.7.3 Eliminating sum values

Consider what happens when doing case discrimination on a boxed-tag style of sum type representation, such as is used when there are multiple value-carrying constructors. Given a value x of sum type, one fetches its tag into a variable y, then does a conditional branch on y; at this point, the difficulty is in relating the outcome of the conditional branch to the refined type of x. One solution is to use a "macro" TAL instruction to code for the load-compare-branch instruction sequence. We wanted to avoid all such macro instructions since they hinder some compiler optimizations such as instruction scheduling. We use type quantification and singleton types to keep track of the implicit dataflow.

Consider the following user-defined datatype

$$\text{datatype } \mathsf{T} = A \mid B \mid C \text{ of } \mathsf{int} \mid D \text{ of } \mathsf{int} * \mathsf{T}$$

which can be represented in LTAL as:

$$\mathsf{T} = \mu\alpha.(\mathsf{range}(0,2) \cup [\mathsf{int}_{=}(\overline{0}), \mathsf{int}] \cup [\mathsf{int}_{=}(\overline{1}), \mathsf{int}, \alpha]).$$

This representation is shown pictorially in Figure 3.10. Since the datatype T has two value-carrying constructors (C and D), the tag field cannot be saved. This representation is similar to the $\mathsf{intlist}_3$ representation showed before.

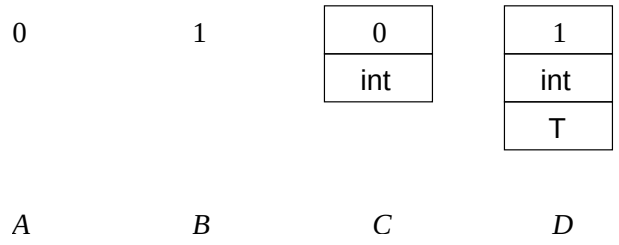


Figure 3.10: LTAL datatype representation example. The boxed values are represented as $\boxed{\cdot}$.

LTAL		SPARC	
	$v_0 = \text{cunfold}(v)$		
	$(\beta, v_0') = \mathit{testbox}(v_0)$		`subcc` $d, 256$
	ifboxed (v_0') *then* (v_1, l_{CD}) *else* (v_2, l_{AB})		`bge` l_{CD}
$l_{AB}:$	$\ldots$	$l_{AB}:$	$\ldots$
$l_{CD}:$	$(\alpha_1, v_3) = \mathit{open}(v_1)$	$l_{CD}:$	
	$t = \mathit{gettag}(v_3, 0)$		`ld` $[d], d_t$
	$\mathit{cmpcc}(t, 0)$		`subcc` $d_t, 0, \%g0$
	iftag $(=)$ $\{v_3\}$ *then* (v_3', l_C) *else* (v_3'', l_D)		`be` l_C
$l_D:$	$\ldots$	$l_D:$	$\ldots$
$l_C:$	$\ldots$	$l_C:$	$\ldots$

Figure 3.11: Datatype tag discrimination example. (Variables v_0, v, v_0', v_1, v_2, v_3, v_3', and v_3'' are all assigned register d, and variable t is assigned register d_t.)

"Switching" on sum values in source program

$$\begin{array}{lll} \text{case}(v : \mathsf{T}) & \text{of} & A \Rightarrow e_A \\ & | & B \Rightarrow e_B \\ & | & C(x) \Rightarrow e_C \\ & | & D(x, y) \Rightarrow e_D \end{array}$$

is translated to the LTAL and SPARC instruction sequence in Figure 3.11.

We need to generate code that tests v to decide which branch to take. Each test and each branch should be an explicit LTAL instruction. We first test whether v is boxed or not. From our assumption that no pointers point to the first 256 words in the memory, if v is a small integer (less than 256), then it is unboxed. That is, it is either A or B. Otherwise it is C or D.

The type checking rules used for datatype tag discrimination are shown in Figure 3.12. Instruction *testbox* performs this test and sets condition codes. Instruction *ifboxed* examines the condition codes and rebinds two fresh variables v_1 and v_2 with refined types for boxed and unboxed cases, respectively. Variable v_1 has type $\exists\alpha.\mathsf{hastag}(\alpha, [\mathsf{int}_=(\overline{0}), \mathsf{int}] \cup [\mathsf{int}_=(\overline{1}), \mathsf{int}, \tau])$, which means it is tagged (we do not know the tag yet). Variable v_2 has type $\mathsf{range}(0, 2)$, which means it is either 0 or 1. Both v_1 and v_2 are forced to be assigned the same register as v_0, so no machine instruction is needed to move v_0 to v_1 or v_2.

In the unboxed case, we further test if v_2 is 0 or 1, which is easy. In the boxed case, we need to test the tag of v_1. Variable v_1 hides the type of its tag by existential types. We first open v_1 to v_3 and bind a brand new type variable α_1. Again, no SPARC instruction is needed because v_1 and v_3 are assigned the same register. Variable v_3 has type $\mathsf{hastag}(\alpha_1, [\mathsf{int}_=(\overline{0}), \mathsf{int}] \cup [\mathsf{int}_=(\overline{1}), \mathsf{int}, \tau])$.

Instruction *gettag* extracts the tag t and gives it type $\mathsf{int}_=(\alpha_1)$. Then *cmpcc* checks if tag t is 0 and set condition-code environment to be $\mathsf{cc\_cmp}(\alpha_1, \overline{0})$. Instruction *iftag* checks condition codes set by *cmpcc*, rebinds two new variables v_3' and v_3'' as aliases of v_3 and does conditional branch. Specifically, the type checking rule of the *iftag* instruction checks that: cc is $\mathsf{cc\_cmp}(\tau_0, \overline{0})$, v_3 is of type $\mathsf{hastag}(\tau_0', \tau)$, and $\tau_0 = \tau_0'$; in this example, both τ_0 and τ_0' are α_1. Then it refines the types of v_3' and v_3'' to $[\mathsf{int}_=(\overline{0}), \mathsf{int}]$ and $[\mathsf{int}_=(\overline{1}), \mathsf{int}, \tau]$, respectively. This refinement rules out

$$\frac{LRT; \rho; \phi \vdash v : \tau \quad \phi' = (\phi \backslash v), v' : \mathsf{int}_{=}(\alpha) \cap \tau \quad cc' = \mathsf{cc\_testbox}(\alpha)}{LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{(\alpha, v') = testbox(v)\}\ (\rho; \mathscr{H}; \phi'; cc')}\ \mathit{InstrTestbox}$$

$$\frac{\begin{array}{c} LRT; \rho; \phi \vdash v : \mathsf{int}_{=}(\tau_\alpha) \cap (\mathsf{int}_{=}(\overline{0}) \cup \ldots \cup \mathsf{int}_{=}(\overline{n-1}) \cup \tau') \\ \tau' = \tau_1 \cup \tau_2 \cup \ldots \cup \tau_m \\ cc = \mathsf{cc\_testbox}(\tau_\alpha) \qquad n < 256 \\ \tau_i = (\mathsf{field}(0, \mathsf{int}_{=}(tag_i))) \cap \tau_i' \ \ (\text{for all } 1 \leq i \leq m) \\ LRT; \rho; \mathscr{H}; \phi, v_1 : \tau'; cc \vdash_\ell l_1 \\ LRT; \rho; \mathscr{H}; \phi, v_2 : \mathrm{range}(0, n); cc \vdash_\ell l_2 \end{array}}{LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{ifboxed\ (v)\ then\ (v_1, l_1)\ else\ (v_2, l_2)\}\ (\_; \_; \_; \_)}\ \mathit{InstrIfboxed}$$

$$\frac{LRT; \rho; \phi \vdash v : \exists \alpha : \kappa.\tau}{LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{(\alpha, v_0) = open(v)\}\ (\rho, \alpha : \kappa; \mathscr{H}; \phi, v_0 : \tau; cc)}\ \mathit{InstrOpen}$$

$$\frac{LRT; \rho; \phi \vdash v' : \mathsf{hastag}(\tau_{tag}, \tau_u) \quad \phi' = (\phi \backslash v), v : \mathsf{int}_{=}(\tau_{tag})}{LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{v = gettag(v')\}\ (\rho; \mathscr{H}; \phi'; cc)}\ \mathit{InstrGettag}$$

$$\frac{LRT; \rho; \phi \vdash v_1 : \mathsf{int}_{=}(\tau_1) \quad LRT; \rho; \phi \vdash v_2 : \mathsf{int}_{=}(\tau_2)}{LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{cmpcc(v_1, v_2)\}\ (\rho; \mathscr{H}; \phi; \mathsf{cc\_cmp}(\tau_1, \tau_2))}\ \mathit{InstrCmpcc}$$

$$\frac{\begin{array}{c} LRT; \rho; \phi \vdash v : \mathsf{hastag}(\tau_\alpha, \tau_u) \\ cc = \mathsf{cc\_cmp}(\tau_\alpha, \overline{i}) \\ \tau = \tau_1 \cup \tau_2 \cup \ldots \cup \tau_n \\ \tau_i = \mathsf{field}(0, \mathsf{int}_{=}(tag_i)) \cap \tau_i' \ \ (\text{for all } 1 \leq i \leq n) \\ \tau_t = \bigcup_{1 \leq j \leq n} \tau_j \ \ \text{where } i\ \pi\ tag_j \text{ holds} \\ \tau_f = \bigcup_{1 \leq k \leq n} \tau_k \ \ \text{where } i\ \pi\ tag_k \text{ does not hold} \\ LRT; \rho; \mathscr{H}; \phi, v_1 : (\mathsf{field}(0, \mathsf{int}_{=}(\tau_\alpha))) \cap \tau_t; cc \vdash_\ell l_1 \\ LRT; \rho; \mathscr{H}; \phi, v_2 : (\mathsf{field}(0, \mathsf{int}_{=}(\tau_\alpha))) \cap \tau_f; cc \vdash_\ell l_2 \end{array}}{LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{iftag\ (\pi)\ \{v\}\ then\ (v_1, l_1)\ else\ (v_2, l_2)\}\ (\_; \_; \_; \_)}\ \mathit{InstrIftag}$$

Figure 3.12: Rules for datatype tag discrimination.

disjuncts by the result of comparing tags with integers. A constraint solver as in DTAL [Xi and Harper, 2001] is overkill for our purpose.

The connection between a tagged value and its tag is established by existential types, since every time we open a variable of type $\exists\alpha.\mathsf{hastag}(\alpha, \tau)$ and assign it to some variable v, we get a fresh type variable α' (α_1 in the above example), and only v's type contains the new type variable α' in the first conjunct ($\mathsf{field}(0, \alpha')$), and only by instruction $gettag(v, 0)$ can we get a variable of type α'.

For simplicity we use linear search here. LTAL also permits binary search; to do an indexed jump we would need to extend LTAL, but our underlying semantic model will permit this in a modular way.

3.8 Heap Allocation

In this section we briefly present the heap allocation model used in LTAL and the FPCC/ML compiler. Chen [2004, Chapter 4] gives a more detailed discussion of the model, including record allocation, known- and unknown-length array allocation, and their type checking.

Like SML/NJ, our compiler allocates closures and records in registers or on the heap; we don't push and pop the stack. At present, our type system (like most TALs) also does not accommodate reasoning about garbage collection either. We intend to handle stacks and GC in the future, after we develop a unified theory of stack and heap deallocation (probably based on a region calculus).

As in SML/NJ, with so much heap allocation we need extremely efficient, in-line allocation of records. We model the allocable heap memory as a large contiguous region bounded by two pointers, *allocptr* and *limitptr*. Heap allocation is broken

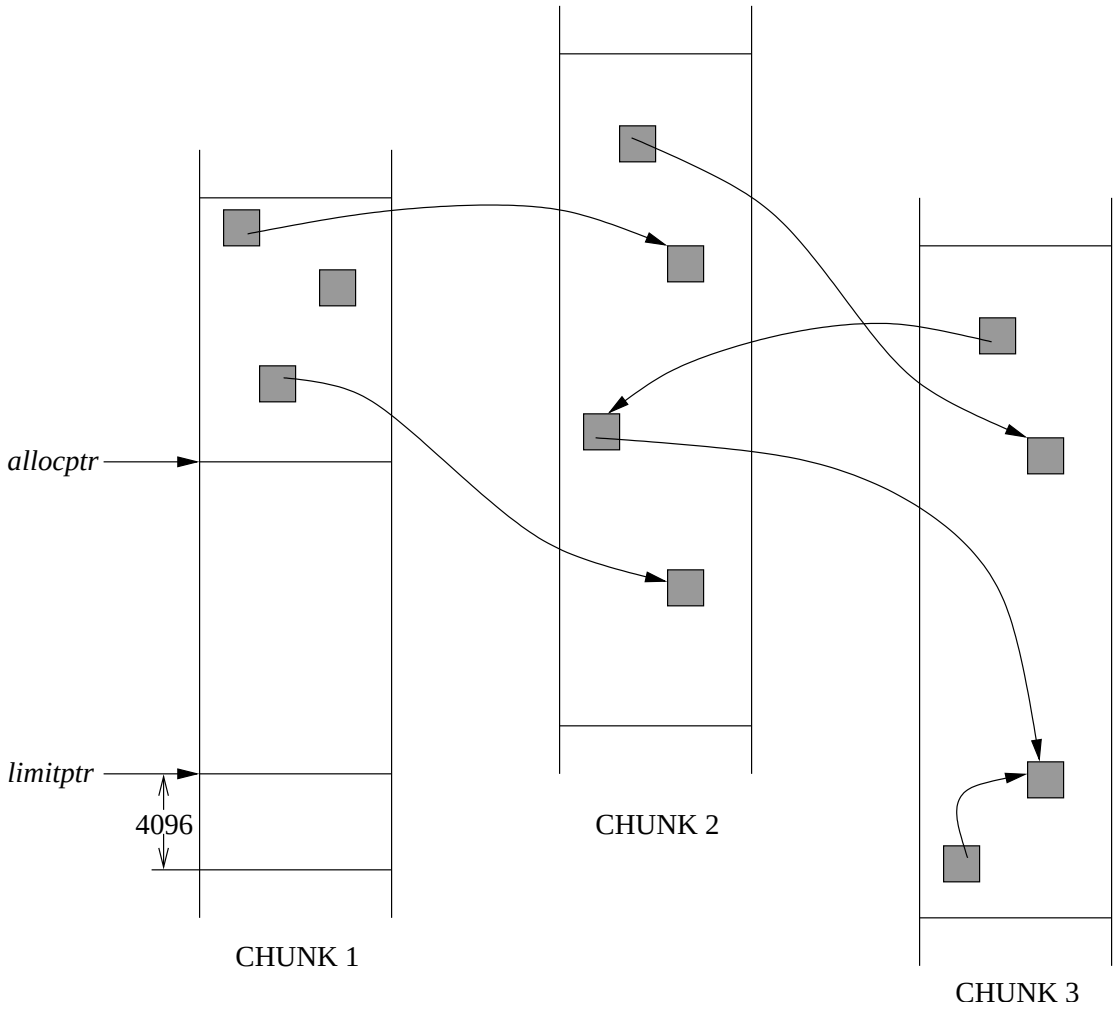


Figure 3.13: SML/NJ Heap allocation model.

into two steps: first, test whether there is enough memory for allocation; second, initialize memory.

The heap allocation model is shown in Figure 3.13. Before the runtime system starts executing a program, it reserves a chunk of memory, and sets the *allocptr* to the lowest address of the memory chunk, and the *limitptr* the highest address (minus a constant $C = 4096$). When the program needs n memory words, where $4n \leq C$, it tests whether *allocptr* $\leq$ *limitptr*; if so, then at least n words must

LTAL		SPARC	
l_0 :	*testmem*(3)	l_0 :	`subcc` *allocptr*, *limitptr*, %$g0$
	iffull then l_1 *else* l_2		`bg` l_1
l_2 :	*init*$(0, v_0)$	l_2 :	`st` $d_0, [allocptr + 0]$
	init$(1, v_1)$		`st` $d_1, [allocptr + 4]$
	init$(2, v_2)$		`st` $d_2, [allocptr + 8]$
	$v =$ *record*		`mov` *allocptr*, d
	inc_allocptr(3)		`add` *allocptr*, 12, *allocptr*
...	...	...	...
l_1 :	...	l_1 :	...

Figure 3.14: Heap allocation example.

be available. Then it fills in n words consecutively to addresses from *allocptr* to *allocptr* $+ 4n - 4$, then increases *allocptr* by $4n$.

The LTAL instruction sequence in Figure 3.14 creates a 3-field record $[v_0, v_1, v_2]$ and assigns it to v. The corresponding SPARC instructions are on the right side of the table (d, d_0, d_1, d_2 are registers assigned to LTAL variables v, v_0, v_1, v_2).

Block l_0 tests if there are at least 3 words in the memory for allocation; after the *testmem* comparison the condition-code environment is cc_testmem(3). Then the branch instruction *iffull* "consumes" this condition code, and statically guarantees 3 words in the fall-through case (memory is not full).

Block l_2 initializes the three newly allocated words. Instruction *init*(i, v_i) initializes the word whose address is *allocptr* $+ 4i$ with v_i. Instruction $v =$ *record* copies *allocptr* to v and v gets the record type. Instruction *inc_allocptr*(n) increases *allocptr* by $4n$.

The instruction sequence for allocation is not fixed. The instruction scheduler can shuffle these instructions with others, as long as certain constraints hold.

An allocation environment $\mathscr{H}$ is used to check heap allocation. It consists of

$$\frac{0 \le n \le 1024}{LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{testmem(\mathrm{n})\}\ (\rho; \mathscr{H}; \phi; \mathsf{cc\_testmem}(n))}\ \mathit{InstrTestmem}$$

$$\frac{\begin{array}{c} cc = \mathsf{cc\_testmem}(n) \quad LRT; \rho; \mathscr{H}; \phi; cc \vdash_\ell l_1 \\ LRT; \rho; (n, -1, \top); \phi; cc \vdash_\ell l_2 \end{array}}{LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{iffull\ then\ l_1\ else\ l_2\}\ (\_; \_; \_; \_)}\ \mathit{InstrIffull}$$

$$\frac{\begin{array}{c} LRT; \rho; \phi \vdash v_i : \mathsf{int}_=(i) \quad 0 \le i < n \quad m' = max(m, i) \\ LRT; \rho; \phi \vdash v : t_i \quad t' = t \cap (\mathsf{field}(4i, t_i)) \end{array}}{LRT \vdash (\rho; (n, m, t); \phi; cc)\ \{init(v_i, v)\}\ (\rho; (n, m', t'); \phi; cc)}\ \mathit{InstrInit}$$

$$\frac{}{LRT \vdash (\rho; (n, m, t); \phi; cc)\ \{v = record\}\ (\rho; \mathscr{H}; \phi, v : t; cc)}\ \mathit{InstrRecord}$$

$$\frac{LRT; \rho; \phi \vdash v : \mathsf{int}_=(n') \quad m < n' \le n}{\begin{array}{c} LRT \vdash (\rho; (n, m, t); \phi; \mathsf{cc\_testmem}(k)) \\ \{inc\_allocptr(v)\} \\ (\rho; (n - n', -1, \top); \phi; \mathsf{cc\_none}) \end{array}}\ \mathit{InstrIncAllocptr1}$$

$$\frac{LRT; \rho; \phi \vdash v : \mathsf{int}_=(n') \quad m < n' \le n \quad cc \ne \mathsf{cc\_testmem}(k)}{\begin{array}{c} LRT \vdash (\rho; (n, m, t); \phi; cc) \\ \{inc\_allocptr(v)\} \\ (\rho; (n - n', -1, \top); \phi; cc) \end{array}}\ \mathit{InstrIncAllocptr2}$$

Figure 3.15: Rules for allocation instructions.

three parts: the number of words that are guaranteed to be available in the memory, the largest index of initialized fields, and the type of the partial record initialized so far. We don't need the initialization flags used in TALx86 [Morrisett et al., 1999a].

The typing rules for the allocation instructions are shown in Figure 3.15. The judgement $LRT; \rho; \mathscr{H}; \phi; cc \vdash_\ell l$ states that the signature of block l matches the

current environment; for heap allocation, the environment $\mathscr{H}$ must hold. If this judgement holds, it is safe to jump to block l, and safe to allocate certain size of memory (without out-of-heap testing) specified by the environment $\mathscr{H}$. Instructions *testmem* and *iffull* establish the allocation environment in which the *init* instructions type-check. The compiler can (and does) optimize by making one *iffull* cover the sequential allocation of several different records in a control-flow path that covers several basic blocks. The parameter m of $\mathsf{codeptr}$ conveys the necessary information about how much memory is guaranteed to remain.

A tuple type $[\tau_0, \tau_1, \ldots, \tau_{n-1}]$ is represented in LTAL as

$$(\mathsf{field}(0, \tau_0)) \cap (\mathsf{field}(1, \tau_1)) \cap \ldots \cap (\mathsf{field}((n-1), \tau_{n-1})).$$

If v has this type, then the word located at memory address v has type τ_0, at address $v+4$ type τ_1, etc. (assuming the word size is 4). When a field is initialized by a *init* instruction, one more conjunct (a field type) is added into the type of the partial record in the allocation environment.

After initialization, the *allocptr* is copied to a variable (with record type) by instruction v = *record*, and then the *allocptr* is adjusted to point to the next available memory word by instruction *inc_allocptr*. After instruction *inc_allocptr*, the condition codes set by *testmem* are invalid because *allocptr* has been changed. So we reset the condition-code environment if it is *testmem*.

In the above example, $\mathscr{H}$ is $(3, -1, \top)$ when checking function l_2. The number 3 means l_2 needs 3 words in the heap; the second number -1 means no fields has been initialized; the type $\top$ means none of the 3 words is initialized. The environment $\mathscr{H}$ becomes $(3, 0, \mathsf{field}(0, t_0))$ after instruction *init*$(0, v_0)$, $(3, 1, (\mathsf{field}(0, t_0)) \cap (\mathsf{field}(4, t_1)))$

after instruction *init*$(4, v_1)$, and $(3, 2, (\mathsf{field}(0, t_0)) \cap (\mathsf{field}(4, t_1)) \cap (\mathsf{field}(8, t_2)))$ after instruction *init*$(8, v_2)$, where t_0, t_1, and t_2 are the types of v_0, v_1, and v_2, respectively. Variable v gets type $(\mathsf{field}(0, t_0)) \cap (\mathsf{field}(4, t_1)) \cap (\mathsf{field}(8, t_2))$, which is the third part of $\mathscr{H}$ at this point, after instruction $v =$ *record*. Instruction *inc_allocptr*(3) clears $\mathscr{H}$ to be $(0, -1, \top)$.

3.9 Instructions

LTAL has a number of instructions to allow efficient type checking of heap allocation processes, position-independent code, user-defined data type tag discrimination, condition codes, and polymorphic code blocks. The LTAL instructions are listed in Figure 3.16.

We briefly explain their informal meanings and type checking below:

***open*:** The *open* instruction has no runtime effect. The compiler assigns the old and new variables to the same register, and only the type is changed. It is opposite to the cpack coercion.

***move*:** The statement $v = v'$ is a move instruction if the registers assigned to variables v and v' are different. If v and v' are assigned to the same register, it has no runtime effect, but copies the type of v' to v.

ALU instructions: LTAL has a set of standard ALU instructions such as addition, subtraction, and multiplication.

***sethi*:** If an integer is too big to fit in an instruction as the immediate operand field, the *sethi* is used to set the high bits first. The semantics of instruction *sethi*

Instructions

ι	::=	$(\alpha, v') = open(v)$	*no instruction*
	|	$v' = v$	*move, or nop*
	|	$v = v_1 \ op \ v_2$	*ALU instructions*
	|	$v = sethi(n)$	*sethi*
	|	$v = load(v_1)$	*load*
	|	$v = store(v_1)$	*store*
	|	$v = addradd(v_1, v_2)$	*add*
	|	$v = select(v_1, v_2)$	*load*
	|	$v = gettag(v)$	*load*
	|	$init(v_i, v)$	*store*
	|	$v = record$	*move*
	|	$inc\_allocptr(v)$	*add*
	|	$call(v, [\tau_1, \ldots, \tau_n])$	*jump*
	|	$calln(l, [\tau_1, \ldots, \tau_n])$	*fall through*
$\star$	|	$cmpcc(v_1, v_2)$	*subcc*
$\star$	|	$(\alpha, v_1') = cmpcci(v_1, v_2)$	*subcc*
$\star$	|	$(\alpha, v') = testbox(v)$	*subcc*
$\star$	|	$testmem(n)$	*subcc*
$\star$	|	*if* (π) *then* l_1 *else* l_2	*branch*
$\star$	|	*iffull then* l_1 *else* l_2	*branch*
$\star$	|	*ifboxed* (v) *then* (v_1, l_1) *else* (v_2, l_2)	*branch*
$\star$	|	*ifboxedone* (v) *then* (v_1, l_1) *else* (v_2, l_2)	*branch*
$\star$	|	*iftag* (π) $\{v\}$ *then* (v_1, l_1) *else* (v_2, l_2)	*branch*

Figure 3.16: LTAL syntax: Instructions. Marked $\star$ operators are specific to machines with condition codes.

is the same as SPARC *sethi* instruction. The instruction $v = sethi(n)$ zeroes the least significant 10 bits of variable v's register, and puts the immediate n in the high 22 bits.

***load*:** The LTAL instruction $v = load(v_1)$ maps to a SPARC load instruction. It loads v_1 (in memory) into v (in register). In the new typing environment, variable v gets v_1's type and bindings whose variables are assigned in the same register as v are killed.

***store*:** The LTAL instruction $v = store(v_1)$ maps to a SPARC store instruction. It stores v (in register) into v_1 (in memory). The type checking rule is very similar to that of *load* instruction.

***addradd*:** The *addradd* is for address arithmetic. It is used for type checking position-independent code. The instruction $v = addradd(v_1, v_2)$ assigns v_1+v_2 to v, where v_1 is a label value and v_2 is a *vdiff* value. Operand v_1 is of label type $\mathsf{addr}(l)$ for some label l. Operand v_2 is a known integer, which is the difference between two labels. The type of v_2 is $\mathsf{diff}(l_1, l_2)$ for some labels l_1 and l_2, and the value of v_2 is $l_1 - l_2$ which is a compile-time known integer. For type checking position-independent code, v_1 is usually the base label, and v_2 is the offset of a label from the base. See Section 3.10 for the details of position-independent code type checking.

***select*:** The *select* statement corresponds to a memory load machine instruction. It loads a record field.

***gettag*:** The *gettag* statement also corresponds to a memory load machine instruction, but it loads a tag for some sum data type. For example, the instruction $A \leftarrow gettag(B)$ loads the tag of B into A, where B should have type $\mathsf{hastag}(\tau_{tag}, \tau_u)$, for some tag type τ_{tag} and union type τ_u. The result type of A is $\mathsf{int}_=(\tau_{tag})$.

***init*:** The *init* is used to initialize a record field, and it maps to a memory store instruction.

***record*:** The instruction $v \leftarrow record$ moves *allocptr*, which resides in a dedicated register, to variable v, whose type will be a record type. The *record* instruction

maps to a memory store instruction if variable v is a memory location, and to a bitwise OR instruction if variable v resides in a register.

***inc_allocptr*:** The instruction *inc_allocptr*(i) increment the *alloc* pointer by i, and invalidates the condition code environment cc_testmem.

***call*:** The instruction *call*$[\sigma](v)$ maps to a SPARC `ba` (branch aways) instruction or `jmpl` (jump and link) instruction depending on whether v is a variable residing in some register or a label value. The substitution σ is applied to the environments for type checking.

***calln*:** The instruction *calln* is fall-through. It maps to no SPARC instructions. The target label must be at the same address as the *calln* instruction.

***cmpcc*:** The *cmpcc* maps to a SPARC `subcc` instruction, and updates the condition code environment to cc_cmp.

***testbox*:** The *testbox* instruction corresponds to a SPARC `subcc` instruction. It compares a variable residing in some register to integer 256 to decide whether the variable is a pointer or not. The implicit assumption here is that all pointers have address values greater than 256. This instruction updates the condition code environment to cc_testbox.

***testmem*:** The *testmem* instruction also corresponds to a SPARC `subcc` instruction. It compares the *alloc* pointer to *limitptr* (the limit pointer as shown in Figure 3.13) to decide whether there are free memory cells or not. This instruction updates the condition code environment to cc_testmem.

Branch instruction *if*: The *if* statement corresponds to a SPARC branch instruction. Its type checking rules check the argument types of both branches, but does not refine any types or environments in either branch as the *iffull*, *ifboxed*, *ifboxedone*, and *iftag* instructions do.

Branch instruction *iffull*: The *iffull* statement corresponds to the SPARC branch instruction `bcc`. Its type checking rule checks the argument types of both branches, and checks that the condition code environment is cc_testmem and has enough free memory slots. The type checker then remembers this fact in the heap allocation environment.

Branch instruction *ifboxed*: The *ifboxed* statement corresponds to the SPARC branch instruction `bcc`. Its type checking rule checks the argument types of both branches, and checks that the condition code environment is cc_testbox and with valid type. The type checker then refines the types of the variable being tested for both branches depending on whether it is boxed or not.

For example, the instruction $A = \mathit{testbox}(B)$ sets the type of variable A to $(\underline{0} \cap \tau_b^{[\uparrow]})$, where $\underline{0}$ is the de Bruijn index implementation of a fresh type variable, τ_b is the type of value B, and $\uparrow$ is the shift operator in the explicit substitution calculus [Abadi et al., 1990]. The *testbox* instruction binds a fresh type variable. After this instruction, the new environments, including typing environment ϕ, heap allocation environment $\mathscr{H}$, and condition code environment cc, have one new type variable and its kind. The condition code environment is set to $\mathsf{cc\_testbox}(\underline{0})$.

In the type checking rule of instruction *ifboxed* (v) *then* (v_1, l_1) *else* (v_2, l_2), we check that the condition code environment is $\mathsf{cc\_testbox}(\tau)$, and the type of v

is $\tau \cap \mathsf{sum}(\tau_r, \tau_s)$ whose first component τ matches the type in the condition code environment. Then we refine the type of v to either $\exists.\mathsf{hastag}(\underline{0} \cap \tau_s^{[\uparrow]})$ or τ_r and bind them to v_1 or v_2, respectively, depending on whether v is boxed or not. We also check that the argument types match for branches to labels l_1 and l_2. We use type variables (de Bruijn indices in our implementation) and intersection types to check integrity of type safety when the testing of boxedness and branching on boxedness are separated instructions. We use a similar trick to type check pairs of *testmem* and *iffull* instructions and pairs of *cmpcc* and *iftag* instructions. Chen and Tarditi [2005] have subsequently used this to type check method lookup and call in virtual table in object-oriented languages [Chen and Tarditi, 2005].

Branch instruction *ifboxedone*: The *ifboxedone* instruction is a special case of *ifboxed*. Its type checking rule does an additional check that the type τ_s above is not a union type; that is, there is only one boxed case in the sum type. In this case, the compiler optimizes the data representation of the sum data type to save the tag field and remove one layer of boxing as shown in Figure 3.9.

Branch instruction *iftag*: The *iftag* (π) $\{v\}$ *then* (v_1, l_1) *else* (v_2, l_2) statement corresponds to a SPARC branch instruction. The *cmpcc* instruction compares a variable to the tag of a sum data type (tagged union) and sets the condition code (environment) to $\mathsf{cc\_cmp}$. The type checking rule of *iftag* checks the argument types of both branches, and checks that the condition code environment is $\mathsf{cc\_cmp}(\tau_{tag}, i)$ and matches v's type $\mathsf{hastag}(\tau_{tag}, \tau_u)$. Then the type of v is refined depending on whether or not the tag matches, and the new type is bound to v_1 or v_2, respectively.

The sophisticated rule shown in Section 3.3 is a typical instruction type checking rule in our actual implementation. For the sake of clarity, we present a simple version of the actual rules implemented. The complete instruction typing rules are listed in Appendix A.2.

3.10 Don't Trust the Linker!

To avoid the need to reason about possible bugs in the link-loader, we arrange that each compilation unit needs no link-editing, and links to others using closures, in the style of SML/NJ [Blume and Appel, 1997, §3]. We must avoid the need for a linker to do relocation. Our safety policy says, "a program is safe if, no matter where we load it in memory, it will never access an illegal address or execute an illegal instruction" [Appel, 2001].

PCC systems are most useful in applications where untrusted code shares the same address space with trusted code; in such situations, position-independent code is desirable because it makes the linker flexible.

Position-independent code must use relative addresses instead of absolute ones. The problem arises when we move a label into a register or store it in memory, to make a function-pointer or a closure. The value of the label depends on where the code is loaded.

We adopt the solution that SML/NJ uses, but we show how to type-check it. Each function takes a *base* parameter, which is the start address of its own machine code in the memory. We keep the *base* address of the current function in a register, and calculate the addresses of labels as offsets from *base*. When a function f is called, the address f is passed as its own *base* argument.

In the body of a function f, moving a label g to variable v is implemented as $v = addradd(base, g - f)$, where $g - f$ is a constant computed by the compiler. Instruction *addradd* is translated to the SPARC *add* instruction, and used only for address arithmetic.

$$\frac{\begin{array}{c} LRT; \rho; \phi \vdash v_1 : \mathsf{addr}(f) \\ LRT; \rho; \phi \vdash v_2 : \mathsf{diff}(g, f) \\ \phi' = (\phi \backslash v), v : \mathsf{addr}(g) \end{array}}{LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{v = addradd(v_1, v_2)\}\ (\rho; \mathscr{H}; \phi'; cc)}\ \mathit{InstrAddrAdd}$$

To type-check position-independent code, we introduce type constructors addr and diff. The former gives a type to a label and the latter types the difference between two labels. For example, in the above example $v = addradd(base, g - f)$, variable *base* has type $\mathsf{addr}(f)$; the compile-time known constant $g - f$, which is represented as a value $vdiff(g, f)$, has type $\mathsf{diff}(g, f)$; and the typing rule for *addradd* will give type $\mathsf{addr}(g)$ to v.

When a function f is called in a compilation unit other than where it is defined, its label is (statically) unknown at the call site. Then the type of its *base* cannot be addr. We use existential types to hide the *base* type; the type of f becomes $\exists\beta.\mathsf{codeptr}[\vec{\alpha} : \vec{\kappa}](m, [base : \beta, \ldots])$. To make sure that f itself is passed to its *base* when f is called, we make f have type $\exists\beta.(\beta \cap \mathsf{codeptr}[\vec{\alpha} : \vec{\kappa}](m, [base : \beta, \ldots]))$.

As an important optimization, when a function is called only by direct jumps from known locations, it does not need its own *base* argument—it can use the *base* of one of its known callers. This avoids *addradd* instructions in local loops and branches.

Category	*Constructors & operators*
SPARC instruction constructors	196
SPARC instruction decoding rules	263
Coercion operators	32
Coercion rules	46
Explicit substitution calculus	59
Environment constructors	63
LTAL type operators	21
LTAL instruction operators	43
Type refinement rules	82
Kind operators	17
Kind checking rules	36
Type wellformedness rules	41
Local environment management	86
Static arithmetic calculations	55
Rules for parsing LRT maps	16
Structural type matching heuristics	38
Branch checking rules	17
LTAL instruction constructors	52
Instruction typing rules	53
Total	1,216

Table 3.1: LTAL calculus statistics.

3.11 Measurements

3.11.1 Size

The LTAL calculus is a large engineering artifact, just like the compiler that produces it and the SPARC machine that consumes it. It comprises (at the current state of implementation) approximately 1200 operators and rules. The statistic data are summarized in Table 3.1. The first column gives brief description of various constructors and rules. The second column shows the number of constructors and rules.

A typical large rule, such as the one shown in Section 3.3, is quantified over a dozen variables and has a dozen premises. In all, the current LTAL type checker is 4,163 lines of (non-blank, non-comment) Prolog-like source code. The machine-checked proof of the soundness of all the LTAL rules (which is nearing completion) is over 143,400 lines of higher-order logic as represented in the Twelf system. The axioms comprise 1,957 lines, almost all of which is the specification of the SPARC instruction set architecture.

The compiler from core ML to LTAL and SPARC machine code is written in ML; its size (including blank lines and comments) is 50k lines of the SML/NJ (version 110.35) front end (unmodified); 1.8k lines of code copied and modified from the implementation of the SML/NJ interactive top-level loop; 2.7k lines to translate FLINT to NFLINT; 7.8k lines to translate NFLINT to LTAL; 1.2k lines to interface of MLRISC; and approximately 50k lines of the MLRISC system[5] itself, of which 400 lines are new or modified to support our more-general annotation interface.

3.11.2 Performance

We compared our performance[6] to that of SML/NJ (version 110.35) on two small benchmarks: Life (adapted from the Standard ML benchmark suite) and RedBlack, which uses balanced trees to do queries on integer sets. The results are shown in Table 3.2.

Our compile time is not competitive (2.998 seconds to compile Life compared to 0.49 seconds for the production release of SML/NJ); we have not engineered our

[5]The MLRISC software has several other analyses, optimizations, and target machine specifications that we did not use and that we don't count here.

[6]The compile and run time is measured on Sun UltraSPARC E250, 400 MHz. The safety checking time is measured on 2.2 GHz Pentium 4.

Benchmark	redblack	life
SML/NJ Compile time	0.300	0.490 sec.
SML/NJ Run time	0.013	0.262
FPCC Compile time	0.955	2.998
FPCC Run time	0.014	0.407
FPCC/SMLNJ slowdown	1.036	1.555
Safety check time in		
SICStus	0.183	0.432
Flit	1.32	2.19
Twelf	1018	>3600
Sparc instrs.	870	1816
LTAL tokens	34278	57670
Coercion tokens	17%	23%

Table 3.2: FPCC/ML compiler, LTAL, and Flit performance.

compiler algorithms as necessary for a production compiler. Run time is almost as good as SML/NJ. Currently we do not garbage collect; SML/NJ spends 0.02% of its time garbage-collecting on these benchmarks. SML/NJ's better performance is probably because it has more sophisticated liveness-based closure conversion and fills branch-delay slots. Other than that, the optimizations performed by the FPCC/ML compiler are just about as sophisticated and comprehensive as those of SML/NJ.

To measure *safety check time,* we translate our lemmas into Prolog rules and time the execution in SICStus Prolog. As an alternative, we have built a minimal-size interpreter, Flit, for syntax-directed lemmas; it is much simpler than Prolog because it doesn't require backtracking [Appel et al., 2002; Wu et al., 2003]. Twelf also builds in a logic programming engine. We measured the safety checking time in all three systems. Twelf is not designed for performance, but its advanced features make it a convenient tool for us to develop machine-checkable proofs in LF. Flit is about five times slower than the optimizing SICStus Prolog, and is fast enough for

the intended application. The performance of proof-checking in Flit will be further discussed in Chapter 5.

Simple encodings should be able to represent LTAL in a few bits per token, so the LTAL expression should not be significantly bigger than the machine-language program. At present, however, we represent LTAL expressions as LF terms, and encode them in the form of directed acyclic graphs (DAGs) [Appel et al., 2002, 2003]. Eliminating the LTAL coercions—thus requiring some backtracking in the type checker—could save about 20% in LTAL size. The builders of SpecialJ [Colby et al., 2000] and TALx86 [Morrisett et al., 1999a] have devoted substantial effort to reducing proof size—not just removing coercions but getting the checker to reconstruct other data as well. Clearly, there is some engineering to be done in this respect, although we would not want to complicate any part of the checker that is in the trusted base.

3.12 Related Work

TAL [Morrisett et al., 1998, 1999b] demonstrated the idea of typed assembly language, but was too limited for practical programming languages. Extensions of this work supported stack allocation [Morrisett et al., 2002] and implemented a more realistic calculus (TALx86) [Morrisett et al., 1999a] for compiling a safe C-like language to Intel IA32 assembly language. DTAL [Xi and Harper, 2001] added a restricted form of dependent types to TAL to support array bound check elimination and datatype tag discrimination. These implementations have soundness proved by hand about abstractions of subsets of the systems that are actually implemented; the proofs cannot be machine-checked. These TALs each have a macroinstruction

"malloc" for heap allocation, and TALx86 has another macro "btagi" which tests tags and branches.

Hamid et al. [2002] proposed a syntactic approach to build machine-checkable foundational proofs. They designed Featherweight Typed Assembly Language (FTAL), mapped each valid machine state to a well-typed FTAL program, and related transition of machine states to evaluation of FTAL programs by a machine-checked syntactic metatheorem. Crary [2003] has built a more substantial TALT, with a machine-checked syntactic metatheorem proving progress and preservation; he uses simulation to relate his typed calculus to the "bare machine" untyped step relation.

Chapter 4

Machine-Checkable Soundness Proofs for LTAL

In this section, we give an overview of the semantic techniques we used to build a machine-checkable soundness proof for LTAL. We give semantic models to types, instructions, and typing judgements, and prove type checking rules as lemmas with respect to machine specification and logic axioms. The semantic models allow a type checking derivation to be interpreted as a machine-checkable safety proof at the machine level.

4.1 Overview

In both Necula's PCC [Necula and Lee, 1996; Necula, 1997] and Morrisett's TAL [Morrisett et al., 1998, 1999b] systems, type checking rules are trusted as axioms. In other words, the type systems used in their systems do not have a (machine-checkable) soundness proof. For example, in the TAL system, there are 13 kinds

of typing judgements and many typing rules are with similar complexity as the following one:

$$\frac{\Psi;\Delta;\Gamma \ \vdash_{\text{TAL}} v : \forall[\,].\Gamma' \quad \Delta \ \vdash_{\text{TAL}} \Gamma \leq \Gamma'}{\Psi;\Delta;\Gamma \ \vdash_{\text{TAL}} \texttt{jmp}\ v} \text{ (s-jmp)}$$

This rule is intuitively "correct" based on the semantics of the jump instruction, and there is a paper-and-pencil proof of soundness [Morrisett et al., 1999b]. However, the TAL described in their published paper is not the TALx86 [Morrisett et al., 1999a] that they actually implemented. Any misunderstanding of the semantics could lead to errors in the type system. League et al. [2003] found an unsound proof rule in the SpecialJ [Colby et al., 2000] type system. In the process of refining our own TAL [Chen et al., 2003], we routinely find and fix bugs that can lead to unsoundness.

Since errors in the Trusted Computing Base (TCB) can be exploited by malicious code, it is useful to minimize the TCB. A foundational approach is to move the entire type system out of TCB by proving its soundness and by verifying that type-checking implies the safety theorem. We give models to types and judgements so that both typing rules and the type-safety theorem can be proved and mechanically verified in a theorem-proving system [Appel, 2001; Wu et al., 2003; Tan et al., 2004].

The rest of this chapter is organized as follows. We first introduce the logic and logical framework we used to build machine checkable proofs in Section 4.2. We then describe the machine architecture specification in Section 4.3. The safety specification is presented in Section 4.4. After that, we build semantic models for types in Section 4.5, and prove the soundness of LTAL in Section 4.6. Finally, we present our implementation in Section 4.7 and discuss related work in Section 4.8.

4.2 Logic and Logical Framework

In order to build machine checkable proofs, one must choose a formal logic and an implementation of the logic to manipulate proofs written in the logic. We choose higher-order logic as the object logic since it is expressive and permits concise proofs. The LF logical framework [Harper et al., 1993] is chosen as the meta-logic to encode higher-order logic.

LF is a dependent type theory based on λ-calculus with type families and $\beta\eta$-equality. It has three levels of terms: objects, types, and kinds. Types classify objects and kinds classify type families. LF is a framework for defining logics [Harper et al., 1993]. The framework is general enough to represent logics of interest; we use it to encode higher-order logic [Appel, 2001].

```
tp   : type.
tm   : tp -> type.
form : tp.
num  : tp.
arrow: tp -> tp -> tp.    %infix right 14 arrow.
pair : tp -> tp -> tp.
pf   : tm form -> type.
```

In LF, "`type`" is a keyword for declaring an LF type (meta-logical type), and "`->`" is the meta-logical function type. In the above LF code, `tp` is declared as a type in the meta logic LF, and it classifies object logic types. Our object logic has primitive types `form` and `num` for formulas and numbers, respectively. The constructor `tm` converts a term of object logic type T (T is of meta-logical type `tp`) into a term of meta-logical type `tm` T. For any formula A of meta-logical type `tm form`, proofs of A are encoded as terms of meta-logical type `pf(`A`)`. The constructors `arrow` and `pair` are used to build function types and tuples, respectively, in the object logic. The constructor `arrow` is declared infix to make it more readable.

Then we introduce constructors and definitions in our object logic, and prove lemmas based on them.

```
lam : (tm T1 -> tm T2) -> tm (T1 arrow T2).
@   : tm (T1 arrow T2) -> tm T1 -> tm T2.
      %infix left 20 @.
imp : tm form -> tm form -> tm form.
      %infix right 10 imp.
forall : (tm T -> tm form) -> tm form.

imp_i: (pf A -> pf B) -> pf (A imp B).
imp_e: pf (A imp B) -> pf A -> pf B.

forall_i: ({x:tm T} pf (A x)) -> pf (forall A).
forall_e: pf (forall A) -> {x:tm T} pf (A x).

and : tm form -> tm form -> tm form =
  [a][b] forall [c] (a imp b imp c) imp c.
  %infix right 12 and.

and_i: pf A -> pf B -> pf (A and B) =
 [p1: pf A]
 [p2: pf B]
 forall_i [c: tm form]
 imp_i [p3] imp_e (imp_e p3 p1) p2.

and_e1: pf (A and B) -> pf A =
  [p1: pf (A and B)]
  imp_e (forall_e p1 A)
  (imp_i [p2: pf A] imp_i [p3: pf B] p2).

imp_trans : pf (A imp B) -> pf (B imp C) -> pf (A imp C) =
  [p1][p2] imp_i [p3] imp_e p2 (imp_e p1 p3).

imp_refl : pf (A imp A) = imp_i [p1] p1.

imp_true : pf B -> pf (A imp B) = [p1] imp_i [p2] p1.
```

The `lam`, `@`, `imp`, and `forall` are constructors for λ-abstraction, function application, logical implication, and universal quantification in our object logic. Next, we define the introduction and elimination rules (e.g. `imp_i` and `imp_e`) for these constructors. Finally, we can introduce definitions and lemmas based on construc-

tors previously defined. The definition `and` and its introduction and elimination rules (`and_i` and `and_e1`) are type checked for validity. The lemmas `imp_trans`, `imp_refl`, and `imp_true` are proved and checked.

The proof checking in LF is based on the *formulae-as-types* principle (as know as Curry-Howard correspondence) [Howard, 1980]. A formula or theorem is encoded as a type in the LF type theory, and a proof of the theorem is an LF term of the LF types that encodes the theorem. Thus, the proof checking in the object logic is reduced to the LF type checking.

Twelf [Pfenning and Schürmann, 1999, 2002] is an implementation of LF. We use Twelf for our development of machine checkable proofs. Twelf has many useful features, such as type inference and mode analysis, which make it a convenient tool for us to develop and manipulate machine-checkable proofs in higher-order logic (encoded in the meta-logic LF).

With many advanced features, Twelf is a very good choice for our development. Twelf is not, however, trustworthy or minimal in terms of system size and features. Because we want to build high-assurance system and don't want to include a large proof checker in the TCB, we implement a simple yet efficient LF proof checker in Flit, which is presented in Chapter 5. Flit also implements a simple logic programming engine for efficient proof checking [Wu et al., 2003].

4.3 Machine Instruction Specification

Our machine model consists of a set of formulas in higher-order logic that specify the decoding and operational semantics of instructions. Our safety policy specifies which addresses may be loaded and stored by the program (memory safety) and

defines what the code safety means. Our machine model and safety policy are trusted and are small enough to be "verifiable by inspection."

In our model, a machine state (r, m) consists of a register bank r and a memory m, which are modeled as functions from numbers (register numbers and addresses) to numbers (contents). A machine instruction is modeled by a relation between two machine states (r, m) and (r', m') before and after execution of the instruction [Michael and Appel, 2000]. For example, the add instruction $r_i \leftarrow r_j + r_k$ is modeled as the following relation:[1]

$$\mathrm{add}(i, j, k) \stackrel{\mathrm{def}}{=}$$
$$\lambda r, m, r', m'.\ r'(i) = r(j) + r(k) \ \wedge\ (\forall x \neq i.\ r'(x) = r(x)) \wedge\ m' = m$$

Since we want to prove safety of machine code, which is just a sequence of integers (representing machine instructions), we must model the decode relation to connect instruction words to their actual meanings. The decode rule in Section 3.3.1 illustrates the idea, but we need to model the decode relation for every instruction of the machine. The decode relation is specified as follows: Some number w decodes to an instruction *instr* if [Michael and Appel, 2000; Appel, 2001]

[1] Our step relation first increments the program counter pc, then executes an instruction. Thus, the semantics of add instruction does not include the semantics of incrementing the pc.

$$
\begin{aligned}
&\mathrm{decode}(w, \mathit{instr}) \stackrel{\text{def}}{=} \\
&\quad (\exists i, j, k. \\
&\qquad 0 \le i < 2^5 \ \wedge \ 0 \le j < 2^5 \ \wedge \ 0 \le k < 2^5 \ \wedge \\
&\qquad w = 3 \cdot 2^{26} + i \cdot 2^{21} + j \cdot 2^{16} + k \cdot 2^0 \ \wedge \\
&\qquad \mathit{instr} = \mathrm{add}(i, j, k)) \\
&\quad \vee (\exists i, j, c. \\
&\qquad 0 \le i < 2^5 \ \wedge \ 0 \le j < 2^5 \ \wedge \ 0 \le c < 2^{16} \ \wedge \\
&\qquad w = 12 \cdot 2^{26} + i \cdot 2^{21} + j \cdot 2^{16} + c \cdot 2^0 \ \wedge \\
&\qquad \mathit{instr} = \mathrm{load}(i, j, c)) \\
&\quad \vee \ \ldots
\end{aligned}
$$

where the ellipsis denotes many other instructions of the machine.

The machine operational semantics is modeled by a step relation $\mapsto$ that steps from one state (r, m) to another state (r', m') [Michael and Appel, 2000], where the state (r', m') is the result of first decoding the current machine instruction, incrementing the program counter and then executing the machine instruction.

$$
\begin{aligned}
(r, m) \mapsto (r', m') \ \stackrel{\text{def}}{=} \ & \exists \mathit{instr}.\ \mathrm{decode}(r(\mathrm{pc}), \mathit{instr}) \\
& \wedge \ \mathrm{upd}(r, \mathrm{pc}, r(\mathrm{pc}) + 4, r'') \\
& \wedge \ \mathit{instr}(r'', m, r', m')
\end{aligned}
$$

where upd predicate increments the program counter pc (assuming the instruction size is 4), and the result register bank state is r''.

An important property of our step relation is that it is deliberately partial: It omits any step that would be illegal under the safety policy. For example, the load instruction is specified by

$$\text{load}(i,j,c) \stackrel{\text{def}}{=} \lambda r, m, r', m'.\ r'(i) = m(r(j)+c) \ \wedge\ (\forall x \neq d.\ r'(x) = r(x)) \\ \wedge\ m' = m \ \wedge\ \text{readable}(r(j)+c).$$

Suppose in some state (r, m) the program counter points to a load instruction that would, if executed, load from an address that is unreadable according to the safety policy. Then, since our load instruction requires that the address must be readable, there will not exist (r', m') such that $(r, m) \mapsto (r', m')$.

4.4 Safety Specification

As stated in the previous section, our step relation is deliberately partial; some states, in which the program counter $r(\text{pc})$ points to an illegal instruction or $r(\text{pc})$ points to a legal machine instruction that violates our safety policy, have no successor states. This mixing of machine semantics and safety policy is to follow the standard practice in type theory [Wright and Felleisen, 1994] so that we can get a clean and uniform definition of safety property.

Using the partial step relation, we can define a safe machine state as a state that cannot lead to a stuck state.

$$\text{safe-state}(r,m) \stackrel{\text{def}}{=} \\ \forall r', m'.\ \ (r,m) \mapsto^* (r',m') \ \ \Rightarrow\ \exists r'', m''.\ \ (r',m') \mapsto (r'',m'')$$

where $\mapsto^*$ denotes zero or more steps.

To show safe-state(r, m), it suffices to prove that the state is "safe for n steps," for any natural number n.

$$\text{safe-n-state}(n, r, m) \stackrel{\text{def}}{=}$$
$$\forall r', m'.\ \forall j < n.\ (r, m) \mapsto^j (r', m') \ \Rightarrow\ \exists r'', m''.\ (r', m') \mapsto (r'', m'')$$

where $\mapsto^j$ denotes j steps being taken.

A machine-language program is just a sequence of integers (each representing a machine instruction); we state that a program p is loaded at a location l in memory m if

$$\text{loaded}(p, m, l) \stackrel{\text{def}}{=} \forall i \in \text{dom}(p).\ m(i + l) = p(i)$$

Finally we define program safety as follows. Assume that programs are written in position-independent code. A program is *safe* if, no matter where we load it in memory and the machine state meets the initial precondition ϕ_0, we get a safe state [Appel, 2001]:

$$\text{safe}(p) \stackrel{\text{def}}{=}$$
$$\forall r, m, \mathit{start}.\ \text{loaded}(p, m, \mathit{start}) \ \wedge\ r(\text{pc}) = \mathit{start} \ \wedge\ (m, r) : \phi_0$$
$$\Rightarrow\ \text{safe-state}(r, m)$$

The initial precondition ϕ_0 specifies, among other things, the initial state of the register bank and memory. It also states that the return address is a label to which it is safe to jump. Our current initial precondition is simple enough such that it can be described in our logic without type constructors since LTAL or other types are not in the TCB. We are currently investigating how to augment the TCB with

some type constructors so that more sophisticated initial precondition and runtime interface can be specified.

The safety theorem is parametrized by the application program. We only need two operators to specify a machine-language program, that is, cons and nil for sequences of integers representing machine instructions. Let ; be the "cons" operator. For some machine-language program[2]

```
2551193600 ;
2181292040 ;
2214748172 ;
2416058369 ;
2450522113 ;
2176860160 ;
16777216 ;
nil
```

the safety theorem is

```
safe ( 2551193600 ; ... ; 16777216 ; nil )
```

Suppose *PROOF* is a proof of the above theorem. To check the validity of the proof, we type check the following LF term:

```
safe_thm: pf (safe (2551193600 ; ... ; 16777216 ; nil)) = PROOF.
```

Appel et al. [2003] measured the size of safety specification in our system. The result is shown in Table 4.1. In our safety specification, there are 1,206 definitions encoded in 1,865 lines of code in Twelf. The definitions used directly or indirectly to specify the safety theorem need to be trusted, and thus are part of the safety specification. Therefore, all definitions and constructors up to the definition of `safe` are part of trusted code base. On the other hand, definitions specified after the

[2]This is the SPARC program compiled from the ML function `fun f(x)=x+1` by our FPCC/ML compiler.

Safety Specification	*Lines*	*Definitions*
Logic	135	61
Arithmetic	160	94
Machine Syntax	460	334
Machine Semantics	1,005	692
Safety Predicate	105	25
Total	1,865	1,206

Table 4.1: Safety specification.

definition of `safe` and used in the proof of the safety theorem do not need to be trusted since they are defined and checked for validity before they are used in other definitions and proofs.

4.5 Semantic Models of Types

In this section, we give a brief description of the semantic models of types [Appel and Felty, 2000; Appel and McAllester, 2001].

Appel and Felty [2000] build set-theoretic models for types. A *state* is a pair (a, m), where m is a memory (including the register bank) and a is the set of allocated addresses of dynamic memory allocation. A *value* is a pair (s, x) of a state s and an integer x (typically representing an address or root pointer). This model can handle records, addresses arithmetic, function pointers, intersection and union types, covariant recursive types, etc., but cannot handle contravariant recursive types.

Appel and McAllester [2001] invented the *indexed* model of types that can describe contravariant recursive types. In the indexed model, a type is not a set of values; instead, it is a set of pairs (k, v), where k is the approximation index (non-

$$
\begin{array}{rcl}
\mathsf{int} & \stackrel{\text{def}}{=} & \{\langle k, m, x\rangle \mid \text{true}\} \\
\mathsf{int}_{=}(\overline{n}) & \stackrel{\text{def}}{=} & \{\langle k, m, x\rangle \mid x = n\} \\
\mathsf{box}\,(\tau) & \stackrel{\text{def}}{=} & \{\langle k, m, x\rangle \mid x \in \text{dom}(m) \wedge \text{readable}(x) \\
 & & \qquad \wedge\ \langle k-1, m, m(x)\rangle \in \tau\} \\
\mathsf{field}(i, \tau) & \stackrel{\text{def}}{=} & \{\langle k, m, x\rangle \mid (x+i) \in \text{dom}(m) \wedge \text{readable}(x+i) \\
 & & \qquad \wedge\ \langle k-1, m, m(x+i)\rangle \in \tau\} \\
\mathsf{codeptr}(\phi) & \stackrel{\text{def}}{=} & \{\langle k, m, x\rangle \mid \forall j, r.\ j < k\ \wedge\ r(\text{pc}) = x\ \wedge\ (m, r) :_j \phi \\
 & & \qquad \Rightarrow\ \text{safe-n-state}(j, r, m)\}
\end{array}
$$

Figure 4.1: The indexed model of types.

negative integer) and v is a value. Intuitively, a pair $(k, v) \in \tau$ means the value v has type τ within k steps of computation; that is, it k-approximately belongs to type τ.

Types are defined such that they are closed under approximation; that is, if $(k, v) \in \tau$, then $(j, v) \in \tau$ for any $j < k$. We use $v :_k \tau$ as a syntactic sugar for $(k, v) \in \tau$. We write $v : \tau$ to mean $v :_k \tau$ is true for any k. A value v is a tuple (a, m, x) of the set of allocated addresses, memory (including the register bank), and an integer (typically denoting the root address). For the sake of simplicity, sometimes we omit the allocated-address set, and use $\text{dom}(m)$ instead when necessary. The indexed model of some types is shown in Figure 4.1.

Any value is of type int since any memory content can be viewed as an integer or binary number. The type $\mathsf{int}_{=}(\overline{n})$ specifies that the integer is exactly n. The type $\mathsf{box}\,(\tau)$ states that the root pointer is a readable address whose content is of type τ under approximation $k - 1$. This is because it takes one computation step (a memory load instruction) to dereference a boxed value. The meaning of field type is similar except that there is an offset. The $\mathsf{codeptr}$ type states that the root address is a label to which it is safe to jump. Specifically, it says that if the current

$$
\begin{array}{rcl}
\top_\phi & \stackrel{\text{def}}{=} & \{\langle k, m, \vec{x}\rangle \mid \text{true}\} \\
\bot_\phi & \stackrel{\text{def}}{=} & \{\langle k, m, \vec{x}\rangle \mid \text{false}\} \\
\{n : \tau\} & \stackrel{\text{def}}{=} & \{\langle k, m, \vec{x}\rangle \mid \langle k, m, x_n\rangle \in \tau\} \\
\phi_1 \cap \phi_2 & \stackrel{\text{def}}{=} & \{\langle k, m, \vec{x}\rangle \mid \langle k, m, \vec{x}\rangle \in \phi_1 \wedge \langle k, m, \vec{x}\rangle \in \phi_2\} \\
\phi[n \mapsto \tau] & \stackrel{\text{def}}{=} & \{\langle k, m, \vec{x}\rangle \mid \exists y.\ \langle k, m, \vec{x}[n \mapsto y]\rangle \in \phi \ \wedge \ \langle k, m, x_n\rangle \in \tau\}
\end{array}
$$

Figure 4.2: The indexed model of environments (vector values).

machine state satisfies precondition ϕ with any index $j < k$, it is safe to run j steps starting from the root address x.

Moreover, we often need to judge not only scalar values such as a singleton integer but also vector values such as the register bank type, typing environments, and code pointer preconditions (a list of arguments and their types). Vector values are modeled in a similar way except that the root pointer is a vector, a function from numbers to values. For example, $(m, r) : \phi$ means that the register bank satisfies ϕ. Another use of vector types is the label environment Γ, which summarizes the preconditions of all labels. In this case, $\vec{x}$ is the identity vector id which maps label l to itself. Thus, $(m, \text{id}) : \{l : \mathsf{codeptr}(\phi)\}$ means that label l itself has type $\mathsf{codeptr}(\phi)$.

The indexed model of vector values is shown in Figure 4.2. Any value belongs to $\top_\phi$, and no value belongs to $\bot_\phi$. The singleton environment only cares about the nth slot of the vector and states that its content has type τ. The intersection of two environments $\phi_1 \cap \phi_2$ states that the value satisfies both environments. The extension of environment with a new binding is represented as $\phi[n \mapsto \tau]$.

With the semantic model of types and environments, the typing rules can be proved as lemmas [Appel and Felty, 2000; Appel and McAllester, 2001]. For example, the following codeptr elimination rule is proved as a lemma.

$$\frac{(m,x) :_{k+1} \mathsf{codeptr}(\phi) \quad r(\mathrm{pc}) = x \quad (m,r) :_k \phi}{\text{safe-n-state}(k,r,m)}\ \textit{CodePtr-E}$$

The rule means that if (1) value (m, x) is of type $\mathsf{codeptr}(\phi)$ to approximation $k+1$, (2) the current program counter is x, and (3) the current memory and register bank (m, r) meets ϕ to approximation k, then it is safe to execute k steps under the current machine state (m, r).

4.6 Safety Proof

Figure 4.3 shows a program, in the LTAL and SPARC assembly language, compiled by the FPCC/ML compiler. The LTAL program has two basic blocks, each of which is annotated by a precondition. To make it simple and readable, we have omitted some type annotations, such as coercions, inside the basic blocks. We have also omitted the map from variables to registers.

Type annotations are generated by a type-preserving compiler from source language types. They serve as a specification (types as specifications). On the other hand, these type annotations are not verified yet; they are the invariants that the compiler believes. They need to be verified through a sound type system.

In the LTAL type system, we have typing judgements for programs, basic blocks, individual instructions, and so on. In order to prove type checking rules as lemmas, we must define the meaning of typing judgements. Informally, we define the following models. The model of the program typing judgement is that all labels are safe to execute with respect to their preconditions (although it suffices to ensure that the start label is safe). The model of the basic block typing judgement is that the

LTAL	SPARC
$l_0:$ ϕ_0	`l0:`
$v_{10} = 0$	`mov 0,%o4`
$v_{11} = addradd(v_4, 8)$	`add %o7,8,%g1`
$v_{19} = lbladd(v_2, l_1 - l_0)$	`add %o1,l1-l0,%g2`
$v_{12} = v_{19}$	
$v_8 = v_{10}$	
$v_7 = v_{11}$	
$v_6 = v_{10}$	
$v_5 = v_3$	
$calln(l_1)$	
$l_1:$ ϕ_1	`l1:`
$v_9 = v_5 + 1$	`add %o0,1,%o0`
$v_{13} = open(v_7)$	
$v_{14} = v_{13}$	`mov %g1,%o1`
$v_{15} = v_{13}$	
$v_{16} = v_{14}$	
$v_{17} = v_6$	
$v_{18} = v_9$	
$call(v_{15})$	`jmp [%g1+%g0]`
	`nop`

Figure 4.3: An example LTAL program. (This program is compiled by the FPCC/ML compiler from ML function "`fun f(x)=x+1`".)

basic block in consideration is safe for at least $k+1$ steps assuming all the other basic blocks are safe for k steps. Let's define

$$\begin{aligned}
\mathsf{instr}\,(i) &\stackrel{\text{def}}{=} \{\langle k, m, x\rangle \mid \text{decode}(m(x), i)\} \\
\Delta &\stackrel{\text{def}}{=} \{0 : \mathsf{instr}\,(i_0)\} \cap \{4 : \mathsf{instr}\,(i_1)\} \cap \ldots \\
\Gamma &\stackrel{\text{def}}{=} \{l_0 : \phi_0\} \cap \{l_1 : \phi_1\} \cap \ldots \\
\Delta \subset \Gamma &\stackrel{\text{def}}{=} \forall k, m.\ (m, \text{id}) :_k \Delta \ \Rightarrow (m, \text{id}) :_k \Gamma
\end{aligned}$$

The $\mathsf{instr}\,(i)$ is an indexed type that relates a memory content and the instruction it represents. The Δ is the indexed type representation of machine-language

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{loaded}(p,m,0)}{(m,\text{id}):\Delta}\,(7) \quad \dfrac{\vdots}{\Delta \subset \Gamma}\,(8) \quad \Gamma \subset \{0 : \mathsf{codeptr}(\phi_0)\}}{(m,\text{id}) : \{0 : \mathsf{codeptr}(\phi_0)\}}\,(6)}{(m,0) : \mathsf{codeptr}(\phi_0)}\,(5)}{\forall k.\ (m,0) :_k \mathsf{codeptr}(\phi_0)}\,(4)}{(m,0) :_{k+1} \mathsf{codeptr}(\phi_0)}\,(3a) \quad \dfrac{(m,r) : \phi_0}{\forall k.\ (m,r) :_k \phi_0}\,(3b) \quad r(\text{pc}) = 0}{\forall k.\ \text{safe-n-state}(k,r,m)}\,(2)}{\text{safe-state}(r,m)}\,(1)$$

Figure 4.4: Outline of safety proof.

program (a sequence of integers encoding machine instructions). The Γ is the label environment (precondition) for each label (basic block) in the code. The subtyping relation $\Delta \subset \Gamma$ states that code Δ respects invariant Γ under any approximation k, which is exactly the meaning of the program typing judgement. Note that id is the identity vector which maps root addresses (labels in this case) to themselves.

The proof outline is shown in Figure 4.4. This is a proof of safe(p) according to its definition. The assumptions are loaded$(p, m, 0)$, $r(\text{pc}) = 0$, and $(m, r) : \phi_0$. For the sake of simplicity, we assume the start address is 0 here. For Step (1), to prove (r, m) is safe, it suffices to prove that (r, m) is safe for an arbitrary k steps. Step (2) is justified by rule *CodePtr-E* in Section 4.5. Step (3a) is by universal instantiation. Step (3b) is by definition. Step (4) is the unfolding of the syntactic sugar of $(m, 0) : \mathsf{codeptr}(\phi_0)$. Step (5) is by the definition of the singleton environment. Step (6) is by the transitivity of subtyping. Step (7) can be easily proved by unfolding definitions. Step (8) is by type checking the corresponding LTAL and SPARC program. The model of program typing judgement should be strong enough to prove (8). Please see Chapter 3 and Section 5.3.4 for the LTAL

type checking rules that establish this subtyping relation. Tan [2005] and Tan et al. [2004] explain in more detail the semantic models that construct the proof of this subtyping relation.

We briefly explain the semantic techniques we used to establish the subtyping relation in Step (8) above. The subtyping relation $\Delta \subset \Gamma$ can be proved by induction over the number of execution steps that are safe starting from labels in Γ. The base case is trivial because it is safety to run zero step from any label. In the inductive case, for the current label l we prove that it is safe to execute at least $k+1$ steps assuming that it is safety to execute at least k steps starting from any label. This is established by checking individual instructions in the current block. Suppose there is at least one real (not virtual) instruction in the current block. Then we can prove that it is safe to run at least one step starting from the current label by checking individual instructions in the current block. Furthermore, the last instruction in the current block must be a branch, jump, or return instruction that transfers the control to some label l'. By induction, we know that it is safe to execute at least k steps starting from label l'. Therefore, we conclude that it is safe to execute at least $k+1$ steps starting from the current label l. We check this for each label, and thus establish the subtyping relation as required.

Take the LTAL program in Figure 4.3 for example, there are two labels, l_0 and l_1, and two basic blocks. The block labeled l_0 has three real instructions and six virtual instructions (coercions). The block l_1 has four real instructions, including the `nop` instruction at the end, and five virtual instructions. For the base case $k = 0$, it is trivially true because any program is safe to run 0 step. For the inductive case, we assume that the labels l_0 and l_1 are both safe for executing k steps starting from them, respectively. By checking the real instructions in the body of basic block l_0,

we conclude that it is safe to execute at least 3 steps starting from l_0. Since the last instruction is fall-through to block l_1, which is safe for k steps, we conclude that it is safe to execute at least $k+1$ steps starting from l_0. Similarly, we can conclude that it is safe to execute at least $k+1$ steps starting from label l_1. What is worth mentioning is that the last instruction of the basic block labeled l_1 is a `jmp` instruction. The target label is in a register since variable v_{15} is in a register. This target label is actually the return address to which it is safe to jump as we specify in the initial precondition ϕ_0. Thus we conclude that label l_1 is also safe. By induction on the number k, we conclude that the program is safe for executing any number of steps.

Building a semantic model for a large calculus such as LTAL and proving its soundness are rather intricate. Interested readers should refer to several papers and PhD theses [Appel and Felty, 2000; Appel and McAllester, 2001; Ahmed et al., 2002; Ahmed, 2004; Swadi, 2003; Tan et al., 2004]. Appel and Felty [2000] and Appel and McAllester [2001] present an (indexed) semantic model of types; Ahmed et al. [2002] and [Ahmed, 2004] extend the model for general reference types; Swadi [2003] introduces Typed Machine Language (TML) and builds an abstraction layer on which the semantic model of LTAL is based; Tan et al. [2004] and Tan [2005] give a more detailed description of the semantic models of machine instructions and basic blocks in LTAL.

4.7 Implementation

Proofs are written and machine-checked in the theorem-proving system—Twelf [Pfenning and Schürmann, 1999, 2002] and Flit [Appel et al., 2002; Wu et al., 2003].

Currently, we have 1865 lines of axioms of the logic, arithmetic, the specification of the SPARC machine, and the safety specification; and about 165k lines of proof in total including lemmas of logic, arithmetic, sets, lists, conventions of machine states, semantic model of types, machines instructions, and the LTAL calculus.

We have built many layers of abstraction to make the large proof implementation as modular as possible. Abstract modules such as mathematic sets, lists, and theory of arithmetic are developed. These modules are all based on higher-order logic implemented in LF. We have also implemented conventions of machine states and semantic model of types and machine instructions. Among these abstractions, two significant ones are TML and LTAL. TML is an expressive typed calculus for proving properties of low-level programs such as machine code, but it does not have an efficient type checking algorithm since the type system is too expressive to have decidable type checking algorithm. LTAL, however, has a syntax-directed type checking algorithm as we presented in Chapter 3. LTAL is also designed for checking low-level programs, but admits efficient type checking. TML is useful for building semantic models of low-level calculus such as LTAL and for proving its soundness, while LTAL is efficient enough to be used as the interface between the compiler and the proof checker.

Table 4.2 presents the breakdown of the proofs in our system according to abstractions and modules. The first column is modules. The number in the second column is the lines of code including comments and blank lines. The number of lines of code in the third column does not include comments and blank lines. In total, we have approximately 143.4k lines of code, not including comments and blank lines. In particular, the syntactic implementation of LTAL in LF is about 4,160 lines of code. These are constructors declarations and Prolog-like clauses that encode the

Modules	*Lines*	*Useful Lines*
Safety specification	2,939	1,957[3]
Logic, arithmetic, & algebra	7,744	5,241
Sets, relations, & partial functions	9,229	8,086
Lists, vectors, trees	12,338	11,024
Miscellaneous	6,239	5,522
Machine conventions	25,890	22,321
Machine states	4,014	3,500
Abstract machine instructions	9,358	8,294
TML	54,800	48,124
LTAL	34,378	29,345
LTAL (syntax only)	6,641	4,163
Total	166,929	143,414

Table 4.2: FPCC proof statistics.

LTAL type checking rules. The breakdown of these constructors and rules is shown in Table 3.1.

4.8 Related Work

The semantic approach to proving the soundness of logical and type systems has been around for decades. Schmidt [1986], Gordon [1988], and Wahab [1998] prove the soundness of Hoare logic based on the denotational semantics. Such verification has been mechanized in HOL [Gordon, 1988]. Loop invariants are specified in first-order or higher-order logic and cannot be derived automatically, so the approach does not scale to large programs.

Appel and Felty [2000] apply the semantic approach to PCC and construct a semantic model to types and machine instructions in higher-order logic, and proved

[3]This number includes the axioms for floating point number arithmetic, while the number presented in Table 4.1 and reported by Appel et al. [2003] does not.

soundness by proving the typing rules as lemmas. This semantic model has been extended to include general recursive types [Appel and McAllester, 2001] and mutable references [Ahmed et al., 2002].

Hamid et al. [2002] and Crary [2003] follow the syntactic approach to prove type soundness. The syntactic approach has two stages. First, a typed assembly language is designed and its operational semantics is specified on top of an abstract machine. Then the syntactic type-soundness theorems are proved on this abstract machine following the scheme presented by Wright and Felleisen [1994]. At the second stage, they use a relation to simulate the operations between the typed abstract machine and the untyped concrete architecture.

Chapter 5

Foundational Proof Checking with Small Witnesses

Proof checkers for proof-carrying code (and similar systems) can suffer from two problems: huge proof witnesses and untrustworthy proof rules. No previous design has addressed both of these problems simultaneously. In this chapter, we show the theory, design, and implementation of a proof-checker that permits small proof witnesses and machine-checkable proofs of the soundness of the system.

5.1 Introduction

In a proof-carrying code system [Necula, 1997], or in other proof-carrying applications [Appel and Felten, 1999], an untrusted prover must convince a trusted checker of the validity of a theorem by sending a proof. Two of the potential problems with this approach are that the proofs might be too large, and that the checker might not be trustworthy. Each of these problems has been solved separately; in this chapter

we show how to solve them simultaneously. The general approach is to write a logic program that has a machine-checked semantic correctness proof. The logic program encodes the proof inference system. This technique can be used in other domains (besides "proof-carrying") to write logic programs with machine-checked guarantees of correctness.

5.1.1 Small proof witnesses

Necula has a series of results on reducing proof size [Necula and Lee, 1998; Necula and Rahul, 2001]. He represents logics, theorems, and proofs in the LF logical framework [Harper et al., 1993]. But the natural representation of an LF proof contains redundancy (common subexpressions) that can cause exponential blowup if the proofs are written in the usual textual representation. Necula's LF_i data structure [Necula and Lee, 1998] eliminated most of this redundancy, leading to reasonable-sized proof terms.

In the PCC framework, given a machine-language program, the proof is of a theorem that the program obeys some safety property. It's natural to compare the size of the representation of the proof witness to the size of the binary machine-language program. Necula's LF_i proof witnesses were about 4 times as big as the programs whose properties they proved.

Pfenning's Elf and Twelf systems [Pfenning, 1994; Pfenning and Schürmann, 1999] are implementations of the LF logical framework. In these systems, proof-search engines can be represented as logic programs, much like (dependently typed, higher-order) Prolog. Elf and Twelf can build proof witnesses automatically if the rule set is encoded as a logic program. If each logic-program clause is viewed as an inference rule, then the proof witness is a trace of the successful goals and subgoals

executed by the logic program. That is, the proof witness is a tree whose nodes are the names of clauses and whose subtrees correspond to the subgoals of these clauses.

Necula's theorem provers were written in this style, originally in Elf and later in a logic-programming engine that he built himself. In later work, he moved the prover clauses into the trusted checker. In principle, proof witnesses for such a system can be just a single bit, meaning, "A proof exists: search and ye shall find it." However, to guarantee that proof-search time (in the trusted checker) would be small, Necula invented *oracle-based checking* [Necula and Rahul, 2001]: The untrusted prover would record a sequence of bits that recorded which subgoals failed (and therefore, where backtracking was required). This bitstream serves as an "oracle" that the trusted checker can use to avoid backtracking. The oracle bitstream need not be trusted; if it is wrong, then the trusted checker will choose the wrong clauses to satisfy subgoals, and will fail to find a proof.

Using oracle-based checking, the proof witness (the oracle bitstream) is about 1/8 the size of the machine code.[1] The key idea is to run a simple Prolog engine in the trusted proof checker; the oracle is just an optimization to ensure that the checker doesn't run for too long.

[1]Unfortunately, this statistic is somewhat misleading. A "pure" PCC system would transmit two components from an untrusted code producer to a code consumer: a machine-language program and a proof witness. The SpecialJ proof-carrying Java system on which Necula measured oracle-based checking transmits three components: the machine code, the proof, and a Java "class file". The Java class file, as is usual in any Java system, contains descriptions of the types of all procedures (methods) in the program, including formal parameter and result types. These method types help guide the proof search. However, the "1/8 size" figure does not include the Java class files. In our FPCC/ML system system, all auxiliary type information needed by the checker is contained within the LTAL expressions whose size we report.

5.1.2 Trustworthy checkers

Necula's oracle-based checker for PCC comprises approximately 26,000 lines of code:

23,000	Verification-condition generator, written in C
1,400	LF proof checker, written in C
800	Oracle-based Prolog interpreter, in C
700	Axioms for type system, written in LF
26,000	Total trusted lines of code

The largest component is the verification condition generator (VC-Gen), which traverses the machine-language program and extracts a formula in logic, the *verification condition*, which is true only if the program obeys a given safety policy.

This 26,000 lines forms the trusted code base (TCB) of the system: Any bug in the TCB may cause an unsafe program to be accepted. The large VC-Gen component is a concern, but so are the axioms of the type system: If the type system is not sound, then unsafe programs will be accepted. League et al. [2003] have shown that one of the SpecialJ typing rules is unsound.

The goal of our research [Appel, 2001] is to check proofs of program safety using a much smaller TCB. We do this by eliminating the VC-Gen component—we reason directly about machine code in higher-order logic, instead of the two-step process of extracting the verification condition and then proving it; and we write the rules of our type system as machine-checkable lemmas, instead of axioms. We have shown that the TCB for a proof-carrying code system can be reduced below 2700 lines, as follows [Appel et al., 2002]:

803	LF proof checker, written in C
135	Axioms & definitions of higher-order logic, in LF
160	Axioms & definitions for arithmetic, in LF
460	Specification of SPARC instruction encodings, in LF
1,005	Specification of SPARC instruction semantics, in LF
105	Specification of safety predicate, in LF
2,668	Total trusted lines of code

Unfortunately, in this prototype system the proof witnesses are huge: The DAG representation of a safety proof of a program might be 1000 times as large as the program. Proof size is approximately linear in the size of the program,[2] so this factor of 1000 will not grow substantially worse for larger programs. However, while this early prototype is useful in showing how small the TCB can be made, it is impractical for real applications because the proof witnesses are too big.

5.1.3 Synthesis

We will show that Necula's insight (run a resource-limited Prolog engine in the trusted checker) can be combined with our paranoia (don't trust the logic programming rules used by such a Prolog engine) to make a PCC checker with small witnesses and a small trusted base.

Our approach is as follows. We write a type-checking algorithm in a subset of Prolog with no backtracking and with efficiently indexed dynamic atomic clauses.

[2]Technically, proof size is roughly proportional to the size of the program multiplied by the average number of live variables on entry to a basic block; this is superlinear but much less than quadratic, for typical programs.

We show that the operators of such a Prolog program can be given a semantics in higher-order logic, such that the soundness of each clause can be proved as a machine-checkable lemma. We show that this Prolog subset is adequate for writing efficient type-checkers for PCC and for other "proof-carrying" applications.

Our trusted checker is sent the Prolog clauses, with machine-checkable soundness proofs; it checks these proofs before installing the clauses. Then it is sent a theorem to check (i.e., in a PCC application, the safety of a particular machine-language program) and a small proof witness. The Prolog program traverses the theorem and proof witness; this traversal succeeds only if the theorem is valid.

The TCB size of our new checker is 3034 lines of code, only 366 lines larger than our previous prototype. It mainly includes all the components of our previous system (2668 lines) plus a concise implementation of an interpreter (282 lines of C code) for our Prolog subset.

5.2 Semantic Proofs of Horn Clauses

We will illustrate our approach using an example—a type checker for a very simple programming language. In this example we illustrate the following points, which are common to many proof-carrying applications:

- The specification of the theorem to be proved is quite simple *(in this case, that the program evaluates to an even number).*

- The proof technique involves the definition of a carefully designed set of predicates that allow a simple, syntax-directed decision procedure *(in this case, we define a syntax-directed type system for* evenness *and* oddness*).*

$$
\begin{array}{llll}
\mathit{type} & \tau & ::= & \mathsf{even} \mid \mathsf{odd} \\
\mathit{decl} & d & ::= & \cdot \mid \mathsf{let}\ x = e; d \\
\mathit{expr} & e & ::= & x \mid n \mid e_1 + e_2 \\
\mathit{prog} & p & ::= & (d; e)
\end{array}
$$

Figure 5.1: Syntax of even-odd system.

- The syntax-directed rules are provable, from the definitions of the operators, as machine-checkable lemmas in the underlying higher-order logic (this is what *foundational* means: The rules are provable from the foundations of logic).

- The syntax-directed rules require management of a symbol table, or *context,* that would lead to a quadratic algorithm if implemented naively; we want a linear-time prover, and we'll show how to make one.

- The language being type checked in a proof-carrying code system (or in proof-carrying authentication) is the output of another program—the compiler (or a prover). Such languages don't need all of the syntactic sugar that human-readable languages have, and processing them is therefore easier.

5.2.1 Example: even-valued expressions

Consider a simple calculus for expressions with constants, variables, addition, and let-binding, as shown in Figure 5.1.

A program consists of a list of declarations and an expression. An expression is either a variable, a natural number, or the sum of two expressions. Here is an example:

$$\mathsf{let}\ x = 4\ ;\ \mathsf{let}\ y = x + 8\ ;\ x + y$$

There are two declarations followed by an expression; the program evaluates to 16.

$$
\begin{array}{lcl}
\mathit{Var} & \stackrel{\text{def}}{=} & \mathit{Num} \\
\mathit{State} & \stackrel{\text{def}}{=} & \mathit{Var} \rightarrow \mathit{Num} \rightarrow \mathit{Form} \\
\mathit{Decl} & \stackrel{\text{def}}{=} & \mathit{State} \rightarrow \mathit{Form} \\
\mathit{Exp} & \stackrel{\text{def}}{=} & \mathit{State} \rightarrow \mathit{Num} \rightarrow \mathit{Form} \\
\mathit{Program} & \stackrel{\text{def}}{=} & \langle \mathit{Decl}, \mathit{Exp} \rangle \\
 & & \\
(d; e) & \stackrel{\text{def}}{=} & \langle d, e \rangle \\
\cdot & \stackrel{\text{def}}{=} & \lambda s.\ \text{true} \\
\mathsf{let}\ x = e; d & \stackrel{\text{def}}{=} & \lambda s.\ d\ s \wedge (\forall a. e\ s\ a \Rightarrow s\ x\ a) \\
x & \stackrel{\text{def}}{=} & \lambda s. \lambda a.\ s\ x\ a \\
n & \stackrel{\text{def}}{=} & \lambda s. \lambda a.\ a = n \\
e_1 + e_2 & \stackrel{\text{def}}{=} & \lambda s. \lambda a.\ \exists a_1. \exists a_2.\ e_1\ s\ a_1\ \wedge\ e_2\ s\ a_2\ \wedge\ a = a_1\ \mathit{plus}\ a_2 \\
 & & \\
\mathit{safe} & \stackrel{\text{def}}{=} & \lambda p.\ \forall s.\ \mathit{fst}(p)\ s\ \Rightarrow\ \exists a.\ \mathit{snd}(p)\ s\ a\ \wedge\ \text{isEven}(a)
\end{array}
$$

Figure 5.2: Safety specification.

5.2.2 Safety specification

In this simple example, we define that a "safe" program is one that evaluates to an even number. In order to define the safety theorem, we need to know what a program means and how to evaluate a program. The safety predicate, along with a conventional denotational semantics of the language in consideration, is shown in Figure 5.2.

All of these definitions are treated as axiomatic by our checker; that is, they are *trusted*. We have predefined types *Num* for numbers and *Form* for formulas (or propositions). Variables are represented as numbers. An abstract machine *State* maps a variable to its content, i.e. a number. A program is a pair of a declaration and an expression; its semantics is the pair of semantics of the corresponding declaration and expression.[3] Declaration *Decl* is a predicate on states. Expression

[3]An alternative denotation for a program is a number, resulting from applying the state after the declaration to the expression.

Exp is a predicate on a state and a number; that is, given a state the expression evaluates to a number. The semantics of concrete expressions is straightforward from definitions.

Finally, the safety theorem is based on the semantics of language constructs.[4] Given a program p, it is "safe" if: For any states s, if the declaration of the program, i.e. $fst(p)$, holds on s, then there exists a number a such that the expression of the program, i.e. $snd(p)$, evaluates to a and a is even.

5.2.3 Type checker

The typing rules appear in Figure 5.3. There are three kinds of typing judgements. The judgement for a program $\vdash_p$ checks that the program evaluates to a number whose type is τ. The declaration judgement $\vdash_d$ states that, assuming the environment built so far, and assuming the remaining declarations hold, the expression has a certain type. The expression judgement $\vdash_e$ asserts that an expression has certain type under typing context Γ.

These typing rules can be read as a Prolog-like logic program. Each rule is a clause of the logic program. The conclusion of a rule is the head of the clause, and each premise of the rule is a subgoal. The typing rules are designed such that the conclusions of these typing rules are disjoint. Therefore, when running the type checker (as a logic program) there is no need to backtrack; we say that such a type system is *syntax-directed*.

Furthermore, if we give denotational semantics expressed in higher-order logic to typing judgements such as $\vdash_p$, $\vdash_d$, and $\vdash_e$, each typing rule can be proved as a lemma

[4]For our PCC application, there are only two language constructs for the machine code to be proved safe. The machine code is a sequence of integers encoding machine instructions; so we only need *cons* and *nil*.

$$\frac{\vdash_p p : \mathsf{even}}{safe(p)}\ SafeTy \qquad \frac{\cdot \vdash_d (d; e) : \tau}{\vdash_p (d; e) : \tau}\ ProgTy$$

$$\frac{\Gamma \vdash_e e_1 : \tau_1 \qquad \Gamma[x : \tau_1] \vdash_d (d; e) : \tau}{\Gamma \vdash_d (\mathsf{let}\ x = e_1; d; e) : \tau}\ DeclConsTy$$

$$\frac{\Gamma \vdash_e e : \tau}{\Gamma \vdash_d (\cdot; e) : \tau}\ DeclNilTy \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash_e x : \tau}\ VarTy$$

$$\frac{\Gamma \vdash_e e_1 : \tau_1 \qquad \Gamma \vdash_e e_2 : \tau_2 \qquad \tau_1 \boxplus \tau_2 = \tau}{\Gamma \vdash_e e_1 + e_2 : \tau}\ PlusTy$$

$$\frac{}{\mathsf{even} \boxplus \mathsf{even} = \mathsf{even}}\ \boxplus ee \qquad \frac{}{\mathsf{odd} \boxplus \mathsf{odd} = \mathsf{even}}\ \boxplus oo$$

$$\frac{}{\mathsf{even} \boxplus \mathsf{odd} = \mathsf{odd}}\ \boxplus eo \qquad \frac{}{\mathsf{odd} \boxplus \mathsf{even} = \mathsf{odd}}\ \boxplus oe$$

Figure 5.3: Typing rules with static context.

in the system, thus its soundness is guaranteed with respect to the foundations of logic. The denotational semantics of typing judgements is given in Figure 5.4. Proofs of the typing rules are quite straightforward and thus omitted here. The denotational semantics of the type operators are part of the safety *proof*, not part of the safety specification. That is, they are *not* trusted. It is straightforward to prove the safety theorem from the conclusion of type checking rule *ProgTy* if we pass τ even when invoking the type checker, as shown in the *SafeTy* rule.

Our checker will determine the validity of the safety predicate by determining whether a proof exists. It will not construct such a proof as a data structure; instead, it will traverse a trace of such a proof, composing lemmas in a syntax-directed way. We call our set of lemmas a *type system*: Our machine-checked safety proof of a program P consists of (1) a proof of soundness for the type system, and (2) the successful syntax-directed execution of the typing clauses as applied to P.

$$
\begin{array}{lcl}
\mathit{Ty} & \stackrel{\text{def}}{=} & \mathit{Num} \rightarrow \mathit{Form} \\
\mathit{Env} & \stackrel{\text{def}}{=} & \mathit{State} \rightarrow \mathit{Form} \\
\mathsf{even} & \stackrel{\text{def}}{=} & \lambda x.\exists n.\ \mathrm{isInt}(n) \wedge x = 2n \\
\mathsf{odd} & \stackrel{\text{def}}{=} & \lambda x.\exists n.\ \mathrm{isInt}(n) \wedge x = 2n + 1 \\
 & & \\
\vdash_p p : \tau & \stackrel{\text{def}}{=} & \forall s.\ \mathit{fst}(p)\ s \Rightarrow \exists a.\ \mathit{snd}(p)\ s\ a \wedge \tau\ a \\
\Gamma \vdash_d (d; e) : \tau & \stackrel{\text{def}}{=} & \forall s.\ (d\ s \wedge \Gamma\ s) \Rightarrow \exists a.\ (e\ s\ a \wedge \tau\ a) \\
\Gamma \vdash_e e : \tau & \stackrel{\text{def}}{=} & \forall s.\ \Gamma\ s \Rightarrow \exists a.\ (e\ s\ a \wedge \tau\ a) \\
\tau_1 \boxplus \tau_2 = \tau & \stackrel{\text{def}}{=} & \forall n_1.\forall n_2.\ \tau_1\ n_1 \Rightarrow \tau_2\ n_2 \Rightarrow \tau\ (n_1 + n_2) \\
\Gamma[x : \tau] & \stackrel{\text{def}}{=} & \lambda s.\ \Gamma\ s \wedge \exists a.\ s\ x\ a \wedge \tau\ a \\
\Gamma(x) = \tau & \stackrel{\text{def}}{=} & \forall s.\ \Gamma\ s \Rightarrow \exists a.\ s\ x\ a \wedge \tau\ a
\end{array}
$$

Figure 5.4: Definitions of types and judgements.

Efficiency and proof size problem. When type checking a program, we build a type environment, or *context*, from the declarations for variables that appear in the expression. The rules for traversing a list of declarations and building the corresponding type contexts are *DeclConsTy* and *DeclNilTy*. When a variable is encountered, we look up its type in the context. However, the typing rule *VarTy* does not specify a context lookup algorithm. Consider the following variable type-lookup rules.

$$\frac{}{\Gamma[x : \tau] \vdash x : \tau}\ \mathit{VarTyHit}$$

$$\frac{\Gamma \vdash x : \tau \qquad x \neq y}{\Gamma[y : \tau'] \vdash x : \tau}\ \mathit{VarTyMiss}$$

Suppose the context is simply organized as a list in these two rules; each element of the list is a pair: a variable and its type. Then each context lookup takes linear time, and type-checking a whole program will take quadratic time. Correspondingly,

the size of the generated proof for a lookup operation is linear with respect to the size of the context, and thus the safety proof (and also the proof checking time) for a program has a quadratic blowup. Our experiment with the even-odd example shows that naive implementation cannot check even medium-size program, while the efficient algorithm, which will be described in the next section, scales very well. This algorithm still has a provably sound semantic model, but generates concise proofs and admits efficient proof checking.

5.3 Effective Context Management

As we have explained, we avoid sending large proofs to the trusted checker by sending a proof scheme with a soundness proof for the proof scheme. We want the proof scheme to "execute" efficiently, that is, in linear time with respect to the size of the program-safety-theorem being proved. And we want the proof schemes to be written in the "smallest possible" Prolog-like language: What set of language features are useful?

Here we will show an efficient proof scheme for contexts; because this scheme requires dynamic clauses in the Prolog subset, we have included a limited form of dynamic clauses in our language design.

5.3.1 Dynamic clauses and local assumptions

Many logic programming systems provide a facility for managing dynamic clauses at run time. In Prolog, users can *assert* a fact or clause into database or *retract* a clause dynamically. The assert/retract mechanism can be expensive if the dynamic clause in consideration is not atomic (i.e., has subgoals) because the dynamic

clause has to be compiled and integrated into the program's decision trees. If the dynamic clause is atomic, with input-mode arguments that are integers or hashable, the assert/retract operation can be cheap: Prolog systems usually provide efficient support for asserting and retracting an atomic clause by using hash tables. That is, asserting, retracting, and querying indexable atomic clauses can be done in constant time per operation.

In the LF logical framework [Harper et al., 1993], or its implementation Twelf [Pfenning, 1991; Pfenning and Schürmann, 1999], one can use local assumptions [Pfenning and Schürmann, 2002] to check dynamic clauses into database. Since these assumptions are local, their dynamic scopes control their lifetimes; there is no need to provide an explicit retract mechanism. A clause of the form $\{x : \tau\}\ A\ x \rightarrow B\ x$ introduces a local assumption $A\ x$ into the context and then solves the goal $B\ x$ under this assumption.[5] When proof search on goal B has finished, assumption A is automatically retracted. That is, Twelf uses a dynamically well-scoped version of assert/retract. One can use Prolog assert/retract mechanism to simulate Twelf's local assumptions, however. We can give semantics to local assumptions and generate concise proofs so that clauses are guaranteed to be correct.

Local assumptions are particularly effective—efficient, secure (with a provably sound model), and concise—when we need to deal with big environments and generate proofs of lookups in these environments.

5.3.2 Typing rules

In this subsection, we present an efficient type checking algorithm for environment management using dynamic clauses. The semantics is presented in the next sub-

[5]It is a dependent type on local parameter x.

$$\frac{\vdash_p p : \mathsf{even}}{safe(p)}\ SafeTy \qquad \frac{d \vdash_d (d; e) : \tau}{\vdash_p (d; e) : \tau}\ ProgTy$$

$$\frac{\Gamma \vdash_e e_1 : \tau_1 \qquad bind(x, \tau_1, \Gamma) \ \rightarrow\ \Gamma \vdash_d (d; e) : \tau}{\Gamma \vdash_d (\mathsf{let}\ x = e_1; d; e) : \tau}\ BindTy$$

$$\frac{\Gamma \vdash_e e : \tau}{\Gamma \vdash_d (\cdot; e) : \tau}\ BindNil \qquad \frac{bind(x, \tau_1, \Gamma)}{\Gamma \vdash_e x : \tau}\ VarTy$$

$$\frac{\Gamma \vdash_e e_1 : \tau_1 \qquad \Gamma \vdash_e e_2 : \tau_2 \qquad \tau_1 \boxplus \tau_2 = \tau}{\Gamma \vdash_e e_1 + e_2 : \tau}\ PlusTy$$

$$\frac{}{\mathsf{even} \boxplus \mathsf{even} = \mathsf{even}}\ \boxplus ee \qquad \frac{}{\mathsf{odd} \boxplus \mathsf{odd} = \mathsf{even}}\ \boxplus oo$$

$$\frac{}{\mathsf{even} \boxplus \mathsf{odd} = \mathsf{odd}}\ \boxplus eo \qquad \frac{}{\mathsf{odd} \boxplus \mathsf{even} = \mathsf{odd}}\ \boxplus oe$$

Figure 5.5: Typing rules with dynamic context.

section. Figure 5.5 shows the type checking rules with a dynamic environment management scheme.

The rule *ProgTy* calls a declaration checking rule and passes declaration d to it. The declaration d appears twice in the premise. The declaration checking rules traverse one d, and the other d is used to pass the original declaration all the way to the expression checking rules.

The rule *BindTy* requires some explanation. It first checks that the expression e_1 has type τ_1, then asserts this fact as a dynamic clause (or local assumption) $bind(x, \tau_1, \Gamma)$ and continues type checking.

When type checking a variable expression, we try rule *VarTy* to match the previous checked-in local assumptions. The lookup operation takes constant time and the proof generated for it is concise. The $\boxplus$ rules remain the same as before.

In a conventional Prolog implementation that supports efficient assert/retract

operations for atomic dynamic clauses like $bind(x, \tau_1, \Gamma)$, the type checking algorithm above is linear. Moreover, it is provably sound as we will show next.

5.3.3 Foundational semantics and proofs

The safety specification remains the same as presented in Figure 5.2. The definitions of types and typing judgements remain untouched except for $\vdash_d$ and the new constructor *bind*.

$$\begin{array}{lcl} \Gamma \vdash_d (d; e) : \tau & \stackrel{\text{def}}{=} & \forall s.\ (\Gamma \sqsubseteq d \wedge \Gamma\ s) \Rightarrow \exists a.\ (e\ s\ a \wedge \tau\ a) \\ bind(x, \tau, \Gamma) & \stackrel{\text{def}}{=} & \forall s.\ \Gamma\ s \Rightarrow \exists a.\ (s\ x\ a \wedge \tau\ a) \\ d_1 \sqsubseteq d_2 & \stackrel{\text{def}}{=} & \forall s.\ d_1\ s \Rightarrow d_2\ s \end{array}$$

The semantics of dynamic clause $bind(x, \tau, \Gamma)$ is very similar to that of the static binding operator $\Gamma[x : \tau]$ and lookup operator $\Gamma(x) = \tau$. It serves both purposes. From these definitions it is straightforward to prove the typing rules as lemmas and the safety theorem can be proved from the successful type checking of a program from the goal $\vdash_p (d; e) : \mathsf{even}$. Here we give the proof for rule *BindTy*.

Lemma 5.3.3.1 (*BindTy*)

$$\frac{\Gamma \vdash_e e_1 : \tau_1 \qquad bind(x, \tau_1, \Gamma)\ \rightarrow\ \Gamma \vdash_d (d; e) : \tau}{\Gamma \vdash_d (\mathsf{let}\ x = e_1; d; e) : \tau}\ \mathit{BindTy}$$

PROOF: By definition of $\vdash_d$, for all state s, we assume $\Gamma \sqsubseteq (\mathsf{let}\, x = e_1; d)$ and $\Gamma\ s$, then we prove $\exists a.\ (e\ s\ a \wedge \tau\ a)$. This can be obtained from $\Gamma \vdash_d (d; e) : \tau$. In order to use this fact, we need to prove the local assumption $bind(x, \tau_1, \Gamma)$, which can be proved from the premise $\Gamma \vdash_e e_1 : \tau_1$ and the assumption $\Gamma \sqsubseteq (\mathsf{let}\, x = e_1; d)$. □

The machine-checkable proof in LF for this rule can be found in Section 5.6.

5.3.4 Dynamic clauses in the real LTAL

In LTAL, we use dynamic clauses to efficiently maintain various environments such as label, register, and type maps. These maps are shown in Figure 3.2 as LRT. An LTAL program is a tuple $\langle L, R, T, B\rangle$, where L is a label map, R is a register map, T is a type definition map, and B is a list of basic blocks.

The type checker for LTAL programs starts by parsing the LRT maps so that later they can be looked up efficiently. These maps could be quite large for real-life programs, and often the type checker needs to look up the value of a label, the register that a variable is assigned to, or the content of a type definition. Naive implementation, such as sequential search, of parsing and lookup rules for various environments is straightforward, but not efficient. In our actual implementation, we use dynamic clauses to efficiently maintain various environments. The type checking rules for parsing these maps are shown in Figure 5.6.

The rule *SafeTheorem* states that the LTAL type checking must establish the safety theorem. The root rule of the LTAL type checker is the rule *ProgTy* which calls the label binding parsing rules. There two rules, *BindLabCons* and *BindLabNil*, for processing the label environment L. The *BindLabCons* rule matches if the current label environment in processing is not empty. Its subgoal has a dynamic clause $bindLab(l_1, a_1, H)$. This dynamic clause is asserted at run time whenever the rule *BindLabCons* is matched. The dynamic clauses have dynamic scope, and thus if later the type checker wants to look up the value (which is an address) of a label l, it can simply invoke a subgoal as $bindLab(l, a, H)$ and a will have the value after the subgoal completes. This lookup operation takes constant time since dynamic clauses are compiled into hash tables by the the underlying logic programming system. The

$$\frac{\vdash_p P\ H}{\text{safe}(P)}\ \mathit{SafeTheorem} \qquad \frac{\vdash_l P\ \langle L, R, T, B\rangle\ L}{\vdash_p P\ \langle L, R, T, B\rangle}\ \mathit{ProgTy}$$

$$\frac{\mathit{bindLab}(l_1, a_1, H)\ \rightarrow\ \vdash_l P\ H\ L}{\vdash_l P\ H\ (l_1 \mapsto a_1, L)}\ \mathit{BindLabCons}$$

$$\frac{\vdash_r P\ \langle L, R, T, B\rangle\ R}{\vdash_l P\ \langle L, R, T, B\rangle\ \text{nil}}\ \mathit{BindLabNil}$$

$$\frac{\mathit{bindReg}(x_1, r_1, H)\ \rightarrow\ \vdash_r P\ H\ R}{\vdash_r P\ H\ (x_1 \mapsto r_1, R)}\ \mathit{BindRegCons}$$

$$\frac{\vdash_t P\ \langle L, R, T, B\rangle\ T}{\vdash_r P\ \langle L, R, T, B\rangle\ \text{nil}}\ \mathit{BindRegNil}$$

$$\frac{\mathit{bindTy}(\mathscr{D}_1, \tau_1, H)\ \rightarrow\ \vdash_t P\ H\ T}{\vdash_t P\ H\ (\mathscr{D}_1 \mapsto \tau_1, T)}\ \mathit{BindTyCons}$$

$$\frac{\vdash_b P\ \langle L, R, T, B\rangle\ B}{\vdash_t P\ \langle L, R, T, B\rangle\ \text{nil}}\ \mathit{BindTyNil}$$

$$\frac{LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{(\iota_1; \ldots; \iota_k)\}\ (\rho'; \mathscr{H}'; \phi'; cc')}{\vdash_b P\ \langle L, R, T, B\rangle\ (l[\vec{\alpha} : \vec{\kappa}](m, cc, \phi) = \iota_1; \ldots; \iota_k)}\ \mathit{BlockCons}$$

$$\frac{}{\vdash_t P\ \langle L, R, T, B\rangle\ \text{nil}}\ \mathit{BlockNil}$$

Figure 5.6: LTAL typing rules for environment management with dynamic context.

rule *BindLabNil* matches if the label environment is empty and it calls the register environment processing rules. The register and type definition environments are processed in a similar way.

After processing these environments, the type checker invokes the basic block checking rules which simply call instruction checking rules for checking the body with preconditions as the current typing environments. Since the dynamic clauses are dynamically scoped, and the basic block checking rules and instruction checking

rules are invoked by the rule *BindTyNil* directly and by other environment processing rules indirectly, we can efficiently query dynamic clauses in the basic block and instruction type checking rules. The individual instruction checking rules are presented in Section 3.7, 3.8, 3.9, and Appendix A.2.

5.4 Logic Programming Engine

For developing our semantic proofs of soundness we use Twelf, a sophisticated system with many useful features: In addition to an LF type checker, it contains a type reconstruction algorithm that permits users to omit many explicit parameters, a proof-search algorithm (which is like a higher-order Prolog interpreter), constraint regimes (e.g., linear programming over the exact rational numbers), mode analysis of parameters, a meta-theorem prover, a pretty-printer, a module system, a configuration system, an interactive Emacs mode, and more. We have found many of these features useful in proof development, but Twelf is certainly not a minimal proof checker; we would like to avoid the need to trust it. However, since Twelf does construct explicit proof objects internally, we can extract these objects to send to our minimal checker.

The previous section shows that efficient syntax-directed type-checking uses certain logic-programming constructs (dynamic clauses) but not others (backtracking), and that each Horn clause can be proved sound as a lemma in higher-order logic. This section describes a suitable logic programming interpreter implemented in Flit, our trusted LF proof checker. The logic programming engine is implemented by Stump [Wu et al., 2003]. Other aspects of Flit are described in Appel et al. [2002].

A type checking lemma (a rule together with its semantic proof) is represented

in LF as "$name : \tau = exp$." The type τ encodes the type checking rule, and *exp* is a term of type τ. By the Curry-Howard isomorphism [Howard, 1980], term *exp* is a proof of the theorem that τ encodes. The *name* stands for the whole term *exp* with type τ, *i.e.* the theorem and the proof.

The first step is to check the validity of the proof "*exp*." Our checker Flit includes a simple LF checker which is used to check that *exp* has LF type τ. Flit LF checker is simpler than Twelf since Twelf does type inference while Flit does not. We ask the adversary to send explicitly typed LF terms instead of implicitly typed terms; explicitly typed LF terms can be constructed by Twelf's type inference module.

After LF type checking, the proof term "*exp*" is not useful anymore. Flit runs a simple logic programming engine to interpret the type checking rules as a logic program, which type checks input machine programs.

To achieve a concise and efficient implementation, we impose several restrictions on the form of goals and programs. If these are violated, the interpreter will remain sound but may fail to be complete. Specifically, Flit's logic programming language makes the following assumptions:

Atomic dynamic clauses. Flit does not allow non-atomic dynamic clauses. Dynamic clauses are mainly used to efficiently maintain various environments. For this purpose, atomic dynamic clauses suffice.

Bounded execution. To avoid dynamic memory allocation during the logic program execution, Flit uses a fixed-size memory to run logic programs. The only purpose of this restriction is to simplify the logic programming engine and thus simplify Flit, the trusted checker.

Determinism. Every subgoal in the input logic program is solved, if solvable,

by the first matching clause in the set of static clauses and active dynamic clauses. Note that dynamic clauses follow the dynamic scoping rule. Under this condition of determinism, Flit does not need backtracking mechanism. And our practice shows that backtracking can often be avoided during type checking when the type checker is carefully engineered.

Bounded indices. Let's define the *index* of a dynamic clause to be its first argument. We require indices of dynamic clauses are small natural numbers and distinct from each other. This allows simple and efficient indexing of dynamic clauses.

Prolog interpreters typically enter atomic dynamic clauses in hash table for efficient matching, using one of the predicate's arguments as the hash key. Our logic programs can be written with this very restricted form of clause indexing.

Example. The even-odd proof scheme of Figure 5.5 is a logic program that conforms to these restrictions. The proof scheme (1) executes in linear time and space, and it is (2) syntax-directed. Its dynamic clauses $bind(x, \tau, \Gamma)$ are all atomic. In our implementation of this proof scheme, we put the x argument of $bind(x, \tau, \Gamma)$ in the first position to conform to the *bounded indices* rule; and all the indices x are manifest constants that are small integers. Our LTAL proof scheme used in the real PCC system also obeys these restrictions.

A logic program is presented to Flit's logic programming engine as a set of LF terms, represented using an expression data structure [Appel et al., 2002]. Flit first transforms the logic clauses into a format that is convenient for executing logic programs, and then runs the logic program [Wu et al., 2003].

5.5 Proof Witnesses

Our even-odd example is overly simplistic in that there is a syntax-directed decision procedure for the main safety theorem: For an expression E, if the formula $safe(E)$ is true, then the proof is easily found. In a real proof-carrying code application, the program E is in machine language; loops and recursion in the program, and quantified types in the type system, make type inference impossible.

Thus, in a PCC application, the input to the prover includes the program E and also an untrusted hint H. The hint provides loop invariants, type annotations, and other information which can be used by the prover. Because the hint is provided by the same adversary who provides the program, H cannot be assumed accurate, but it can still be useful in constructing the proof.

We will illustrate using the even-odd example. Let us provide a hint H which is a list of type annotations, $x_1 : \tau_1$, $x_2 : \tau_2, \ldots, x_n : \tau_n$. We will write a prover that uses this hint (even though for this simple language the hint is not necessary). The root goal is now $\vdash_p \ H \ E$ instead of $safe(E)$.[6]

In addition to running the logic program on the root query $\vdash_p \ H \ E$, the checker verifies a (static) proof of the lemma,

$$\frac{\vdash_p \ H \ E}{safe(E)} \ .$$

We can't use this as a logic-programming rule, i.e. we can't use $safe(E)$ as our query, because then the logic program would have to "guess" H, which could require unbounded backtracking. The hint H serves as a *proof witness* for E, in conjunction with the Prolog program (i.e. proof scheme) and its semantic soundness proof.

[6]The text representation of the predicate $\vdash_p$ is "judge_prog" as we presented in Chapter 2.

	Axioms	stage 1
Trusted↑	Expression Operators	
Untrusted↓	Semantic Model	stage 2
	Hint Operators	
Proof scheme	Clauses	
Theorem to be proved	Expression	stage 3
Proof witness	Hint	

Table 5.1: Layers of specification and proof.

5.5.1 Layers of specification and proof

To handle proof-checking with hints, the checker software must process separately several layers of specification, semantics, proof, and logic-programming clauses. The layers of specification and proof are shown in Table 5.1. It is useful to think in terms of a *proof consumer* and an *adversary*.

Stage 1. The proof consumer specifies the *Axioms* of a logic, and defines the kinds of theorems she wants to check—that is, the language of expressions for which she wants safety theorems—by defining *Expression Operators*. One of the expression operators must be a predicate called *safe*.

Stage 2. Then the adversary sends a proof scheme, that is, a logic program (the syntactic type checker in the even-odd example). This program manipulates goals expressed using the *Expression Operators* and the *Hint Operators*. All the hint operators must be defined in terms of the underlying logic—the adversary is not permitted to add uninterpreted operators to the logic. All the *Clauses* of the logic program must be proved as derived lemmas in the logic, from the definitions of the expression and hint operators, as Lemma 5.3.3.1 does.

The *Semantic Model*, sent by the adversary, is simply a set of supporting definitions and lemmas, defined in terms of the underlying logic, that can be useful in defining the hint operators and the clauses.

The adversary may define as many hint operators and clauses as he likes; however, there must be one operator called $\vdash_p$, and the semantic model must contain a lemma of the form,

$$\frac{\vdash_p \ H \ E}{safe(E)} \ .$$

The proof consumer uses the logical framework LF to check the wellformedness of all the definitions and the proofs of all the lemmas. Then she loads the *Clauses* into the subset-Prolog interpreter.

Stage 3. Finally, the adversary sends an *Expression* and a *Hint*. The consumer needs to verify that the expression obeys her desired safety property—this was the point of the whole exercise!—and she will do it using the adversary's proof scheme. Since the proof scheme was proved sound (and she has checked the proof), then if the logic program completes successfully, then $safe(E)$ must be valid.

For the even-odd system, the implementation of these stages is shown in Table 5.2; sample source code written in Twelf is in Section 5.6.

What is a proof witness? Stage 1 (loading axioms and safety predicate) needs to be done only once per safety policy. In a PCC application, stage 2 (loading the proof scheme) would need to be done when there are substantial modifications to the the untrusted compiler. Stage 3 is repeated for each compiled program sent from the compiler to the consumer. Clearly, any work done in stages 1 and 2 can

Axioms

$\dfrac{A \Rightarrow B \quad A}{B}$ imp_e $\qquad \dfrac{\forall x.A(x)}{A(B)}$ $\forall$_e $\qquad$ *et cetera*

Expression Operators

(Figure 5.2)

Var	$\stackrel{\text{def}}{=}$	Num
$State$	$\stackrel{\text{def}}{=}$	$Var \to Num \to Form$
$Decl$	$\stackrel{\text{def}}{=}$	$State \to Form$
Exp	$\stackrel{\text{def}}{=}$	$State \to Num \to Form$
$Prog$	$\stackrel{\text{def}}{=}$	$\langle Decl, Exp \rangle$
$(d; e)$	$\stackrel{\text{def}}{=}$	$\langle d, e \rangle \qquad \cdot \stackrel{\text{def}}{=} \lambda s.\ \text{true}$
let	$\stackrel{\text{def}}{=}$	$\lambda x.\lambda e.\lambda d.\ \lambda s.\ d\ s \wedge (\forall a.e\ s\ a \Rightarrow s\ x\ a)$
x	$\stackrel{\text{def}}{=}$	$\lambda s.\lambda a.\ s\ x\ a \qquad n \stackrel{\text{def}}{=} \lambda s.\lambda a.\ a = n$
$+$	$\stackrel{\text{def}}{=}$	$\lambda e_1.\lambda e_2.\ \lambda s.\lambda a.\ \exists a_1.\exists a_2.e_1\ s\ a_1\ \wedge\ e_2\ s\ a_2\ \wedge a = a_1\ plus\ a_2$
$safe$	$\stackrel{\text{def}}{=}$	$\lambda p.\ \forall s.\ fst(p)\ s\ \Rightarrow \exists a.\ snd(p)\ s\ a\ \wedge \text{isEven}(a)$

Semantic Model

(Figure 5.4 and Section 5.3.3)

Ty	$\stackrel{\text{def}}{=}$	$Num \to Form \qquad Env \stackrel{\text{def}}{=} State \to Form$
$\exists!$	$\stackrel{\text{def}}{=}$	$\lambda F.\ \exists x.\ F\ x\ \wedge\ \forall y.\ F\ y\ \Rightarrow\ x = y$
upd	$\stackrel{\text{def}}{=}$	$\lambda x.\lambda a.\lambda s.\ \lambda y.\lambda b.\ \text{if}\ (x = y)\ (a = b)\ (s\ y\ b)$
	...	
even	$\stackrel{\text{def}}{=}$	$\lambda x.\exists n.\ \text{isInt}(n) \wedge x = 2n$
odd	$\stackrel{\text{def}}{=}$	$\lambda x.\exists n.\ \text{isInt}(n) \wedge x = 2n + 1$
$\vdash_p$	$\stackrel{\text{def}}{=}$	$\lambda h.\lambda p.\lambda \tau.\ \forall s.\ fst(p)\ s \Rightarrow \exists a.snd(p)\ s\ a \wedge \tau\ a$
$\vdash_d$	$\stackrel{\text{def}}{=}$	$\lambda \Gamma.\lambda h.\lambda d.\lambda e.\lambda \tau.\ \forall s.\ (\Gamma \sqsubseteq d \wedge \Gamma\ s) \Rightarrow \exists a.\ (e\ s\ a \wedge \tau\ a)$
$\vdash_e$	$\stackrel{\text{def}}{=}$	$\lambda \Gamma.\lambda e.\lambda \tau.\ \forall s.\ \Gamma\ s \Rightarrow \exists a.\ (e\ s\ a \wedge \tau\ a)$
$\boxplus$	$\stackrel{\text{def}}{=}$	$\lambda \tau_1.\lambda \tau_2.\lambda \tau.\ \forall n_1.\forall n_2.\tau_1\ n_1 \Rightarrow \tau_2\ n_2 \Rightarrow \tau\ (n_1 + n_2)$
$bind$	$\stackrel{\text{def}}{=}$	$\lambda x.\lambda \tau.\lambda \Gamma.\ \forall s.\ \Gamma\ s \Rightarrow \exists a.\ (s\ x\ a \wedge \tau\ a)$
$\sqsubseteq$	$\stackrel{\text{def}}{=}$	$\lambda d_1.\lambda d_2.\ \forall s.\ d_1\ s \Rightarrow d_2\ s$

Hint Operators

Ty Env even odd typeof $\cdot$

Clauses

(Figure 5.5)

$safe(p) \ \leftarrow\ \vdash_p p : \text{even}.$
$\vdash_p (d; e) : \tau \ \leftarrow\ d \vdash_d (d; e) : \tau.$
$\Gamma \vdash_d (\text{typeof}\ x : \tau_1; h) \parallel (\text{let}\ x = e_1; d)\ ; e : \tau \ \leftarrow$
$\quad \Gamma \vdash_e e_1 : \tau_1 \ \leftarrow$
$\quad (bind(x, \tau_1, \Gamma) \ \rightarrow\ \Gamma \vdash_d (h \parallel d\ ; e) : \tau).$
$\Gamma \vdash_d (\cdot \| \cdot; e) : \tau \ \leftarrow\ \Gamma \vdash_e e : \tau.$
$\Gamma \vdash_e x : \tau \ \leftarrow\ bind(x, \tau_1, \Gamma).$
$\Gamma \vdash_e e_1 + e_2 : \tau \ \leftarrow\ \Gamma \vdash_e e_1 : \tau_1 \ \leftarrow\ \Gamma \vdash_e e_2 : \tau_2 \ \leftarrow\ \tau_1 \boxplus \tau_2 = \tau.$
$\text{even} \boxplus \text{even} = \text{even}. \qquad \text{even} \boxplus \text{odd} = \text{odd}.$
$\text{odd} \boxplus \text{odd} = \text{even}. \qquad \text{odd} \boxplus \text{even} = \text{odd}.$

Expression

let $x = 4$; let $y = x + 8$; $x + y$

Hint

typeof x even (typeof y even $\cdot$)

Table 5.2: Proof scheme for even-odd system. Not shown are the proofs (in higher-order logic) of all the clauses.

be amortized over many executions of stage 3. Although the foundational proof derives from information transmitted in stages 2 and 3, in measuring the effective size of proof witnesses we can consider just the *Hint* sent in stage 3.

5.6 Machine Checkable Proofs

To illustrate the format of machine-checked soundness proofs of the type-checking clauses, here we will show the proofs related to the rule *BindTy* (Lemma 5.3.3.1). Note this is the version with hints we described in Section 5.5; the rule without hints is quite similar.

Since the proof is written in LF, we begin with a brief introduction to LF. LF is based on the λ-calculus with dependent types, and it has syntactic entities at three levels: objects, types, and kinds. Types classify objects and kinds classify families of types. A deductive system is represented in LF using the judgements-as-types and derivations-as-terms principle [Harper et al., 1993]: Judgements (theorems) are represented as types, and derivations (proofs) are represented as terms whose type is the representation of the judgement (theorem) that they prove. In this way proof checking of the object logic is reduced to type checking of the LF terms.

In general, a definition in LF has the form: $name : \tau = exp.$ including the dot. The type τ encodes the theorem to be proved, and *exp* is a term of type τ. By the judgements-as-types and derivations-as-terms principle, term *exp* is a proof of the theorem that τ encodes. And the *name* stands for the whole term *exp* with type τ, *i.e.* the theorem and the proof. LF and Twelf also permit introducing constructors with the form $name : \tau$.

The entire machine-checkable proof in LF is shown in Figure 5.7. The notation

```
check_decl_cons:
    |-d (typeof V Tv HINT) (let V Ev D) Gamma E T <-
    |-e Gamma Ev Tv <-
    (bind V Tv Gamma
       -> |-d HINT D Gamma E T) =
 [p1: bind V Tv Gamma -> |-d HINT D Gamma E T]
 [p2: |-e Gamma Ev Tv]
 |-d_i [s]
       [p3: pf (sub_env @ Gamma @ (let V Ev D))]
       [p4: pf (Gamma @ s)]
 cut (bind_i [s_v]
             [p7: pf (Gamma @ s_v)]
      |-e_l p2 p7 [a_v]
             [p5: pf (Ev @ s_v @ a_v)]
             [p6: pf (Tv @ a_v)]
      cut (let_e1 (sub_env_e p3 p7) p5)
             [p8: pf (s_v @ c V @ a_v)]
      exists_i a_v
      (and_i p8 p6))
    [p10: bind V Tv Gamma]
 cut (sub_env_i [s']
                [p12: pf (Gamma @ s')]
      let_e2 (sub_env_e p3 p12))
    [p20: pf (sub_env @ Gamma @ D)]
 |-d_e (p1 p10) p20 p4.
```

Figure 5.7: Machine-checkable proof of *BindTy* in LF.

"`[x:t]A`" denotes $\lambda \mathtt{x} : \mathtt{t}.\, \mathtt{A}$. In the proof above we first introduce two λ-bindings; that is, we assume that the two premises of the typing rule hold. Then we use the `|-d` introduction rule `|-d_i` to get a proof of

```
|-d (typeof V Tv HINT) (let V Ev D) Gamma E T,
```

i.e. the conclusion.

The rule `|-d_i` introduces three λ-bindings: `s`, `p3`, and `p4`. Note that the type of `s` is omitted and Twelf will reconstruct it to a *State* type. Lemma `cut` is as follows:

```
cut: pf A -> (pf A -> pf B) -> pf B =
  [p1:pf A][p2:pf A -> pf B] imp_e (imp_i p2) p1.
```

The `imp_i` and `imp_e` (*modus ponens*) are introduction and elimination lemmas for implication. In general, the lemma `cut` means if we have a proof of A, and a function which maps a proof of A to a proof of B, then we can get a proof of B. This is similar to `imp_e` or *modus ponens*, but `cut` uses LF function type `->` instead of object implication. When using `cut`, we first prove some formula A, then bind this proof (give it a name so that we can refer to it later) and continue to prove the goal (B in this case). The `@` is the object logic level term application.

5.7 Scaling Up to Foundational PCC

The even-odd type system is just a toy example to demonstrate some of the principles. Our real applications are in proof-carrying code and distributed authorization. Our checking system scales up to these examples quite well, as we will explain.

In our application to foundational PCC, the hint H is an expression in the LTAL calculus presented in Chapter 3, and the expression E is a machine-language program, that is, a sequence of 32-bit natural numbers.

Figure 2.1 shows the major components of our foundational proof-carrying code framework. The *LTAL clauses* are a set of clauses in our restricted Prolog subset. *Axioms & Architecture Spec* are preloaded into our *Checker* and must be trusted as axioms and trusted definitions.[7] Between these two components are proofs, based on the axioms, of all the *LTAL clauses*.

A source program is compiled into a machine-code program and an LTAL expression. The compiler is not trusted, because it is a large program that may have bugs. The trusted checker receives the LTAL clauses, along with their soundness

[7] A trusted definition is one that is used in the statement of the theorem to be proved; an untrusted definition is used only in the proof.

proofs in higher-order logic; checks the soundness proofs; and then runs the LTAL checker, which is a syntax-directed computation in our subset Prolog.

LTAL is presented in Chapter 3; the LTAL semantic model is briefly presented in Chapter 4. In this chapter, we focus on the aspects of the LTAL calculus that enable it to be type-checked by our tiny trusted checker.

Because a source-language programmer never sees the LTAL program, we can design the LTAL calculus to be checkable in our very restricted language. To use the checker's limited support for dynamic clauses, we have arranged the LTAL so that: All identifiers in LTAL are small integers. No variables have the same identifier. Program labels, local variables, and type abbreviations are represented by disjoint sets of integers. To make the LTAL type system entirely syntax-directed, we use explicit coercions to guide the typing rules, instead of relying on subtyping which would require a search.

We use the simple and limited arithmetic provided by the checker: addition, multiplication, and truncating division on 32-bit natural numbers. Other operators are synthesized, such as $A > B$ by `div` B A 0, using truncating division.

The LTAL typing rules, such as the one shown in Section 3.3, though bigger and more complicated than the rules we presented for the even-odd system, can be executed by our simple subset Prolog interpreter.

5.8 Experimental Results

We have measured our trusted checker on the even-odd microbenchmark and on some small but nontrivial LTAL benchmarks. Gross statistics about these proof schemes are shown in Table 5.3.

	EvenOdd	*LTAL*	
Core Axioms	341	341	lines of LF
Application-specific	10	1522	lines of LF
Expression Operators	40	2	lines of LF
Semantic Model	218	∼100,000	lines of LF
Hint Operators	10	500	lines of LF
Clauses	12	3,500	lines of LF
Expression	$\sim 7N$	$\sim 2N$	tokens
Hint	$\sim 4N$	$\sim 30N$	tokens

Table 5.3: Measurements—system size.

Lines of LF does not include blank lines and comments. Expression sizes for EvenOdd are measured with N as the number of declarations, each declaration of the form $\mathsf{let}\ x_i = x_j + x_k$; which is 7 tokens per declaration. Expression sizes for LTAL are measured with N as the number of machine instructions (32-bit integers) in the program to be proved safe, with two tokens per integer, for example:

```
2551193600 ; 2181292040 ; 2214748172 ; ... ; nil
```

From this it should be clear why LTAL has only two *Expression Operators*; everything shown in Figure 3.2 and 3.16 is actually *Hint Operators*.

The logic program is the set of LTAL typing rules. There are several hundred LTAL *clauses* or typing rules, some of which take dozens of lines to write down, such as the one we showed in Section 3.3 for the SPARC add instruction. The LTAL *semantic model*, which provides proofs of all these clauses, is rather intricate and is the subject of several other papers and PhD theses [Appel and Felty, 2000; Appel and McAllester, 2001; Ahmed et al., 2002; Ahmed, 2004; Swadi, 2003; Tan et al., 2004].

Since the clauses are written in a subset of Prolog, we can execute them in a standard Prolog system. For each benchmark, we compare execution time in

Input size	SICStus	Twelf	Flit
EvenOdd			
$N = 100$	0.002	0.99	0.01
$N = 1000$	0.030	> 3600	0.05
$N = 10000$	1.460		0.26
LTAL			
$N = 32$	0.005	1.21	0.43
$N = 870$	0.183	1018	1.32
$N = 1816$	0.432	> 3600	2.19

Table 5.4: Measurements—safety checking performance.

the (highly optimized) SICStus Prolog compiler with execution time in the Flit interpreter. The results are shown in Table 5.4.

All times are in seconds on a 2.2 GHz Pentium 4. Twelf is not designed for performance, but its advanced features make it a convenient tool for us to develop machine-checkable proofs in LF. Flit is faster than SICStus Prolog for large EvenOdd examples; EvenOdd is unrealistic because the Prolog program has only a few simple clauses. Parsing the expression and hint contributes a significant portion of execution time for EvenOdd examples in SICStus Prolog. And also, the dynamic clause indexing in Flit is tailored to our specific applications; it could be more efficient for our examples than general purpose Prolog systems. Checking LTAL, Flit is about five times slower than SICStus; this performance may be acceptable in the intended application.

Of course, execution in SICStus loses the benefits of the tiny trusted base: In that mode we don't mechanically connect the soundness proof for the LTAL clauses to the actual SICStus execution, and the SICStus Prolog compiler and interpreter also become part of the trusted base.

Table 3.2 in Chapter 3 compares the proof-checking time of the life benchmark

with the time necessary for the ML compiler to generate the program. For applications where the output of a compiler is to be checked by a trusted checker, it's desirable that checking time be small compared to compile time. SML/NJ can compile this benchmark in 0.49 seconds; our LTAL-generating FPCC/ML compiler takes 3.0 seconds (the slowdown is partly because it takes extra time to preserve types, and mostly because we have not engineered the back end for speed). LTAL type-checking takes 0.43 seconds in SICStus and 2.2 seconds in Flit.

The Flit software currently comprises about 1169 lines of C code: the 803 lines described in Section 5.1.2 for parsing axioms, loading proof graphs, and LF checking the proofs have grown to 852 lines; our new logic-program interpreter is about 282 lines, and there are about 35 lines to manage the stages described in Section 5.5.1.

Necula's oracle-based Prolog interpreter [Necula and Rahul, 2001] is about 800 lines of C code. It should be straightforward to use our style of LF proof-checking of Prolog clauses, but use oracle-based execution instead of our interpreter. Then, instead of an 1169-line C program, we would have a 1700-line program. In such a system, the proof witnesses would be just as tiny as Necula's, and the trusted base would be somewhat larger than that of the system we have described in this paper.

Our initial implementation of Flit has no garbage collector. Checking the $N = 1816$ LTAL example consumes approximately 4 million heap nodes without garbage collection. To scale Flit to significantly larger inputs, garbage collection would be necessary. Our implementation of an allocator with two-space copying collector is 70 lines of C code.

5.9 Conclusion

To make a trustworthy proof-checker with small witnesses, one should define a language for proof-schemes, with a way to represent and check soundness theorems for the proof schemes; then one should implement an interpreter to execute the proof scheme on the theorem and the witness.

Pollack explained much of this in "How to believe a machine-checked proof" [Pollack, 1998]:

> ... I suggest that the "programming language" for the checking program be a logical framework [such as] the Edinburgh Logical Framework .... we [could] program a checker in the internal language of the framework .... The question then arises: where will we find a believable implementation of a logical framework?

We ask you to believe very little. Our implementation is based on LF, higher-order logic, and a small subset of pure Prolog, all of which are well understood; and our implementation is about as small as possible—that is, to trust our system there are less than 1200 lines of code that you have to understand.

Chapter 6

Conclusion and Future Work

In summary, the adoption of a low-level typed assembly language, construction of its semantic model and machine-checkable soundness proof, and integration of a simple logic programming engine in the proof checker are our main design choices, and they serve as the interfaces between the compiler, the proof checker and the proofs. The main contribution of the thesis is the design of these interfaces.

We have designed a syntactic low-level typed assembly language, called LTAL, with a semantic model that backs up its soundness with a machine-checkable proof. The semantic modeling technique makes LTAL easily and safely extensible. It has a rich set of expressive constructors, yet its type-checking is decidable and syntax-directed. We have implemented a prototype compiler (by Chen and Fang [Chen et al., 2003] based on SML/NJ) that transforms core ML programs to SPARC code annotated with LTAL programs.

In a Proof-Carrying Code (PCC) system, an untrusted prover (code producer) must convince a trusted checker (code consumer) of the validity of a theorem by sending a proof. The proof has to be checked by a trusted checker. The proof

checking in our system is mainly done in two steps [Wu et al., 2003]. First, the LF proofs for the LTAL type checking rules are checked; this is standard LF proof checking. Second, the LTAL type checker (the set of LTAL type checking rules written in the fashion of logic clauses) is interpreted by a logic programming engine.

To this end, we have built a tiny and trustworthy proof-checker, called Flit [Appel et al., 2002; Wu et al., 2003], that permits small proof witnesses and machine-checkable proofs of the soundness of the system. Flit includes an efficient LF proof checker and a simple yet efficient logic programming engine that implements a subset Prolog. The LF checker is used to verify the soundness proof of the type system chosen by the compiler or user, and the logic programming engine is used to interpret the verified type checker to check machine code together with some proof hints from the compiler. The LTAL type checker is written in such a way that it can be interpreted by Flit logic programming engine (without backtracking).

In the future, there are several directions to extend our Foundational Proof-Carrying Code (FPCC) system. One direction is to build FPCC systems for object-oriented languages such as Java and C# based on the current system. The core part of LTAL and the soundness proof should be reusable.

Another direction is to strengthen the current safety policy and to build FPCC systems that carry proofs for stronger properties. To specify stronger safety properties at machine level, sometimes we need stronger constructors, such as types, in our Trusted Computing Base (TCB). We are currently investigating how to extend our TCB to include some type constructors so that we can specify interfaces between two low-level code modules.

Appendix A

LTAL Static Semantics

A.1 Coercion Rules

$$\frac{}{\rho; LRT \vdash_c \tau \overset{\text{cid}}{\hookrightarrow} \tau} \; \textit{CoerceId}$$

$$\frac{\rho; LRT \vdash_c \tau \overset{c_2}{\hookrightarrow} \tau' \qquad \rho; LRT \vdash_c \tau' \overset{c_1}{\hookrightarrow} \tau''}{\rho; LRT \vdash_c \tau \overset{c_1 \circ c_2}{\hookrightarrow} \tau''} \; \textit{CoerceComposition}$$

$$\frac{}{\rho; LRT \vdash_c \tau[\mu\alpha : \kappa.\tau/\alpha] \overset{\text{cfold}[\mu\alpha:\kappa.\tau]}{\hookrightarrow} \mu\alpha : \kappa.\tau} \; \textit{CoerceFold}$$

$$\frac{}{\rho; LRT \vdash_c \mu\alpha : \kappa.\tau \overset{\text{cunfold}}{\hookrightarrow} \tau[\mu\alpha : \kappa.\tau/\alpha]} \; \textit{CoerceUnfold}$$

$$\frac{\tau_1 : \kappa}{\rho; LRT \vdash_c \tau_2[\tau_1/\alpha] \overset{\text{cpack}[\tau_1, \exists\alpha:\kappa.\tau_2]}{\hookrightarrow} \exists\alpha : \kappa.\tau_2} \; \textit{CoercePack}$$

$$\frac{}{\rho; LRT \vdash_c \tau_u \overset{\text{cinjection}(\mathsf{sum}(\tau_r,\tau_u))}{\hookrightarrow} \mathsf{sum}(\tau_r, \tau_u)} \; \textit{CoerceSumInjection}$$

$$\frac{}{\rho; LRT \vdash_c \mathsf{sum}(\tau_r, \bot) \overset{\text{csum2range}}{\hookrightarrow} \tau_r} \; \textit{CoerceSum2Range}$$

$$\frac{\tau \text{ is not a union type}}{\rho; LRT \vdash_c \mathsf{sum}(\bot, \tau) \overset{\text{csum2boxedone}}{\hookrightarrow} \tau} \; \textit{CoerceSum2Boxedone}$$

$$\frac{\begin{array}{c}\tau = \tau_1 \cup \tau_2 \cup \ldots \cup \tau_n \\ \tau_i = \mathsf{field}(0, \mathsf{int}_=(t_i)) \cap \tau_i' \;\; (\text{for all } 1 \le i \le n) \\ (\alpha \text{ is a fresh type variable})\end{array}}{\rho; LRT \vdash_c \tau \overset{\text{csum2hastag}}{\hookrightarrow} \exists\alpha.\mathsf{hastag}(\alpha, \tau)} \; \textit{CoerceSum2Hastag}$$

$$\frac{\tau \text{ is neither a union type, nor a bottom type}}{\rho; LRT \vdash_c \mathsf{hastag}(\tau_{tag}, \tau) \overset{\text{cunhastag}}{\hookrightarrow} \tau} \; \textit{CoerceUnhastag}$$

$$\frac{n_1 \le n < n_2}{\rho; LRT \vdash_c \overline{n} \overset{\text{crange}[n_1,n_2]}{\hookrightarrow} \mathsf{range}(n_1, n_2)} \; \textit{CoerceSingleton2Range}$$

$$\frac{}{\rho; LRT \vdash_c \overline{n} \overset{\text{c2int32}}{\hookrightarrow} \mathsf{int}_{32}} \; \textit{CoerceSingleton2Int32}$$

$$\frac{}{\rho; LRT \vdash_c \mathsf{range}(n_1, n_2) \overset{\text{c2int32}}{\hookrightarrow} \mathsf{int}_{32}} \; \textit{CoerceRange2Int32}$$

$$\frac{i \neq 0}{\rho; LRT \vdash_c \mathsf{int}_=(i) \overset{\text{ci2nz}}{\hookrightarrow} \mathsf{int}_{\neq}(0)} \; \textit{CoerceSingleton2Nonzero}$$

$$\frac{}{\rho; LRT \vdash_c \tau_1 \overset{\text{cinj1}\,[\tau_1 \cup \tau_2]}{\hookrightarrow} \tau_1 \cup \tau_2} \; \textit{CoerceInjectionLeft}$$

$$\frac{}{\rho; LRT \vdash_c \tau_2 \overset{\text{cinj2}\,[\tau_1 \cup \tau_2]}{\hookrightarrow} \tau_1 \cup \tau_2} \; \textit{CoerceInjectionRight}$$

$$\frac{}{\rho; LRT \vdash_c \tau_1 \cap \tau_2 \overset{\text{cproj1}}{\hookrightarrow} \tau_1} \; \textit{CoerceProjectionLeft}$$

$$\frac{}{\rho; LRT \vdash_c \tau_1 \cap \tau_2 \overset{\text{cproj2}}{\hookrightarrow} \tau_2} \; \textit{CoerceProjectionRight}$$

$$\frac{}{\rho; LRT \vdash_c \mathsf{def}(\mathscr{D}) \overset{\text{cname}}{\hookrightarrow} T(\mathscr{D})} \; \textit{CoerceName}$$

$$\frac{}{\rho; LRT \vdash_c T(\mathscr{D}) \overset{\text{cdef}(\mathscr{D})}{\hookrightarrow} \mathsf{def}(\mathscr{D})} \; \textit{CoerceDef}$$

$$\frac{\rho; LRT \vdash_c \tau_1 \overset{c_1}{\hookrightarrow} \tau_1' \qquad \rho; LRT \vdash_c \tau_2 \overset{c_2}{\hookrightarrow} \tau_2'}{\rho; LRT \vdash_c \tau_1 \cup \tau_2 \overset{\text{cunion}(c_1,c_2)}{\hookrightarrow} \tau_1' \cup \tau_2'} \; \textit{CoerceUnion}$$

$$\frac{\rho; LRT \vdash_c \tau \overset{c_1}{\hookrightarrow} \tau_1 \quad \rho; LRT \vdash_c \tau \overset{c_2}{\hookrightarrow} \tau_2}{\rho; LRT \vdash_c \tau \overset{\text{cinters}(c_1,c_2)}{\hookrightarrow} \tau_1 \cap \tau_2} \; \textit{CoerceIntersection}$$

$$\frac{\rho; LRT \vdash_c \tau \overset{c}{\hookrightarrow} \tau'}{\rho; LRT \vdash_c \mathsf{field}(\tau_i, \tau) \overset{\text{cfield}(c)}{\hookrightarrow} \mathsf{field}(\tau_i, \tau')} \; \textit{CoerceField}$$

$$\frac{LRT \vdash l : \mathsf{codeptr}[\vec{\alpha} : \vec{\kappa}](m, cc, \vec{v} : \vec{\tau})}{\rho; LRT \vdash_c \mathsf{addr}(l) \overset{\text{caddr2code}}{\hookrightarrow} \mathsf{codeptr}[\vec{\alpha} : \vec{\kappa}](m, cc, \vec{v} : \vec{\tau})}\ \textit{CoerceAddr2Code}$$

$$\frac{}{\rho; LRT \vdash_c \mathsf{offset}(0, \tau) \overset{\text{coffset0}}{\hookrightarrow} \tau}\ \textit{CoerceOffset0}$$

$$\frac{}{\rho; LRT \vdash_c \tau \overset{\text{c2offset0}}{\hookrightarrow} \mathsf{offset}(0, \tau)}\ \textit{Coerce2Offset0}$$

$$\frac{\begin{array}{c}\tau' \equiv \tau^{[\uparrow^n]}, \text{ where } n = |\alpha_1 : \kappa_1; \vec{\alpha} : \vec{\kappa}| \\ m[\tau'] = m' \quad cc[\tau'] = cc' \quad \phi[\tau'] = \phi' \\ \text{where } [\cdot] \text{ denotes type application}^1\end{array}}{\rho; LRT \vdash_c \mathsf{codeptr}[\alpha_1 : \kappa_1; \vec{\alpha} : \vec{\kappa}](m, cc, \phi) \overset{\text{cptapp}(\tau)}{\hookrightarrow} \mathsf{codeptr}[\vec{\alpha} : \vec{\kappa}](m', cc', \phi')}\ \textit{CoercePtapp}$$

A.2 Instruction Typing Rules

$$\frac{LRT; \rho; \phi \vdash v : \exists \alpha : \kappa.\tau}{LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{(\alpha, v_0) = open(v)\}\ (\rho, \alpha : \kappa; \mathscr{H}; \phi, v_0 : \tau; cc)}\ \textit{InstrOpen}$$

$$\frac{\begin{array}{c}LRT; \rho; \phi \vdash v : \tau \\ \phi' = \phi, v : \tau, \text{ if } \mathrm{rmap}(v) = \mathrm{rmap}(v') \\ \phi' = (\phi \backslash v), v : \tau, \text{ if } \mathrm{rmap}(v) \neq \mathrm{rmap}(v')\end{array}}{LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{v = v'\}\ (\rho; \mathscr{H}; \phi'; cc)}\ \textit{InstrMove}$$

[1] We use de Bruijn index representation and explicit substitution calculus [Abadi et al., 1990] notation in this rule.

$$\frac{LRT; \rho; \phi \vdash v_1 : \mathsf{int} \quad LRT; \rho; \phi \vdash v_2 : \mathsf{int}}{LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{v = v_1\ op\ v_2\}\ (\rho; \mathscr{H}; (\phi \backslash v), v : \mathsf{int}; cc)}\ InstrALU$$

$$\frac{LRT; \rho; \phi \vdash v_1 : \mathsf{int}_{=}(\overline{n_1}) \quad LRT; \rho; \phi \vdash v_2 : \mathsf{int}_{=}(\overline{n_2}) \quad n = n_1 + n_2}{LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{v = v_1 +_i v_2\}\ (\rho; \mathscr{H}; (\phi \backslash v), v : \mathsf{int}_{=}(\overline{n}); cc)}\ InstrALUi$$

$$\frac{}{LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{v = sethi(n)\}\ (\rho; \mathscr{H}; (\phi \backslash v), v : \mathsf{int}_{=}(\overline{n * 4096}); cc)}\ InstrSethi$$

$$\frac{LRT; \rho; \phi \vdash v' : \tau}{LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{v = load(v')\}\ (\rho; \mathscr{H}; (\phi \backslash v), v : \tau; cc)}\ InstrLoad$$

$$\frac{LRT; \rho; \phi \vdash v' : \tau}{LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{v = store(v')\}\ (\rho; \mathscr{H}; (\phi \backslash v), v : \tau; cc)}\ InstrStore$$

$$\frac{LRT; \rho; \phi \vdash v_1 : \mathsf{addr}(f) \quad LRT; \rho; \phi \vdash v_2 : \mathsf{diff}(g, f)}{LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{v = addradd(v_1, v_2)\}\ (\rho; \mathscr{H}; (\phi \backslash v), v : \mathsf{addr}(g); cc)}\ InstrAddrAdd$$

$$\frac{LRT; \rho; \phi \vdash v_1 : \mathsf{field}(i, \tau) \quad LRT; \rho; \phi \vdash v_2 : \mathsf{int}_{=}(\overline{i})}{LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{v = select(v_1, v_2)\}\ (\rho; \mathscr{H}; (\phi \backslash v), v : \tau; cc)}\ InstrSelect$$

$$\frac{LRT; \rho; \phi \vdash v' : \mathsf{hastag}(\tau_{tag}, \tau_u) \quad \phi' = (\phi \backslash v), v : \mathsf{int}_{=}(\tau_{tag})}{LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{v = gettag(v')\}\ (\rho; \mathscr{H}; \phi'; cc)}\ \mathit{InstrGettag}$$

$$\frac{\begin{array}{c} LRT; \rho; \phi \vdash v : \tau_i \quad LRT; \rho; \phi \vdash v_i : \mathsf{int}_{=}(\bar{i}) \\ 0 \leq i < n \quad m' = max(m, i) \quad \tau' = \tau \cap (\mathsf{field}(4i, \tau_i)) \end{array}}{LRT \vdash (\rho; (n, m, \tau); \phi; cc)\ \{init(v_i, v)\}\ (\rho; (n, m', \tau'); \phi; cc)}\ \mathit{InstrInit}$$

$$\frac{}{LRT \vdash (\rho; (n, m, t); \phi; cc)\ \{v = record\}\ (\rho; \mathscr{H}; (\phi \backslash v), v : t; cc)}\ \mathit{InstrRecord}$$

$$\frac{LRT; \rho; \phi \vdash v : \mathsf{int}_{=}(n') \quad m < n' \leq n}{\begin{array}{c} LRT \vdash (\rho; (n, m, t); \phi; \mathsf{cc\_testmem}(k)) \\ \{inc\_allocptr(v)\} \\ (\rho; (n - n', -1, \top); \phi; \mathsf{cc\_none}) \end{array}}\ \mathit{InstrIncAllocptr1}$$

$$\frac{LRT; \rho; \phi \vdash v : \mathsf{int}_{=}(n') \quad m < n' \leq n \quad cc \neq \mathsf{cc\_testmem}(k)}{\begin{array}{c} LRT \vdash (\rho; (n, m, t); \phi; cc) \\ \{inc\_allocptr(v)\} \\ (\rho; (n - n', -1, \top); \phi; cc) \end{array}}\ \mathit{InstrIncAllocptr2}$$

$$\frac{\begin{array}{c} LRT; \rho; \phi \vdash v : \mathsf{codeptr}([\alpha_1 : \kappa_1, \ldots, \alpha_j : \kappa_j](m, cc', v_1 : \tau_1', \ldots, v_n : \tau_n')) \\ LRT; \rho; \phi \vdash v_i : \tau_i'[\sigma] \ \ (\text{for all } 1 \leq i \leq n) \\ \sigma = \tau_1 \cdot \tau_2 \cdot \ldots \tau_j \cdot \mathrm{id} \quad cc = cc'[\sigma] \\ \mathscr{H} = (n_1, n_2, \tau) \qquad n_1 \geq m[\sigma] \end{array}}{LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{call(v, [\tau_1, \ldots, \tau_j])\}\ (\_; \_; \_; \_)}\ \mathit{InstrCall}$$

$$\frac{\begin{array}{c} LRT \vdash l : \mathsf{codeptr}([\alpha_1 : \kappa_1, \ldots, \alpha_j : \kappa_j](m, cc', v_1 : \tau'_1, \ldots, v_n : \tau'_n)) \\ LRT; \rho; \phi \vdash v_i : \tau'_i[\sigma] \;\; (\text{for all } 1 \leq i \leq n) \\ \sigma = \tau_1 \cdot \tau_2 \cdot \ldots \tau_j \cdot \mathrm{id} \quad cc = cc'[\sigma] \\ \mathscr{H} = (n_1, n_2, \tau) \qquad n_1 \geq m[\sigma] \end{array}}{LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{calln(l, [\tau_1, \ldots, \tau_j])\}\ (\_; \_; \_; \_)}\ InstrCalln$$

$$\frac{}{LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{cmp(v_1, v_2)\}\ (\rho; \mathscr{H}; \phi; \mathsf{cc\_none})}\ InstrCmp$$

$$\frac{LRT; \rho; \phi \vdash v_1 : \mathsf{int}_=(\tau_1) \quad LRT; \rho; \phi \vdash v_2 : \mathsf{int}_=(\tau_2)}{LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{cmpcc(v_1, v_2)\}\ (\rho; \mathscr{H}; \phi; \mathsf{cc\_cmp}(\tau_1, \tau_2))}\ InstrCmpcc$$

$$\frac{LRT; \rho; \phi \vdash v : \tau \quad \phi' = (\phi \backslash v), v' : \mathsf{int}_=(\alpha) \cap \tau \quad cc' = \mathsf{cc\_testbox}(\alpha)}{LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{(\alpha, v') = testbox(v)\}\ (\rho; \mathscr{H}; \phi'; cc')}\ InstrTestbox$$

$$\frac{0 \leq n \leq 1024 \quad cc' = \mathsf{cc\_testmem}(n)}{LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{testmem(n)\}\ (\rho; \mathscr{H}; \phi; cc')}\ InstrTestmem$$

$$\frac{LRT; \rho; \mathscr{H}; \phi; cc \vdash_\ell l_1 \quad LRT; \rho; \mathscr{H}; \phi; cc \vdash_\ell l_2}{LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{if\ (\pi)\ then\ l_1\ else\ l_2\}\ (\_; \_; \_; \_)}\ InstrIf$$

$$\frac{\begin{array}{c} cc = \mathsf{cc\_testmem}(n) \quad LRT; \rho; \mathscr{H}; \phi; cc \vdash_\ell l_1 \\ LRT; \rho; (n, -1, \top); \phi; cc \vdash_\ell l_2 \end{array}}{LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{iffull\ then\ l_1\ else\ l_2\}\ (\_; \_; \_; \_)}\ InstrIffull$$

$$\frac{\begin{array}{c} LRT; \rho; \phi \vdash v : \mathsf{int}_=(\tau_\alpha) \cap (\mathsf{int}_=(\overline{0}) \cup \ldots \cup \mathsf{int}_=(\overline{n-1}) \cup \tau') \\ \tau' = \tau_1 \cup \tau_2 \cup \ldots \cup \tau_m \qquad cc = \mathsf{cc\_testbox}(\tau_\alpha) \qquad n < 256 \\ \tau_i = (\mathsf{field}(0, \mathsf{int}_=(tag_i))) \cap \tau_i' \;\; (\text{for all } 1 \leq i \leq m) \\ LRT; \rho; \mathscr{H}; \phi, v_1 : \tau'; cc \vdash_\ell l_1 \\ LRT; \rho; \mathscr{H}; \phi, v_2 : \text{range}(0, n); cc \vdash_\ell l_2 \end{array}}{LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{ifboxed\ (v)\ then\ (v_1, l_1)\ else\ (v_2, l_2)\}\ (\_; \_; \_; \_)}\ \mathit{InstrIfboxed}$$

$$\frac{\begin{array}{c} LRT; \rho; \phi \vdash v : \mathsf{hastag}(\tau_\alpha, \tau_u) \\ cc = \mathsf{cc\_cmp}(\tau_\alpha, \overline{i}) \\ \tau = \tau_1 \cup \tau_2 \cup \ldots \cup \tau_n \\ \tau_i = \mathsf{field}(0, \mathsf{int}_=(tag_i)) \cap \tau_i' \;\; (\text{for all } 1 \leq i \leq n) \\ \tau_t = \bigcup_{1 \leq j \leq n} \tau_j \;\; \text{where } i\ \pi\ tag_j \text{ holds} \\ \tau_f = \bigcup_{1 \leq k \leq n} \tau_k \;\; \text{where } i\ \pi\ tag_k \text{ does not hold} \\ LRT; \rho; \mathscr{H}; \phi, v_1 : (\mathsf{field}(0, \mathsf{int}_=(\tau_\alpha))) \cap \tau_t; cc \vdash_\ell l_1 \\ LRT; \rho; \mathscr{H}; \phi, v_2 : (\mathsf{field}(0, \mathsf{int}_=(\tau_\alpha))) \cap \tau_f; cc \vdash_\ell l_2 \end{array}}{LRT \vdash (\rho; \mathscr{H}; \phi; cc)\ \{iftag\ (\pi)\ \{v\}\ then\ (v_1, l_1)\ else\ (v_2, l_2)\}\ (\_; \_; \_; \_)}\ \mathit{InstrIftag}$$

Bibliography

Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lèvy. Explicit substitutions. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '90)*, pages 31–46. ACM Press, 1990.

Amal J. Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Department of Computer Science, Princeton University, 2004.

Amal J. Ahmed, Andrew W. Appel, and Roberto Virga. A stratified semantics of general references embeddable in higher-order logic. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS '02)*, pages 75–86. IEEE Computer Society, July 2002.

Andrew W. Appel. Foundational proof-carrying code. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*, pages 247–258. IEEE Computer Society, 2001.

Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *CCS '99: Proceedings of the 6th ACM Conference on Computer and Communications Security*, pages 52–62, New York, November 1999. ACM Press.

Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00)*, pages 243–253, New York, January 2000. ACM Press.

Andrew W. Appel and David B. MacQueen. A Standard ML compiler. In *Proceedings of the Functional Programming Languages and Computer Architecture*, pages 301–324. Springer-Verlag, 1987. ISBN 3-540-18317-5. Lecture Notes In Computer Science (Vol. 274).

Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In *Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming (PLILP '91)*, pages 1–13. Springer-Verlag, 1991. Lecture Notes in Computer Science (Vol. 528).

Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5):657–683, September 2001.

Andrew W. Appel, Neophytos Michael, Aaron Stump, and Roberto Virga. A trustworthy proof checker. In Iliano Cervesato, editor, *Foundations of Computer Security Workshop*, pages 37–48. DIKU, July 2002.

Andrew W. Appel, Neophytos G. Michael, Aaron Stump, and Roberto Virga. A trustworthy proof checker. *Journal of Automated Reasoning*, 31:231–260, 2003.

Andrew W. Appel and Daniel C. Wang. JVM TCB: Measurements of the trusted computing base of Java virtual machines. Technical Report CS-TR-647-02, Princeton University, 2002.

Matthias Blume and Andrew W. Appel. Lambda-splitting: A higher-order approach to cross-module optimizations. In *Proc. ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, pages 112–124, New York, June 1997. ACM Press.

Frederick Brooks. *The Mythical Man-Month.* Addison-Wesley, Boston, 1975.

Bor-Yuh Evan Chang, Adam Chlipala, George C. Necula, and Robert R. Schneck. The open verifier framework for foundational verifiers. In *TLDI '05: Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 1–12, New York, NY, USA, 2005. ACM Press. ISBN 1-58113-999-3.

Juan Chen. *A Low-Level Typed Assembly Language with a Machine-checkable Soundness Proof.* PhD thesis, Department of Computer Science, Princeton University, 2004.

Juan Chen and David Tarditi. A simple typed intermediate language for object-oriented languages. In *Proceedings of the 32th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*, pages 38–49, New York, NY, USA, 2005. ACM Press. ISBN 1-58113-830-X.

Juan Chen, Dinghao Wu, Andrew W. Appel, and Hai Fang. A provably sound TAL for back-end optimization. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)*, pages 208–219, New York, June 2003. ACM Press.

Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Ken Cline, and Mark Plesko. A certifying compiler for Java. In *Proceedings of the ACM SIGPLAN*

2000 Conference on Programming Language Design and Implementation (PLDI '00), pages 95–107, New York, June 2000. ACM Press.

Karl Crary. Toward a foundational typed assembly language. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*, pages 198–212, New York, January 2003. ACM Press.

Drew Dean, Edward W. Felten, Dan S. Wallach, and Dirk Balfanz. Java security: Web browers and beyond. In Dorothy E. Denning and Peter J. Denning, editors, *Internet Beseiged: Countering Cyberspace Scofflaws*. ACM Press (New York), October 1997.

Lal George. MLRISC: Customizable and reusable code generators. Technical report, Bell Laboratories, May 1997.

M.J.C. Gordon. Mechanizing programming logics in higher-order logic. In G.M. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automatic Theorem Proving*, pages 387–439, Berlin, 1988. Springer-Verlag.

Nadeem Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS '02)*, pages 89–100. IEEE Computer Society, July 2002.

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and

J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, Inc., New York, NY, 1980.

Christopher League, Zhong Shao, and Valery Trifonov. Precision in practice: A type-preserving Java compiler. In *Proceedings of the 12th International Conference on Compiler Construction (CC '03), Lecture Notes in Computer Science 2622*. Springer-Verlag, April 2003.

Allen Leung and Lal George. *MLRISC Annotations*. Bell Laboratories. Available online at http://cm.bell-labs.com/cm/cs/what/smlnj/compiler-notes/annotations.ps.

Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Boston, 1996.

Brian Martin. Social aspects of the Love Bug virus. Available online at http://www.attrition.org/~jericho/works/security/lovebug.html, 2000.

Gary McGraw and Greg Morrisett. Attacking malicious code: A report to the infosec research council. *IEEE Software*, 17(5):33–41, September 2000.

Neophytos G. Michael and Andrew W. Appel. Machine instruction syntax and semantics in higher-order logic. In *CADE-17: 17th International Conference on Automated Deduction*, pages 7–24, Berlin, June 2000. Springer-Verlag. Lecture Notes in Artificial Intelligence (LNAI 1831).

Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A real-

istic typed assembly language. In *Second ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, New York, 1999a. ACM Press. INRIA Technical Report 0288, March 1999.

Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 12(1):43–88, January 2002.

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*, pages 85–97, New York, January 1998. ACM Press.

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999b.

George Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*, pages 106–119, New York, January 1997. ACM Press.

George Necula and Peter Lee. Safe kernel extensions without runtime checking. In *2nd USENIX symposium on Operating System Design and Implementation*, Seattle, October 1996.

George C. Necula and Peter Lee. Efficient representation and validation of proofs. In *Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science (LICS '98)*, pages 93–104. IEEE Computer Society, 1998.

George C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '01)*, pages 142–154, New York, January 2001. ACM Press.

Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

Frank Pfenning. Elf: A meta-language for deductive systems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction (CADE-12)*, pages 811–815, Nancy, France, June 1994. Springer-Verlag. Lecture Notes in Artificial Intelligence (LNAI 814).

Frank Pfenning and Carsten Schürmann. System description: Twelf—a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag. Lecture Notes in Artificial Intelligence (LNAI 1632).

Frank Pfenning and Carsten Schürmann. *Twelf User's Guide (Version 1.4)*. Carnegie Mellon University, 2002.

David S. Platt. *Introducing Microsoft .NET*. Microsoft Press, 2001.

Robert Pollack. How to believe a machine-checked proof. In G. Sambin and J. Smith, editors, *Twenty Five Years of Constructive Type Theory*. Oxford University Press, 1998.

Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Boston, 1986.

Ravi Sethi. *Programming Languages: Concepts and Constructs*. Addison-Wesley, Reading, Mass., 1989.

Zhong Shao. An overview of the FLINT/ML compiler. In *Proceedings of the 1997 ACM SIGPLAN Workshop on Types in Compilation*, New York, June 1997. ACM Press.

Zhong Shao and Andrew W. Appel. A type-based compiler for Standard ML. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI '95)*, pages 116–129, New York, 1995. ACM Press. ISBN 0-89791-697-2.

Robert M. Slade. The Melissa macro virus. Available online at http://sun.soci.niu.edu/~rslade/melissa.txt, 1999.

Kedar N. Swadi. *Typed Machine Language*. PhD thesis, Department of Computer Science, Princeton University, 2003.

Kedar N. Swadi and Andrew W. Appel. Typed machine language and its semantics. Available online at http://www.cs.princeton.edu/~appel/papers, July 2001.

Gang Tan. *A Compositional Logic for Control Flow and Its Application in Foundational Proof-Carrying Code*. PhD thesis, Department of Computer Science, Princeton University, 2005.

Gang Tan, Andrew W. Appel, Kedar Swadi, and Dinghao Wu. Construction of a semantic model for a typed assembly language. In *Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '04)*, pages 30–43. Springer-Verlag, January 2004. Lecture Notes in Computer Science (LNCS 2937).

D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: a type-directed optimizing compiler for ML. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation (PLDI '96)*, pages 181–192, New York, 1996. ACM Press. ISBN 0-89791-795-2.

M. Wahab. Verification and abstraction of flow-graph programs with pointers and computed jumps. Research Report CS-RR-354, Department of Computer Science, University of Warwick, Coventry, UK, 1998.

Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, pages 203–216, New York, 1993. ACM Press. ISBN 0-89791-632-8.

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

Dinghao Wu, Andrew W. Appel, and Aaron Stump. Foundational proof checkers with small witnesses. In *Proceedings of the Fifth ACM-SIGPLAN International*

Conference on Principles and Practice of Declarative Programming (PPDP '03), pages 264–274, New York, August 2003. ACM Press.

Hongwei Xi and Robert Harper. A dependently typed assembly language. In *Proceedings of the 2001 ACM SIGPLAN International Conference on Functional Programming (ICFP '01)*, pages 169–180, New York, September 2001. ACM Press.