A Low-level Typed Assembly Language with a Machine-checkable Soundness Proof

Juan Chen

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

June 2004

Abstract

To reason about mobile machine code safety, Proof-Carrying Code framework requires machine code accompanied by a proof of safety. Typed assembly languages provide a way to automatically generate such safety proofs. But the soundness proofs of most existing typed assembly languages are hand-written and cannot be machine-checked, which is worrisome for such large calculi.

In this dissertation I will explain a low-level typed assembly language (LTAL) with a semantic model that proves LTAL's soundness with a machine-checkable proof. 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.

I will also explain a prototype system that uses LTAL to compile core ML to Sparc code and generate safety proofs. I will show how we were able to build type-preserving back end based on an untyped one, without restricting low-level optimizations and without knowledge of any type system pervading the instruction selector and register allocator.

Acknowledgments

First of all I would like to thank my advisor Professor Andrew W. Appel, who has given tremendous help and support during my stay at Princeton. He is always a source of advice and guidance when I need it. Thanks also to the two readers David Walker and Zhong Shao who spent so much time on reading my thesis and gave me very insightful suggestions, and to the two non-readers David August and Sharad Malik for their constructive comments.

I have benefited greatly from working and talking with the other SIP group members, Amal Ahmed, Neophytos Michael, Xinming Ou, Kedar Swadi, Gang Tan, and Dinghao Wu. Thanks to Hai Fang who implemented the translation from FLINT to NFLINT in the compiler.

I am also grateful to all the professors, staff members and graduate students who make the computer science department cozy and nourish.

Thanks to DARPA grant F30602-99-1-0519 and NSF grant CCR-9974553 for providing financial support.

Last I would thank my parents and my husband for their love and understanding, and my baby twin girls who have brought endless joy to my life.

To my husband Hua and my twin girls Amy and Amanda

Contents

List of Figures

Chapter 1

Introduction

The explosive growth of the Internet brings us a globally connected computing environment. It makes possible very large-scale and complex software systems, since we can share resources and information. But at the same time, we become more vulnerable to attacks. First, it is much easier to attack a connected network than an isolated host (physical access is not a necessity). Second, the damage spreads fast and widely. In March 1999, the Computer Emergency Response Team (CERT) received reports from about 300 organizations that more than 100,000 individual hosts were infected by Melissa virus over a weekend [3]. One site reported receiving 32,000 copies of infected messages within 45 minutes. Third, system failures and malicious attacks cause more disastrous damages, as we are more dependent on the Internet. Think about what it would be like if the banks, flight control systems, or the defense systems have to shut down because of attacks. The security problems of the Internet impose great challenges on the research community. In this thesis, I will address one important topic—how to verify that the code we run is safe. Part of this thesis work has been described in a paper [19].

1.1 Code Safety

When we get a piece of code, before running it on local machines, we would like some safety guarantees about the code. For example, it does not delete important files, or send private data to outsiders.

Given the size and complexity of modern software, verifying code safety is a very challenging problem. There are many software products from various vendors running even on a single PC, some of which are huge. The Windows XP operating system has about 40 million lines of code. Such large and complicated software products certainly contain bugs. Recently Microsoft has issued quite a few patches to prevent bugs in the Windows operating system from being utilized to attack the machines. It is even more difficult for the end users to verify the executables because they normally don't have the source code.

The situation gets worse with the wide usage of mobile code. There are many forms of mobile code, including Java applets, web script, and ActiveX. They play important roles in e-commence and e-business. For example, many online shopping website requires clients use Java-enabled browsers. A browser can download Java applets from a remote server and run them on the local machine, sometimes even without the awareness of the users.

Mobile code provides convenience, flexibility, and efficiency. First, we can share code. Second, most mobile code can run in heterogeneous systems, so porting it to various environments is not an issue. Next, we can easily add or replace mobile components to extend a system. Last, we can improve performance by properly distributing computation. For instance, communication between cellular phones and servers is expensive and unreliable. It is sometimes beneficial to download small applications to cellular phones and run them locally to reduce communication with servers.

Since we cannot avoid using these software products, we need mechanisms to provide some assurance about their safety. What is safe may have different meanings depending on the systems and applications. It is formally defined by a *safety policy* in a specific setting. Security mechanisms should provide support for specifying safety policies and enforcing them.

The Trusted Computing Base (TCB) is the totality of the system parts that are responsible for enforcing security policies. In order for the TCB to enforce correctly a security policy, the TCB should not contain bugs that could lead to security holes. Therefore, a smaller and simpler TCB is generally better, since it is easier to test and reason about.

1.2 Traditional Approaches

Traditional approaches to ensure code safety include:

Code Signing Cryptography provides a method for identification. The code writer can digitally sign his code before distributing it. After receiving the code, the code user verifies whether it comes from a trusted entity and it is not altered during transmission. If so, the code is run as trusted. Microsoft's Authenticode provides utilities to sign and check codes [1].

The disadvantages of code signing are: first, code signing only tells us the origin of the code, but does not guarantee that the code is safe; second, it needs a complex hierarchy of certification authorities and key distribution.

Virus Scanning Many commercial tools (such as those provided by McAfee and Norton) can scan for viruses or suspicious code, by looking for "signatures" (often specific bit patterns). But these tools only scan for known viruses, not even unknown variants or disguises of existing viruses.

Firewall Firewalls define the boundary of an "internal network". They examine everything that comes into the internal network, and decide access according to its identity. A complex system might use many firewalls, and it is very difficult to verify that the firewalls really implement a safety policy, and that bypassing the firewalls is impossible. Sometimes a system administrator needs special analysis tools to analyze firewall configurations [39, 2].

All the above techniques examine surface-level code properties. They cannot provide strong enough guarantees about the code behavior. Another traditional approach—hardware and OS protection—does a better job.

Hardware-OS Protection Hardware and operating system protection might be one of the most widely used approaches. To protect memory, each process has its own address space, and has no access to the memory used by others. The mapping from each process's address space to physical memory is managed by the kernel. Furthermore, by separating system mode from user mode, only through specific interfaces (system calls) can a user program access kernel-protected resources.

This protection is very effective on system resources such as memory, files, and I/O devices, but its disadvantages are obvious: the safety policies that can be enforced are limited (it only protects those resources accessed through system calls); context switch between different processes is very expensive.

Recently, language-based security has attracted more and more attention because it is based more on the code semantics than traditional approaches, thus it provides more flexibility and stronger safety guarantees.

1.3 Language-based Security

Schneider *et al.* defined language-based security as "mechanisms based on programming analysis and modification" [53]. The advantage of language-based security is that it checks the intrinsic properties of untrusted code, and it could support more fine-grained controls than traditional approaches.

1.3.1 Safe Languages

There are many type-safe languages, such as Java, ML, and Modula-3 [29, 41, 48]. The dynamic semantics of a safe language specifies explicitly desired behaviors and undesired ones. Type safety guarantees that well-typed programs do not have specific undesired behaviors. As one application of type safety, the SPIN extensible operating system allows kernel extensions written with Modula-3 and compiled by a trusted compiler [15]. The result of the compiler can be trusted to be safe and run in the kernel. This approach relies on the assumption that the compiler compiles a safe source program to a safe target program, which is often broken because of the size and the complexity of the compiler.

1.3.2 Java Verification

Java enforces its security model through the Java Virtual Machine (JVM): the bytecode verifier allows only valid bytecode to pass; the class loader defines separate name spaces for trusted classes and for untrusted ones, so that they don't interfere with each other; the security manager checks and restricts access to crucial system resources, such as disk and network connections. This model ensures type safety and memory safety.

After bytecode is verified, a Just-In-Time (JIT) compiler or an interpreter translates it to native code for execution. Since this translation is after security verification, the JIT compiler or the interpreter needs to be trusted, as do the runtime system and core libraries that are necessary to run the program. Any bugs in these components might be taken advantage of by a malicious attacker to translate safe bytecode to unsafe native code.

Therefore, the JIT compiler is in the TCB, which could make the TCB size more than 100,000 lines of code [13]. Such a large-scale software is very likely to contain bugs. To make things worse, there are no complete formal proofs proving that the Java security model does not have any bugs in the design[1]. As a matter of fact, many security bugs have been found [22].

1.3.3 Inlined Reference Monitor

A reference monitor runs in parallel with a program, observes the program's behavior, and stops the program when the program is about to violate the security policy [52]. The framework is shown in Figure 1.1. During the execution of a program, each security "sensitive" operation is checked by the reference monitor before it is executed. The program continues to run if the operation is valid. Otherwise, it terminates.

[1] There are several proofs of the soundness of the Java type system [23, 59, 49], but they reason about only subsets of Java.

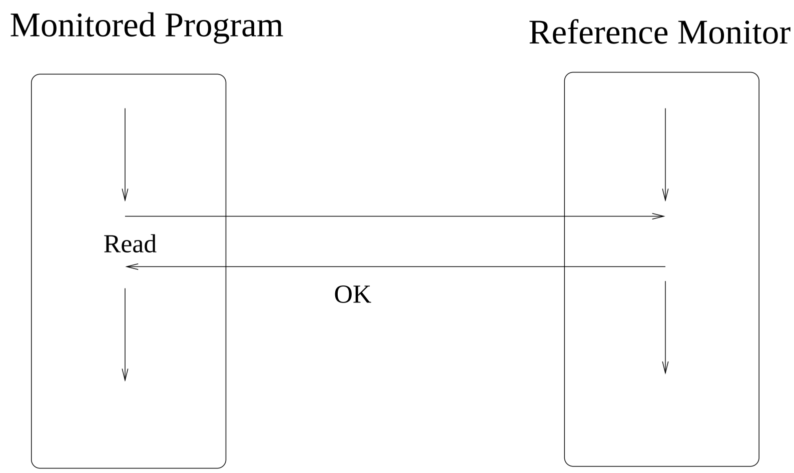

Figure 1.1: Reference Monitor

When the reference monitor and the monitored program are in separate address spaces, an expensive context switch is involved when control transfers between them before and after security checks.

To remove the performance cost, we can *inline* the reference monitor: let it share the same address space with the monitored program. Before executing a program, we can rewrite it to insert code that implements a monitor. The result program is guaranteed not to violate the security policy. Software Fault Isolation (SFI) uses this approach to ensure memory safety [64]. New code is inserted before each memory operation in a program. The inserted code checks at run time that the memory operation accesses only certain predefined memory regions.

Schneider extended SFI to enforce safety policies that could be specified as security automata (finite-state automata) [24, 25]. An inlined reference monitor is implemented as an automaton simulation, and merged into the monitored program by a rewriter. The automaton simulation could be simplified by program analysis.

Schneider's automata can express a subset of safety properties (bad things will not happen), but not liveness properties (good things will eventually happen) [52].

Bauer *et al.* added more functionality to security automata, so that they can specify more safety properties [14, 37].

IRM performs checks at run time. The run time overhead could be substantial (amount to 20% in some cases for SFI [64]).

1.3.4 Proof-Carrying Code

Necula and Lee proposed Proof-Carrying Code (PCC) to guarantee the safety of untrusted code [47, 46]. The key idea is to accompany a piece of native code with a safety proof. A code consumer, where untrusted code runs, specifies safety rules; a code producer, who provides the code, generates a formal proof that proves the code obeys the safety rules. The code consumer gets both the code and the safety proof, and checks the proof before executing the code.

Checking the compiler output makes it unnecessary to trust the compiler any more. Imagine how complicated a production-quality optimizing compiler could be; it is much easier to check the output than to verify the compiler itself.

Necula and Lee's PCC framework is shown in Figure 1.2. The compiler compiles Java bytecode to native code (annotated with type information), and generates a safety proof as well. A Verification Condition Generator (VCGen) extracts from the annotated native code a Verification Condition (VC, a predicate in first-order predicate logic). A proof checker checks whether the safety proof proves VC. VC has the property that if it holds, the native code is safe. VCGen examines a machine-code program instruction by instruction and calculates the weakest preconditions for each instruction in Hoare-logic style.

The proofs are encoded and checked in Logical Framework (LF) [31]. Predicates are presented as types in LF, and proofs as expressions. Proof checking is type

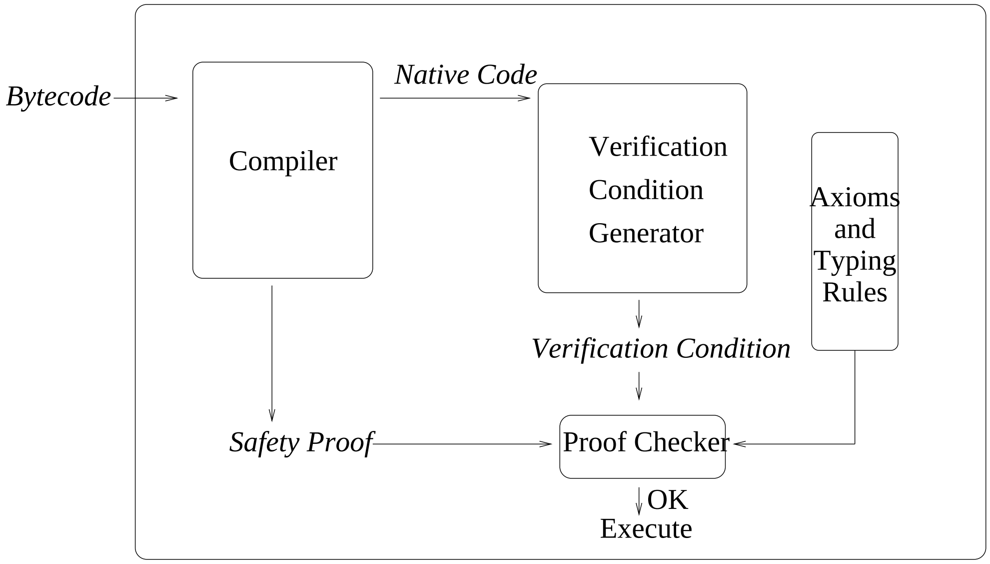

Figure 1.2: Proof-Carrying Code Framework

checking in LF: to check whether a proof *pf* proves a predicate P, we only need to check whether the type of the representation of *pf* is the representation of P.

PCC is a very appealing framework for statically reasoning about code safety:

- PCC is a general framework. The code consumer does not care about how the proofs are generated. It even accepts hand-written proofs.

- PCC is tamper-proof. If the code and/or the safety proof are modified accidentally or maliciously, the proof checker either rejects the safety proof, or the modified code is still safe to run. Neither case will harm the code consumer.

- The code producer normally has more knowledge about program safety than the code consumer. The work of the code consumer is dramatically reduced since it only does simple and fast proof-checking.

- There is no runtime cost. The proof is checked before the code is executed. We can run the code as many times as we want after checking.

The TCB consists of the VCGen, which has to generate the right VC (safety theorem), the proof checker, which should only allow valid proofs, and the axioms and typing rules, which are the base of the proofing system.

This VC-based verification builds the type system and machine instruction semantics into the algorithm for formulating the safety predicate. As a result, first, it restricts the languages the code producer can use; second, the type system needs to be trusted, but it does not have a formal soundness proof; third, VCGen must be trusted to generate the right formula, but it is a large program (23,000 lines of C code [13]), thus difficult to trust as bug-free.

1.3.5 Typed Assembly Language

Compilers that translate source programs to target language and construct safety proofs are called "certifying compilers". Typed Assembly Language (TAL) proposed by Morrisett *et al.* is an instance of certifying compilation that constructs type-safety proofs [45].

The key idea of typed assembly language is to preserve type information throughout compilation. Traditional untyped compilers throw away types after type-checking source programs. Today some compilers translate source programs to statically typed intermediate languages, and uses type information to guide optimizations [54, 62, 38, 34, 35]. But these compilers do not preserve types to the target languages. At some stage, type information is gone. In contrast, typed assembly languages keep type information to the assembly level. The compiler translates a well-typed source program to a well-typed assembly program (see Figure 1.3).

Type-preserving compilation has been proved to be very beneficial: first, it enables many optimizations in the compiler, such as unboxing and tag-free garbage

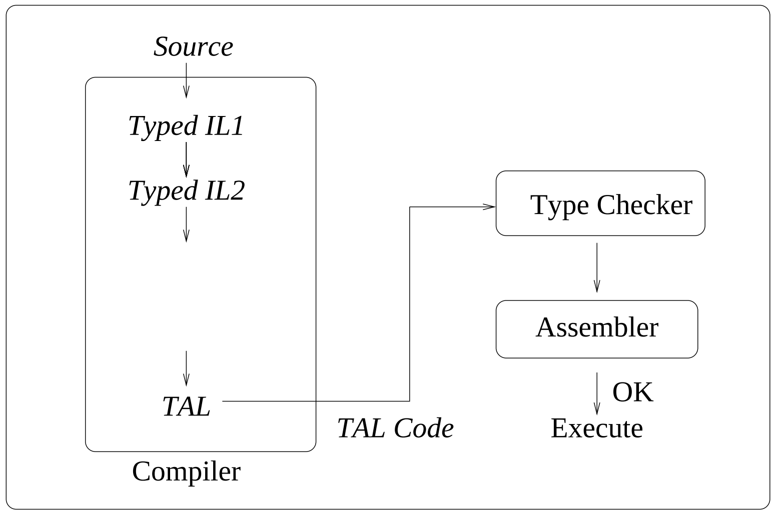

Figure 1.3: Typed Assembly Language Framework

collection; second, type checking each intermediate language and target language provides a way to debug compiler transformations and optimizations; third, a typed target-language program translated from untrusted code can serve as a type-safety proof of itself. Well-typedness of the target-language program can guarantee that it does not violate abstractions enforced by the type system. And the compiler generates this safety proof automatically.

The first TAL was based on a conventional RISC assembly language, from a very simple source language with polymorphism, functions and records [45]. There were many extensions after that. Morrisett *et al.* added stack-allocation, linking, higher-order constructors to support compilation from safe C-based source language Popcorn to Intel IA32-based assembly language TALx86 [43]. To reason about optimizations such as array bounds checking, Xi and Pfenning provided in the type system dependent types and integer arithmetic [69]. Hamid *et al.* proposed a feather-weight TAL with a machine-checkable syntactic soundness proof [30]. Crary extended the syntactic approach with machine-checked soundness proof [21].

Like Necula and Lee's PCC system, TALs rely on the soundness of their type systems. Some previous work has informally proved the soundness of some typed assembly languages as a metatheorem [44, 43, 69]. It is hard to manage the soundness proofs and avoid errors when scaling up to realistic type systems for real compilers.

Another problem with many TAL extensions is "macro instructions". Each macro instruction will be expanded to a sequence of machine instructions after type checking. The sequence is like a "black box" to the type checker. Therefore, the intermediate states during executing the sequence can not be modeled, and the type system can not reason about optimizations performed on these sequences, such as instruction scheduling.

1.3.6 Foundational Proof-Carrying Code

The motivation of Foundational Proof-Carrying Code (FPCC) 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. The TCB of the FPCC system consists of the specification of the safety policy, machine instruction semantics, and the proof checker. In the current implementation, it is less than 2,700 lines of code [12, 68], of which more than half is the specification of the Sparc instruction set. To make the TCB minimal, FPCC uses Church's higher-order logic with a few axioms of arithmetic, gives types a semantic model to move the type system out of the TCB, and models machine instructions by a step relation between machine states; a VCGen is avoided entirely [9].

In order to support contravariant recursive datatypes and mutable fields, FPCC models types as predicates on states, approximation indices [11], and type levels [6]. There is an abstraction layer, Typed Machine Language (TML) [58], to hide the complex semantic models for types. TML provides a rich set of constructors for types, type maps, 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 type checking for it 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: LTAL constructors have a semantic model in terms of TML.

The FPCC system follows a TAL approach for automatic proof generation. Part of this thesis work is the design of a low-level typed assembly language (LTAL). LTAL is the interface between the compiler and the checker: the compiler compiles a source program to machine code annotated by an LTAL program.

FPCC needs its own typed assembly language because it intends 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 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 LTAL does not accommodate the filling of branch-delay slots on the 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.

The soundness of LTAL typing rules is being proved not by a metatheorem as in TAL, but by their semantic model [58, 60], bottom up: first use higher-order logic with axioms for arithmetic to prove lemmas about machine instructions and types, then prove the TML typing rules based on these lemmas, then prove the soundness of LTAL typing rules in the TML model. Each typing rule is represented as a derived lemma in the logic. The soundness proof is nearing completion. We have not seen intrinsic difficulties in the unfinished part, only some engineering work.

LTAL benefits from its semantic model in many aspects: first, it is more scalable. Adding new rules that can be described in the semantic model generally does not affect the soundness of existing rules, which from our experience was proved very convenient in evolving the design. Second, it is more secure because the TCB is smaller without the typing rules in it. Third, TML connects LTAL to real machine instruction semantics, thus bridges the gap between typed assembly language and machine language.

The FPCC framework is shown in Figure 1.4. A source program is compiled into a machine-code program and an LTAL program. The "code consumer" receives the LTAL rules, along with their soundness proof; checks the soundness proof [12, 68]; and then runs the LTAL checker, which is a simple computation (like Prolog but without backtracking and with only a very limited form of unification).

LTAL is not intended as a universal TAL. Instead, it is extensible. The 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. The first version of the semantic model was very simple [9]. Adding contravariant recursive types [11] and mutable record fields [6] did violate previous assumptions and require nonmodular rewrites. But now the model is very powerful and general: none of the existing LTAL soundness proofs will need to be touched when more operators are necessary to handle extensible sums, various kinds of exception handling

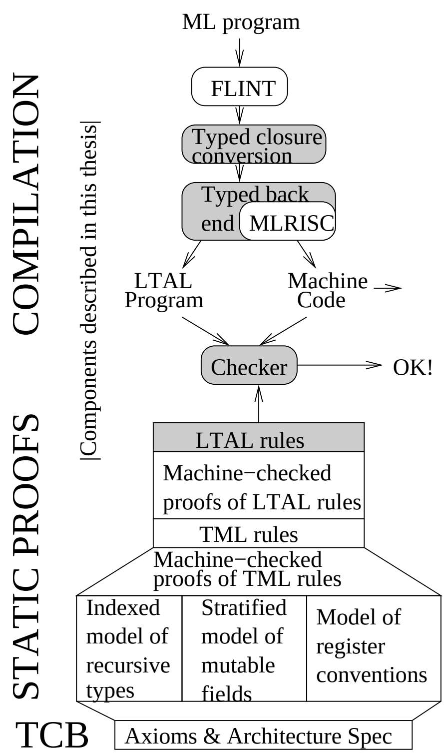

Figure 1.4: Foundational PCC Framework

mechanisms, various kinds of multidimensional arrays (with or without pointer indirections), or arbitrary predicates on scalar values. However, a more powerful model might be required to suppport Java objects.

1.3.7 FPCC-ML Compiler

The FPCC-ML compiler transforms core ML (ML without the module system) into Sparc code with LTAL annotations. At present it omits exceptions and strings. The compiler was built based on the Standard ML of New Jersey system.

There are several stages: the compiler reuses the front end of SML/NJ to translate source ML programs to FLINT (a typed intermediate language based on F_ω) [54]. The newly built typed CPS-conversion and closure conversion phases generate NFLINT (a typed intermediate language like Morrisett's λ_C [45]). The next few phases break down complex instructions, build basic blocks, and insert coercions to get machine-independent LTAL programs. The back end 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 [26]. The difficulty is to make MLRISC preserve and manipulate type information, without rewriting the MLRISC or making it dependent on the particular type system. Fortunately, MLRISC already had some support for an annotation mechanism [36] that permits "comments" on the instructions; the FPCC-ML compiler generalized this mechanism and used it to propagate types.

1.3.8 Checker

The 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 [12, 68]. The checker reviews 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. 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.

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 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. The underlying semantic model can prove that well-typedness implies safety:

```
judge_prog H P -> safe_program P.
```

The predicate `safe_program` is the machine-level safety policy. When the checker succeeds on the goal `judge_prog H P`, it applies this lemma to get a proof of `safe_program P`.

1.4 Overview of the thesis

The main contribution of this work is:

- The design of LTAL—a syntactic low-level typed assembly language whose soundness is backed up by a semantic model with a machine-checkable proof;
- The implementation of FPCC-ML—a certifying compiler that transforms core ML programs to Sparc code as well as LTAL annotations. The safety proofs for the Sparc programs are constructed based on the LTAL annotations.

The semantic model and soundness proof of LTAL is other people's work. I will explain only the basic ideas and will not present any formal proofs. The features in core ML that we haven't supported yet include exceptions, strings, floating point numbers, references and mutally recursive datatypes. I speculate that LTAL does not need a significant change to support these features.

The rest of the thesis is organized as follows:

Chapter 2 explains the features, syntax and typing rules of LTAL. The key difficulty (and achievement) in designing LTAL is to make LTAL very expressive, yet keep type checking simple and efficient.

Chapter 3 explains the semantic model of LTAL types and typing rules. The model provides a semantic approach to prove the soundness of the LTAL type system. Types are translated to predicates in the underlying logic and typing rules are lemmas. All the proofs can be machine-checked.

Chapter 4 describes memory allocation in LTAL. During heap allocation, the type checker needs to keep track of the status of the heap and the structures being initialized. Modeling each step of allocation enables several optimizations macro instructions do not allow. I will explain allocation of both records and arrays.

Chapter 5 addresses user-defined datatypes. The compiler uses tagged unions to represent datatypes. When the compiler does tag discrimination, it needs to "remember" the connection between tags and their associated values even after fetching tags. Also, to be able to refine types after checking tags, the type checker needs some data-flow analysis. I will show how to encode the connection and data-flow analysis in the type system.

Chapter 6 demonstrates how position-independent code is achieved in LTAL. Position-independent code leaves the linker out of the TCB. I present the transformation and the typing rules.

Chapter 7 gives an overview of the certifying compiler that generates Sparc code and LTAL programs from core ML programs. The emphasis is building the typed back end based on an existing untyped one.

Finally, Chapter 8 summarizes the contributions of the thesis, and discusses some related work and future directions.

Chapter 2

Low-level Typed Assembly Language

As shown in Chapter 1, FPCC generates safety proofs automatically by type-preserving compilation that translates well-typed source programs to well-typed assembly programs and rejects ill-typed source programs. The typed assembly language should be capable of expressing the source language core ML (with higher-order functions, polymorphism and user-defined datatypes), low-level compiler optimizations (such as instruction scheduling), and target architecture Sparc (with condition codes). It needs to reason about (type-check) each single operation of the Sparc architecture, which makes the type system complicated. But the type-checking should still be simple and efficient to make it practical. None of the previous typed assembly languages have all these features. In this chapter, I explain Low-level Typed Assembly Language (LTAL), the typed assembly language used in FPCC with all the desired features. I briefly describe the syntax and the type-checking. More details will be explained later in the thesis.

2.1 LTAL Features

LTAL design and implementation has the following desirable properties, some of which are shared by some other TAL and PCC systems:

Compiles a "real" source language. LTAL handles 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. The compiler generates high-quality Sparc code.

Foundational specification. The logical specification of the safety property guaranteed by the system is concise and independent of any type system: the prototype system guarantees memory safety and that only a certain subset of Sparc instructions will be executed [8]. Furthermore, the specification relates to the actual *machine language* to be executed—not assembly language— instruction encodings are modeled and checked explicitly.

Machine-checked proof. LTAL has a machine-checked soundness proof (mostly finished)—that is, if an LTAL program type-checks, the corresponding machine code is safe. Unlike any other TAL or PCC system, this proof is with respect to a minimal set of axioms, the largest part of which is a specification (in logic) 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, the soundness proof is checkable in a very minimal logic: the trusted base of the system (including axioms, machine specification, and a C program implementing LF checking) is less than 2700 lines of code [12, 68], 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 discrimination, or memory allocation. This inhibits optimizations such as hoisting and scheduling.[1] Each of the LTAL instructions corresponds to at most one machine instruction.

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 the prototype compiler does not exploit all of this power yet).

Position-independent code. To avoid the need to trust a linker, the compiler generates and checks typed position-independent code—even in the presence of long jumps and of operations that move code addresses into pointer variables and closures[2].

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, and making type-checking a bit more efficient.

Syntax-directed. Type-checking LTAL is syntax-directed; that way, if a compiler generates a well-typed LTAL program it doesn't have to worry about whether the checking algorithm will be smart enough to find a proof.

[1] In other systems, these optimizations can be done by a trusted assembler, but need to be trusted bug-free, whereas the FPCC system does not need to trust either the assembler or the optimizer.

[2] As in SML/NJ, compilation units in FPCC-ML link to each other by using closures.

Key:
○ partially
⊙ nearly
● completely

	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 reprs.	8 Dataflow analysis	9 Position-independent code	10 Basic blocks	11 Syntax-directed checking
SpecialJ [20]	●	●				○		●			
TALx86 [45]	⊙	●				○	●	●		●	○
DTAL [69]								●			
FTAL [30]			⊙	●							●
TALT [21]		●	⊙	⊙			●	●	○		
Our LTAL	⊙	●	●	⊙	●	⊙	●	●	●	●	●

Figure 2.1: Comparison of typed assembly languages

Table 2.1 shows the comparison of LTAL to other TALs with regard to the implementation of these features. This table lists only those features we need most. Of course there are other features for which some other TALs have better support than LTAL. In this sense the comparison is incomplete. I don't claim that LTAL is better than all other TALs, but that LTAL is more suitable for our system.

2.2 Syntax

LTAL is a calculus with conventional features such as variable names and scoping rules. The LTAL syntax is shown in Figures 2.2, 2.4 and 2.5.

$$
\begin{array}{lll}
\kappa & ::= \Omega \mid \Omega_N & \textbf{Kinds} \\
\tau & ::= \alpha \mid \mathrm{int} \mid \exists\alpha:\kappa.\tau \mid \mu\alpha:\kappa.\tau \mid \mathrm{boxed} & \textbf{Types} \\
 & \mid \ \overline{n} \mid \tau_1 + \tau_2 \mid \mathrm{freem} & \\
 & \mid \ \mathrm{field}\ i\ \tau \mid \tau_1 \cap \tau_2 \mid \tau_1 \cup \tau_2 \mid \mathrm{int}_\pi\ \tau \mid \mathrm{range}[\tau_1, \tau_2] & \\
 & \mid \ \mathrm{codeptr}[\alpha_1:\kappa_1, \ldots, \alpha_j:\kappa_j](m, cc,\ v_1:\tau_1, \ldots, v_n:\tau_n) & \\
 & \mid \ \mathrm{addr}\ l \mid \mathrm{diff}(l_1, l_2) \mid \mathrm{def}\,\Bbbk \mid \mathrm{array}(\tau_1, \tau_2) \mid \mathrm{offset}\ i\ \tau & \\
\pi & ::= \ = \ \mid \ \neq \ \mid \ > \ \mid \ \geq \ \mid \ < \ \mid \ \leq & \textbf{Arith. Compares} \\
cc & ::= \mathrm{cc\_cmp}(\tau_1, \tau_2) & \textbf{ConditionCodes} \\
 & \mid \ \mathrm{cc\_testbox}\ \tau \mid \mathrm{cc\_none} & \\
v & ::= x \mid i \mid l \mid c(v) \mid \mathrm{vdiff}(l_1, l_2) & \textbf{Values} \\
c & ::= \mathrm{cid} \mid c_1 \circ c_2 \mid \mathrm{cpack}(\tau_1, \tau_2) & \textbf{Coercions} \\
 & \mid \ \mathrm{cfold}[\tau] \mid \mathrm{cunfold} \mid \mathrm{crange}[n_1, n_2] & \\
 & \mid \ \mathrm{cinj1}\ [\tau] \mid \mathrm{cinj2}\ [\tau] \mid \mathrm{cproj1} \mid \mathrm{cproj2} \mid \mathrm{c2inters}(c_1, c_2) & \\
 & \mid \ \mathrm{cname} \mid \mathrm{cdef}\ \Bbbk & \\
 & \mid \ \mathrm{c2int32} \mid \mathrm{cptapp}[\tau] \mid \mathrm{coffset0} \mid \mathrm{c2offset0} & \\
 & \mid \ \mathrm{caddr2code} \mid \mathrm{csum2hastag} &
\end{array}
$$

Figure 2.2: LTAL Syntax-types and coercions.

2.2.1 Kinds

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.

- Ω_N is the kind of singleton integer types.
- Ω is for all other types. Types of any value or expression in a program should be of kind Ω.

LTAL does not have a sub-kinding relation. Ω_N is not considered a sub-kind of Ω, that is, type τ of kind Ω_N does not mean that τ is of kind Ω. LTAL has type constructors that coerce types of kind Ω_N to types of kind Ω.

2.2.2 Types

LTAL has a set of expressive type constructors:

- Type variables $\alpha, \beta, \ldots$[3] are treated abstractly in LTAL. LTAL does not support intensional analysis of type variables as Harper and Morrisett did [32]. Two type variables are equal if and only if they are the same.

- Currently, **int** is for 32-bit integers.

- LTAL uses existential types $\exists\alpha : \kappa.\tau$ to represent function closures, tagged sum values, position-independent code, and so on.

- Recursive types $\mu\alpha : \kappa.\tau$ are used for user-defined recursive datatypes.

- Type **boxed** describes pointers pointing to heap values. The boxed type does not specify the content of the heap value the pointer points to.

- Singleton integer literal types $\overline{\mathbf{n}}$ are introduced for data flow analysis. They help in reasoning about integer arithmetic at the type level.

- Types $\tau_{\mathbf{1}} + \tau_{\mathbf{2}}$ are used for singleton integer arithmetic. Both τ_1 and τ_2 should be of kind Ω_N.

- Type **freem** describes the currently available free space on the heap.

- **field** type describes a field of a record. Value v has type field $i\ \tau$ means the word located at memory address $v + i$ has type τ.

[3] The implementation uses de Bruijn indices, but in this presentation I will show named variables.

- Intersection types are like "AND". Value v has type $\tau_1 \cap \tau_2$ means v has type τ_1 and v has type τ_2.

  LTAL uses intersection types and field types to describe records. A record type $[\tau_0, \tau_1, \ldots, \tau_{n-1}]$ is represented in LTAL as (field 0 τ_0) $\cap$ (field 4 τ_1) $\cap \ldots \cap$ (field $(4n-4)$ τ_{n-1})[4].

- Union types are like "OR". Value v has type $\tau_1 \cup \tau_2$ means v has type τ_1 or v has type τ_2. It is natural to represent user-defined datatypes with union types (see Chapter 5 for details).

- Integer predicates $\mathbf{int}_\pi\, \tau$ refine the integer type. The subscript π is an integer comparison operator such as $=$ or $\leq$. Type τ should be of kind Ω_N.

  Value i has type $\mathrm{int}_\pi\, \overline{n}$ means $i\, \pi\, n$ is true. For example, 3 has type $\mathrm{int}_=\, \overline{3}$, and it also has type $\mathrm{int}_<\, \overline{6}$.

- Type $\mathbf{range}[n_1, n_2]$ is an abbreviation for $(\mathrm{int}_\geq\, \bar{n}_1) \cap (\mathrm{int}_<\, \bar{n}_2)$. Integer i has type $\mathrm{range}[\bar{n}_1, \bar{n}_2]$ when $n_1 \leq i < n_2$.

- Polymorphic "code pointer" type $\mathbf{codeptr}[\vec{\alpha} : \vec{\kappa}](m, cc, v_1 : \tau_1, \ldots, v_n : \tau_n)$ models basic blocks (with their live variables) and functions (with their formal parameters), where $\vec{\alpha} : \vec{\kappa}$ is a list of type variables, m is the available memory size guaranteed at this point, cc is the condition code requirement, and $v_i : \tau_i$ are the formal parameters.

- Type $\mathbf{addr}\ l$ describes the beginning address of basic block l.

[4]In this thesis I assume that the word size is 4.

- Type **diff**(l_1, l_2) means the difference between beginning addresses of basic blocks l_1 and l_2. LTAL cannot use only integers for addresses because of position-independent code.

  Types addr and diff are introduced for label arithmetic, which will be explained in Chapter 6.

- Type **def** refers to a type expression by a name $\Bbbk$; in the implementation, names 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.

- Type **array**(τ_1, τ_2) describes a τ_2 array of size τ_1. The size specifies how much space the array occupies. With size in the array type, it is possible to break apart the "allocArray" macro instruction and to reason about array bounds checking.

  Reference type ref τ can be expressed with array type array$(\overline{4}, \tau)$, since the space one word occupies is 4 bytes.

- Value v has type **offset** $i\ \tau$ means $v + i$ has type τ. Offset types are used for address arithmetic.

2.2.3 Condition Codes

LTAL has a special category *cc* to capture the condition code status (on machines with condition codes):

- **cc_cmp**(τ_1, τ_2) represents the condition codes set by comparing two integers of types $\mathrm{int}_= \tau_1$ and $\mathrm{int}_= \tau_2$ respectively. Both τ_1 and τ_2 should be of kind Ω_N.

- **cc_testbox** τ represents the condition codes set by comparing a sum value (whose type is τ) with a magic number that the compiler uses to differentiate pointers from small integers. The FPCC-ML compiler has the convention that all pointers are greater than or equal to 256. It represents constant data constructors as integers less than 256, and value-carrying constructors as pointers. Thus, comparing a sum value with 256 tells us whether the value is a constant constructor or a value-carrying one.

- **cc_none** is for arbitrary condition code status. It is used to describe condition codes that are unknown or that we don't care about.

The type-check can refine types by keeping track of the condition code status. Normally, when a comparison is followed by a conditional branch, we can get more information about the two operands of the comparison according to whether it branches or not. The type-checker could use the information to refine the types of the operands. For example, if we compare variable v of type int with integer 5 and then do a branch-if-equal. If the control transfers to another block, we know $v = 5$. Otherwise, $v \neq 4$. Either way, v's type can be refined.

The type refinement happens only at the conditional branch, and it needs the information about the two operands passed from the comparison. This information is carried by the condition code status in LTAL. The comparison instruction sets the condition code status and the conditional branch consumes it and refines types.

2.2.4 Values

A value can be a variable x, an integer i, a label l, a coerced value $c(v)$, or a **vdiff** value. Vdiff values represent the difference between two labels, and are used only for typed position-independent code.

Variables 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.

2.2.5 Coercions

A coercion changes only the static type of a value; it has no runtime effect, as it follows subtyping relations in the underlying model. A coercion c defines a type transformation function f_c. Applying c to value v of type τ results in another value $c(v)$ of type $f_c(\tau)$. Type τ and $f_c(\tau)$ should be compatible; more accurately, it should be provable in the underlying model 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.

LTAL has the following coercions:

- **cid** is the identity coercion. It transforms type τ to τ.

- Coercion $c_2 \circ c_1$ composes two coercions. It transforms type τ_0 to τ_2, if c_1 transforms τ_0 to τ_1 and c_2 transforms τ_1 to τ_2. Like a standard function composition representation, the right coercion is applied first.

- **cpack**(τ_1, τ_2) creates existential types. Type τ_2 should be an existential type, of format $\exists\alpha : \kappa.\tau_f$. This coercion transforms type τ to τ_2, if τ_1 has kind κ, and $\tau = \tau_f[\tau_1/\alpha]$. $\tau_f[\tau_1/\alpha]$ means replacing every appearance of α in τ_f with τ_1.

- **cfold**$[\tau]$ creates recursive types. Type τ should be a recursive type, of format $\mu\alpha : \kappa.\tau'$. This coercion transforms type τ_0 to τ, if $\tau_0 = \tau'[\tau/\alpha]$, the unfolded type of τ.

- **cunfold** unfolds a recursive type. It transforms type $\mu\alpha : \kappa.\tau$ to its unfolded type $\tau[(\mu\alpha : \kappa.\tau)/\alpha]$.

- **crange**$[n_1, n_2]$ transforms type $\overline{n}$ to range$[\bar{n}_1, \bar{n}_2]$, if $n_1 \leq n < n_2$.

- **cinj1**$[\tau_1 \cup \tau_2]$ transforms type τ_1 to union type $\tau_1 \cup \tau_2$.

- **cinj2**$[\tau_1 \cup \tau_2]$ transforms type τ_2 to union type $\tau_1 \cup \tau_2$.

- **cproj1** projects the first conjunct of an intersection type. It transforms $\tau_1 \cap \tau_2$ to τ_1.

- **cproj2** projects the second conjunct of an intersection type. It transforms $\tau_1 \cap \tau_2$ to τ_2.

- **c2inters**(c_1, c_2) creates an intersection type. It transforms type τ to $\tau_1 \cap \tau_2$, if c_1 transforms τ to τ_1 and c_2 transforms τ to τ_2.

- **cname** expands a type abbreviation name to the type expression the name stands for. It transforms type def $\Bbbk$ to τ, if $\Bbbk$ is declared to be an abbreviation of τ.

- **cdef** $\mathbb{k}$ is the opposite to cname. It transforms τ to def $\mathbb{k}$, if $\mathbb{k}$ is declared to be an abbreviation of τ.

- **c2int32** transforms a refined integer type (or a range type) to type int.

- **cptapp**$[\tau]$ partially instantiates a polymorphic function. When applied to a codeptr type, it replaces every appearance of the first type variable in the codeptr type with τ.

- **coffset0** transforms type (offset 0 τ) to τ.

- **c2offset0** transforms type τ to (offset 0 τ).

- **caddr2code** transforms type $\text{int}_=$ (addr l), which is the type of the beginning address of function l, to $\text{codeptr}[\vec{\alpha} : \vec{\kappa}](m, cc, v_1 : \tau_1, \ldots, v_n : \tau_n)$, if there is a function declaration:

  $$l[\vec{\alpha} : \vec{\kappa}](m, cc, v_1 : \tau_1, \ldots, v_n : \tau_n) = \iota_1; \ldots; \iota_k$$

  This coercion indicates the equivalence of the base address of function l and the code pointer l.

- **csum2hastag** transforms type τ to $\exists\alpha : \Omega_N.(\text{field } 0\ (\text{int}_=\ \alpha)) \cap \tau$. It should be applied to a union type, each disjunct of which has format (field 0 $\text{int}_=\ \bar{i}) \cap \ldots$, meaning it is tagged i.

I list some coercion rules in Figure 2.3. Other rules will be shown in later chapters when they are used. The coercion typing judgment $\rho; LRT \vdash_c \tau \overset{C}{\hookrightarrow} \tau'$ means that under kind environment ρ and maps LRT (explained in Section 2.2.8), coercion C changes τ to τ'.

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

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

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

Figure 2.3: Selected coercion rules

The cproj1 rule is justified by a semantic model containing the subtyping rule $\tau_1 \cap \tau_2 \subset \tau_1$. And the cproj2 rule is by $\tau_1 \cap \tau_2 \subset \tau_2$. The composition rule is the transitivity of subtyping.

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.

LTAL solves 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 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 $v : \tau$, the type-checker examines each binding $v' : \tau'$ in the current value binding environment Φ and removes it from Φ if v' is assigned the same register as v, which means v' should no longer be live. Notation $(\Phi \backslash v), v : \tau$ is used to represent this operation; it can be seen in premise (9) of the big rule in Section 2.3. When there is no ambiguity, $(\Phi \backslash v), v : \tau$ is abbreviated to

$\Phi, v : \tau$. Note that $\backslash$ is not an LTAL operator. It is a meta notation used by the type-checker.

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

2.2.6 Instructions

LTAL has a machine-independent core and a machine-dependent extension. Each target machine requires the addition of machine-specific operators and rules. LTAL has the following instructions:

- **Open** is essentially a coercion that transforms an existential type to a non-existential one. Instruction $(\alpha, v') = \text{open}(v)$ binds a new type variable α (the hidden part in the existential type) and a new variable v' (coerced from v). The open instruction "expands" to zero Sparc instructions. It is designed as an instruction (instead of an ordinary coercion) because it binds a new type variable.

- The move instruction $v' = v$ expands to a Sparc move instruction, if the source v and the target v' are not assigned the same register.

- The move-with-coercion instruction $v' = v$ corresponds to zero Sparc instructions, if v and v' are assigned the same register.

- ALU instructions $v = v_1 \;\; op \;\; v_2$ expand to Sparc ALU instructions. Operator $+_i$ is specialized for singleton integer addition. Instruction $v = v_1 +_i v_2$ gives v type $\text{int}_= (n_1 + n_2)$ if v_1 has type $\text{int}_= n_1$ and v_2 has type $\text{int}_= n_2$. Other operators are for normal integer arithmetic.

$op ::= + \mid +_i \mid - \mid * \mid /$		**Arith. Ops**
		Instructions
ι	$::= (\alpha, v') = \text{open}(v)$	*no instruction*
	$\mid\ v' = v$	*move* $\text{R}(v) \neq \text{R}(v')$
	$\mid\ v' = v$	*no instruction* $\text{R}(v) = \text{R}(v')$
	$\mid\ v = v_1\ \ op\ \ v_2$	*ALU instructions*
	$\mid\ v = \text{sethi}(n)$	*sethi*
	$\mid\ \text{store}(v_i, v)$	*store*
	$\mid\ v = \text{record}$	*move*
	$\mid\ \text{inc\_alloc}(v)$	*add*
	$\mid\ v = \text{load}(v_1, v_2)$	*load*
	$\mid\ v = \text{addradd}(v_1, v_2)$	*add*
	$\mid\ \text{arrStart}[\tau]$	*no instruction*
	$\mid\ \text{storeA}(v_i, v)$	*store*
	$\mid\ v = \text{sub}(v_a, v_i)$	*load*
	$\mid\ \text{update}(v_a, v_i, v)$	*store*
	$\mid\ \text{call}(v, [\tau_1, \ldots, \tau_n])$	*jump*
	$\mid\ \text{calln}(l, [\tau_1, \ldots, \tau_n])$	*fall through*
$\star$	$\mid\ \text{cmp}(v_1, v_2)$	*subcc*
$\star$	$\mid\ \text{cmpr}(v_1, v_2)$	*subcc*
$\star$	$\mid\ (\alpha, v') = \text{testbox}(v)$	*subcc*
$\star$	$\mid\ \text{testAvail}$	*subcc*
$\star$	$\mid\ \text{testFull}(v)$	*subcc*
$\star$	$\mid\ \text{if}(\pi)\text{ then } l_1 \text{ else } l_2$	*branch*
$\star$	$\mid\ \text{ifinitA then } (l_1) \text{ else } (l_2)$	”
$\star$	$\mid\ \text{iffull then } l_1 \text{ else } l_2$	”
$\star$	$\mid\ \text{ifboxed}\{v\} \text{ then } (v_1, l_1) \text{ else } (v_2, l_2)$	”
$\star$	$\mid\ \text{iftag}(\pi)\{v\} \text{ then } (v_1, l_1) \text{ else } (v_2, l_2)$	”

Figure 2.4: LTAL Syntax-instructions. Marked $\star$ operators are specific to setting and branching on condition codes.

- Instruction $v = \textbf{sethi}(n)$ loads 32-bit integer n to v.
- **store** initializes a field of the record currently being initialized. Instruction store(v_i, v) writes v to the field at offset v_i from the beginning of the record.
- **record** assigns an allocated and initialized structure (record or array) to a variable. Instruction $v =$ record makes v point to the newly allocated structure on the heap.
- **inc_alloc** updates heap status at the end of allocation. Instruction inc_alloc(v) increases by v the *allocptr*, which defines the boundary between the allocated and the unallocated area.
- **load** loads a field of a record to a variable. Instruction $v = \text{load}(v_1, v_2)$ loads the memory word at address $v_1 + v_2$ to v. Variable v_1 points to the beginning of the record, and v_2 is the offset of the field to be loaded.
- **addradd** is used for address arithmetic. Instruction $v = \text{addradd}(v_1, v_2)$ assigns $v_1 + v_2$ to v, where v_1 is a label value and v_2 is an integer.
- **arrStart** $[\tau]$ sets up the allocation environment before allocating a τ array.
- **storeA** initializes an element of the array currently being initialized. Instruction storeA(v_i, v) stores value v to the array element at offset v_i.
- **sub** is the array subscript operator. Instruction $v = \text{sub}(v_a, v_i)$ loads the memory word at address $v_a + v_i$ to v. Variable v_a points to the beginning of the array, and v_i is the offset of the element to be loaded.
- **update** (v_a, v_i, v) updates an array element at address $v_a + v_i$ with v.

- **call**$(v, [\tau_1, \ldots, \tau_n])$ jumps to function v, with v's type variables instantiated by $\tau_1, \ldots, \tau_n$. Passing of value parameters (as distinguished from type parameters) is explained in Section 2.2.7.

- **calln**$(l, [\tau_1, \ldots, \tau_n])$ is used for "call by fall-through," which generates no code. This instruction should be followed by the declaration of block l.

- **cmp**(v_1, v_2) compares two integers v_1 and v_2, and sets condition code cc_none. The types of v_1 and v_2 are int, and there is no need for type refinement. This comparison is useful when the user cares about the control flow (which branch to take), but the checker does not need to track it for safety.

- **cmpr**(v_1, v_2) compares two singleton integers v_1 and v_2, and sets condition code cc_cmp(n_1, n_2) if v_1 has type $\text{int}_= n_1$ and v_2 has type $\text{int}_= n_2$. Later v_1's type could be refined according to the result of the comparison.

- Instruction $(\alpha, v') =$ **testbox**(v) compares a sum value v with a number set by the compiler that differentiates small integers and pointers (256 in the implementation), rebinds v to v', and sets condition codes. Variable v' is assigned the same register as v. The only difference between v' and v is the types. The testbox instruction is explained in Section 5.3.3.

- **testAvail** calculates the available heap space and assigns it to a reserved register r_{al}.

- **testFull** tests to see if there is sufficient heap space for allocation. Instruction testFull(v) compares the available heap space (register r_{al}) with the required space v for allocation, and sets condition codes.

- Instruction **if**(π) then l_1 else l_2 is the normal conditional branch without type refinement. It branches to l_1 if the condition codes satisfy π. The other target l_2 is the fall-through case. For example, if(>) will be mapped to the Sparc branch if greater than instruction.

- **ifinitA** then l_1 else l_2 is translated to the Sparc bge (branch if greater or equal) instruction. It is used to branch when an array is completely initialized. See Section 4.4.4 for details.

- **iffull** tests whether there is enough space for allocation and sets up the allocation environment if there is enough space for allocation on the heap. Instruction iffull then l_1 else l_2 falls through to l_2 if there is enough space.

- **ifboxed** refines a sum value's type according to whether it is a constant constructor (represented as a small integer) or a value-carrying constructor (represented as a pointer). Instruction ifboxed$\{v\}$ then (v_1, l_1) else (v_2, l_2) checks the condition codes set by comparing sum value v with 256, rebinds v to v_1 in branch l_1 when $v \geq 256$ (v is a pointer). Otherwise it rebinds v to v_2 in the fall-through case l_2 (v is a small integer). Both v_1 and v_2 have types refined from v's type.

- **iftag** specializes type refinement for datatype tag discrimination. Instruction iftag$(\pi)\{v\}$ then (v_1, l_1) else (v_2, l_2) checks the condition codes set by tag-checking, and rebinds v to new variables v_1 and v_2 with refined types in l_1 and l_2 respectively.

Each LTAL instruction maps to at most one Sparc instruction. Several LTAL instructions with different typing rules may map to the same Sparc instruction.

$$
\begin{array}{llr}
 & & \textbf{Basic block} \\
B & ::= l[\alpha_1 : \kappa_1, \ldots, \alpha_j : \kappa_j](m, cc, v_1 : \tau_1, \ldots, v_n : \tau_n) = \iota_1; \ldots; \iota_k & \\
LRT & ::= (L, R, T) & \textbf{Environments} \\
L & ::= \{l_1 \mapsto a_1, \ldots, l_n \mapsto a_n\} & \textbf{label map} \\
R & ::= \{x_1 \mapsto r_1, \ldots, x_n \mapsto r_n\} & \textbf{register map} \\
T & ::= \{\mathbb{k}_1 \mapsto \tau_1, \ldots, \mathbb{k}_n \mapsto \tau_n\} & \textbf{type abbrev. map} \\
P & ::= (LRT, \vec{B}) & \textbf{Program}
\end{array}
$$

Figure 2.5: LTAL Syntax-programs.

2.2.7 Functions

The function declaration

$$l[\alpha_1 : \kappa_1, \ldots, \alpha_j : \kappa_j](m, cc, v_1 : \tau_1, \ldots, v_n : \tau_n) = \iota_1; \ldots; \iota_k$$

defines a function (basic block) with label l, type parameters $\alpha_1, \ldots, \alpha_j$ which are of kind $\kappa_1, \ldots, \kappa_j$ respectively, 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. If a function specifies m words and allocates no more than m words, there is no need to test the memory availability. Otherwise, it has to check explicitly whether 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 codeptr$[\alpha_1 : \kappa_1, \ldots, \alpha_j : \kappa_j](m, cc, v_1 : \tau_1, \ldots, v_n : \tau_n)$. Each function is closed in the sense that there are no free type variables or value variables. LTAL uses continuation-passing style, thus functions never return.

When the above function l is called, the caller has to satisfy all l's requirements. It has to instantiate all l's type variables with types of the right kind, make sure there is at least m free space on the heap, set up proper condition codes, and pass the actual paramters to formals.

Suppose the formal parameters $v_1, v_2, \ldots, v_n$ of l are assigned registers $r_1, r_2, \ldots, r_n$ respectively, and the actual paramters $v'_1, v'_2, \ldots, v'_n$ are in register $r'_1, r'_2, \ldots, r'_n$ correspondingly. When l is called, before the control transfers to l, we must first do register marshalling to move $r'_1, r'_2, \ldots, r'_n$ to $r_1, r_2, \ldots, r_n$ respectively. The marshalling is completed by a sequence of move instructions (and store and load instruction when spilling is necessary). For example, the following Sparc instructions arrange registers before calling l, suppose there is no spilling and no register dependency between the formals and the actuals.

$\ldots$
mv r'_1, r_1
mv r'_2, r_2
$\ldots$
mv r'_n, r_n
ba l

To make the correspondence between LTAL and Sparc, we cannot let the call instruction handle marshalling, nor use a single instruction to move all registers. Therefore, the calling convention of LTAL requires that the register marshalling be done by a sequence of explicit LTAL move instructions before a function is called. These LTAL move instructions correspond to the Sparc instructions that move actuals to formals. But LTAL uses variables, not registers. What should be the destination of the LTAL move instructions, and how does the type-checker know the actuals are moved to the right registers? The compiler moves the actuals to local variables that have the same names as the callee's formal parameters, and the register allocator assigns the same register to variables with the same name, regardless of their scopes. As a result, the LTAL move instructions arrange that the actuals are moved to the right registers.

For example, the following LTAL instructions complete a call to l with actual parameters $v'_1, v'_2, \ldots, v'_n$, suppose there is no spilling and no dependency between the registers used by the formals and the actuals. Note that $v_1, v_2, \ldots, v_n$ are local variables in the caller. Their scopes are disjoint with l's formals $v_1, v_2, \ldots, v_n$. They only have same names, thus are assigned the same registers by the register allocator.

LTAL	Sparc
$\ldots$	$\ldots$
$v_1 = v'_1$	**mv** r'_1, r_1
$v_2 = v'_2$	**mv** r'_2, r_2
$\ldots$	$\ldots$
$v_n = v'_n$	**mv** r'_n, r_n
call(l, [])	**ba** l

2.2.8 Environments

Triple LRT represents three environments that keep auxiliary information for type-checking: label environment L maps labels to addresses (offset from the beginning of the program); register environment R maps variables to temporaries (registers or spill locations); type abbreviation environment T maps type abbreviations to their expansions.

2.2.9 Programs

An LTAL program consists of the above environments and a set of function declarations. The entry point of a program is the first declared function.

2.3 Type-Checking

The low-level type and term constructors in LTAL make the type system expressive. Yet LTAL needs 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, LTAL has been made completely syntax-directed. There are no subtyping rules; instead, LTAL uses coercions to avoid nondeterministic choices during type-checking. I explain various typing judgments, and then show a sample typing rule in this section. More rules are shown later.

The typing judgment for values $LRT; \rho; \Phi \vdash v : \tau$ means value v has type τ under environment $LRT; \rho; \Phi$. Triple LRT is part of the program (see Figure 2.5). Kind environment ρ maps type variables bound so far to their kinds (the implementation uses de Bruijn numbers, so ρ is just a list of kinds). Value environment Φ maps variables to their types.

The judgment $LRT \vdash (\rho; \hbar; \Phi; cc)\ \{\iota\}\ (\rho'; \hbar'; \Phi'; cc')$ means after instruction ι is executed, environment $(\rho; \hbar; \Phi; cc)$ becomes $(\rho'; \hbar'; \Phi'; cc')$. The construction $\Phi, v : \tau$ augments Φ with a new binding $v : \tau$ and keeps the bindings other than v unchanged. The heap-allocation environment $\hbar$ is explained in Chapter 4. Environment cc specifies the current status of condition codes.

As an example I will show a simplified rule for an LTAL *add* instruction. In Chapter 6 I will show another typed version of *add*. These two different typed versions of *add* expand to the same Sparc machine instruction. The first rule I 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 type-checking.

$$\frac{LRT; \rho; \Phi \vdash x : \mathrm{int} \quad LRT; \rho; \Phi \vdash y : \mathrm{int}}{LRT \vdash (\rho; \hbar; \Phi; cc)\ \{z = x + y\}\ (\rho; \hbar; \Phi, z : \mathrm{int}; cc)}$$

In fact, this rule is dramatically simplified for clarity. The full version looks like this:

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

Premises (1) and (2) state that both x and y have type int_{32}, the 32-bit integer type. 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 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 20 are 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.

Value y can be either a register or an immediate. The checker uses a predicate *match_reg_or_imm* in premise (8) to match either case. So y_m can be either (rmode r_y) for some register r_y or (imode i) for some integer constant i.

Premise (9) states the relation between the value typing context before and after execution of the current instruction. Before the type of variable z is added into the context, all aliases of z should be killed since they are not live anymore, which is what $\Phi\backslash z$ does.

Premise (10) will be explained in the next subsection.

The conclusion is like a Hoare-logic judgment. In 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 $\hbar$; value context Φ becomes Φ' after execution; the machine code at location ℓ' is P'.

2.3.1 Instruction decoding

The *decode_list* relation in premise (10) maps an instruction word to a higher-level instruction with semantic meaning. Specifically, it says that the instruction word at the beginning of P with length $\ell' - \ell$ is an add instruction i_ADD(r_x, y_m, r_z). Instruction encoding is checked with rules such as the following:

1 0	Z	0 0 0 0 0 0	X	0	0 0 0 0 0 0 0 0	Y

32 30 25 19 14 13 5 0

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

This rule is not an axiom of the system, it is a lemma derived from a more concise and readable definition of instruction encodings [40]. 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.

2.4 An Example

Figure 2.6 shows a slightly simplified LTAL function compiled from ML source:

$$\text{fun } f\ x\ =\ x\ +\ 1$$

The right column is the corresponding Sparc instructions.

The function is labeled l. The head declaration means:

- $[\,]$: it does not have any type variables (not polymorphic).
- $\overline{0}$: it does not allocate on the heap.
- cc_none: it does not care about condition codes when called.
- parameters: it has three formal parameters:

LTAL	Sparc
$l : [\,], \overline{0}, \text{cc\_none},$	$l :$
$\begin{bmatrix} v1 : \text{int}_{=}\,(\text{addr } l), \\ v2 : \exists\alpha : \Omega_N.((\text{int}_{=}\,\alpha) \cap (\text{codeptr}[](\overline{0}, \text{cc\_none}, [v4 : \text{int}_{=}\,\alpha, \\ \hfill v5 : \text{int}]))), \\ v3 : \text{int} \end{bmatrix}$	
$(1) v6 = v3 + 1$	**add** $r3, 1, r0$
$(2) (\beta, v7) = \text{open } v2$	
$(3) v8 = \text{cproj1}(v7)$	
$(4) v9 = \text{cproj2}(v7)$	
$(5) v4 = v8$	
$(6) v5 = v6$	
$(7) \text{call}(v9, [\,])$	**jmp** $r2$

Figure 2.6: An Example

- $v1$: the start address, which points to the beginning of l (the first instruction of l).
- $v2$: continuation, which consumes the result $x + 1$ (parameter $v5$)[5].
- $v3$: the integer x to be incremented.

To make position-independent code, each function takes one parameter—the start address of itself, which points to its first instruction. When a value v is a function pointer, v is its start address. The address v should be passed as one of the arguments of the function v, when v is called. The type of the continuation $v2$ guarantees that exactly $v2$ is passed when $v2$ is called. This is further explained in Chapter 6.

[5]The compiler uses continuation-passing style [7], which abstracts the rest of computation within a continuation function.

In the body of the function,

- Instruction (1) increments $v3$ and assigns the result to $v6$.
- Instruction (2) opens the continuation package.
- Instructions (3) and (4) coerce the continuation to get two views: the start address and the function pointer.
- Instruction (5) and (6) assign actual parameters to formal parameters of function pointer $v9$.
- Instruction (7) calls the continuation function pointer $v9$.

Only (1) and (7) have corresponding Sparc instructions. Variables $v2, v4, v7, v8, v9$ are assigned the same register $r2$, thus, no Sparc instructions are needed for (2)-(5). For the same reason, instruction (6) has no corresponding Sparc instructions.

Type-checking this LTAL function goes as follows: first, the type-checker initializes the value environment with the bindings of the formals v_1, v_2 and v_3. Instruction (1) assigns $v6$ type int. Instruction (2) opens $v2$, binds new type variable β of kind Ω_N, and creates a new variable $v7$ of type $(\mathrm{int}_= \beta) \cap (\mathrm{codeptr}[\ ](\overline{0}, \mathrm{cc\_none}, [v4 : \mathrm{int}_= \beta, v5 : \mathrm{int}]))$. Instruction (3) gives $v8$ type $\mathrm{int}_= \beta$, the first conjunct of $v7$'s type, and instruction (4) gives $v9$ $\mathrm{codeptr}[\ ](\overline{0}, \mathrm{cc\_none}, [v4 : \mathrm{int}_= \beta, v5 : \mathrm{int}])$, the second conjunct. Value $v4$ gets $v8$'s type after instruction (5), and $v5$ gets $v6$'s after instruction (6). When $v9$ is called in instruction (7), the checker checks that: a) $v9$ is of codeptr type $\mathrm{codeptr}[\ ](\overline{0}, \mathrm{cc\_none}, [v4 : \mathrm{int}_= \beta, v5 : \mathrm{int}])$; b) the local variable $v4$ is of type $\mathrm{int}_= \beta$; and c) the local variable $v5$ is of type int.

2.5 Implementation

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, including 196 machine-language Sparc instruction constructors (many of which are not used by the compiler and could be deleted from the checker), 263 Sparc instruction decoding rules, 30 coercion operators and 49 coercion rules, 48 explicit-substitution operators and reduction rules, 41 types and constructors for such things as label-maps and register-maps, 27 type operators (union, intersection, field, etc.), 69 rules for type refinement, 98 rules for wellformedness of types, 73 operators and rules for local environment management, 44 operators and rules for static arithmetic calculations, 38 rules for parsing the label, register, and type maps, 50 structural matching heuristics for type expressions, 51 LTAL instruction constructors, and 53 typing rules for instructions.

A typical large rule, such as the one shown in Section 2.3, is quantified over a dozen variables and has a dozen premises. In all, the current LTAL type-checker is 3900 lines of (non-blank, non-comment) Prolog-like source code. It is almost certain that such a large calcus has bugs, and that the non-machine checkable soundness proof of the type system has bugs. Therefore, it is very important to have a machine checkable soundness proof. The machine-checked proof of the soundness of all the LTAL rules (which is nearing completion) is over 133,000 lines of higher-order logic as represented in the Twelf system. The axioms comprise 1850 lines, almost all of which is the specification of the Sparc instruction set.

Chapter 3

Soundness of the LTAL Type System

In this chapter, I briefly explain the semantic model for LTAL types and typing rules and how to prove program safety based on the semantic model. This is not my work, but the work of other people in the FPCC project. The purpose of this chapter is only to give a flavor of our semantic approach. Many details are omitted. Readers could refer to a paper [60] and a thesis [58] for more explanation.

3.1 Soundness of Type Systems

A sound type system has the desirable guarantee that well-typed programs do not go wrong, that is, if a program type-checks with respect to the typing rules of the type system, it will not have undesired behaviors when executed. Which behaviors are undesired is defined by the safety policy, in terms of the dynamic semantics of the language.

Traditionally, the soundness of a type system is proved by a syntactic approach [66]. In order to prove that a type system is sound, we need to prove two properties—progress and preservation. Progress means if a program is well-typed, it can continue to execute one more step. Preservation means if a program P is well-typed, and it executes one more step to another program P', then P' is well-typed. By progress and preservation, we can conclude that a program can safely continue executing (until it stops at a desired state, for example, it evaluates to a value). An ill-typed program will get stuck at some place where it can not take one step further.

Most TALs have syntactic soundness proofs that are not machine-checkable. We need to trust that these proofs contain no bugs. For a full-scale typed assembly language as LTAL, we need stronger guarantee.

While it might be possible to produce machine-checkable soundness proof uses syntactic soundness proofs [30, 21], our machine-checkable soundness proof uses a semantic approach. We give a semantic model to its types and typing rules. Types are modeled as predicates and typing rules are modeled as lemmas in the underlying higher-order logic. Also the soundness guarantee—well-typed programs do not go wrong—is proved as a theorem in the logic. All the proofs are with respect to a minimal set of axioms of the higher-order logic, arithmetic and machine instruction semantics, and all can be machine-checked. The LTAL type system is totally out of the TCB.

Before I explain the semantic model of types and typing rules, I will first describe the semantics of machine instructions, since ultimately LTAL instructions will be mapped to machine instructions, and proving LTAL typing rules is based on the instruction semantics.

3.2 Modeling Machine Instructions

Sparc machine instructions are modeled as step relations between machine states [40]. A machine state (r, m) consists of the register bank r and the memory m. Both r and m are functions from numbers to numbers: r maps register indices to register contents, and m maps addresses to memory contents. An instruction is a relation between machine states before and after executing the instruction.

For example, the Sparc add instruction is modeled as:

$$add\ r_j, r_k, r_i = \lambda r.\lambda m.\lambda r'.\lambda m'.m = m' \wedge r'(i) = r(j) + r(k) \wedge \forall x \neq i\ r'(x) = r(x)$$

The machine state before execution is (r, m), and the one after execution is (r', m'). The memory is not changed since the add instruction does not touch the memory $(m = m')$. After instruction $add\ r_i = r_j + r_k$, the value of register i becomes the sum of register j and k $(r'(i) = r(j) + r(k))$. All other registers are the same $(\forall x \neq i.\ r'(x) = r(x))$.

The Sparc load instruction can be modeled as:

$$\begin{aligned} ld\ [r_j, k], r_i &= \lambda r.\lambda m.\lambda r'.\lambda m.m = m' \wedge readable(r(j) + k) \\ &\quad \wedge r(i) = m(r(j) + k) \wedge \forall x \neq i\ r(x) = r(x) \end{aligned}$$

Load instruction does not change the memory either. It requires that the address $r(j) + k$ be readable $(readable(r(j) + k))$. After load instruction, register i has the same contents as the memory word at address $r(j)+k$ $(r(i) = m(r(j)+k))$. All other registers are unchanged. Notice that the relation specified by load instruction is partial: if the state before execution does not satisfy the condition $readable(r(j)+k)$, there is no next state. This property prevents a program from reading memory it does not have access to.

The step relation between machine states $\mapsto$ contains two parts : syntax and semantics of machine instructions. The syntax part is instruction decodings. The semantics part is the transition of machine states, as in examples add and load. $(r, m) \mapsto (r', m')$ is true if the memory word the program counter points to decodes to some instruction ι, and (r, m) steps to (r', m') according to the semantics of ι. Zero or more steps are represented by $\mapsto^*$. An informal definition of $\mapsto$ is as follows (each disjunct represents a machine instruction):

$$
\begin{array}{rcl}
(r, m) \mapsto (r', m') & = & (\exists w, i, j, k. \\
& & m(r(PC)) = w \\
& & \wedge r'(PC) = r(PC) + 4 \\
& & \wedge w \text{ decodes to } \mathit{add}\ r_i, r_j, r_k \\
& & \wedge (\mathit{add}\ r_i, r_j, r_k)(r, m, r', m')) \\
& & \vee(\text{load } \ldots) \vee (\text{store } \ldots) \vee \ldots
\end{array}
$$

A machine state (r, m) is *stuck* if there exists no state (r', m') such that $(r, m) \mapsto (r', m')$.

3.3 Modeling Types

A type is a predicate on indexed values $\langle k, m, v\rangle$, where m is the memory, v is the root pointer (start address of a value), and k is an index. Predicate $\langle m, v\rangle :_k \tau$ is a syntactic sugar for $\tau\ k\ m\ v$, which means value $\langle m, v\rangle$ k-approximately has type τ, that is, if $\langle m, v\rangle$ is put into a place where a value of type τ is required, we cannot observe any difference within k steps of execution. We use the notation $\langle m, v\rangle : \tau$ if $\langle m, v\rangle :_k \tau$ for any k.

$$
\begin{array}{rcl}
\text{int} & \equiv & \lambda k.\lambda m.\lambda v.\text{true} \\
\text{int}_{=}\, n & \equiv & \lambda k.\lambda m.\lambda v.v = n \\
\text{field } i\ \tau & \equiv & \lambda k.\lambda m.\lambda v.(v+i) \in \text{dom}(m) \wedge \text{readable}(v+i) \wedge \tau\ (k-1)\ m\ (v+i) \\
\text{codeptr}(\phi) & \equiv & \lambda k.\lambda m.\lambda v.\forall j, r.\ j < k\ \wedge\ r(\text{pc}) = v\ \wedge\ (m,r) :_j \phi\ \Rightarrow\ \text{safe}_n(j,r,m)
\end{array}
$$

Figure 3.1: Model for Types

$$
\begin{array}{rcl}
\{n : \tau\} & \equiv & \lambda k.\lambda m.\lambda \vec{v}.(m, \vec{v}(n)) :_k \tau \\
\phi_1 \cap \phi_2 & \equiv & \lambda k.\lambda m.\lambda \vec{v}.(m, \vec{v}) :_k \phi_1 \wedge (m, \vec{v}) :_k \phi_2
\end{array}
$$

Figure 3.2: Model for Environments

The definitions of some types are shown in Figure 3.1.

Since every value in a Sparc register can be treated as an integer, applying int to any value is true. The model of $\text{int}_{=}\, n$ makes sure only constant n has this type. $\langle m, v\rangle :_k \text{field } i\ \tau$ means the address $v+i$ should be readable and $\langle m, m(v+i)\rangle :_{k-1} \tau$. The index is $k-1$ because there should be a load instruction that loads $m(v+i)$. $\langle m, l\rangle :_k \text{codeptr}(\phi)$ means if the control goes to location l, for any index $j < k$ and any register bank r, if the machine state (m, r) j-approximately satisfies the precondition ϕ, it would be safe to run j steps from l (what $\text{safe}_n(j, r, m)$ means). Thus, if $\langle m, l\rangle : \text{codeptr}(\phi)$, it is safe to run any number of steps from l if ϕ is satisfied.

Type environments are modeled in the same way as types, except that the root pointer v in the model of types becomes a vector (a function from indices to contents). $(m, \vec{x}) :_k \phi$ means for each binding $n : \tau$ in ϕ, $(m, \vec{x}(n)) :_k \tau$. Figure 3.2 shows the definitions of some type environments.

Singleton type environment $\{n : \tau\}$ means the nth slot of the vector has type τ. Intersection of type environments $\phi_1 \cap \phi_2$ specifies multiply slots of the vector. $\{n_1 : \tau_1\} \cap \{n_2 : \tau_2\} \cap \ldots \cap \{n_i : \tau_i\}$ is often represented as $\{n_1 : \tau_1, n_2 : \tau_2, \ldots, n_i : \tau_i\}$.

3.4 Program Safety

The guideline of designing the semantic model of LTAL typing rules is to make these rules, and the program safety theorem provable. First, I explain what program safety means in the FPCC system.

The safety theorem says a program is safe if, wherever it is loaded in the memory, when the precondition is satisfied and the program counter points to the beginning of the program, it can safely execute any number of steps without getting stuck[1]. To simplify the notation, in this section, I assume each program starts at memory address 0. Thus, the program safety theorem is

$$
\begin{aligned}
\text{safe\_prog}(C) \quad = \quad & \forall r, m.(\text{loaded}(m, C) \wedge r(PC) = 0 \wedge (m, r) : \phi_0) \\
& \Rightarrow (\forall r', m'.(r, m) \mapsto^* (r', m') \Rightarrow \exists r'', m''.(r', m') \mapsto (r'', m''))
\end{aligned}
$$

FPCC-ML compiles a source program to a machine-instruction program C with type annotation Γ. C is a list of integers (machine instructions), $\{0 : i_1, 4 : i_1, \ldots, 4m : i_m\}$. Γ is the collection of preconditions of labels in the program, $\{l_0 : \text{codeptr}(\phi_0), l_1 : \text{codeptr}(\phi_1), \ldots, l_n : \text{codeptr}(\phi_n)\}$ ($l_0, l_1, \ldots, l_n$ are labels).

From the definition of codeptr type, we know $l : \text{codeptr}(\phi)$ means if the program counter is at location l and the precondition ϕ is satisfied, it is safe to run any number of steps (without getting stuck). Thus, we only need to prove $0 : \text{codeptr}(\phi_0)$ to guarantee the safety of a program C annotated by invariants $\Gamma = \{l_0(= 0) : \text{codeptr}(\phi_0), l_1 : \text{codeptr}(\phi_1), \ldots, l_n : \text{codeptr}(\phi_n)\}$. $0 : \text{codeptr}(\phi_0)$ is part of Γ. So

[1] Here we reason about programs that infinitely loop. "Getting stuck" means that the next instruction would violate the memory safety specification. Every program could be transformed to this style with continuations.

if we could prove the stronger guarantee that program C respects the invariants Γ (formalized as $C \subset \Gamma$), we can guarantee $0 : \text{codeptr}(\phi_0)$ and then the program C is safe. The proof tree is roughly as follows:

$$\cfrac{\cfrac{C \subset \Gamma \qquad \Gamma \subset \{0 : \text{codeptr}(\phi_0)\}}{C \subset \{0 : \text{codeptr}(\phi_0)\}}\,(2)}{\text{safe\_prog}(C)}\,(1)$$

Step (1) comes from the definition of codeptr type. Step (2) uses the transitivity of the subtype relation.

The well-typedness of a program C is given a semantic model $C \subset \Gamma$. The process of type-checking C, in effect, builds a proof tree of $C \subset \Gamma$: every typing construct (typing judgement, type environment) is given a semantic model in the logic, and every application of typing rules could be regarded as applying the corresponding lemmas. The typing derivation becomes a machine-checkable safety proof of the program.

The theorem $C \subset \Gamma$ is proved by induction over the number of safe execution steps from labels l_0, l_1, $\ldots$, l_n in Γ.

The base case is trivial, since it is safe to run 0 steps from any label.

The inductive case is proved by inspecting instructions in each block. Assume that each label is safe for k steps given that its precondition is satisfied. Take a block l_i of n_i instructions with precondition ϕ_i and postcondition (the precondition of its target) ϕ_j. Suppose we can prove that it is safe to execute n_i instructions in l_0 given ϕ_i, and ϕ_j is satisfied at the end of the block, from the induction hypothesis, it is safe to execute k steps from l_j, thus it is safe to execute $n_i + k$ steps from l_i. We assume each block has at least one instruction, therefore, $n_i + k \geq k + 1$, and the inductive case is proved. LTAL can also handle empty basic blocks, or basic blocks containing only coercions that have no runtime effect by using subtyping. I won't discuss it here.

3.4.1 Basic Block Rules

Based on the inductive technique of proving program safety, the well-typedness of a block B has a semantic model that: it is safe to execute $k+1$ steps from labels defined in B given it is safe to execute k steps from each label in the program.

3.4.2 Instruction Rules

Typing rules of instructions are given a semantic model similar to basic blocks. For example, if there is an add instruction at location l, and the precondition and the postcondition are ϕ and ϕ' respectively, then the typing judgement $\Gamma; l \vdash \{\phi\} r_i = r_j + r_k \{\phi'\}$ means that when ϕ is satisfied, it is safe to execute $k+1$ steps from l given that it is safe to run k steps from $l+4$ and other labels in Γ.

The proof of each typing rule consists of two parts: first, prove that it is safe to execute the instruction if the precondition is satisfied; second, after execution of the instruction, the postcondition is satisfied. Both parts are derived from the semantics of the corresponding machine instruction.

3.5 TML Abstraction

The semantic model is rather involved in order to express recursive types and mutable references [11, 6]. We add another abstraction, Typed Machine Language (TML), to hide the details of the semantic model from LTAL and to make proofs modular [58]. TML defines a rich set of syntactic type constructors and subtyping rules. All syntactic rules in TML have machine-checked proofs.

The syntax of TML is shown in Figure 3.3.

		Types	
τ, ϕ, Γ	$::=$	$\top \mid \bot$	top, bottom
	$\mid$	$\mathrm{codeptr}(\phi)$	first-order continuation
	$\mid$	$\mathrm{offset}(n, \tau)$	address arithmetic
	$\mid$	$\mathrm{id}(\tau)$	identity
	$\mid$	$\mathrm{box}(\tau)$	immutable reference
	$\mid$	$\mathrm{ref}(\tau)$	mutable reference
	$\mid$	$\mathrm{rec}\ \tau$	recursive type
	$\mid$	$\tau \cap \tau' \mid \tau \cup \tau'$	intersection, union
	$\mid$	$\forall\ \tau \mid \exists\ \tau$	kinded universal and existential
	$\mid$	$\{n : \tau\}$	singleton type map
	$\mid$	$\phi \backslash n$	restricted type map
	$\mid$	$\mathrm{int}_\pi\, n$	refined integer
	$\mid$	$n_1\ op\ n_2$	singleton integer arithmetic
	$\mid$	$\underline{n}$	type variable index
		Instructions	
ι	$::=$	$\mathrm{instr}(\Gamma, \phi, \phi')$	instruction constructor

Figure 3.3: TML Syntax

TML serves as a semantic basis for LTAL: LTAL types and environments are translated to TML types, LTAL coercions are translated to TML subtyping rules, and LTAL instructions are translated to TML instructions. TML instructions abstract the semantics of machine instructions and explicitly give preconditions and postconditions. For example, the TML load instruction is defined roughly as:

$$\text{tml-load}(i, j, c) = \forall \Gamma, \phi.\mathrm{instr}(\Gamma, \phi \cap j : field(c, \tau), \phi \cap i : \tau)$$

Type instr means if a machine state (r, m) satisfies ϕ_1, and the current instruction has type instr(Γ, ϕ_1, ϕ_2), then the next machine state after executing the instruction satisfies ϕ_2. Γ is used for jump instructions. For example, in the TML load instruction, the precondition specifies that j should have a field type (the definition of field type is in Figure 3.1), and the postcondition specifies that after $m(j + c)$ is loaded to i, i would have type τ. TML instructions capture the preservation property of machine instructions, and are used to prove the soundness of LTAL typing rules.

Chapter 4

Heap Allocation

In this chapter, I explain how to allocate a record or an array on the heap without macro instructions. LTAL provides a set of primitive instructions that model every allocation step. Each of these instructions is simple enough to be mapped to at most one Sparc instruction. Type-checking these instructions requires some bookkeeping, because the type-checker needs to model intermediate states during allocation. The LTAL type-checker uses an allocation environment to keep track of the heap status and the partially initialized structure.

4.1 Heap Model

Like SML/NJ, FPCC-ML compiler allocates closures, records, and arrays in registers or on the heap. At present, the LTAL type system (like most TALs) does not accommodate reasoning about garbage collection. So I will not reason about any information in a record or an array that is used to support garbage collection, for example, the head descriptors specifying which field is a pointer. In the future, we

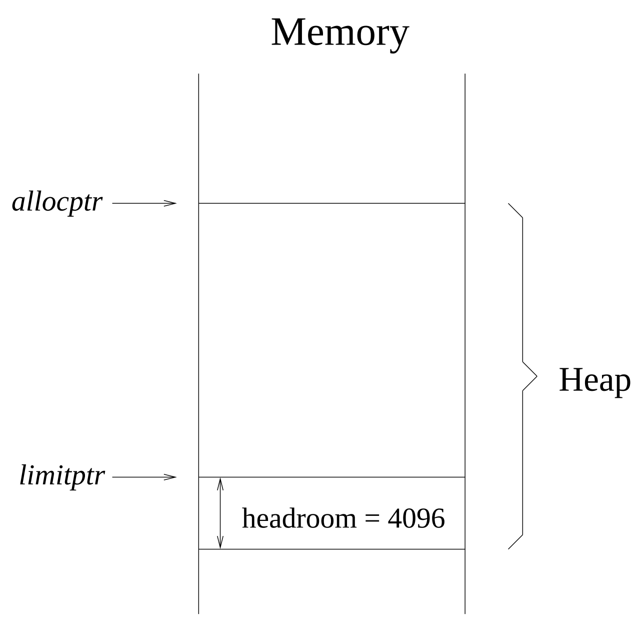

Figure 4.1: Heap Model

intend to handle stacks and GC with a unified theory of stack and heap deallocation (probably based on a region calculus [65]).

As in SML/NJ, with so much heap allocation the compiler needs extremely efficient, in-line allocation of heap values. The allocable heap memory is modeled as a large contiguous region bounded by two pointers, the *allocptr* and the *limitptr*. Heap allocation is broken into two steps: first, test whether there is enough memory for allocation; second, initialize memory in order and bump the *allocptr*.

Before the runtime system starts executing a program, it reserves a chunk of memory as the heap, and sets the *allocptr* to the lowest address of the memory chunk, and the *limitptr* the highest address (minus a constant headroom = 4096) (see Figure 4.1). GC should be invoked when the available space on the heap is less then headroom. The cushion between the *limitptr* and the real heap limit makes allocation of small structures more efficient.

When the program needs n space, and $n \leq$ headroom, testing whether there is enough space can be accomplished by a single instruction that compares the *limitptr* and the *allocptr*. If $limitptr \geq allocptr$, at least n space must be available. Then the n words right after the *allocptr* are initialized word by word. Finally the *allocptr* is increased by n, pointing to the next available location.

When $n >$ headroom or n is unknown at compile time, testing out-of-heap is a bit more complicated. Two instructions are needed: first, calculate the difference $avail = limitptr - allocptr$; then compare whether $avail \geq n$. There is enough memory on the heap if $avail \geq n$.

4.2 Untyped Allocation

Before I describe the LTAL instructions and typing rules for heap allocation, I use three examples to illustrate how allocation is implemented in Sparc assembly instructions.

4.2.1 Record Allocation

Figure 4.2 shows a code segment that allocates a three-field record $[r_0, r_1, r_2]$ and assigns the record to register r.

The heap status during the allocation process is illustrated in Figure 4.3.

Block l_0 tests whether there is enough space on the heap. Since the space needed to allocate the record is 12 (3 words), which is smaller than the headroom 4096, we only need to test whether $limitptr - allocptr < 0$. Instruction (1) compares the *limitptr* with the *allocptr*, and instruction (2) branches to block l_1 if $limitptr - allocptr < 0$ (no enough memory), otherwise it falls through to block l_2.

	l_0 :	
(1)	**subcc** $limitptr, allocptr, r_{al}$	% $r_{al} = limitptr - allocptr$
(2)	**bl** l_1	% if $limitptr - allocptr < 0$, goto l_1
	l_2 :	
(3)	**st** $r_0, [allocptr + 0]$	% $M[allocptr + 0] = r_0$
(4)	**st** $r_1, [allocptr + 4]$	% $M[allocptr + 4] = r_1$
(5)	**st** $r_2, [allocptr + 8]$	% $M[allocptr + 8] = r_2$
(6)	**mov** $allocptr, r$	% $r = allocptr$
(7)	**add** $allocptr, 12, allocptr$	% $allocptr = allocptr + 12$
	...	
	l_1 : ...	% memory not enough for allocation

Figure 4.2: Untyped Record Allocation

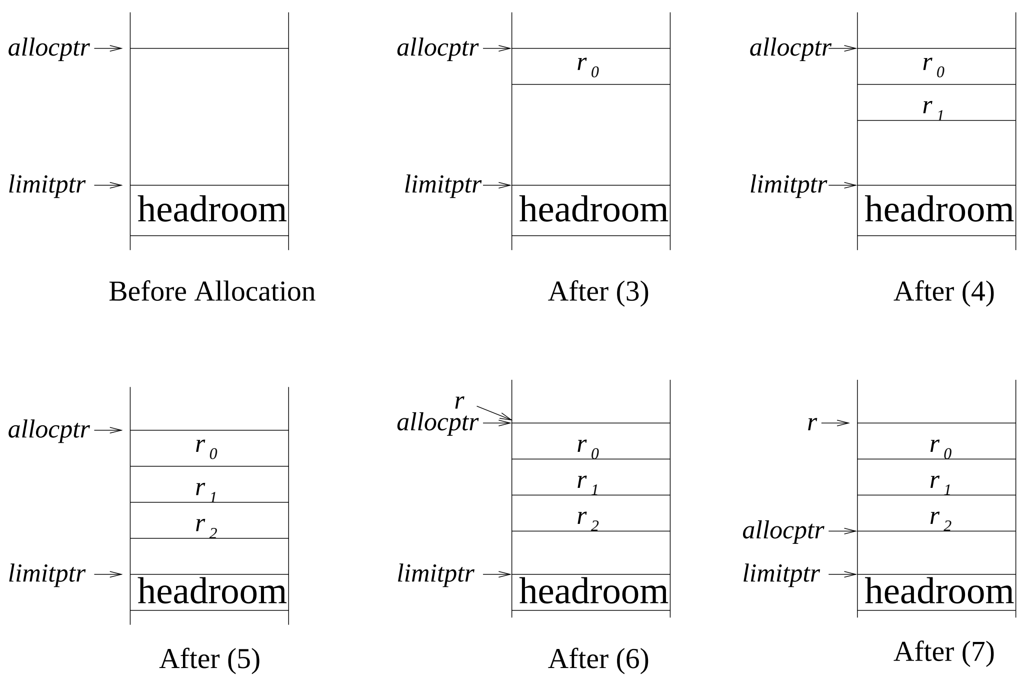

Figure 4.3: Heap Status during Record Allocation

	l_0 :
(1)	**subcc** $limitptr, allocptr, r_{al}$ % $r_{al} = limitptr - allocptr$
(2)	**bl** l_1 % if $limitptr - allocptr < 0$, goto l_1
	l_2 :
(3)	**st** $r_0, [allocptr + 0]$ % $M[allocptr + 0] = r_0$
(4)	**st** $r_0, [allocptr + 4]$ % $M[allocptr + 4] = r_0$
(5)	**st** $r_0, [allocptr + 8]$ % $M[allocptr + 8] = r_0$
(6)	**mov** $allocptr, r$ % $r = allocptr$
(7)	**add** $allocptr, 12, allocptr$ % $allocptr = allocptr + 12$
	...
	l_1 : ...

Figure 4.4: Known-length Array Allocation

Block l_2 initializes the record field by field. The new record occupies three words right after the *allocptr*. Instruction (3) stores r_0 in the first field at address $allocptr + 0$, and instruction (4) stores r_1 in the second field at address $allocptr + 4$, and instruction (5) stores r_2 in the third field at address $allocptr + 8$. Then instruction (6) assigns the *allocptr* to register r. As a result, r points to the beginning of the new record. Instruction (7) increases the *allocptr* by 12, the size of the new record. After (7) the *allocptr* points to the next available free space on the heap and this allocation is finished.

4.2.2 Known-length Array Allocation

Allocating a known-length array could be done in a similar way to allocating a record, by first testing out-of-heap and then initializing element by element.

Figure 4.4 shows an instruction sequence that allocates a three-element array with initial value r_0. It is very similar to the instruction sequence in Figure 4.2.

Block l_0 tests whether there is enough space for a three-element array by comparing the *limitptr* and the *allocptr*. If there is enough space, block l_2 then initializes three elements and adjusts the *allocptr* in the end. The only difference between the instruction sequences in Figure 4.4 and Figure 4.2 is that the initialization in Figure 4.4 uses the same initial value for each element.

We could use an initialization instruction to initialize each element of a known-length array, therefore, n store instructions to initialize an n-element array. But for a large array, for example, an array with hundreds or thousands of elements, this initialization style would be very tedious. An alternative is to use a loop. For an unknown-size array, a loop for initialization is not an alternative, but a necessity.

4.2.3 Unknown-length Array Allocation

Figure 4.5 shows how to allocate an array of size r_l with initial value r_0, where r_l is not known at compile time. Here r_l is not the number of elements in the array, but 4 (the word size) times the element number, that is, the size of the memory the array occupies[1].

Block l_0 tests for heap exhaustion. Since whether the required space is less than the headroom or not is unknown, simply comparing the *limitptr* and the *allocptr* is not enough. We need to compare the free space on the heap with the required space. Instruction (1) calculates the free space on the heap $limitptr - allocptr$. Instruction (2) compares $limitptr - allocptr$ with the required space r_l. If $limitptr - allocptr \geq r_l$, there is enough space for allocation, and the control falls through to block l_2; otherwise, instruction (3) branches to block l_1.

Initializing each element in the array requires a loop. The loop index is the offset of the array element to be initialized next. It goes from 0 to $r_l - 4$, and is

[1]We assume each element in the array is a single word, such as an integer or a pointer to a boxed data structure.

	l_0 :	
(1)	**subcc** $limitptr, allocptr, r_{al}$	% $r_{al} = limitptr - allocptr$
(2)	**cmp** r_{al}, r_l	% compare r_{al} and the array size r_l
(3)	**bl** l_1	% if $r_{al} < r_l$, goto l_1
	l_2 :	
(4)	**mov** $r_i, 0$	% initialize loop index r_i with 0
	test :	
(5)	**subcc** $r_i, r_l, \%g0$	% compare loop index r_i and size r_l
(6)	**bge** *done*	% if $r_i \geq r_l$, goto *done*
	loop :	% not all elements are initialized
(7)	**st** $r_0, [allocptr + r_i]$	% $M[allocptr + r_i] = r_0$
(8)	**add** $r_i, 4, r_i$	% increment r_i with 4, the word size
(9)	**ba** *test*	% goto *test*
	done :	% all elements are initialized
(10)	**mov** $allocptr, r_a$	% $r_a = allocptr$
(11)	**add** $allocptr, r_l, allocptr$	% increase the *allocptr* by r_l
	...	
	l_1 : ...	% memory not enough for allocation

Figure 4.5: Unknown-length Array Allocation

incremented by 4 each time, since the first element of the new array is at address $allocptr + 0$, the second at $allocptr + 4$, and the last at $allocptr + r_l - 4$. The termination condition is $r_i \geq r_l$.

Block l_2 initializes the loop index r_i with 0 (instruction (4)).

Block *test* tests whether the loop is done. Instruction (5) compares the loop index r_i with the array size r_l. If $r_i \geq r_l$, every element has been initialized, then the control goes to *done*. Otherwise it falls through to *loop*.

Block *loop* runs an iteration of the initialization loop. Instruction (7) initializes the array element at address $allocptr + r_i$. Instruction (8) increases the loop index by 4. Instruction (9) jumps back to *test*.

When the control arrives at block *done*, the array has been completely initialized. Instruction (10) assigns the *allocptr* to r_a and r_a points to the beginning of the new array. Instruction (11) adds r_l to the *allocptr* and the *allocptr* points to the next free space.

4.3 Macro Instructions

4.3.1 DTAL

Dependently Typed Assembly Language has a macro instruction *newarray* for array allocation [69]. Instruction newarray[τ] r, l, v_0 creates an τ array of length l, initializes each element with v_0 (of type τ) and assigns the array to r. This instruction does all allocation steps in one macro, and gives the new array type τ array(l), read as an τ array of length l. After type-checking and before execution, the macro expands to a bunch of instructions that test the heap and initialize.

4.3.2 TALx86

When allocating a record, TALx86 first reserves space on the heap and then initializes each field [43]. The first step is done by a macro instruction *malloc* that calls a runtime routine, and the second by a sequence of initialization instructions.

TALx86 uses initialization flags to indicate which field has been initialized and which has not. The *malloc* macro specifies explicitly the type of the target record, with all flags cleared. After one field is initialized, the corresponding flag in the record type is set.

The following TALx86 instruction sequence allocates a two-field record. The right side is the type of the new record.

$$\begin{array}{l|l} \text{malloc } 8, \langle B4, B4\rangle & [B4^u, B4^u] \\ \text{mov } [r+0], 3 & [B4^{rw}, B4^u] \\ \text{mov } [r+4], 4 & [B4^{rw}, B4^{rw}] \end{array}$$

Instruction malloc reserves two words on the heap, and puts a pointer to the new space into a register. The next two mov instructions initialize the two fields respectively. Type B4 means 4-byte integers. The superscripts of B4 are initialization flags: u stands for uninitialized and rw stands for readable and writable. The *malloc* instruction specifies the record's type. The flag for each field is u at this point. After a field is initialized, the corresponding flag is changed from u to rw, indicating it can be read or written. The type-checker does not allow reading or writing uninitialized fields.

TALx86 also allocates and initializes arrays with macro instructions.

4.3.3 TALT

Like TALx86, TALT also allocates space with a macro instruction *malloc* and initializes a new record field by field [21]. But it does not use initialization flags, thus requires that the initialization instruction sequence be uninterrupted. The type-checker enters a special state after *malloc*, and it returns to normal state after initialization is finished. No other instructions are allowed during allocation.

4.3.4 Orderly Lambda Calculus

Peterson *et al.* proposed an orderly lambda calculus to express the memory layout in great details [50]. The heap is modeled as a contiguous block of memory bounded by allocation pointer and limit pointer, just like the LTAL heap model. The allocation process is also divided into reservation, initialization and heap status update. The reservation is done by a macro instruction *reserve* similar to *malloc*. The type-checker uses an ordered context (an ordered list of bindings from heap allocations to types) to describe the memory region of the new record that is currently being initialized.

With the power of orderly linear logic, the calculus could express memory adjacency, size preservation and pointer indirection, but one limitation of this work is that it lacks the ability to express array allocation because the type-checker needs to know the size of an object to be allocated.

4.3.5 Problems with Macro Instructions

With the existence of macro instructions, no optimizations could propagate to the sequences these instructions expand to, such as constant propagation. Furthermore, the instruction scheduler cannot insert other instructions into the sequences. When creating a record with many fields, unbreakable allocation sequences will increase the register pressure. We have to compute all the fields and put them in registers before the allocation starts. An alternative is to let the instruction scheduler optimize the allocation by computing and initializing one field at a time. Figure 4.3.5 shows two pseudocode sequences for allocating record $r = [x_0+y_0, x_1+y_1, \ldots, x_{n-1}+y_{n-1}]$. The left allocation cannot be interrupted and requires that registers $r_0, r_1, \ldots, r_{n-1}$ be all live during the allocation. The right one is optimized by interleaving computation with initialization, and uses only r_0 for computation.

The problem with macro instructions lies in the fact that the type system is not expressive enough to describe each step of allocation.

Unbreakable Allocation	Optimized Allocation
$r_0 = x_0 + y_0$	$r_0 = x_0 + y_0$
$r_1 = x_1 + y_1$	$r[0] = r_0$
$r_2 = x_2 + y_2$	$r_0 = x_1 + y_1$
$r_3 = x_3 + y_3$	$r[1] = r_0$
$\ldots$	$\ldots$
$r_{n-1} = x_{n-1} + y_{n-1}$	$\ldots$
$r[0] = r_0$	$\ldots$
$r[1] = r_1$	$\ldots$
$\ldots$	$\ldots$
$r[n-2] = r_{n-2}$	$r_0 = x_{n-1} + y_{n-1}$
$r[n-1] = r_{n-1}$	$r[n-1] = r_0$

Figure 4.6: Un-optimized and Optimized Allocation Sequences

In order to give the compiler as much flexibility as possible to optimize the generated code, LTAL chooses not to use macro instructions, but to make explicitly each allocation step, and it has an expressive type system to type-check each step. Therefore, LTAL instruction sequence for allocation is not fixed. The instruction scheduler can shuffle the allocation instructions with others.

The FPCC-ML compiler can (and does) optimize by coalescing multiple tests for out-of-heap. It makes one test cover the sequential allocation of several different records (and known-length arrays) in a control-flow path that covers several basic blocks. This optimization can only be done when the tests are explicit; it is imposible with macro instructions.

4.4 LTAL Heap Allocation

To express each allocation step, at term level, LTAL has a set of allocation instructions. As I described in Section 4.1, the *allocptr* and the *limitptr* play very important roles in allocation and appear in machine instructions. But in LTAL they are implicit—they do not appear in LTAL programs. LTAL allocation instructions implicitly manipulate them and update heap status. The status of these two registers are kept in the allocation environment. Therefore, the type-checker can get their status easily.

At type level, LTAL needs to keep track of the available space on the heap and the type of the partial structure being initialized.

The LTAL type-checker uses an allocation environment $\hbar$ to check heap allocation, both for records and for arrays. The allocation environment consists of three parts (τ_1, τ_2, τ): τ_1 is the size of the space guaranteed to be available on the heap; τ_2 is the size of the space initialized so far for the partial structure being allocated; and τ is the type of the partial structure. Both τ_1 and τ_2 are types instead of integers to accommodate allocation of unknown-size structures (arrays). Both should be of kind Ω_N. The type-checking does not need the initialization flags used in TALx86 [45].

In the underlying semantic model, a virtual register *boundary* points to the frontier of allocation. Before allocation, *boundary* is the same as the *allocptr*. After each initialization instruction, *boundary* is bumped by 4. After record instruction, the *allocptr* meets *boundary*. Another virtual register *limit* points to the real limit of the heap ($limit = limitptr + 4096$).

The semantic model uses existential types for allocation environment. LTAL allocation environment (m, n, τ) is represented as TML type environment $\{limitptr - allocptr \geq m, boundary - allocptr = n, allocptr : \tau, limit - limitptr = 4096, \ldots\}$[2]. $limitptr - allocptr \geq m$ guarantees at least m free space between the *allocptr*

[2]For clarity, I use equations in the environment, not the same as notations used in TML.

and the *limitptr*. $boundary - allocptr = n$ indicates n space is initialized, and $allocptr : \tau$ means the partial structure has type τ.

The rest of this section explains LTAL allocation instructions and their typing rules, grouped by their functionality.

4.4.1 Testing for Heap Exhaustion

Instructions

We have seen in the untyped allocation examples instructions that test whether there is enough space on the heap:

- an instruction that compares the *limitptr* with the *allocptr* (instruction (1) in Figures 4.2, 4.4 and 4.5)
- an instruction that compares $limitptr - allocptr$ with the required space (instruction (2) in Figure 4.5)
- an instruction that does conditional branch in case of no enough memory (instruction (2) in Figure 4.2 and 4.4, instruction (3) in Figure 4.5).

LTAL has three instructions that correspond to them respectively.

- Instruction **testAvail** computes the difference between the *limitptr* and the *allocptr*, assigns the result to a reserved register r_{al}, and sets condition codes.

  Instead of reserving register r_{al} for $limitptr - allocptr$, an alternative approach is to assign the result to a variable and treat the variable the same as any other variable in register allocation. Now $limitptr - allocptr$ is accessible from a variable in the program, and it should be invalidated after each allocation since the *allocptr* is changed after each allocation. The type-checker should prevent the program from preserving $limitptr - allocptr$ at some point (store it into memory or hide it using existential types), and using it later when the heap status has changed. This requires more complicated syntax and semantics; therefore, LTAL simply reserves a register for testAvail, and LTAL programs cannot manipulate the register directly since programs use variables instead of registers and no variables can be assigned register r_{al}.

- Instruction **testFull**(v) compares register r_{al} $(limitptr - allocptr)$ with v (the required space), and sets condition codes. If $r_{al} \geq v$, there is enough memory.

  Instruction testFull should always follow instruction testAvail, as in Figure 4.5, because testFull uses register r_{al} which can only be set by testAvail. I will explain how the type-checker makes sure of this order next.

- Instruction **iffull** then l_1 else l_2 branches according to the condition codes set by testing out-of-heap.

Typing Rules

The main point of the typing rules for testing instructions is: when there is enough space on the heap, the type-checker should set up an allocation environment that allows initializations that follow.

Figure 4.7 shows the typing rules for the three testing instructions. The judgement $LRT; \rho; \hbar; \Phi; cc \vdash_{\ell} l$ states that the current environment matches the signature of block l (see Appendix A for its complete typing rule). If this judgement holds, it is safe to jump to block l. Type **freem** represents $limitptr - allocptr$.

$$\frac{\hbar = (\tau, \overline{0}, \text{boxed})}{LRT \vdash (\rho; \hbar; \Phi; cc)\ \{\text{testAvail}\}\ (\rho; \hbar; \Phi; \text{cc\_cmp}(\text{freem}, \overline{0}))}$$

$$\frac{LRT; \rho; \Phi \vdash v : \text{int}_= \tau_n}{\begin{array}{c}LRT \vdash (\rho; (\tau, \overline{0}, \text{boxed}); \Phi; \text{cc\_cmp}(\text{freem}, \overline{0}))\ \{\text{testFull(v)}\}\\ (\rho; (\tau, \overline{0}, \text{boxed}); \Phi; \text{cc\_cmp}(\text{freem}, \tau_n))\end{array}}$$

$$\frac{\begin{array}{l}cc = \text{cc\_cmp}(\text{freem}, \tau_n) \qquad LRT; \rho; (\tau, \overline{0}, \text{boxed}); \Phi; cc \vdash_\ell l_1\\ LRT; \rho; (\tau_n + \overline{4096}, \overline{0}, \text{boxed}); \Phi; cc \vdash_\ell l_2\end{array}}{LRT \vdash (\rho; (\tau, \overline{0}, \text{boxed}); \Phi; cc)\ \{\text{iffull then } l_1 \text{ else } l_2\}\ (\rho; (\tau, \overline{0}, \text{boxed}); \Phi; cc)}$$

Figure 4.7: Typing Rules for Testing Instructions

Instructions testAvail, testFull, and iffull establish the allocation environment in which the initialization instructions type-check.

Instruction testAvail is translated to Sparc instruction subcc $limitptr, allocptr, r_{al}$. It sets condition code cc_cmp(freem, $\overline{0}$). Once $limitptr \geq allocptr$, there is at least 4096 space available. TestAvail can not appear during an on-going allocation, as enforced by the requirement in the typing rule that the second type of the allocation environment be 0 and the third be boxed. This means no allocation can start until the previous structure is finished.

Instruction testFull(v) maps to the Sparc instruction subcc $r_{al}, r_v, \%g0$ if v is assigned register r_v. It is used for allocating large (greater than headroom) or size-unknown structures. This instruction sets condition codes cc_cmp(freem, n) where n is the value of v (normally the required space for allocation).

Instruction testFull uses register r_{al}, thus when testFull is executed r_{al} should contain the desired value $limitptr - allocptr$. Only instruction testAvail can assign this value to r_{al}. We should guarantee that a testFull instruction always follows a testAvail instruction. The typing rule of testFull enforces this by requiring the condition code cc_cmp(freem, $\overline{0}$) in the precondition. This condition code can ultimately only come from testAvail[3].

Instruction iffull consumes the condition codes cc_cmp(freem, τ_n) set by a testAvail or a testFull instruction. If the condition codes are not satisfied (freem $\geq \tau_n$), it establishes the guaranteed available space $\tau_n + 4096$ in the allocation environment for l_2 where initialization happens .

In TML, the condition codes cc_cmp(freem, τ) could be specified as the type of virtual register CC in the type environment $\{d = \mathit{limitptr} - \mathit{allocptr}, r_{al} = d, \mathrm{CC} : \mathit{cmp}(d, \tau), \ldots\}$.

Using the semantics, the above three typing rules could be reasoned about as follows:

testAvail: TestAvail is mapped to TML instruction tml_subcc($\mathit{limitptr}, \mathit{allocptr}, r_{al}$). Suppose in the precondition the *allocptr* has type int$_=$ m_a and the *limitptr* has type int$_=$ m_l, this TML instruction tml_subcc would give the virtual register CC a type $\mathit{cmp}(m_l, m_a)$, which is the same as $\mathit{cmp}(m_l - m_a, 0)$.

testFull: TestFull(v) is mapped to TML instruction tml_subcc($r_{al}, r, \%g0$). Suppose in the precondition r_{al} has type int$_=$ d and r has type int$_=$ τ_n. The TML instruction would give CC type $\mathit{cmp}(d, \tau_n)$.

iffull: If the precondition ϕ says $\{d = \mathit{limitptr} - \mathit{allocptr}, r_{al} = d, \mathit{limit} - \mathit{limitptr} = 4096, \mathrm{CC} : \mathit{cmp}(d, \tau_n)\}$, and iffull branches to label l_1, we need to prove that $\phi[CC \mapsto \mathit{cmp}(d, \tau_n) \cap ge]$ satisfies the precondition of l_1. The type $\mathit{cmp}(d, \tau_n) \cap$

[3] Instruction testFull(0) also generates condition code status cc_cmp(freem, $\overline{0}$), but its precondition still requires cc_cmp(freem, $\overline{0}$). Only testAvail instruction "generates" this condition code status.

ge implies $d \geq \tau_n$. Along with the judgements of the *allocptr*, the *limitptr*, r_{al} and *limit*, we could prove that $limit - allocptr \geq \tau_n + 4096$.

4.4.2 Record Allocation

Instructions

As shown in Figure 4.2, allocating a record is completed by initializing each field (instructions (3) - (5)), followed by assigning the *allocptr* to the destination register (instruction (6)), then followed by increasing the *allocptr* (instruction (7)). LTAL has the following corresponding instructions:

- Instruction $\mathbf{store}(v_i, v)$ stores value v to the memory word at address $allocptr + v_i$. It initializes the field at offset v_i of the record being allocated.

- Instruction $v = \mathbf{record}$ assigns the *allocptr* to v. At this point, every field of the new structure should be completely initialized. The *allocptr* is unchanged during initialization. Therefore, it points to the beginning of the initialized structure when the "record" instruction is executed. After the record instruction, v points to the initialized structure. The *allocptr* moves on afterwards by instruction incAlloc.

- Instruction $\mathbf{incAlloc}(v)$ adds v to the *allocptr*. It should follow a record instruction. The value v should be the size of the structure that is just initialized. After incAlloc, the *allocptr* points to the next available word on the heap.

$$\frac{LRT;\rho;\Phi\vdash v':\mathrm{int}_{=}\,\bar{i} \qquad LRT;\rho;\Phi\vdash v:t_i \qquad t' = t\cap(\text{field } i\ t_i)}{LRT\vdash(\rho;(\tau_m,\tau_n,t);\Phi;cc)\ \{\text{store}(v',v)\}\ (\rho;(\tau_m,\tau_n+\bar{4},t');\Phi;cc)}$$

$$\frac{}{LRT\vdash(\rho;(\tau_m,\tau_n,\tau);\Phi;cc)\ \{v=\text{record}\}\ (\rho;(\tau_m,\tau_n,\tau);\Phi,v\!:\!\tau;cc)}$$

$$\frac{LRT;\rho;\Phi\vdash v:\mathrm{int}_{=}\,\tau_n}{\begin{array}{c}LRT\vdash(\rho;(\tau_m,\tau_n,\tau);\Phi;\text{cc\_cmp}(\text{freem},k))\ \{\text{incAlloc } v\}\\ (\rho;(\tau_m-\tau_n,\bar{0},\text{boxed});\Phi;\text{cc\_none})\end{array}}$$

$$\frac{LRT;\rho;\Phi\vdash v:\mathrm{int}_{=}\,\tau_n}{\begin{array}{c}LRT\vdash(\rho;(\tau_m,\tau_n,\tau);\Phi;cc)\ \{\text{incAlloc } v\}\\ (\rho;(\tau_m-\tau_n,\bar{0},\text{boxed});\Phi;cc)\end{array}}$$

Figure 4.8: Rules for Record Allocation Instructions

Typing Rules

When type-checking these instructions, the most important thing that the type-checker does is to update the heap status in the allocation environment. The typing rules for the record allocation instructions are shown in Figure 4.8.

When a field of the current partial record is initialized by a store instruction, the initialized space (second part of the allocation environment) is increased by 4, and one more conjunct (a field type) is added into the type of the partial record (third part of the allocation environment).

After initialization, the *allocptr* is copied to a variable (with the desired record type) by instruction v = record, and then the *allocptr* is adjusted to point to the next available memory word by instruction incAlloc. After instruction incAlloc, the type-checker subtracts the allocated space from the first part of the allocation environment—the available space, resets the second part to 0, and resets the third part to the empty record, i.e. "boxed". Instruction incAlloc invalidates the condition codes set by testAvail and testFull because the *allocptr* has been changed. So

LTAL	Sparc
l_0 :	l_0 :
testAvail	**subcc** $limitptr, allocptr, r_{al}$
iffull then l_1 else l_2	**bl** l_1
l_2 :	l_2 :
store$(0, v_0)$	**st** $r_0, [allocptr + 0]$
store$(4, v_1)$	**st** $r_1, [allocptr + 4]$
store$(8, v_2)$	**st** $r_2, [allocptr + 8]$
$v =$ record	**mov** $allocptr, r$
incAlloc 12	**add** $allocptr, 12, allocptr$
...	...
l_1 : ...	l_1 : ...

Figure 4.9: Record Allocation Example

if before incAlloc, the condition-code status is cc_cmp(freem, k) which can set only by testAvail or testFull, it is reset to cc_none afterwards.

The LTAL store instruction is mapped to the Sparc store instruction, the LTAL record instruction to the Sparc move instruction, and the LTAL incAlloc instruction to the Sparc add instruction. The typing rules in Figure 4.8 could be proved from the semantics of the corresponding machine instructions.

Instruction $v =$ record can be used to create aliases during the initialization of a record. But the uses of those aliases may be restricted, because we don't track alias information which many TALs including TALx86 do not track either. See Section 4.4.5 for discussion about aliases.

Type-checking the Record Allocation Example

Figure 4.9 shows an LTAL instruction sequence that corresponds to the Sparc sequence in Figure 4.2. Variables v, v_0, v_1, v_2 are assigned registers r, r_0, r_1, r_2 respectively.

LTAL	Allocation Environment
l_0 :	$(\tau_m, \overline{0}, \text{boxed})$
testAvail	
	$(\tau_m, \overline{0}, \text{boxed})$
iffull then l_1 else l_2	
	$(\tau_m, \overline{0}, \text{boxed})$
l_2 :	$(\overline{4096}, \overline{0}, \text{boxed})$
store$(0, v_0)$	
	$(\overline{4096}, \overline{4}, \text{field } \overline{0}\ \tau_0)$
store$(4, v_1)$	
	$(\overline{4096}, \overline{8}, (\text{field } \overline{0}\ \tau_0) \ \cap \ (\text{field } \overline{4}\ \tau_1))$
store$(8, v_2)$	
	$(\overline{4096}, \overline{12}, (\text{field } \overline{0}\ \tau_0) \ \cap \ (\text{field } \overline{4}\ \tau_1) \ \cap \ (\text{field } \overline{8}\ \tau_2))$
$v = \text{record}$	
	$(\overline{4096}, \overline{12}, (\text{field } \overline{0}\ \tau_0) \ \cap \ (\text{field } \overline{4}\ \tau_1) \ \cap \ (\text{field } \overline{8}\ \tau_2))$
incAlloc 12	
	$(\overline{4084}, \overline{0}, \text{boxed})$
...	
l_1 : ...	

Figure 4.10: Checking Record Allocation

In block l_0, instruction testAvail assigns $limitptr - allocptr$ to r_{al} and sets condition codes. Then the branch instruction **iffull** "consumes" the condition codes, and branches to l_1 if there is no enough memory on the heap for the record. Otherwise it falls-through to l_2.

Block l_2 initializes the new record. Instruction **store**(i, v_i) initializes the word at address $allocptr + i$ with v_i. Three store instructions initialize three fields with v_0, v_1, and v_2 respectively. Instruction $v = \text{record}$ copies the *allocptr* to v. Instruction incAlloc 12 increases the *allocptr* by 12.

Figure 4.10 shows how the allocation environment changes during type-checking this sequence. Variables v_0, v_1, v_2 are of type τ_0, τ_1, τ_2 respectively.

In block l_0, instruction testAvail sets the condition codes cc_cmp(freem, $\overline{0}$). Instruction iffull sets an allocation environment that statically guarantees 4096 space for the fall-through case l_2.

At the beginning of l_2, the allocation environment ($\overline{4096}, \overline{0}$, boxed)indicates that: 4096 space is guaranteed, none of it is initialized, and the type of the record is boxed. After each store instruction, one more field type is added as a conjunct to the record type. The original boxed type disappears after the first field type is added, because field type implies boxed type. More precisely, field type is a subtype of boxed type in the underlying model. After the three store instructions, the record has type (field $\overline{0}$ τ_0) $\cap$ (field $\overline{4}$ τ_1) $\cap$ (field $\overline{8}$ τ_2) . The record instruction assigns this type to v. The incAlloc instruction resets the allocation environment. It deducts the allocated space 12 from the available space 4096. Now the available space becomes 4084.

4.4.3 Known-length Array Allocation

The Sparc instruction sequences in Figures 4.2 and 4.4 are very similar, but type-checking these two sequences is different. This is because LTAL uses the same set of instructions for allocating known-length arrays and unknown-length ones. Using intersection types for partial arrays in type-checking is unacceptable, since the type-checker does not know how many conjuncts an unknown-size array would have.

Notice that the type of each array element is the same. The only thing that needs to keep track of in the type of the partial array is how much has been initialized. LTAL uses a fixed format array(τ_i, τ) for the type of the partial array, where τ_i is the space of initialized elements, and τ is the type of elements. At the beginning of

$$\frac{}{LRT \vdash (\rho; (\tau_m, \overline{0}, \text{boxed}); \Phi; cc)\ \{\text{arrStart}[\tau]\}\ (\rho; (\tau_m, \overline{0}, \text{array}(\overline{0}, \tau)); \Phi; cc)}$$

$$\frac{LRT; \rho; \Phi \vdash v_i : \text{int}_{=}\, \tau_n \qquad \tau_n' = \tau_n + \overline{4} \qquad LRT; \rho; \Phi \vdash v : \tau}{\begin{array}{c} LRT \vdash (\rho; (\tau_m, \tau_n, \text{array}(\tau_n, \tau)); \Phi; cc)\ \{\text{storeA}(v_i, v)\} \\ (\rho; (\tau_m, \tau_n', \text{array}(\tau_n', \tau)); \Phi; cc) \end{array}}$$

Figure 4.11: Rules for Array Allocation Instructions

array allocation, the initialized space is 0, thus the partial array has type array$(0, \tau)$. Then after one more element is initialized, τ_i is increased by 4, until in the end τ_i is the same as the desired size. Assigning the *allocptr* to the destination register and increasing the *allocptr* are the same as record allocation.

To set the initial array type array$(0, \tau)$, LTAL has an instruction arrStart that maps to no Sparc instruction. Initializing an element of an array is implemented by an LTAL instruction storeA, which has a different typing rule from the store instruction used for record initialization.

Instructions

- Instruction **arrStart**$[\tau]$ sets up the initial array type in the allocation environment before initializing a τ array.

- Instruction **storeA**(v_i, v) stores value v to the memory word whose address is *allocptr* $+ v_i$. It initializes the element at offset v_i of the array.

Typing Rules

The typing rules for arrStart and storeA are shown in Figure 4.11. The instructions testing for heap exhaustion sets up an allocation environment $(n, \overline{0}, \text{boxed})$

when there is at least n space on the heap (see Figure 4.7). Before initializing an array, instruction arrStart[τ] coerces the array type (third type in the allocation environment) from boxed to array$(\overline{0}, \tau)$. When an element is initialized by a storeA instruction, the type of the array changes from array(n, τ) to array$(n + \overline{4}, \tau)$.

The semantic model of LTAL type array(τ_n, τ) is based on the TML type boxed$\cap$ $\forall_{i \in [0, \tau_n)}$offset$(i, \tau)$, meaning if a value v has type array(τ_n, τ), then $m(v + i) : \tau$ for any $0 \leq i < \tau_n$. Therefore, the arrStart rule could be proved by TML subtyping boxed $\subset$ array$(\overline{0}, \tau)$ for any τ.

The storeA instruction is mapped to TML's store instruction. The storeA rule is proved by the semantics of machine instruction store and the fact that $\forall_{i \in [0, \tau_n)}$offset$(i, \tau) \cap$ offset$(\tau_n + 4, \tau) \subset \forall_{i \in [0, \tau_n + 4)}$offset$(i, \tau)$.

Type-checking the Known-length Array Example

Figure 4.12 shows the LTAL instruction sequence that corresponds to the Sparc code segment in Figure 4.4. LTAL variables v_0 and a are assigned registers r_0 and r_a respectively. Variable v_0 has type τ.

In block l_2, instruction arrStart first sets up the initial array type. Then the three storeA instructions initialize the three elements. Instruction record lets a point to the beginning of the new array, and finally the *allocptr* is increased by 12 (the size of the new array) by instruction incAlloc.

Figure 4.13 shows the change of the allocation environment during type-checking.

Similar to the record allocation example, block l_1 sets up an allocation environment $(\overline{4096}, \overline{0}, \text{boxed})$ for block l_2.

In block l_2, the first instruction arrStart coerces the allocation environment to $(\overline{4096}, \overline{0}, \text{array}(\overline{0}, \tau))$. After instruction storeA$(0, v_0)$, the allocation environment

LTAL	Sparc
l_0 :	l_0 :
testAvail	**subcc** $limitptr, allocptr, r_{al}$
iffull then l_1 else l_2	**bl** l_1
l_2 :	l_2 :
arrStart$[\tau]$	
storeA$(0, v_0)$	**st** $r_0, [allocptr + 0]$
storeA$(4, v_0)$	**st** $r_0, [allocptr + 4]$
storeA$(8, v_0)$	**st** $r_0, [allocptr + 8]$
a = record	**mv** $allocptr, r_a$
incAlloc 12	**add** $allocptr, 12, allocptr$
$l_1 : \ldots$	$l_1 : \ldots$

Figure 4.12: LTAL Instruction Sequence for Known-length Array Example

becomes $(\overline{4096}, \overline{4}, \text{array}(\overline{4}, \tau))$, meaning the first element of the array is initialized. After the next two storeA instructions, the allocation environment becomes $(\overline{4096}, \overline{12}, \text{array}(\overline{12}, \tau))$. The next record instruction assigns a type array$(\overline{12}, \tau)$, an τ array of size 12 (three elements). The incAlloc instruction subtracts 12 from the available space 4096, and resets the other two parts of the allocation environment.

4.4.4 Unknown-length Array Allocation

Type-checking unknown-length array allocation is more complicated because array initialization is completed by a loop.

To model the unknown size of the array, LTAL uses polymorphism over the size: the size is represented as a type variable in the initialization function.

Modeling the initialized space needs another type variable because its value changes during each loop iteration. Since array elements are initialized sequentially, the value of the loop index is equal to the initialized space. Therefore, the LTAL checker uses a single type variable to indicate the value of the loop index and the

LTAL	Allocation Environment
l_0 :	$(\tau_m, \overline{0}, \text{boxed})$
testAvail	
	$(\tau_m, \overline{0}, \text{boxed})$
iffull then l_1 else l_2	
	$(\tau_m, \overline{0}, \text{boxed})$
l_2 :	$(\overline{4096}, \overline{0}, \text{boxed})$
arrStart$[\tau]$	
	$(\overline{4096}, \overline{0}, \text{array}(\overline{0}, \tau))$
storeA$(0, v_0)$	
	$(\overline{4096}, \overline{4}, \text{array}(\overline{4}, \tau))$
storeA$(4, v_0)$	
	$(\overline{4096}, \overline{8}, \text{array}(\overline{8}, \tau))$
storeA$(8, v_0)$	
	$(\overline{4096}, \overline{12}, \text{array}(\overline{1}2, \tau))$
$a =$ record	
	$(\overline{4096}, \overline{12}, \text{array}(\overline{1}2, \tau))$
incAlloc 12	
	$(\overline{4084}, \overline{0}, \text{boxed})$
...	
l_1 : ...	

Figure 4.13: Checking Known-length Array Allocation

initialized space in the allocation environment. Each time an element is initialized and the loop index is increased, the partial array type in the allocation environment changes accordingly. If τ_n space is initialized, the array should have type array(τ_n, τ). This array type is refined to an array type with desired size when the loop is finished. The type refinement is done by a new LTAL instruction ifinitA.

Instructions

- Instruction **ifinitA** then l_1 else l_2 is translated to Sparc bge instruction. It branches to l_1 with the allocation environment coerced, if the array initialization is finished (indicated by the loop index $\geq$ the array size).

Typing Rules

The typing rules of cmpr and ifinitA are shown in Figure 4.14.

Instruction cmpr is used for singleton integer comparison. It sets condition codes cc_cmp(τ_1, τ_2) when comparing two values of type $\text{int}_= \tau_1$ and $\text{int}_= \tau_2$ respectively. In a typical use, τ_1 is a type variable and τ_2 is a singleton literal such as $\overline{0}$. The comparison result will be used to refine the type variable.

Instruction ifinitA should follow an instruction that compares the loop index i and the array size l. Suppose i has type $\text{int}_= \tau_n$ and l has type $\text{int}_= \tau_l$. The comparison sets condition codes cc_cmp(τ_n, τ_l). After instruction ifinitA then l_1 else l_2, the allocation environment $(\tau_m, \tau_n, \text{array}(\tau_n, \tau))$ is coerced to $(\tau_m, \tau_l, \text{array}(\tau_l, \tau))$ in l_1 when $\tau_n \geq \tau_l$.

If branch ifinitA is taken, the condition codes would imply $\tau_n \geq \tau_l$, and then we could prove that $\text{array}(\tau_n, \tau) \subset \text{array}(\tau_l, \tau)$.

$$\frac{LRT;\rho;\Phi \vdash v_1 : \mathrm{int}_= \tau_1 \quad LRT;\rho;\Phi \vdash v_2 : \mathrm{int}_= \tau_2}{LRT \vdash (\rho;\hbar;\Phi;cc)\ \{\mathrm{cmpr}(v_1,v_2)\}\ (\rho;\hbar;\Phi;\mathrm{cc\_cmp}(\tau_1,\tau_2))}$$

$$\frac{\begin{array}{l} cc = \mathrm{cc\_cmp}(\mathrm{int}_= \tau_n, \mathrm{int}_= \tau_l) \qquad LRT;\rho;(\tau_m,\tau_l,\mathrm{array}(\tau_l,\tau));\Phi;cc \vdash_\ell l_1 \\ LRT;\rho;(\tau_m,\tau_n,\mathrm{array}(\tau_n,\tau));\Phi;cc \vdash_\ell l_2 \end{array}}{\begin{array}{c} LRT \vdash (\rho;(\tau_m,\tau_n,\mathrm{array}(\tau_n,\tau));\Phi;cc)\ \{\mathrm{ifinitA\ then}\ l_1\ \mathrm{else}\ l_2\} \\ (\rho;(\tau_m,\tau_n,\mathrm{array}(\tau_n,\tau));\Phi;cc) \end{array}}$$

Figure 4.14: Rules for cmpr and ifinitA Instructions

Type-checking the Unknown-length Array Example

Figure 4.15 shows an LTAL instruction sequence that corresponds to the instruction sequence in Figure 4.5, which creates an array of size l, initializes each element with v and assigns the array to a. The size l and the initial value v are both parameters. The LTAL instructions are polymorphic over the type of v (type variable α) and l (type variable β).

Figure 4.16 shows the condition code status and the allocation environment during type-checking the instruction sequence in Figure 4.15. Postconditions of branch/call instructions are not shown because they are the same as preconditions.

Block l_0 tests whether there is enough memory on the heap for the unknown-length array. It is a polymorphic function with two type variables: α is the type of array elements, and β is the desired size of the array. This block requires no space guarantees on the heap, since it would test by itself. It takes two parameters: v of type α is the initial value for elements, and l of type $\mathrm{int}_= \beta$ is the size of the array. Type variable β is of kind Ω_N, which cannot be assigned to value l directly. Type constructor $\mathrm{int}_=$ transforms β to a singleton type of kind Ω.

Instruction testAvail assigns $limitptr - allocptr$ to the reserved register r_{al} and sets condition codes $\mathrm{cc\_cmp}(\mathrm{freem}, \overline{0})$. Instruction testFull($l$) compares r_{al} and l,

LTAL	Sparc
$l_0 : [\alpha : \Omega, \beta : \Omega_N], \overline{0}, \text{cc\_none},$	$l_0 :$
$\quad [v : \alpha, l : \text{int}_= \beta]$	
$\quad$ testAvail	$\quad$ **subcc** $\textit{limitptr}, \textit{allocptr}, r_{al}$
$\quad$ testFull(l)	$\quad$ **cmp** r_{al}, r_l
$\quad$ iffull then l_1 else l_2	$\quad$ **bl** l_1
$l_2 : [\alpha : \Omega, \beta : \Omega_N], \beta, \text{cc\_none},$	$l_2 :$
$\quad [v : \alpha, l : \text{int}_= \beta]$	
$\quad i = 0$	$\quad$ **mov** $r_i, 0$
$\quad$ arrStart[α]	
$\quad$ calln($\textit{test}, [\alpha, \beta, \overline{0}]$)	
$\textit{test} : [\alpha : \Omega, \beta : \Omega_N, \gamma : \Omega_N], \beta, \text{cc\_none},$	$\textit{test} :$
$\quad [v : \alpha, l : \text{int}_= \beta, i : \text{int}_= \gamma]$	
$\quad$ cmpr(i, l)	$\quad$ **subcc** $r_i, r_l, \%g0$
$\quad$ ifinitA then *done* else *loop*	$\quad$ **bge** *done*
$\textit{loop} : [\alpha : \Omega, \beta : \Omega_N, \gamma : \Omega_N], \beta, \text{cc\_none},$	$\textit{loop} :$
$\quad [v : \alpha, l : \text{int}_= \beta, i : \text{int}_= \gamma]$	
$\quad$ storeA(i, v)	$\quad$ **st** $r_v, [\textit{allocptr} + r_i]$
$\quad i = i +_i 4$	$\quad$ **add** $r_i, 4, r_i$
$\quad$ call($\textit{test}, [\alpha, \beta, \gamma + \overline{4}]$)	$\quad$ **ba** *test*
$\textit{done} : [\alpha : \Omega, \beta : \Omega_N], \beta, \text{cc\_none}, [l : \text{int}_= \beta]$	$\textit{done} :$
$\quad a =$ record	$\quad$ **mov** $\textit{allocptr}, r_a$
$\quad$ incAlloc l	$\quad$ **add** $\textit{allocptr}, r_l, \textit{allocptr}$
$\quad \ldots$	$\quad \ldots$
$l_1 : \ldots$	$l_1 : \ldots$

Figure 4.15: Array Allocation Example

and sets condition codes cc_cmp(freem, β). Instruction iffull guarantees β on the heap in the fall-through case l_2[4].

Block l_2 initializes the loop index i and the allocation environment. It has the same type variables and value parameters as l_0, but requires β space available on the heap. Instruction $i = 0$ sets loop index i to 0 and gives i type $\text{int}_= \overline{0}$. Instruction arrStart[$\alpha$] corresponds to no Sparc instruction. It only coerces the

[4]Instruction iffull guarantees $\beta + \overline{4096}$ space in l_2, which is more than l_2's requirement β.

LTAL	Cond. Codes	Alloc. Env.
$l_0 : [\alpha : \Omega, \beta : \Omega_N], \overline{0}, \text{cc\_none},$	cc	$(\tau_m, \overline{0}, \text{boxed})$
$\quad [v : \alpha, l : \text{int}_= \beta]$		
$\quad$ testAvail		
	$\text{cc\_cmp}(\text{freem}, \overline{0})$	$(\tau_m, \overline{0}, \text{boxed})$
$\quad$ testFull(l)		
	$\text{cc\_cmp}(\text{freem}, \beta)$	$(\tau_m, \overline{0}, \text{boxed})$
$\quad$ iffull then l_1 else l_2		
$l_2 : [\alpha : \Omega, \beta : \Omega_N], \beta, \text{cc\_none},$	cc_none	$(\beta, \overline{0}, \text{boxed})$
$\quad [v : \alpha, l : \text{int}_= \beta]$		
$\quad i = 0$		
	cc_none	$(\beta, \overline{0}, \text{boxed})$
$\quad$ arrStart$[\alpha]$		
	cc_none	$(\beta, \overline{0}, \text{array}(\overline{0}, \alpha))$
$\quad$ calln$(\textit{test}, [\alpha, \beta, \overline{0}])$		
$\textit{test} : [\alpha : \Omega, \beta : \Omega_N, \gamma : \Omega_N],$	cc_none	$(\beta, \gamma, \text{array}(\gamma, \alpha))$
$\quad \beta, \text{cc\_none},$		
$\quad [v : \alpha, l : \text{int}_= \beta, i : \text{int}_= \gamma]$		
$\quad$ cmpr(i, l)		
	$\text{cc\_cmp}(\gamma, \beta)$	$(\beta, \gamma, \text{array}(\gamma, \alpha))$
$\quad$ ifinitA then *done* else *loop*		
$\textit{loop} : [\alpha : \Omega, \beta : \Omega_N, \gamma : \Omega_N],$	cc_none	$(\beta, \gamma, \text{array}(\gamma, \alpha))$
$\quad \beta, \text{cc\_none},$		
$\quad [v : \alpha, l : \text{int}_= \beta, i : \text{int}_= \gamma]$		
$\quad$ storeA(i, v)		
	cc_none	$(\beta, \gamma + \overline{4}, \text{array}(\gamma + \overline{4}, \alpha))$
$\quad i = i +_i 4$		
	cc_none	$(\beta, \gamma + \overline{4}, \text{array}(\gamma + \overline{4}, \alpha))$
$\quad$ call$(\textit{test}, [\alpha, \beta, \gamma + \overline{4}])$		
$\textit{done} : [\alpha : \Omega, \beta : \Omega_N],$	cc_none	$(\beta, \beta, \text{array}(\beta, \alpha))$
$\quad \beta, \text{cc\_none}, [l : \text{int}_= \beta]$		
$\quad a = \text{record}$		
	cc_none	$(\beta, \beta, \text{array}(\beta, \alpha))$
$\quad$ incAlloc l		
	cc_none	$(\overline{0}, \overline{0}, \text{boxed})$

Figure 4.16: Type-checking Array Allocation

boxed type in the allocation environment to $\text{array}(\overline{0}, \alpha)$ which is desired by type-checking initialization. Then it falls through to block *test*.

Block *test* tests whether initialization is done. It has one more type variable γ, the initialized space, and one more value parameter i of type $\text{int}_= \gamma$, the loop index. The allocation environment should be $(\beta, \gamma, \text{array}(\gamma, \alpha))$.

The first instruction cmpr compares the loop index i with the size l, and sets the condition codes $\text{cc\_cmp}(\gamma, \beta)$. The next instruction ifinitA checks the condition codes. When $\gamma \geq \beta$, it goes to block *done*, and coerces the allocation environment from $(\beta, \gamma, \text{array}(\gamma, \alpha))$ to $(\beta, \beta, \text{array}(\beta, \alpha))$. Otherwise, it falls through to block *loop*.

Block *loop* initializes one element and increases the loop index. It has the same signature as *test*. When one more element is initialized by a storeA instruction, the allocation environment changes from $(\beta, \gamma, \text{array}(\gamma, \alpha))$ to $(\beta, \gamma+\overline{4}, \text{array}(\gamma+\overline{4}, \alpha))$. After i is increased by 4, i has type $\gamma + \overline{4}$ ($+_i$ is a specialized operator for singleton integer addition). The control goes back to *test*, and instantiates the type variable γ of *test* with $\gamma + \overline{4}$. The type variable γ abstracts the value of the loop index, so that each iteration of the loop can be type-checked uniformly.

Block *done* assigns the initialized array to a and updates the *allocptr*. At the beginning of *done*, the allocation environment should be $(\beta, \beta, \text{array}(\beta, \alpha))$, meaning the whole array has been initialized.

Instruction $a = \text{record}$ assigns the *allocptr* to a, and gives a type $\text{array}(\beta, \alpha)$, meaning an α array of size β. Instruction incAlloc adds l to the *allocptr*, and resets the allocation environment to $(\overline{0}, \overline{0}, \text{boxed})$.

4.4.5 Discussion

To simplify the type-checking, LTAL initializes records and arrays sequentially. The type-checker could allow record initialization in arbitrary order by slightly changing the typing rules for allocation instructions. Since the type of the part record is an intersection of each field's type, and the checker can accept any order of the conjuncts, no matter which order the fields are initialized, the allocation environment can track the record type. The checker just needs some checks to make sure the initialization is in valid space. More specificly, when m space is guaranteed, at every store(i, v) instruction, the type-checker should check whether $i < m$, and whether the ith field has been initialized by checking whether the third part of the allocation environment already has a conjunct "field i τ". The type-checker could either reject duplicate initializition or replace the first initialization with the new one. At instruction incAlloc n, the checker should check whether n is big enough so that every initialized field is in the new record. The checker could allow uninitialized fields as long as they are not accessed.

The typing rules in Figures 4.8, 4.11 and 4.14 do not allow arbitrary aliasing of heap-allocated structures. For example, the following instruction sequence is safe, since both v_0 and v_1 point to a two-field record [0, 1] (Figure 4.17), and fetching the second field of v_0 should return 1. But the LTAL type-checker will reject the fetching because when v_0 is assigned by instruction $v_0 =$ record, the checker thinks v_0 is a one-field record, and does not change v_0's type even it knows v_1 has two fields. The checker lacks the ability to track that v_0 and v_1 are actually aliases, thus they should have the same type.

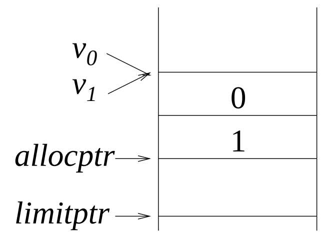

Figure 4.17: Alias Example

$$\text{store}(0, 0)$$
$$v_0 = \text{record}$$
$$\text{store}(4, 1)$$
$$v_1 = \text{record}$$
$$\text{incAlloc}(8)$$
$$v = \text{load}(v_0, 4)$$

The alias type system by Smith, Walker and Morrisett might provide a solution since it is able to track alias information in the type system [57]. Furthermore, it permits reasoning about safe deallocation of memory.

Chapter 5

User-defined Datatypes

In this chapter, I explain how LTAL handles 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 the sum value after tag-checking. LTAL provides flexibility for the compiler writer to choose her preferred style of datatype representation. The representations described in this chapter are not new, but the point is type-checking each aspect of their construction and destruction.

For simplicity, I use the notation $[\tau_0, \tau_1, \ldots, \tau_{n-1}]$ for record types in this chapter, instead of (field 0 τ_0) $\cap$ (field 4 τ_1) $\cap \ldots \cap$ (field $(4n-4)$ τ_{n-1}).

5.1 Untyped Representation and Discrimination

First, I describe the conventional (untyped) datatype representation and tag discrimination.

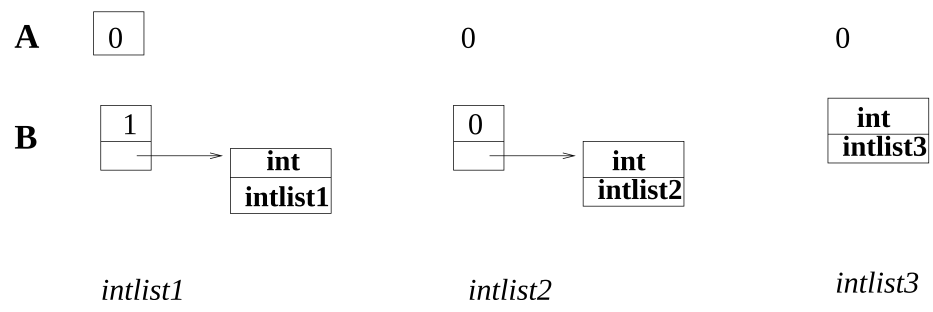

Figure 5.1: Data Representation of *Intlist*

5.1.1 Datatype Representation

There are many different data representations that a compiler can choose from for user-defined datatypes such as *intlist*:

$$\text{datatype } intlist = nil \ \mid cons \text{ of int} * intlist,$$

including the three representations shown in Figure 5.1:

intlist$_1$ The most straightforward representation is to tag each constructor with a small integer, that is, to represent the constructor as a record, with the tag in the first field. *Nil* is tagged 0, and *cons* is tagged 1. *Nil* does not carry any value, thus it is a record with only one field—the tag 0. *Cons* is a two-field record, the first field being the tag 1 and the second field being the carried value (a record of an integer and a list).

intlist$_2$ Assuming that small integers can be distinguished from pointers, the compiler can use small integers to represent constant data constructors, and use pointers for value-carrying constructors: *nil* is represented as an unboxed integer 0, instead of a one-field record of 0; *cons* is a two-field record of tag 0 and the carried value.

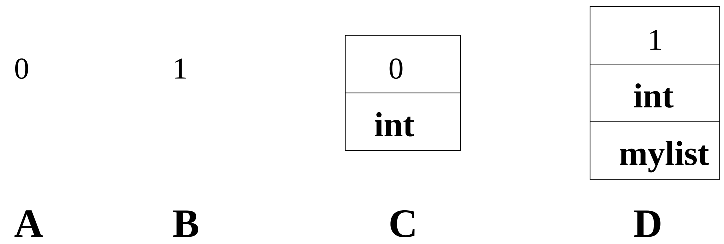

Figure 5.2: Data Representation of *mylist*

intlist$_3$ A datatype with only one value-carrying constructor can be optimized further. If the value-carrying constructor carries an always boxed value, it need not be tagged, since only this constructor is represented as a pointer, and all other constructors are represented as small integers. *Cons* carries a record, which is always boxed, thus its tag can be removed and *cons* has the same representation as the record it carries.

Assuming small integers can be distinguished from pointers, for the user-defined datatype $mylist$ = **A** | **B** | **C** of int | **D** of int $*$ *mylist*, constant constructors A and B can be represented as unboxed integers 0 and 1 respectively. Value-carrying constructors C and D can be tagged with 0 and 1 respectively. Here I choose a flattened representation for D (shown in Figure 5.2). It is a three-field record, whose first field is tag 1 and the rest two fields are the carried value. This representation saves an extra indirection compared with boxing the carried value.

5.1.2 Sum Value Discrimination

To determine which case a sum value belongs to, we need to check the sum value and its tag (if it is tagged).

Discriminating $\textbf{\textit{intlist}}_1$

To check whether a value v of $intlist_1$ is *nil* or *cons*, we need to check whether the tag of v is 0 or 1, since *nil* is tagged 0 and *cons* is tagged 1. Thus we could fetch the first field of v to some register t, and then test t. The following pseudo code implements the discrimination:

$l: \quad v : intlist_1$	
$\quad t = v[0]$	% fetch the tag of v to t
$\quad$ cmp $t, 0$	% test whether t is 0
$\quad$ beq l_{nil}	% if t is 0, goto l_{nil}
$l_{cons} : \ldots$	% t is 1, and v is *cons*
$l_{nil} : \ldots$	% v is *nil*

The type-checker can figure out that the tag t is an integer. A stronger type system with singleton and union types can specify that t is either 0 or 1. Then t is compared with 0, followed by the branch-if-equal instruction. The type-checker can refine the type of t according to whether the control transfers: if yes, $t = 0$; otherwise, $t \neq 0$.

However, this refinement is *not* what we want. We want to refine the type of v, so that we can access v's fields. When $t \neq 0$, v is *cons*. The type-checker should be able to refind v's type and indicate that v is a two-field record, such that reading v's second field (the carried value) is valid. When $t = 0$, the type-checker should inhibit reading v's second field.

The problem here is: we want to refine v's type based on the result of checking t. The connection between v and t is lost after tag-fetching.

Discriminating $\textit{intlist}_3$

Checking whether a value v of $intlist_3$ is *nil* or *cons* does not need tag-checking, because neither case is tagged. It can be done by checking whether v is a small integer or a pointer. A small integer means *nil*, and a pointer means *cons*. Normally a compiler sets up a boundary between small integers and pointers. No pointers are less than the boundary. Therefore, if v is less than the boundary, it is *nil*, otherwise it is *cons*.

The following code tests whether value v of type $intlist_3$ is *nil* or *cons* by comparing v with 256, the boundary between small integers and pointers used in the FPCC-ML compiler. The FPCC-ML compiler arranges that no pointer points to the first 256 space in the memory.

l :	
$\quad v : intlist_3$	
$\quad$ cmp $v, 256$	% compare v with the boundary
$\quad$ bge l_{nil}	% if $v \geq 256$ goto l_{cons}
$l_{nil} : \ldots$	% v is *nil*
$l_{cons} : \ldots$	% v is a pointer, and v is *cons*

Of course in this discrimination case we could also directly check whether v is 0 (*nil*), since there are only two cases here.

Discriminating *mylist*

Checking which case a value v of *mylist* is would need both checks in the previous two examples. First, we check whether v is a small integer to differentiate the unboxed cases A and B and the boxed cases C and D. If v is unboxed, we further

l :	
v : *mylist*	
cmp $v, 256$	% compare v and the boundary
bge l_{CD}	% if v is greater than or equal to the boundary, goto l_{CD}
l_{AB} :	% v is a small integer, and v is either A or B
cmp $v, 0$	% test whether v is 0
beq l_A	% if v is 0, goto l_A
$l_B : \ldots$	% v is 1 (case B)
$l_A : \ldots$	% v is 0 (case A)
l_{CD} :	% v is a pointer, and v is either C or D
$t = v[0]$	% fetch the tag of v to t
cmp $t, 0$	% test whether t is 0
beq l_C	% if t is 0, goto l_C
$l_D : \ldots$	% t is 1 and v is case D
$l_C : \ldots$	% t is 0 and v is case C

Figure 5.3: Untyped Discrimination of *Mylist*

test whether v is 0 (A) or 1 (B). If v is boxed, we then need to fetch the tag of v and check the tag. Tag 0 means C and tag 1 means D. Figure 5.1.2 illustrates the discrimination process.

First, block l tests whether v is a pointer by comparing v with the boundary 256. If $v \geq 256$, v is either C or D, and the control goes to block l_{CD}. Otherwise v is either A or B, and the control falls through to block l_{AB}.

Block l_{AB} further tests whether v is A or B. Since A is represented as unboxed integer 0 and B is unboxed integer 1, v is case A if $v = 0$, otherwise v is B. The first instruction compares v with 0, and the second branch instruction goes to block l_A if $v = 0$, otherwise it goes to block l_B.

In block l_{CD}, we know v is boxed and tagged, either C or D. Value v is tagged 0 if it is C, and it is tagged 1 if it is D. The first instruction fetches the tag of v to t, then the second instruction compares the tag t with 0. If $t = 0$, we know v is case C, and the control goes to block l_C by the conditional branch instruction beq. Otherwise we know v is case D, and the control falls through to block l_D.

In block l_C, we know that v is a two-field record, and the second field is an integer. In block l_D, we know that v is a three-field record, and it is safe to access the second and the third fields. Therefore, tag discrimination should be done correctly in order to safely access the memory: if in the comparison instruction of block l_{CD} 0 is changed to 1, fetching the third field in block l_D would violate memory safety. The type-checker should detect such errors. In order to do this, the checker should refine the type of the sum value according to the result of tag-checking. Instructions cmp $t, 0$ and cmp $t, 1$ would result in different refined types of the sum value v in block l_D, thus the checker could inhibit the unsafe memory access.

5.2 Type-checking Tag Discrimination

Consider the example that discriminates *mylist*. Tag discrimination for the sum value v is done by first fetching the tag into t, then comparing t with integer 0, and then doing a conditional branch on the comparison result. The type-checker should use the result of comparing t with 0 to refine v's type. In block l_C, v should have type $[\mathrm{int}_= 0, \mathrm{int}]$. In block l_D, v should have type $[\mathrm{int}_= 1, \mathrm{int}, \mathit{intlist}]$, so that fetching the third field of v is valid. The reader may remember the problem of refining v's type according to the result of checking t (in Section 5.1.2).

Previous TALs used either macro instructions or dependent types to deal with this problem.

5.2.1 Solution 1: Macro Instructions

Most TALs use macro instructions that does fetching tags and comparison and conditional branch all in one instruction. For example, in TALT, tag discrimination is implemented by instruction **cmpjcc** [21]. Instruction cmpjcc o_1, o_2, k, o_3 is actually a macro for two instructions: cmp o_1, o_2; jcc k, o_3. The first instruction compares o_1 with o_2, and the second instruction branches to o_3 if condition codes set by the comparison satisfy condition k. Normally in tag discrimination, o_1 is of format $M(s, r, 0)$, meaning the value at memory address $r + 0$ (of size s)—the tag of r if r is a tagged sum value, and operand o_2 is an integer. For example, given that v is a tagged sum value, instruction cmpjcc $M(W, r, 0), 0, eq, o$ means comparing the tag of r with 0, and jumping to o if they are equal.

TALT uses singleton integers and dataflow analysis to type-check cmpjcc instructions. The memory mode of Intel x86 allows the tag to be kept in the memory, instead of fetched to a register. Format $M(s, r, 0)$ connects the tag of r and sum value r itself. When checking the cmpjcc $M(s, r, 0), i, k, o$ instruction, the checker rewrites r's type to $\tau_1 \cup \tau_2$, where $r[0]\ k\ i$ holds in τ_1 but not in τ_2. The checker refines r's type to τ_1 in target o and τ_2 in the fall-through case.

5.2.2 Solution 2: Dependent Types

Dependently Typed Assembly Language (DTAL) uses a restricted form of dependent types all the way through the source language DML (ML + dependent types) to the target assembly language DTAL. The programmer gives integer specifications (for example, the length of an array) in terms of types, and the compiler translates these specifications to low-level dependent types and integer constraints. The type-checker decides whether the specifications are satisfied with the help of a constraint solver. Dependent types make it possible to specify and enforce more invariants such as that the length function returns n given an array of length n.

DTAL uses existential types and singleton types to connect a value and its tag. Type $\mu\alpha.(\exists a : nat_n.int(a) * choose(a, unit, int * \alpha))$ in DTAL could represent the sum type $intlist_1$, both constructors tagged. Index variable a states the value of the tag. The sum value is a two-field record, with the first field being the tag and the second field being the carried value. The tag has value a, as specified by its singleton integer type $int(a)$. The second field has type $choose(a, unit, int * \alpha)$, which means $unit$ if a is 0 and $int * \alpha$ if a is 1. The tag t fetched from a value v of type $\tau = \mu\alpha.(\exists a : nat_n.int(a) * choose(a, unit, int * \alpha))$ would have type $int(a)$. Branching on the tag t would add one more constraint $a\ k\ 0$ to the constraint set of the branch target, and $a\ k'\ 0$ to the fall-through case (k is the condition used in the branch, and k' is the opposite of k). With this new constraint about a, the checker could use subtyping to refine type $choose(a, unit, int * \tau)$ to either $unit$ or $int * \tau$.

The constraint solver allows DTAL to express and check many integer invariants, but it makes type-checking complicated and even hard to reason about.

5.3 LTAL Approach

LTAL adopts the dependent type approach of DTAL. It uses existential types to connect sum values and their tags and uses singleton integer types to track the implicit data flow to refine types. But the LTAL type-checker is simpler than DTAL in the sense that it totally avoids a constraint solver. Another difference (also one important feature of LTAL) is that LTAL has no macros for datatype discrimination.

Sparc does not have a memory addressing mode, or branch on registers. Tag-fetching, tag-checking and branch should be implemented by individual instructions. LTAL has explicit instructions for each of them.

5.3.1 Sum Type Representation

First, I will explain how sum types are represented in LTAL.

LTAL has a set of low-level constructors for describing datatypes. Singleton integer literals are used for constant constructors represented as integers and for tags. Union types are used for combining the cases of sum types. Recursive types are used for datatypes with recursive definitions (the datatype appears in the definition of itself).

The representation of $\mathit{intlist}_1$ can be expressed as LTAL type:

$$\mathit{intlist}_1 = \mu\alpha : \Omega.[\mathrm{int}_= 0] \cup [\mathrm{int}_= 1, [\mathrm{int}, \alpha]]$$

Intlist is a recursive type, since the *cons* case contains *intlist* in the definition. That is what the outmost constructor μ is for. The body of the recursive type is a union type, meaning a value of $\mathit{intlist}_1$ is either *nil* or *cons*. *Nil* is represented as $[\mathrm{int}_= 0]$, a record with only one field (the tag). The tag should be 0 as indicated by singleton type $\mathrm{int}_= 0$. *Cons* is represented as $[\mathrm{int}_= 1, [\mathrm{int}, \alpha]]$, a record with two fields: the tag and the carried value. Singleton type $\mathrm{int}_= 1$ means the tag is 1 and record type $[\mathrm{int}, \alpha]$ means *cons* carries a record of an integer (head) and another $\mathit{intlist}_1$ (tail).

$\mathit{Intlist}_2$ can be expressed in LTAL as

$$intlist_2 = \mu\alpha : \Omega.\mathrm{int}_= 0 \cup [\mathrm{int}_= 0, [\mathrm{int}, \alpha]]$$

Nil is represented as $\mathrm{int}_= 0$, an unboxed integer 0. *Cons* has two fields: tag 0 and the carried value.

Intlist$_3$ can be expressed in LTAL as

$$intlist_3 = \mu\alpha : \Omega.\mathrm{int}_= 0 \cup [\mathrm{int}, \alpha]$$

Nil is represented as 0. *Cons* is an untagged record, which has the same representation as the carried value.

Mylist can be expressed in LTAL as

$$mylist = \mu\alpha : \Omega.\mathrm{int}_= 0 \cup \mathrm{int}_= 1 \cup [\mathrm{int}_= 0, \mathrm{int}] \cup [\mathrm{int}_= 1, \mathrm{int}, \alpha].$$

Four disjuncts in the body of the recursive type indicate the four cases: either constant constructor A represented as unboxed integer 0 ($\mathrm{int}_= 0$), or constant constructor B represented as unboxed integer 1 ($\mathrm{int}_= 1$), or constructor C with tag 0 ($[\mathrm{int}_= 0, \mathrm{int}]$), or constructor D with tag 1 ($[\mathrm{int}_= 1, \mathrm{int}, \alpha]$).

With these low-level type constructors, the compiler can explain to the proving system the data representation it uses. Therefore, the compiler writer has much flexibility to choose the representation he wants, even the freedom to choose different representations for different datatypes, as long as the representation for each datatype is consistent.

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

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

Figure 5.4: Typing Rules of cinj1 and cfold

5.3.2 Creating Sum Values

An empty list of $intlist_1$ could be built from a 1-field record $v_0 = [0]$:

	LTAL	Sparc
	$v_0 : [\text{int}_= 0]$	
(1)	$v_1 = \text{cinj1 } ([\text{int}_= 0] \cup [\text{int}_= 1, [\text{int}, intlist_1]])(v_0)$	
	$v_1 : [\text{int}_= 0] \cup [\text{int}_= 1, [\text{int}, intlist_1]]$	
(2)	$v_2 = \text{cfold}[intlist_1](v_1)$	
	$v_2 : intlist_1$	

The only difference between v_0, v_1 and v_2 is coercions. 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 $intlist_1$. It simply checks: (1) whether the type of v_0 is the first part of union type $[\text{int}_= 0] \cup [\text{int}_= 1, [\text{int}, intlist_1]]$ (by the rule of coercion cinj1); (2) whether the type of v_1 is exactly the same as $intlist_1$ with type variable α replaced by $intlist_1$ (coercion cfold).

The typing rules of coercions cinj1 and cfold are shown in Figure 5.4.

The cinj1 rule can be easily proved by the TML subtyping rule in $\tau_1 \subset \tau_1 \cup \tau_2$. The cfold rule could be proved by the subtyping rule *fold* in TML.

The following two LTAL instructions create an empty list of $intlist_3$ by coercing integer 0 to be of type $intlist_3$.

$v_1 = \text{cinj1}[\text{int}_= 0 \cup [\text{int}, intlist_3]](0)$ $\quad v_1 : \text{int}_= 0 \cup [\text{int}, intlist_3]$ $v_2 = \text{cfold}[intlist_3](v_1)$ $\quad v_2 : intlist_3$	**mov** $0, d_1$

Value 0 has type $\text{int}_= 0$. It is coerced by cinj1 to v_1 of type $[\text{int}_= 0 \cup [\text{int}, intlist_3]]$, then v_1 is coerced to v_2 of type $intlist_3$ by cfold. Again, v_1 and v_2 are assigned the same register. No Sparc instruction is needed for coercion from v_1 to v_2.

5.3.3 Eliminating Sum Values

In this section, I show how the previous three discrimination examples are implemented and type-checked in LTAL.

Discriminating *Intlist*$_1$

From Section 5.1.2 we know discriminating a value of $intlist_1$ is done by tag-checking. Section 5.3.1 shows the representation of $intlist_1$:

$$intlist_1 = \mu\alpha : \Omega.[\text{int}_= 0] \cup [\text{int}_= 1, [\text{int}, \alpha]].$$

The discrimination is completed by the following steps in LTAL:

1. Before tag-checking, a value v of type $intlist_1$ should be unfolded to another value v_0 of non-recursive type $[\text{int}_= 0] \cup [\text{int}_= 1, [\text{int}, intlist_1]]$.

2. From the representation of $intlist_1$, we know that v_0 is tagged, but do not know the tag yet. LTAL uses an existential type $\exists\alpha : \Omega_N.\text{field } 0\ (\text{int}_=\ \alpha)$ to represent this. Type variable α represents the value of the tag. It is of kind Ω_N.

   Also v_0 has the union type $[\text{int}_=\ 0] \cup [\text{int}_=\ 1, [\text{int}, \alpha]]$, therefore, the sum value has type $\exists\alpha : \Omega_N.\text{field } 0\ (\text{int}_=\ \alpha) \cap ([\text{int}_=\ 0] \cup [\text{int}_=\ 1, [\text{int}, intlist_1]])$. LTAL uses coercion **csum2hastag** to coerce v_0 to value v_1 and make v_1 have type $\exists\alpha : \Omega_N.\text{field } 0\ (\text{int}_=\ \alpha) \cap ([\text{int}_=\ 0] \cup [\text{int}_=\ 1, [\text{int}, intlist_1]])$[1].

3. v_1 has an existential type. Next v_1 is opened to v_2 of type field 0 $(\text{int}_=\ \beta) \cap ([\text{int}_=\ 0] \cup [\text{int}_=\ 1, [\text{int}, intlist_1]])$, with the new bound type variable β addressing the tag value.

4. Now v_2's tag is explicit in its type. We fetch v_2's tag to t, and assign type $\text{int}_=\ \beta$ to t. Notice that the same type variable β appears both in the type of the tag t and in the type of the sum value v_2. It indicates the connection between t and v_2, because only by fetching the first field of v_2 can we get a value whose type has β in it.

5. Next is tag-checking, comparing the tag t with 0.

6. If $t = 0$, v_2 is *nil*, and the control jumps to some label l_{nil}, with v_2 coerced to a new value v_{nil} of type $[\text{int}_=\ 0]$. Otherwise v_2 is *cons*, and the control fall-through to l_{cons} with v_2 coerced to v_{cons} of type $[\text{int}_=\ 1, [\text{int}, intlist_1]]$.

The above steps 1-6 are implemented by LTAL instructions (1)-(6) in Figure 5.5 respectively. The corresponding Sparc instructions are on the right side of the table.

[1]Type variable α does not appear in the union type, so the $\exists$ quantifier can be lifted to the outmost level.

	LTAL	Sparc
	$l:$	$l:$
(1)	$v_0 = \text{cunfold}(v)$	
(2)	$v_1 = \text{csum2hastag}(v_0)$	
(3)	$(\beta, v_2) = \text{open}(v_1)$	
(4)	$t = v_2[0]$	**ld** $[r], r_t$
(5)	$\text{cmpr}(t, 0)$	**cmp** $r_t, 0$
(6)	$\text{iftag}(=)\{v_2\}$ then (v_{nil}, l_{nil}) else (v_{cons}, l_{cons})	**beq** l_{nil}
	$l_{cons} : \ldots$	$l_{cons} : \ldots$
	$l_{nil} : \ldots$	$l_{nil} : \ldots$

Figure 5.5: LTAL Instruction Sequence for Discriminating $Intlist_1$

Variables v_0, v_1, v_2, v_{nil} and v_{cons} are all coerced values from v. They are assigned the same register as v. Therefore, instructions (1)-(3) map to no Sparc instructions, and no Sparc instructions are needed to move v_2 to v_{nil} or v_{cons} in instruction (6). Instruction iftag makes explicit the sum value v_2 whose type will be refined.

Figure 5.6 shows typing rules for coercions cunfold and csum2hastag, and instructions open and iftag.

The cunfold rule can be proved by subtyping rule $\mu\alpha : \kappa.\tau \subset \tau[\mu\alpha : \kappa.\tau/\alpha]$.

Proving the csum2hastag rule can be done as follows:

1. field 0 $\text{int}_= t_i \subset \exists\alpha : \Omega_N.$field 0 $\text{int}_= \alpha$ by existential-introduction rule.

2. field 0 $\text{int}_= t_i \cap \tau'_i \subset$ field 0 $\text{int}_= t_i \subset \exists\alpha : \Omega_N.$field 0 $\text{int}_= \alpha$ for any τ'_i, by subtyping rule $\tau \cap \tau' \subset \tau$, 1 and the transitivity of the subtyping relation. Thus for all $1 \le i \le n$, $\tau_i \subset \exists\alpha : \Omega_N.$field 0 $\text{int}_= \alpha$.

3. $\tau_1 \cup \tau_2 \cup \ldots \cup \tau_n \subset \exists\alpha : \Omega_N.$field 0 $\text{int}_= \alpha$, by 2 and subtyping rule

$$\frac{\tau_1 \subset \tau \quad \tau_2 \subset \tau}{\tau_1 \cup \tau_2 \subset \tau}$$

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

$$\frac{\begin{array}{l}\alpha \text{ is a fresh type variable}\\ \tau = \tau_1 \cup \tau_2 \cup \ldots \cup \tau_n\\ \tau_i = \text{field } 0 \text{ int}_= t_i \cap \tau_i' \;\; \forall\, 1 \leq i \leq n\end{array}}{\rho; LRT \vdash_c \tau \overset{\text{csum2hastag}}{\hookrightarrow} \exists\alpha : \Omega_N.(\text{field } 0 \text{ int}_= \alpha) \cap \tau}$$

$$\frac{LRT; \rho; \Phi \vdash v : \exists\alpha : \kappa.\tau}{LRT \vdash (\rho; \hbar; \Phi; cc)\ \{(\alpha, v_0) = \text{open}(v)\}\ (\rho, \alpha : \kappa; \hbar; \Phi, v_0 : \tau; cc)}$$

$$\frac{\begin{array}{l}cc = \text{cc\_cmp}(\tau_\alpha, \bar{i}) \qquad LRT; \rho; \Phi \vdash v : (\text{field } 0 \text{ int}_= \tau_\alpha) \cap \tau\\ \tau = \tau_1 \cup \tau_2 \cup \ldots \cup \tau_n \qquad \tau_i = \text{field } 0 \text{ int}_= t_i \cap \tau_i' \;\; \forall\, 1 \leq i \leq n\\ \tau_t = \cup\{\tau_j | 1 \leq j \leq n, t_j\ \pi\ i \neq \text{false}\}\\ \quad \tau_f = \cup\{\tau_j | 1 \leq j \leq n, t_j\ \pi\ i \neq \text{true}\}\\ LRT; \rho; \hbar; \Phi, v_1 : (\text{field } 0 \text{ int}_= \tau_\alpha) \cap \tau_t; cc \vdash_\ell l_1\\ LRT; \rho; \hbar; \Phi, v_2 : (\text{field } 0 \text{ int}_= \tau_\alpha) \cap \tau_f; cc \vdash_\ell l_2\end{array}}{LRT \vdash (\rho; \hbar; \Phi; cc)\ \{\text{iftag}(\pi)\{v\} \text{ then } (v_1, l_1) \text{ else } (v_2, l_2)\}\ (\rho; \hbar; \Phi; cc)}$$

Figure 5.6: Typing Rules for cunfold, csum2hastag, open and iftag

4. $\tau_1 \cup \tau_2 \cup \ldots \cup \tau_n \subset (\exists \alpha : \Omega_N.\text{field } 0 \text{ int}_= \alpha) \cap (\tau_1 \cup \tau_2 \cup \ldots \cup \tau_n)$, by 3 and subtyping rules

$$\frac{}{\tau \subset \tau} \qquad \frac{\tau \subset \tau_1 \quad \tau \subset \tau_2}{\tau \subset \tau_1 \cap \tau_2}$$

5. $\tau_1 \cup \tau_2 \cup \ldots \cup \tau_n \subset \exists \alpha : \Omega_N.(\text{field } 0 \text{ int}_= \alpha \cap (\tau_1 \cup \tau_2 \cup \ldots \cup \tau_n))$, by (4) and the fact that α is a fresh type variable and α does not appear in $\tau_1 \cup \tau_2 \ldots \cup \tau_n$.

The open rule is proved by the corresponding subtyping rule in TML.

Instruction iftag$(\pi)\{v\}$ then (v_1, l_1) else (v_2, l_2) does tag discrimination. It should follow a cmpr instruction that compares the tag of v with an integer. The type-checker checks in iftag instruction that: cc is $\mathbf{cc\_cmp}(\tau_\alpha, \bar{i})$, the result of comparing the tag (of type int$_=$ τ_α) with i (of type int$_=$ $\bar{i}$), v is of type (field 0 τ_α)$\cap\tau$; and it refines the types of v_1 and v_2 to (field 0 τ_α)$\cap\tau_t$ and (field 0 τ_α)$\cap\tau_f$, respectively. This refinement rules out disjuncts by the result of comparing the tag with an integer. Type τ_t is a refinement of τ with no disjuncts in τ that conflict with the condition codes. Type τ_f is the opposite case. For example, if the condition codes are set by comparing v's tag with 0, and π is "equal", τ_t would be the only disjunct with tag 0 (or bottom type in case there is no disjunct with tag 0), and τ_f would be the union of the rest of the disjuncts. A constraint solver as in DTAL [69] is overkill for this purpose.

Figure 5.7 shows the types of each variable during type-checking. Instruction (1) unfolds v to v_0. v_0's type is created by replacing the type variable α in the body of the recursive type of v with $intlist_1$ (cunfold rule). Instruction (2) coerces v_0 to v_1 and makes v_1's tag explicit in its type, indicated by type variable α' (csum2hastag rule). Instruction (3) opens v_1 and binds a new type variable β for the tag. Every

	$l : v : \mu\alpha : \Omega.[\text{int}_= 0] \cup [\text{int}_= 1, [\text{int}, \alpha]]$
(1)	$v_0 = \text{cunfold}(v)$
	$\quad v_0 : [\text{int}_= 0] \cup [\text{int}_= 1, [\text{int}, \mathit{intlist}_1]]$
(2)	$v_1 = \text{csum2hastag}(v_0)$
	$\quad v_1 : \exists\alpha' : \Omega_N.\text{field } 0\ (\text{int}_= \alpha') \cap ([\text{int}_= 0] \cup [\text{int}_= 1, [\text{int}, \mathit{intlist}_1]])$
(3)	$(\beta, v_2) = \text{open}(v_1)$
	$\quad v_2 : \text{field } 0\ (\text{int}_= \beta) \cap ([\text{int}_= 0] \cup [\text{int}_= 1, [\text{int}, \mathit{intlist}_1]])$
(4)	$t = v_2[0]$
	$\quad t : \text{int}_= \beta$
(5)	$\text{cmpr}(t, 0)$
	$\quad$ condition codes: $\text{cc\_cmp}(\beta, \overline{0})$
(6)	$\text{iftag}(=)\{v_2\}$ then (v_{nil}, l_{nil}) else (v_{cons}, l_{cons})
	$l_{cons} : v_{cons} : \text{field } 0\ \text{int}_= \beta \cap [\text{int}_= 1, [\text{int}, \mathit{intlist}_1]]$
	$\quad \ldots$
	$l_{nil} : v_{nil} : \text{field } 0\ \text{int}_= \beta \cap [\text{int}_= 0]$
	$\quad \ldots$

Figure 5.7: Type-checking Discrimination of *Intlist*$_1$

appearance of α' in v_1's type is replaced by β (open rule). Then instruction (4) fetches the tag to t and gives t type $\text{int}_= \beta$, since v_2 has type field 0 $\text{int}_= \beta$. Then instruction (5) compares the tag t and 0 and sets condition codes $\text{cc\_cmp}(\beta, \overline{0})$. Instruction (6) rebinds v_2 to v_{nil} in l_{nil} where condition codes are satisfied, and refines v_{nil}'s type to be (field 0 $\text{int}_= \beta)\cap[\text{int}_= \overline{0}]$ since the other disjunct $[\text{int}_= \overline{1}, [\text{int}, \mathit{intlist}_1]]$ conflicts with the fact that $\beta = 0$. Similarly, in l_{cons}, v_2 is coerced to v_{cons} of type (field 0 $\text{int}_= \beta) \cap [\text{int}_= \overline{1}, [\text{int}, \mathit{intlist}_1]]$.

The connection between a tagged value (such as v_2) and its tag (such as t) is established by existential types and type variables, since every time we open a variable of type $\exists\alpha : \Omega_N.\text{field } 0\ \text{int}_= \alpha \cap \tau$ and assign it to some variable v, we get a fresh type variable α', and only v's type field 0 $\text{int}_= \alpha' \cap \tau$ contains the new type variable α', and only by instruction $\text{load}(v, 0)$ can we get a variable of type $\text{int}_= \alpha'$.

LTAL	Sparc
$v' = \text{cunfold}(v)$	
$(\alpha, v'') = \text{testbox}(v')$	$\mathbf{cmp}\ r, 256$
$\text{ifboxed}\{v''\}$ then (v_{cons}, l_{cons})	$\mathbf{bge}\ l_{cons}$
else (v_{nil}, l_{nil})	
$l_{nil} : \ldots$	$l_{nil} : \ldots$
$l_{cons} : \ldots$	$l_{cons} : \ldots$

Figure 5.8: LTAL Instruction Sequence for Discriminating *Intlist*$_3$

Discriminating ***Intlist***$_3$

Tag-checking in discriminating *Intlist*$_3$ is unnecessary, since neither constructor is tagged. We only need to test whether a value is a pointer or not. A pointer means *cons* and a small integer means *nil*.

Testing whether value v of type *intlist*$_3$ is boxed could be done by comparing v with the boundary between pointers and small integers. LTAL instruction **testbox** does this comparison. Then a condition branch instruction ifboxed consumes the condition codes set by testbox. From the assumption that no pointers point to the first 256 space in the memory, if $v \geq 256$ v is *cons* and we could rebind v to v_{cons} of type $[\text{int}, \mathit{Intlist}_3]$. Otherwise, v is *nil* and we could rebind v to v_{nil} of type $\text{int}_= 0$. As in the branch instruction iftag, v_{nil} and v_{cons} are assigned the same register as v. We don't need move instructions to move v to v_{nil} or v_{cons}.

The LTAL instructions for discriminating *intlist*$_3$ are shown in Figure 5.8.

Like iftag, instruction ifboxed indicates the value whose type will be refined. Since the type refinement is based on the result of testbox, the type-checker should make sure the value used in ifbox is the same as the value in testbox. Otherwise, there could be unsoundness. For example, in the following instruction sequence, v_c is *cons* and v_n is *nil*. Instruction testbox tests whether v_c is boxed. Instruction ifboxed consumes the condition codes, but refines v_n. The condition codes are satisfied, thus the control goes to l_{cons} and v_{cons} coerced from v_n is treated as if it is *cons*, which would allow illegal access of two fields of v_n (in fact v_n is an unboxed integer 0).

$$\begin{array}{l} \text{testbox}(v_c) \\ \text{ifboxed}\{v_n\} \text{ then } (v_{cons}, l_{cons}) \text{ else } (v_{nil}, l_{nil}) \end{array}$$

To prevent this unsoundness instruction testbox(v) binds a new type variable α and rebinds v to v'. The meaning of α is the actual value of v. Again, v' is just an alias of v. They are in the same register. If v has type τ, then after instruction $(\alpha, v') = \text{testbox}(v)$, v' has type $\text{int}_= \alpha \cap \tau$, and the condition codes would be cc_testbox(α), which is the same as $\text{cc\_cmp}(\alpha, \overline{256})$. Type variable α ties the value v' and the condition codes from testbox(v). The checker will check in ifbox that α both appears in the type of the sum value and the condition codes, which guarantees the value we refine is the value we test, since α introduced by testbox only appears in the type of v' and the condition codes.

Instruction ifboxed$\{v\}$ then (v_1, l_1) else (v_2, l_2) consumes the condition codes set by testbox, and rebinds v of type $\text{int}_= \overline{0} \cup \text{int}_= \overline{1} \cup \ldots \cup \text{int}_= \overline{n-1} \cup \tau$ to v_1 of type τ when v is boxed, or rebinds v to v_2 of type range$[0, n]$ when v is unboxed.

Typing rules for instructions testbox and ifboxed are in Figure 5.9.

The testbox rule could be considered as a syntactic sugar for the following steps:

$$\begin{array}{ll} (1) & \text{coerce } v_0 \text{ to } v_0' \text{ of type } \exists\beta.\text{int}_= \beta \cap \tau \\ (2) & (\alpha, v_1) = \text{open}(v_0') \\ (3) & \text{cmpr}(v_1, 256) \end{array}$$

$$\frac{LRT; \rho; \Phi \vdash v : \tau}{LRT \vdash (\rho; \hbar; \Phi; cc)\ \{(\alpha, v') = \text{testbox}(v)\}\ (\rho; \hbar; \Phi, v' : \text{int}_= \alpha \cap \tau; \text{cc\_testbox}\ \alpha)}$$

$$\frac{\begin{array}{l} cc = \text{cc\_testbox}\ \tau_\alpha \\ LRT; \rho; \Phi \vdash v : \text{int}_= \tau_\alpha \cap (\text{int}_= \overline{0} \cup \text{int}_= \overline{1} \cup \ldots \cup \text{int}_= \overline{n-1} \cup \tau') \\ \tau' = \tau_1 \cup \tau_2 \cup \ldots \cup \tau_m \qquad \tau_i = (\text{field } 0 \text{ int}_= t_i) \cap \tau_i' \ \ \forall 1 \le i \le m \\ LRT; \rho; \hbar; \Phi, v_1 : \tau'; cc \vdash_\ell l_1 \qquad LRT; \rho; \hbar; \Phi, v_2 : \text{range}[0, n]; cc \vdash_\ell l_2 \end{array}}{LRT \vdash (\rho; \hbar; \Phi; cc)\ \{\text{ifboxed}\{v\} \text{ then } (v_1, l_1) \text{ else } (v_2, l_2)\}\ (\rho; \hbar; \Phi; cc)}$$

Figure 5.9: Typing Rules for Testbox and Ifboxed

In the ifboxed rule, from the condition codes cc_testbox(τ_α), in the true branch we could reason that $\tau_\alpha \ge 256$, and all disjuncts $\text{int}_= \overline{0}$, $\text{int}_= \overline{1}$, $\text{int}_= \overline{n-1}$ could be eliminated since they combined with $\text{int}_= \tau_\alpha$ do not satisfy $\tau_\alpha \ge 256$. In the false branch, all disjuncts with field 0 $\text{int}_= t_i$ could be eliminated since field types imply no less than 256.

Types of the variables during type-checking are listed in Figure 5.10. By cunfold rule v is unfolded to v' of type $\text{int}_= 0 \cup [\text{int}, \mathit{intlist}_3]$. Then instruction testbox coerces v' to v'' of type $\text{int}_= \beta \cap (\text{int}_= 0 \cup [\text{int}, \mathit{intlist}_3])$, with new bound type variable β in it, and testbox also sets condition codes cc_cmp(β, 0). Instruction ifboxed coerces v'' to v_{cons} of type $[\text{int}, \mathit{intlist}_3]$ in l_{cons}, and coerces v'' to v_{nil} of type $\text{int}_= 0$ in l_{nil}.

Discriminating *Mylist*

Discriminating *mylist* is done by boxing-checking followed by tag-checking. I will show the details in Appendix B.

$$
\boxed{\begin{array}{l}
l : \\
\quad v : \mu\alpha : \Omega.(\mathrm{int}_{=}\, 0 \cup [\mathrm{int}, \alpha]) \\
v' = \mathrm{cunfold}(v) \\
\quad v' : \mathrm{int}_{=}\, 0 \cup [\mathrm{int}, \mathit{intlist}_3] \\
(\beta, v'') = \mathrm{testbox}(v') \\
\quad v'' : \mathrm{int}_{=}\, \beta \cap (\mathrm{int}_{=}\, 0 \cup [\mathrm{int}, \mathit{intlist}_3]) \\
\quad \text{condition codes: cc\_cmp}(\beta,\, 0) \\
\mathrm{ifboxed}\{v''\} \text{ then } (v_{cons}, l_{cons}) \text{ else } (v_{nil}, l_{nil}) \\
l_{cons} : v_{cons} : [\mathrm{int}, \mathit{intlist}_3] \\
\ldots \\
l_{nil} : v_{nil} : \mathrm{int}_{=}\, 0 \\
\ldots
\end{array}}
$$

Figure 5.10: Type-checking Discrimination of $\mathit{Intlist}_3$

5.3.4 Discussion

No macro instructions for LTAL tag-discrimination means that the tag-checking instructions can be mixed with unrelated ones by a scheduler for optimizations.

For simplicity the implementation uses linear search. LTAL also permits binary search. To do an indexed jump would need extending LTAL, but the underlying semantic model will permit this in a modular way. A jump table T for boxed constructors C and D in the above example could be represented as a record $[l_C, l_D]$ of type (field 0 codeptr(Φ_C)) $\cap$ (field 4 codeptr(Φ_D)), where Φ_C and Φ_D are the preconditions for l_C and l_D respectively. Tag discrimination is implemented by first fetching tag t, then jumping to the entry indexed by t in table T. Checking the jumping instruction might need case analysis to make sure every case is safe.

Chapter 6

Position-independent Code

In this chapter, I explain how the compiler generates position-independent code and how the type-checker checks it. Why is position-independence necessary? Many compilers do not generate position-independent code, but output object code with symbols that stand for addresses in it. Later, right before execution, a link-editor would replace the symbols with absolute addresses, and transform the object code to executables. We need to trust the link-editor that it transforms safe object code to safe executables.

To achieve the goal of FPCC—minimal TCB—a link-editor should be avoided if possible. FPCC-ML compiler arranges that each compilation unit needs no link-editing, and links to others using closures, in the style of SML/NJ [16, §3]. Code generated by the compiler can execute no matter where it is loaded. Thus a link-editor is completely unnecessary, and the proving system does not need to reason about any possible bugs in the link-editor. FPCC describes this position-independence in its safety policy as, "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" [8].

Another advantage of position-independent code is that it can be shared directly. In order to reduce resource usage of many copies of common library routines, shared libraries are often dynamically linked. If the libraries are position-independent, sharing is more convenient [27].

For many programs the performance overhead of position-independent code is negligible. The cost is about 3% in benchmarks measured by Tarditi and Diwan [61].

LTAL uses ideas from SML/NJ to achieve position-independence. But more importantly, it provides a way to type-check position-independent code with no PC-relative addressing, even in the setting of separate compilation. SML/NJ does not have typed position-independent code.

6.1 Untyped Position-independent Code

Position-independent code must use relative addresses instead of absolute ones. Relative jumps are not enough to achieve this. The problem arises when we move a label into a register or store it in memory, which is needed to make a function-pointer or a closure. The value of the label depends on where the code is loaded.

For an architecture with PC-relative addressing, such as Intel x86, absolute addresses can be transformed to relative ones very easily (see Section 6.4.1). But for architectures without a PC-relative addressing mode, such as Sparc, the transformation is more involved.

LTAL adopts the solution used in SML/NJ. The basic idea is to transform absolute addresses to relative ones that are relative to a "base" register. Each function takes an extra "base" parameter, which is assigned the base register. When the function is called, the caller puts an appropriate address into the base register.

The base register points to the beginning of a program when the program starts executing.

6.1.1 Escaping Functions

When a function name appears in a non-application place, we call the function *escaping*. The function name might be held in a register or in the memory, and loaded later at arbitrary points; or it might be used in another compilation unit. We cannot know statically all the call-sites of escaping functions.

What should the base parameter of an escaping function mean? What value should be passed to the base parameter when the function is called?

Let us first look at what happens when an escaping function f is called by an compilation unit C2 other than the unit C1 where f is defined. The caller in C2 needs to give the base register of f a proper value (some address in C1), but it knows no addresses in C1 except the function pointer f, which points to the beginning of the function's machine code in the memory. Thus it is natural to make the escaping function take its own start address as its base parameter, and let the caller pass the function pointer to the base parameter.

This is exactly how LTAL is designed. An escaping function f takes a base parameter, which means the base address of the function. The base address is kept in a reserved register, and does not change during the execution of f. In the body of f, the address of another label g is calculated from the base address and the difference between g and f. When f is called, f is passed as its own *base* parameter.

Figure 6.1 illustrates position-independence transformation.

Three functions f, g and h are defined in the compilation unit 1. In function g label f is moved to register r. Function h calls f.

Position-dependent Code	Position-independent Code	Sparc
(Compilation Unit 1)		
$f : [x, y, \ldots]$	$f : [\text{base}_f(= f), x, y, \ldots]$	$f : (r_b = f)$
$\ldots$	$\ldots$	$\ldots$
$g : [\ldots]$	$g : [\text{base}_g(= g), \ldots]$	$g : (r_b = g)$
$r = f$	$r = \text{base}_g + (f - g)$	**add** $r_b, (f - g), r$
$\ldots$	$\ldots$	$\ldots$
$h : [\ldots]$	$h : [\text{base}_h(= h), \ldots]$	$h : (r_b = h)$
	$\text{base}_f = base_h + (f - h)$	**add** $r_b, (f - h), r_b$
call(f)	call(f)	**ba** f
(Compilation Unit 2)		
$l : [v_f, \ldots]$	$l : [\text{base}_l, v_f, \ldots]$	$l : [r_f, \ldots]$
$\ldots$	$\ldots$	$\ldots$
(extract v'_f from v_f)	(extract v'_f from v_f)	(extract r'_f from r_f)
	$\text{base}_v = v'_f$	**mov** r'_f, r_b
call(v'_f)	call(v'_f)	**ba** r'_f

Figure 6.1: Position-independence Transformation

After position-independence transformation, each function k would take an extra parameter base_k, whose value is the same as label k and is kept in a reserved register r_b. In function g, the address f is computed by adding base_g (in register r_b) and $f - g$ (difference between f and g, a constant calculated by the compiler). Both operands are independent of where the code is loaded. In function h before the control transfers to f, h puts the address f in register r_b. Like in g, f is calculated from the base register (which now keeps the value of h) and the difference between f and h. The transformed code is totally position-independent.

In unit 2, function l calls an external function closure v_f. The linkage to the compilation unit that defines v_f is achieved by making v_f a parameter of l. Closure v_f is a record that contains a function pointer and its free variables. Function l first extracts the function pointer v'_f (kept in register r'_f) from closure v_f, and puts v'_f in the base register r_b, and then calls v'_f.

6.1.2 Known Functions

A function whose name only appears in application position is a *known* function. Its call-sites are all known when the compiler compiles the unit where the function is defined.

As an important optimization, when a "known" function is called, it does not need its own base argument—it can use the base of one of its known callers. This avoids an instruction to assign the base register in local loops and branches. For example, in the following table, function k is a known function and it uses its caller f's base argument. When f falls through to k, no instruction is needed to give the base register a new value since it is already in place.

LTAL	Sparc
$f : [\text{base}_f, \ldots]$	$f :$
$\ldots$	$\ldots$
$k : [\text{base}_f, \ldots]$	$k :$
$\ldots$	$\ldots$

Function k will never be put into a register or in a closure, nor will it be called externally. There is no need to build a closure for k.

LTAL	Sparc
(Compilation Unit 1)	
$f : [\text{base}_f, \ldots]$	$f :$
$\quad \ldots$	$\quad \ldots$
$g : [\text{base}_g, \ldots]$	$g :$
$\quad v = \text{addradd}(\text{base}_g, \text{vdiff}(f, g))$	$\quad$ **add** r_b, C_{fg}, r_v
$\quad \ldots$	$\quad \ldots$
$h : [\text{base}_h, \ldots]$	$h :$
$\quad \text{base}_f = \text{addradd}(\text{base}_h, \text{vdiff}(f, h))$	$\quad$ **add** r_b, C_{fh}, r_b
$\quad \text{call}(f)[base_f, \ldots]$	$\quad$ **ba** f

Figure 6.2: Position-independence Example

6.2 LTAL Instructions

The difference between two labels l_1 and l_2 is actually a number. But LTAL could not simply use a number for $l_1 - l_2$, because the type-checker needs to know the two labels. Thus, LTAL uses a value constructor **vdiff**(l_1, l_2) to express $l_1 - l_2$.

Adding the base register with a vdiff value is performed in LTAL by instruction **addradd**. The LTAL instruction addradd(v_1, v_2) is actually $v_1 + v_2$, and is mapped to the Sparc add instruction. But addradd is specialized for address arithmetic. It has a different typing rule than the normal integer arithmetic instruction.

Figure 6.2 shows the LTAL instructions for unit one in Figure 6.1.

6.3 Type-checking Position-independent Code

6.3.1 Moving Labels to Registers

To type-check position-independent code, LTAL introduces new type constructors **addr** and **diff**. The former gives a type to the base address of a label, and the latter gives a type to the difference between two labels. For example, the base parameter base_f of function f is given type addr(f), and value $\text{vdiff}(l_1, l_2)$ is given type $\text{diff}(l_1, l_2)$.

The typing rule of instruction $v = \text{addradd}(\text{base}_f, \text{vdiff}(g, f))$ checks that variable base_f has type $\text{addr}(f)$ and value $\text{vdiff}(g, f)$ has type $\text{diff}(g, f)$, and assigns type $\mathbf{addr}(g)$ to v.

The typing rules for the vdiff value and the addradd instruction are as follows.

$$\frac{}{LRT; \rho; \Phi \vdash \text{vdiff}(g, f) : \text{diff}(g, f)}$$

$$\frac{LRT; \rho; \Phi \vdash v_1 : \text{addr}(f) \qquad LRT; \rho; \Phi \vdash v_2 : \text{diff}(g, f)}{LRT \vdash (\rho; \hbar; \Phi; cc)\ \{v = \text{addradd}(v_1, v_2)\}\ (\rho; \hbar; \Phi, v : \text{addr}(g); cc)}$$

The semantic model of $\text{addr}(l)$ would be the program start address plus the offset of l. Type $\text{diff}(l_1,\, l_2)$ is modeled as $l_1 - l_2$. The addradd rule could be derived from the semantics of add instruction.

The types of variables during type-checking functions in compilation unit one in Figure 6.2 are shown in Figure 6.3.

6.3.2 Calling External Functions

Position-independence transformation changes the types of function closures. In the definition of a function f, the base parameter has type $\text{addr}(f)$. But when 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 type. LTAL uses existential types: the type of f becomes $\exists\beta : \Omega_N.\text{codeptr}[\vec{\alpha} : \vec{\kappa}](m, cc, [\text{base} : \text{int}_= \beta, \ldots])$. The type variable β stands for the base address of f.

(Compilation Unit 1)

$f : [\text{base}_f : \text{addr}(f), \ldots]$

$\quad \ldots$

$g : [\text{base}_g : \text{addr}(g), \ldots]$

$\quad v = \text{addradd}(\text{base}_g, \text{vdiff}(f, g))$

$\quad\quad v : \text{addr}(f)$

$\quad \ldots$

$h : [\text{base}_h : \text{addr}(h), \ldots]$

$\quad \text{base}_f = \text{addradd}(\text{base}_h, \text{vdiff}(f, h))$

$\quad\quad \text{base}_f : \text{addr}(f)$

$\quad \text{call}(f)[base_f, \ldots]$

Figure 6.3: Type-checking Compilation Unit One

In compilation unit two, the type of the extern function pointer v'_f would be like $\exists\beta : \Omega_N.\text{codeptr}[\vec{\alpha} : \vec{\kappa}](m, cc, [\text{base} : \text{int}_= \beta, \ldots])$.

To make sure that an extern function pointer v'_f itself is passed to its base when v'_f is called, β is "anded" to the body of the existential type, thus v'_f is made to have type $\exists\beta : \Omega_N.(\text{int}_= \beta \cap \text{codeptr}[\vec{\alpha} : \vec{\kappa}](m, cc, [\text{base} : \text{int}_= \beta, \ldots]))$. Before v'_f is called, it is opened to v''_f of type $\text{int}_= \beta' \cap \text{codeptr}[\vec{\alpha} : \vec{\kappa}](m, cc, [\text{base} : \text{int}_= \beta', \ldots])$, with new type variable β'. Then v''_f is coerced to v_1 (the base address of f) and v_2 (the function pointer). v_1 has type $\text{int}_= \beta'$ and v_2 has type $\text{codeptr}[\vec{\alpha} : \vec{\kappa}](m, cc, [\text{base} : \text{int}_= \beta', \ldots])$ (see Figure 2.3 for cproj rules). Then v_1 is passed to the base parameter of v_2. β' is an abstract type variable, and only value v_1 coerced from the function pointer has type β' and can be passed to the function pointer v_2.

The LTAL instructions for compilation unit two in Figure 6.1 are shown in Figure 6.4. Type-checking unit two is shown in Figure 6.5.

LTAL	Sparc
(Compilation Unit 2)	
$l : [\mathrm{base}_l, v_f, \ldots]$	$l :$
$\quad\ldots$	$\quad\ldots$
$\quad$(extract v'_f from v_f)	$\quad$(**ld** $\ldots, r_f$)
$\quad(\beta', v''_f) = \mathrm{open}(v'_f)$	
$\quad v_1 = \mathrm{cproj1}(v''_f)$	
$\quad v_2 = \mathrm{cproj2}(v''_f)$	
$\quad\mathrm{base} = v_1$	$\quad$**mov** r_f, r_b
$\quad\mathrm{call}(v_2)$	$\quad$**jmp** $[r_f]$

Figure 6.4: LTAL Instructions for Compilation Unit Two

6.3.3 Making Closures

To make a function closure, we need a function pointer of a codeptr type without any label in it, because it might be called in other compilation units. Each compilation unit knows only the labels defined within itself. Coercion **addr2code** would coerce v of type $\mathrm{addr}(l)$ to a codeptr type $\mathrm{codeptr}[\vec{\alpha} : \vec{\kappa}](m, cc, base : \mathrm{addr}\ l, \vec{v} : \vec{\tau})$, if there is such a function declaration

$$l[\vec{\alpha} : \vec{\kappa}](m, cc, base : \mathrm{addr} l, \vec{v} : \vec{\tau}) = \ldots$$

Coercion **c2inters**$(\mathrm{cid}, \mathrm{caddr2code})$ transforms type $\mathrm{addr}(l)$ to intersection type $\mathrm{addr}(l) \cap \mathrm{codeptr}[\vec{\alpha} : \vec{\kappa}](m, cc, base : \mathrm{addr}(l), \vec{v} : \vec{\tau})$, where the first conjunct is the result of applying cid to $\mathrm{addr}(l)$ and the second conjunct is applying caddr2code to $\mathrm{addr}(l)$. It is proved by subtyping rule

$$\frac{\tau \subset \tau_1 \qquad \tau \subset \tau_2}{\tau \subset \tau_1 \cap \tau_2}$$

(Compilation Unit 2)	
$l : [\text{base}_l : \text{addr}(l), v_f : \ldots, \ldots]$	$l :$
$\quad \ldots$	$\quad \ldots$
$\quad$ (extract v'_f from v_f)	$\quad$ (**ld** $\ldots, r_f$)
$\quad v'_f : \exists\beta : \Omega_N.(\text{int}_= \beta \cap \text{codeptr}[\vec{\alpha} : \vec{\kappa}](m, cc, [\text{base} : \text{int}_= \beta, \ldots]))$	
$\quad (\beta', v''_f) = \text{open}(v'_f)$	
$\quad v''_f : \text{int}_= \beta' \cap \text{codeptr}[\vec{\alpha} : \vec{\kappa}](m, cc, [\text{base} : \text{int}_= \beta', \ldots])$	
$\quad v_1 = \text{cproj1}(v''_f)$	
$\quad v_1 : \text{int}_= \beta'$	
$\quad v_2 = \text{cproj2}(v''_f)$	
$\quad v_2 : \text{codeptr}[\vec{\alpha} : \vec{\kappa}](m, cc, [\text{base} : \text{int}_= \beta', \ldots])$	
$\quad \text{base} = v_1$	$\quad$ **mov** r_f, r_b
$\quad \text{base} : \text{int}_= \beta'$	
$\quad \text{call}(v_2)$	$\quad$ **jmp** $[r_f]$

Figure 6.5: Type-checking Compilation Unit Two

Coercion **cpack** can be used to hide the addr type of the base parameter in the codeptr type, and create an existential type desired for a function pointer in a closure. This rule could be proved by the introduction rule of existential types.

The typing rules of caddr2code, c2inters and cpack are shown as follows.

$$\frac{l \text{ is declared as: } [\vec{\alpha} : \vec{\kappa}](m, cc, \vec{v} : \vec{\tau}) = \ldots}{\rho; LRT \vdash_c \text{addr}(l) \overset{\text{caddr2code}}{\hookrightarrow} \text{codeptr}[\vec{\alpha} : \vec{\kappa}](m, cc, \vec{v} : \vec{\tau})}$$

$$\frac{\rho; LRT \vdash_c \tau \overset{c_1}{\hookrightarrow} \tau_1 \quad \rho; LRT \vdash_c \tau \overset{c_2}{\hookrightarrow} \tau_2}{\rho; LRT \vdash_c \tau \overset{\text{c2inters}(c_1, c_2)}{\hookrightarrow} \tau_1, \tau_2}$$

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

LTAL	Sparc
$v : \mathrm{int}_= \mathrm{addr}(f)$ $v' = \mathrm{c2inter}(\mathrm{cid},\ \mathrm{caddr2code})(v)$ $v' : \mathrm{int}_= \mathrm{addr}(f) \cap \mathrm{codeptr}[\vec{\alpha} : \vec{\kappa}](m, cc, \mathrm{base} : \mathrm{int}_= \mathrm{addr}(f), \ldots)$ $v_f = \mathrm{pack}[\mathrm{addr}(f),$ $\quad\quad \exists \alpha : \Omega_N.\mathrm{int}_= \alpha \cap \mathrm{codeptr}[\vec{\alpha} : \vec{\kappa}](m, cc, \mathrm{base} : \mathrm{int}_= \alpha, \ldots)](v')$ $v_f : \exists \alpha : \Omega_N.\mathrm{int}_= \alpha \cap \mathrm{codeptr}[\vec{\alpha} : \vec{\kappa}](m, cc, \mathrm{base} : \mathrm{int}_= \alpha, \ldots)$	

Figure 6.6: Create a Closure

Figure 6.6 shows LTAL instructions that create a function pointer that could be stored in the closure of function f from a variable of type $\mathrm{int}_= \mathrm{addr}(f)$.

6.4 Related Work

6.4.1 PC-relative Addressing in TALT

Crary's TALT supports PC-relative addressing [21]. Reasoning about PC-relative addressing would make the type system complicated, because the types of PC-relative addresses depend on physical locations. To avoid this complication, TALT introduces a transformation called *delocalization*. Each code block takes a start address in its code pointer type. Delocalization transforms each PC-relative address to an absolute address calculated from the start address and the relative address. Code is always type-checked after delocalization. Thus there is no need to give typing rules to relative addresses. The following typing rule from the paper explains the idea:

$$\frac{\Delta \vdash \Gamma \quad \Delta; \Psi; \Gamma \vdash [v]a}{\Delta; \Psi \vdash v : \mathrm{code}(a, \Gamma)}$$

It means a code block v has type code(a, Γ) if the precondition Γ is valid $(\Delta \vdash \Gamma)$ and the delocalized block type-checks under Γ $(\Delta; \Psi; \Gamma \vdash [v]a)$. The address a in the code type is the start address. Value v consists of a sequence of instructions. The transformation $[v]a$ applies delocalization to each instruction in v.

Position-independent code can be implemented by PC-relative addressing. Actually the start address in TALT is almost the same as that of LTAL for escaping functions. For example, moving a label l to a variable v could be implemented by relative addressing:

$$
\begin{aligned}
&f : \mathrm{code}(f, \Gamma)\\
&\quad v = \mathrm{pcrel}(l - f)\\
&\quad \ldots\\
&l : \ldots
\end{aligned}
$$

Operator **pcrel**(n) computes $PC + n$. Since it appears in the first instruction of f, when executing this instruction, PC should be the same as f. Thus pcrel$(l - f)$ is equal to the start address of l.

The above sequence with relative addressing would be translated to the following one after delocalization:

$$
\begin{aligned}
&f : \mathrm{code}(f, \Gamma)\\
&\quad v = l\\
&\quad \ldots\\
&l : \ldots
\end{aligned}
$$

The delocalization phase computes $(f + (l - f))$ and puts the result l directly in the code.

Notice that the type-checker checks only the transformed sequence, instead of the original one.

The difference between TALT's PC-relative addressing and LTAL's approach is:

- in TALT, all relative addresses are with respect to PC. But in LTAL, they are relative to the base register r_b since Sparc architecture does not provide PC-relative address mode. The program makes sure the base register has a correct value.

- TALT does not type-check the position-independent code to be executed, but the transformed code after delocalization. Thus the delocalization phase must be trusted. LTAL type-checks position-independent code directly. It does not need a trusted transformation like delocalization.

Chapter 7

Certifying Compiler

In this chapter I will explain the certifying compiler FPCC-ML. It translates core ML programs to Sparc machine code and a machine-dependent LTAL program through type-preserving compilation. Both the machine-code program and the LTAL program will be sent to a checker and the checker "type-checks" the machine-code program with the guidance of the LTAL program.

FPCC-ML is based on the SML/NJ compiler. We reuse SML/NJ's front end, we have built a new "middle end", and we have constructed a typed back end based on the untyped back end of SML/NJ.

7.1 SML/NJ

SML/NJ is an industrial-strength compiler that compiles Standard ML programs to machine code of many architectures, including Sparc, Mips, Intel x86 and Alpha [10, 56]. It was first developed by Appel and MacQueen, and then expanded and modified by many other people.

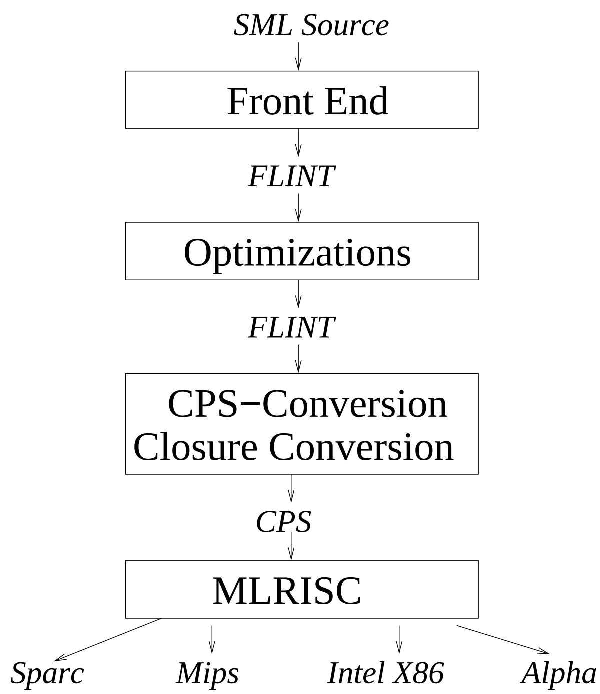

Figure 7.1: Pipeline of SML/NJ

The pipeline of SML/NJ is shown in Figure 7.1. The front end translates a source program to intermediate representation FLINT through parsing and type-checking. The middle end performs some optimizations on FLINT. After CPS- and closure conversion, most type information has been lost from the program. The back end MLRISC takes the untyped result of CPS- and closure conversion, does instruction selection and register allocation, and emits machine code.

The front end first parses SML source programs and builds abstract syntax trees, then it performs type inference and type-checking, followed by the elaboration of the abstract syntax trees to the strongly typed intermediate language FLINT.

During the front end, modules are translated to structures, and higher-order modules are translated to (polymorphic) functions. The front end handles the module system. Later phases only need to deal with the core language constructors.

FLINT is based on a predicative variant of polymorphic calculus $\mathbf{F}_\omega$ [51, 28]. It is designed as a common intermediate representation for various source languages and various architectures. It is expressive and efficient. Several techniques such as hash consing and memoization are used to reduce the cost of representing and manipulating types during compilation.

The middle end performs several conventional optimizations that translate FLINT to FLINT, including optimizations based on control and dataflow analysis, lambda-based reductions, and type specialization that reduces polymorphism.

The CPS-conversion transforms typed FLINT to an untyped representation, CPS. The control flow is made explicit by continuations. Then closure conversion transforms each CPS function so that the free variables of each function are now parameters of the function. The closure representation is optimized to be space efficient [55]. More conventional dataflow optimizations are carried out at this stage.

The back end MLRISC was developed mainly by George and Leung [26]. ML-RISC is aimed to be a generic framework for compiler back ends. It is customizable and retargetable: it can be customized to compile various source languages to various architectures. Most back end modules, including the instruction selection, register allocation, and instruction scheduling, are parameterized over machine specifications so that they can be reused for multiple architectures. To generate a compiler back end, users specialize these modules for their target machine. ML-RISC has been used in many compiler projects including SML/NJ for years, and generates high-performance code.

The untyped CPS representation is first translated to MLTree, a machine-independent representation used by MLRISC as an interface for users. Then an instruction selection module (specialized for the target architecture) creates a flow graph of the target machine instructions from MLTree. The register allocation module assigns physical registers to temporaries. Last the emitter emits machine code.

The whole back end is untyped, including the intermediate representation MLTree and the flowgraph.

MLRISC has an annotation mechanism [36]. Annotations in MLRISC are like comments; in fact, they are emitted as comments in assembly code. They have no runtime effect. One can annotate cells (pseudoregisters that will be mapped to physical registers or memory words), instructions, code blocks, and compilation units. Each annotation describes a property of the construct it annotates, and a construct can have many annotations addressing different properties. MLRISC provides ways to create, append, extract and remove annotations.

Originally MLRISC developers added this mechanism to propagate type information to code optimization phases. Annotations have been used extensively in MLRISC to pass information through the back end without changing existing data structures. Data abstraction hides the representation of client annotations from MLRISC's register allocator and instruction selector.

7.2 Overview of FPCC-ML

The pipeline of FPCC-ML is shown in Figure 7.2.

The FPCC-ML compiler borrows the front end of SML/NJ to translate core ML source programs to FLINT. Then *typed* CPS- and closure conversion convert FLINT to another typed intermediate language NFLINT. Next, NFLINT is trans-

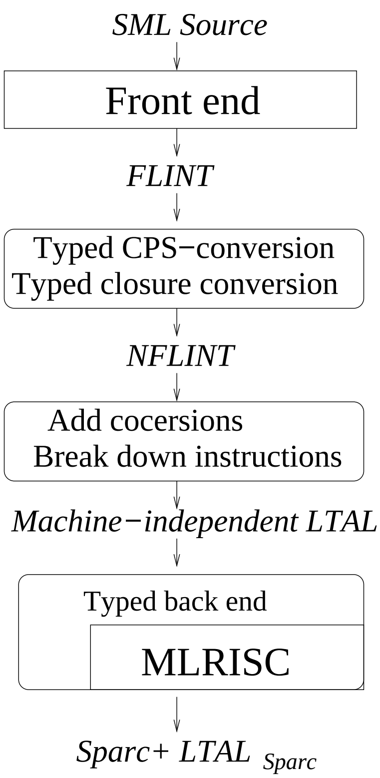

Figure 7.2: Pipeline of FPCC-ML

lated to machine-independent LTAL: complex instructions are broken into sequences of LTAL simple instructions, and coercions are inserted to indicate subtyping relations. Last, the typed back end built on top of MLRISC generates Sparc machine code and an LTAL program that has the same semantics as the machine-code program. The Sparc code could be executed by the FPCC runtime system.

Machine-independent LTAL is actually a different language from $\text{LTAL}_{\text{Sparc}}$, but they have many common constructs. The only difference between the two is that machine-independent LTAL is more abstract. One machine-independent LTAL instruction might correspond to several $\text{LTAL}_{\text{Sparc}}$ instructions (and several Sparc instructions). Also machine-independent LTAL programs don't have the full LRT environments since L and R are only available after the back end. The type-checker of machine-independent LTAL doesn't rely on these two mappings.

The FPCC-ML compiler differs from SML/NJ in several aspects:

- The FPCC-ML compiler preserves type information from the source language all the way down to the machine language. Each compilation stage transforms types along with programs. Each of the intermediate representations FLINT, NFLINT and LTAL is strongly typed, and every intermediate compilation result can be type-checked. The type-preserving compilation was very helpful in finding bugs during the development of the compiler. Many bugs were manifested as type errors and were caught by the type-checker of the result language. Therefore I could pinpoint the very stage where the errors occurred.

- The FPCC-ML compiler serves as a front end of our FPCC system. The emphasis is to generate enough type information so that the proving system could construct machine-checkable safety proofs from machine programs.

Thus I have not paid much attention to performance at this time. FPCC-ML has not included many optimizations in SML/NJ. It could not compete with SML/NJ where many people have made effort to improve the compilation and code generation. But I speculate that FPCC-ML could accommodate many optimizations in SML/NJ and make its performance competitive. Preserving type information during these optimizations should not be a big problem because of the low-level type constructors in LTAL and LTAL's extensibility.

- FPCC-ML does not deal with module system yet, because higher-order modules require higher-order kinds, which the underlying semantic model has not supported yet.

7.3 Typed Intermediate Language NFLINT

The transformation from FLINT to NFLINT was implemented by Hai Fang at Yale University. NFLINT is a simplified version of FLINT, with higher-order kinds removed.

7.3.1 Syntax

Figure 7.3 shows NFLINT syntax. Notice that the types and instructions of NFLINT are standard. Some NFLINT constructors are abstract: they do not specify low-level implementation details. For example, the sum type $\mathrm{sum}(m, \tau_1, \tau_2, \ldots, \tau_n)$ means m unboxed constructors and n boxed constructors which carry values of type $\tau_1, \tau_2, \ldots, \tau_n$ respectively. It does not specify whether these carried values are grouped in a record or flattened. In the former case, a boxed sum value would

$$
\begin{array}{llr}
\tau & ::= \alpha \mid \mathrm{def}\,\Bbbk \mid \mathrm{int} \mid \overline{n} \mid \mathrm{array}(\tau_1, \tau_2) \mid \mathrm{ref}(\tau) & \textbf{Types} \\
& \quad \mid \mu\alpha.\tau \mid \mathrm{sum}(m, \tau_1, \tau_2, \ldots, \tau_n) \mid [\tau_1, \tau_2, \ldots, \tau_n] \mid \exists\alpha.\tau & \\
& \quad \mid \mathrm{code}(\vec{\alpha})[\tau_1, \tau_2, \ldots, \tau_n] & \\
v & ::= x \mid i \mid l & \textbf{Values} \\
op & ::= + \mid - \mid * \mid / & \textbf{Arith. Ops} \\
\iota & ::= x = \mathrm{pack}(\tau_1, \tau, v) \mid (\alpha, v') = \mathrm{open}(v) & \textbf{Instructions} \\
& \quad \mid x = \mathrm{record}(v_1, v_2, \ldots, v_n) \mid x = \mathrm{select}(v, v_i) & \\
& \quad \mid x = \mathrm{fold}[\tau](v) \mid x = \mathrm{unfold}(v) & \\
& \quad \mid x = \mathrm{mkarray}(v_l, v_0) \mid x = \mathrm{sub}(v_a, v_i) \mid \mathrm{update}(v_a, v_i, v) & \\
& \quad \mid x = \mathrm{ref}(v) \mid x = !(v) \mid v := v' & \\
& \quad \mid x = v \mid x = v_1\, op\, v_2 \mid x = \mathrm{inj}[\tau](v) & \\
& \quad \mid \mathrm{switch}(v)[(e_1, e_2, \ldots, e_m), ((x_1, e'_1), (x_2, e'_2), \ldots, (x_m, e'_m))] & \\
& \quad \mid \mathrm{if}(\pi, v_1, v_2) \text{ then } e_1 \text{ else } e_2 & \\
& \quad \mid \mathrm{call}(v)[\tau_1, \tau_2, \ldots, \tau_n][v_1, v_2, \ldots, v_m] & \\
& \quad \mid x = \mathrm{ptapp}(v)[\tau_1, \tau_2, \ldots, \tau_n] &
\end{array}
$$

Figure 7.3: NFLINT Syntax

always be represented as a two-field record: the tag and a pointer to the carried value. The latter case might have boxed sum values represented as records of more than two fields. The implementation could do either way. NFLINT does not reason about position-independence either.

Some NFLINT instructions are complex, such as record, mkarray and switch. They will be expanded to multiple machine instructions.

7.3.2 CPS- and Closure Conversion

FLINT programs are translated to NFLINT by typed CPS- and closure conversion in the style of TAL [44]. FPCC-ML does not have an intermediate language between these two transformations as λ^k in TAL.

In the CPS-conversion, each function has one more parameter, the continuation (the rest of the program). A function does not return. At the end of a function, its continuation is invoked, and the result computed by the function is passed as a parameter to the continuation. The topmost function returns to the operating system. Thus no stack is needed.

A conventional closure conversion creates for each function a closure, a record of the function pointer and the free variables used in the function. The free variables are passed as parameters to the function. Every function is closed after closure conversion.

The typed closure conversion used in FPCC-ML is simpler than the one proposed by Minamide *et al.* [42]. It adopts type-erasure semantics instead of type-passing. A polymorphic function does not need to create another environment for free type variables, nor does the type system need abstract kinds or translucent types for type environments.

7.4 Transformations from NFLINT to Machine-independent LTAL

NFLINT is not low-level enough for assembly languages. NFLINT is translated to machine-independent LTAL by several steps:

- First, complex NFLINT instructions are broken into sequences of LTAL instructions. For example, NFLINT record and mkarray instructions are translated to sequences of LTAL allocation instructions, with an LTAL instruction for each allocation step.

The compiler also needs to choose representations for datatypes, since in LTAL data representations are made explicit. Also tagging and tag-fetching are made explicit and implemented by LTAL instructions when creating and discriminating sum values.

- Next, coercions are added to LTAL programs. This is done through type-unification during type-checking the LTAL program translated from NFLINT. Since the NFLINT program has been type-checked before being translated to LTAL, the LTAL program should also type-check. This stage figures out subtyping and inserts corresponding coercions as hints of subtyping to the LTAL checker.

- The next transformation generates position-independent LTAL code, as explained in Chapter 6.

- Next, functions in a program are reordered to get maximum number of fall-throughs. No Sparc instruction is needed for fall-through. If a function has only one caller, it will be put immediately after the caller, and the call instruction at the end of the caller will become an LTAL calln instruction.

7.5 Typed Back End

We did not originally intend to take advantage of MLRISC, because MLRISC is totally untyped while we need type-preserving transformations. When we learned of MLRISC's annotation mechanism, we tried using annotations to connect the typed representation we need and the untyped one MLRISC uses. The experiment turned out to be rewarding: reusing MLRISC this way is much less work than writing a back end from scratch, and has the advantages that MLRISC provides, such as generating code with good performance and being retargetable.

7.5.1 Annotate MLRISC with LTAL

When an LTAL program is transformed to MLTree, the MLRISC interface language, the compiler annotates the MLTree program with the LTAL program. Later the LTAL annotations are carried through the back end optimizations.

Cells are annotated with variables, MLRISC instructions with LTAL instructions, code blocks with function signatures, and compilation units with type definitions. LTAL programs correspond to MLRISC programs very closely.

However, MLRISC does not take care to maintain annotations through every program transformation. For example, sometimes MLRISC removes annotations of instructions, or creates new instructions without annotations because it does not know what annotations to give them. So the main difficulty in reusing MLRISC is how to restore the missing annotations.

7.5.2 Basic Blocks

Part of the solution to missing annotations is to design LTAL to provide annotations when MLRISC rewrites instructions.

At first LTAL used extended basic blocks: conditional branch instructions such as $\mathbf{if}(v)\ \mathbf{then}(l, \sigma)\ \mathbf{else}(l', \sigma')$ could appear in the middle of a function body. To avoid long jumps, MLRISC would create a new block for the fall-through case and change the "jump" block to be the fall-through case. In the following example, the *neq* case has more than 2^{21} instructions, but the *eq* case does not. MLRISC simply switches the two cases, changing the code in the left column to the one in the right. Label l_3 is for illustration purpose. It does not exist before MLRISC switches branches.

$l_1 : \ldots$	$l_1 : \ldots$
be l_2 (**else** l_3)	**bne** l_3 (**else** l_2)
$(l_3 :)$*neq* case	$(l_2 :)$*eq* case
l_2 : *eq* case	l_3 : *neq* case

Newly created block l_3 needed to be annotated with an LTAL function signature, which MLRISC could not provide since there was no LTAL function that corresponds to this new block. So in the final design LTAL makes each basic block a function. Thus in the above example, when MLRISC moves blocks, the LTAL function signature for l_3 is already there.

The important lesson here is that a good TAL should serve not only as an interface between a compiler and a checker, but also as a useful intermediate language in the back-end phases of the compiler itself. By using basic blocks instead of extended basic blocks, LTAL becomes useful as such an intermediate language.

7.5.3 Hooks

The annotation approach to preserve types needs tight connection between machine code and LTAL annotations. It causes problems when MLRISC breaks annotations. For example, MLRISC transformed instruction $d_1 = 3 + d_2$ to $d_1 = d_2 + 3$ (thus one Sparc instruction suffices), and annotated the new instruction with the annotation of the old one. The LTAL annotation of $d_1 = 3 + d_2$ would be like $v_1 = 3 + v_2$ where v_1 and v_2 are assigned registers d_1 and d_2 respectively. This annotation is invalid for $d_1 = d_2 + 3$. The checker will try to map 3 to d_2 and v_2 to 3, and fail because it cannot match an LTAL immediate and a register, or an LTAL variable and an immediate operand.

This problem results from the fact that MLRISC does not know the meaning of annotations or the connection between annotations and code, thus could not preserve them. Yet MLRISC should not understand annotations because different users give different annotations. The solution of FPCC-ML is that MLRISC users provide hooks (functions) that manipulate annotations, and MLRISC calls those hooks when it transforms the code.

7.5.4 Switch Operands

The commuting transformation (shown above) will call a function of type $annotation \rightarrow annotation$ to restore annotations before it exchanges the two operands. This function takes the old instruction's annotation—an LTAL instruction, and rewrites it to fit the new instruction.

	Before Switching	After Switching
LTAL Annotation	$v_1 = 3 + v_2$	$v_1 = v_2 + 3$
MLRISC Instruction	$d_1 = 3 + d_2$	$d_1 = d_2 + 3$

7.5.5 Load Large Integers

Another case in which MLRISC breaks LTAL annotations is when it splits an instruction into several ones. For example, pseudo-instruction $d = 4097$ is split into two instructions, $d =$ **sethi** 4 and $d = d$ **or** 1, where the first instruction loads the high 22 bits of constant 4097 to the high 22 bits of register d, and the second instruction loads the low 10 bits. In the modified back end, MLRISC calls a function to generate the annotations for the two new instructions.

	Before Transformation	After Transformation
LTAL Annotation	$v = 4097$	$v =$ sethi 4 $v = v + 1$
MLRISC Instruction (v maps to d)	$d = 4097$	$d =$ sethi 4 $d = d$ or 1

7.5.6 Spilling

In case of not enough registers, the register allocator will spill some registers into the spilled area (in the memory), and load them back to registers when needed. This transformation generates new instructions, therefore new LTAL annotations are needed.

Register to Memory

When a register is spilled into the memory, MLRISC calls a hook that creates a new LTAL variable corresponding to the spilled location. A new LTAL move instruction from the variable that maps to the register to the new LTAL variable is generated for the MLRISC store instruction.

	Before Transformation	After Transformation
LTAL Annotation	$v = \ldots$	$v = \ldots$ $v' = v$
MLRISC Instruction (v maps to d and v' to s)	$d = \ldots$	$d = \ldots$ st d, s

Memory to Register

Before using a spilled word, it must be first loaded back to a register. Similar to the spilling phase, a hook creates a new LTAL variable for the register. A new LTAL move instruction annotates the MLRISC load instruction.

	Before Transformation	After Transformation
LTAL Annotation	 $\ldots v \ldots$	$v = v'$ $\ldots v \ldots$
MLRISC Instruction (v maps to d and v' to s)	 $\ldots d \ldots$	ld s, d $\ldots d \ldots$

7.6 Measurements

The compiler from core ML to LTAL+machine code is written in ML; its size (including blank lines and comments) is 50k lines of the Standard ML of New Jersey (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[1] itself, of which 400 lines are new or modified to support our more-general annotation interface.

7.6.1 Performance

We[2] compared our performance[3] to that of SML/NJ 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.

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
Safety check time	0.183	0.432
FPCC Run time	0.014	0.407
FPCC/SMLNJ slowdown	1.036	1.555
Sparc instrs.	870	1816
LTAL tokens	34278	57670
Coercion tokens	17%	23%

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 compiler algorithms as necessary for a production compiler.

[1]The MLRISC software has several other analyses, optimizations, and target machine specifications that are not used and counted here.

[2]In this section I use “we” to represent Dinghao Wu and I. Dinghao Wu has implemented an LTAL checker in Twelf and SICStus Prolog.

[3]Measured on Sun UltraSparc E250, 400 MHz.

Run time is almost as good as SML/NJ. 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.

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 or full unification; checking in Flit is about five times slower than SICStus, but it makes the TCB much smaller since we don't have to trust the Prolog compiler and interpreter [68].

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. Eliminating the coercions—thus requiring some backtracking in the checker—could save about 20% in LTAL size. The builders of SpecialJ [20] and TALx86 [43] 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.

Chapter 8

Summary and Future Work

In summary, this thesis explains the typed assembly language LTAL used in FPCC to reason about code safety. The FPCC-ML compiler compiles a core ML source program to Sparc machine code and an LTAL program through type-preserving compilation, and with the guidance of the LTAL program, the prover constructs for the machine code a safety proof that is machine-checkable in a simple logic from minimal axioms.

8.1 Summary of Contributions

A weakness of previous PCC systems is that the proof-checking infrastructure is too complex to prove sound. LTAL is a syntactic low-level typed assembly language with a semantic model that backs up its soundness with a machine-checkable proof. The semantic modeling technique makes LTAL easily and safely extensible. My thesis work did not include the LTAL soundness proof, which is being done by others in the FPCC project, but it does include the design of the LTAL language.

LTAL is closer than other TALs to machine language that is actually executed, in the sense that it provides more primitive instructions each of which maps to at most one machine instruction. To type-check these primitive instructions, LTAL needs an expressive type system to describe the low-level machine details, yet the type-checking is still decidable and simple.

Also because of these primitive instructions, LTAL instructions are more schedulable than other TALs using macro instructions. The design goal is to give the scheduler as much flexibility as possible.

I have implemented a prototype compiler that transforms core ML programs to Sparc code annotated with LTAL programs. In the compiler an untyped back end preserves types by annotations and hooks.

At this stage, the LTAL language is almost stable for the purpose of compiling core ML to Sparc machine code, but the development of the FPCC-ML compiler is still undergoing. Many features and optimization could be accommodated.

8.2 Future Work

Array Bounds Checking Elimination Currently the LTAL type system provides sufficient support for reasoning about array bound checking elimination, but the compiler has not yet fully implemented this optimization.

The elimination requires that the compiler figure out the variables whose values should be tracked and refine their types using singleton types. One possible way is to let the programmer specify this as in DTAL. Another way is by data flow analysis. All variables whose values will affect an array index will be tracked. This analysis is easier to be done before LTAL since LTAL is too low-level.

Furthermore, reasoning about whether array indices are in range would introduce integer constraint solving. The type-checker should stay simple and decidable and avoid a constraint solver. Thus, the optimizations that LTAL allows are likely to be weaker than that of DTAL.

Garbage Collection To keep the TCB minimal, we would like the garbage collector to be out of the TCB, thus we need to reason about the safety of garbage collection. To support this, we need to extend LTAL, probably adding into LTAL a set of primitive constructors to represent storage. Region-based calculus seems very promising [63, 65, 5].

Other Source Languages LTAL targets ML-like functional programming languages. It provides support for polymorphism, ML-style user-defined datatypes, and closures (to deal with higher-order functions). For other languages such as Java, this is not enough. There is very significant research work on representing classes and objects in a type calculus, mostly typed intermediate language [4, 17, 18, 67, 20, 33]. But there is not yet one calculus that could express machine code and whose soundness proof could be machine-checked. Thus they could not be used in FPCC directly. Also, it could be interesting to investigate how the proving system adapts to the change of source language.

Appendix A

Formal Semantics

A.1 Kinding Rules

$$\frac{}{T;\rho \vdash \alpha \;:\; \rho(\alpha)} \qquad \frac{}{T;\rho \vdash \mathrm{int} \;:\; \Omega} \qquad \frac{}{T;\rho \vdash \mathrm{boxed} \;:\; \Omega}$$

$$\frac{T;\rho,\alpha:\kappa \vdash \tau \;:\; \Omega}{T;\rho \vdash \exists\alpha:\kappa.\tau \;:\; \Omega} \qquad \frac{T;\rho,\alpha:\kappa \vdash \tau \;:\; \Omega}{T;\rho \vdash \mu\alpha:\kappa.\tau \;:\; \Omega}$$

$$\frac{T;\rho \vdash \tau_1 \;:\; \Omega_N \quad T;\rho \vdash \tau_2 \;:\; \Omega_N}{T;\rho \vdash \tau_1 + \tau_2 \;:\; \Omega_N} \qquad \frac{}{T;\rho \vdash \mathrm{freem} \;:\; \Omega_N}$$

$$\frac{}{T;\rho \vdash \overline{n} \;:\; \Omega_N} \qquad \frac{T;\rho \vdash \tau \;:\; \Omega_N}{T;\rho \vdash \mathrm{int}_\pi\,\tau \;:\; \Omega} \qquad \frac{T;\rho \vdash \tau \;:\; \Omega}{T;\rho \vdash \mathrm{field}\; i\; \tau \;:\; \Omega}$$

$$\frac{T;\rho \vdash \tau_1 \;:\; \Omega \quad T;\rho \vdash \tau_2 \;:\; \Omega}{T;\rho \vdash \tau_1 \cap \tau_2 \;:\; \Omega} \qquad \frac{T;\rho \vdash \tau_1 \;:\; \Omega \quad T;\rho \vdash \tau_2 \;:\; \Omega}{T;\rho \vdash \tau_1 \cup \tau_2 \;:\; \Omega}$$

$$\frac{\begin{array}{c}\rho' = \rho,\alpha_1:\kappa_1,\ldots,\alpha_j:\kappa_j \qquad T;\rho' \vdash m \;:\; \Omega_N \\ T;\rho' \vdash_{cc} cc \qquad T;\rho' \vdash \tau_i \;:\; \Omega i = 1,\ldots,n\end{array}}{T;\rho \vdash \mathrm{codeptr}[\alpha_1:\kappa_1,\ldots,\alpha_j:\kappa_j](m, cc,\, v_1:\tau_1,\ldots,v_n:\tau_n) \;:\; \Omega}$$

$$\frac{}{T;\rho \vdash \mathrm{addr}(l) \ : \ \Omega_N} \qquad \frac{}{T;\rho \vdash \mathrm{diff}(l_1, l_2) \ : \ \Omega_N}$$

$$\frac{T;\rho \vdash T(\Bbbk) \ : \ \kappa}{T;\rho \vdash \mathrm{def}\,\Bbbk \ : \ \kappa} \qquad \frac{T;\rho \vdash \tau_1 \ : \ \Omega_N \quad T;\rho \vdash \tau_2 \ : \ \Omega}{T;\rho \vdash \mathrm{array}(\tau_1, \tau_2) \ : \ \Omega}$$

$$\frac{T;\rho \vdash \tau \ : \ \Omega}{T;\rho \vdash \mathrm{offset}\ i\ \tau \ : \ \Omega} \qquad \frac{T;\rho \vdash \tau_1 \ : \ \Omega \quad T;\rho \vdash \tau_2 \ : \ \Omega}{T;\rho \vdash \mathrm{range}[\tau_1, \tau_2] \ : \ \Omega}$$

A.2 Branch to a Function

$$\frac{\begin{array}{c} l \text{ is declared as } l[\alpha_1 : \kappa_1, \ldots, \alpha_j : \kappa_j](m, cc, v_1 : \tau_1, \ldots, v_n : \tau_n) = \iota_1; \ldots; \iota_k \\ n_1 \geq m \qquad cc' = cc \text{ or } cc = \mathrm{cc\_none} \qquad LRT;\rho;\Phi \vdash v_i : \tau_i \ \ \forall\ 1 \leq i \leq n \end{array}}{LRT;\rho;(n_1, n_2, \tau);\Phi;cc' \vdash_\ell l}$$

A.3 Value Rules

$$\frac{}{LRT;\rho;\Phi \vdash i : \mathrm{int}_{=}\,\bar{\bar{i}}} \qquad \frac{}{LRT;\rho;\Phi \vdash \mathrm{vdiff}(g, f) : \mathrm{diff}(g, f)}$$

$$\frac{}{LRT;\rho;\Phi \vdash x : \Phi(x)} \qquad \frac{l \text{ is declared as: } l[\vec{\alpha} : \vec{\kappa}](m, cc, v_1 : \tau_1, \ldots, v_n : \tau_n) = \ldots}{LRT;\rho;\Phi \vdash l : \mathrm{codeptr}[\vec{\alpha} : \vec{\kappa}](m, cc, v_1 : \tau_1, \ldots, v_n : \tau_n)}$$

A.4 Coercion Rules

$$\frac{}{\rho;LRT \vdash_c \tau \overset{\mathrm{cid}}{\hookrightarrow} \tau} \qquad \frac{n_1 \leq n < n_2}{\rho;LRT \vdash_c \overline{n} \overset{\mathrm{crange}[n_1,n_2]}{\hookrightarrow} \mathrm{range}[n_1, n_2]}$$

$$\frac{}{\rho;LRT \vdash_c \mathrm{def}\,\Bbbk \overset{\mathrm{cname}}{\hookrightarrow} T(\Bbbk)} \qquad \frac{}{\rho;LRT \vdash_c T(\Bbbk) \overset{\mathrm{cdef}\ \Bbbk}{\hookrightarrow} \mathrm{def}\,\Bbbk}$$

$$\frac{}{\rho;LRT \vdash_c \overline{n} \overset{\mathrm{c2int32}}{\hookrightarrow} \mathrm{int}} \qquad \frac{}{\rho;LRT \vdash_c \mathrm{range}[n_1, n_2] \overset{\mathrm{c2int32}}{\hookrightarrow} \mathrm{int}}$$

$$\frac{}{\rho; LRT \vdash_c \text{offset } 0\ \tau \overset{\text{coffset0}}{\hookrightarrow} \tau} \qquad \frac{}{\rho; LRT \vdash_c \tau \overset{\text{c2offset0}}{\hookrightarrow} \text{offset } 0\ \tau}$$

$$\frac{}{\rho; LRT \vdash_c \tau_1 \cap \tau_2 \overset{\text{cproj1}}{\hookrightarrow} \tau_1} \qquad \frac{}{\rho; LRT \vdash_c \tau_1 \cap \tau_2 \overset{\text{cproj2}}{\hookrightarrow} \tau_2}$$

$$\frac{\rho; LRT \vdash_c \tau \overset{c_1}{\hookrightarrow} \tau' \quad \rho; LRT \vdash_c \tau' \overset{c_2}{\hookrightarrow} \tau''}{\rho; LRT \vdash_c \tau \overset{c_2 \circ c_1}{\hookrightarrow} \tau''} \qquad \frac{\rho; LRT \vdash_c \tau \overset{c_1}{\hookrightarrow} \tau_1 \quad \rho; LRT \vdash_c \tau \overset{c_2}{\hookrightarrow} \tau_2}{\rho; LRT \vdash_c \tau \overset{\text{c2inters}(c_1,c_2)}{\hookrightarrow} \tau_1, \tau_2}$$

$$\frac{}{\rho; LRT \vdash_c \tau_1 \overset{\text{cinj1}_{[\tau_1 \cup \tau_2]}}{\hookrightarrow} \tau_1 \cup \tau_2} \qquad \frac{}{\rho; LRT \vdash_c \tau_2 \overset{\text{cinj2}_{[\tau_1 \cup \tau_2]}}{\hookrightarrow} \tau_1 \cup \tau_2}$$

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

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

$$\frac{\begin{array}{l}\alpha \text{ is a fresh type variable} \qquad \tau = \tau_1 \cup \tau_2 \cup \ldots \cup \tau_n \\ \tau_i = \text{field } 0 \text{ int}_= t_i \cap \tau_i' \ \ \forall\ 1 \leq i \leq n\end{array}}{\rho; LRT \vdash_c \tau \overset{\text{csum2hastag}}{\hookrightarrow} \exists\alpha : \Omega_N.(\text{field } 0 \text{ int}_= \alpha) \cap \tau}$$

$$\frac{l \text{ is declared as: } [\vec{\alpha} : \vec{\kappa}](m, cc, \vec{v} : \vec{\tau}) = \ldots}{\rho; LRT \vdash_c \text{addr}(l) \overset{\text{caddr2code}}{\hookrightarrow} \text{codeptr}[\vec{\alpha} : \vec{\kappa}](m, cc, \vec{v} : \vec{\tau})}$$

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

A.5 Typing Rules for Instructions

$$\frac{LRT; \rho; \Phi \vdash v : \tau}{LRT \vdash (\rho; \hbar; \Phi; cc)\ \{v = v'\}\ (\rho; \hbar; \Phi, v : \tau; cc)}$$

$$\frac{LRT; \rho; \Phi \vdash v_1 : \mathrm{int} \quad LRT; \rho; \Phi \vdash v_2 : \mathrm{int} \quad op \neq +_i}{LRT \vdash (\rho; \hbar; \Phi; cc)\ \{v = v_1\ op\ v_2\}\ (\rho; \hbar; \Phi, v : \mathrm{int}; cc)}$$

$$\frac{LRT; \rho; \Phi \vdash v_1 : \mathrm{int}_= \overline{n_1} \quad LRT; \rho; \Phi \vdash v_2 : \mathrm{int}_= \overline{n_2}}{LRT \vdash (\rho; \hbar; \Phi; cc)\ \{v = v_1\ \ +_i\ \ v_2\}\ (\rho; \hbar; \Phi, v : \mathrm{int}_= \overline{n_1 + n_2}; cc)}$$

$$\frac{LRT; \rho; \Phi \vdash v_1 : \mathrm{addr}(f) \qquad LRT; \rho; \Phi \vdash v_2 : \mathrm{diff}(g, f)}{LRT \vdash (\rho; \hbar; \Phi; cc)\ \{v = \mathrm{addradd}(v_1, v_2)\}\ (\rho; \hbar; \Phi, v : \mathrm{addr}(g); cc)}$$

$$\frac{}{LRT \vdash (\rho; \hbar; \Phi; cc)\ \{v = \mathrm{sethi}(n)\}\ (\rho; \hbar; \Phi, v : \mathrm{int}_= \overline{n * 4096}; cc)}$$

$$\frac{LRT; \rho; \Phi \vdash v_1 : \mathrm{field}\ i\ \tau \quad LRT; \rho; \Phi \vdash v_2 : \mathrm{int}_= \overline{i}}{LRT \vdash (\rho; \hbar; \Phi; cc)\ \{v = \mathrm{load}(v_1, v_2)\}\ (\rho; \hbar; \Phi, v : \tau; cc)}$$

$$\frac{LRT; \rho; \Phi \vdash v_1 : \mathrm{array}(\tau_l, \tau) \quad LRT; \rho; \Phi \vdash v_2 : \mathrm{int}_{\geq} \overline{0} \cap \mathrm{int}_< \tau_l}{LRT \vdash (\rho; \hbar; \Phi; cc)\ \{v = \mathrm{sub}(v_1, v_2)\}\ (\rho; \hbar; \Phi, v : \tau; cc)}$$

$$\frac{LRT; \rho; \Phi \vdash v_a : \mathrm{array}(\tau_l, \tau) \quad LRT; \rho; \Phi \vdash v_i : \mathrm{int}_{\geq} \overline{0} \cap \mathrm{int}_< \tau_l \quad LRT; \rho; \Phi \vdash v : \tau}{LRT \vdash (\rho; \hbar; \Phi; cc)\ \{\mathrm{update}(v_a, v_i, v)\}\ (\rho; \hbar; \Phi; cc)}$$

$$\frac{\begin{array}{l} \hbar = (n_1, n_2, \tau) \qquad n_1 \geq m \\ LRT; \rho; \Phi \vdash v : \mathrm{codeptr}[\alpha_1 : \kappa_1, \ldots, \alpha_j : \kappa_j](m, cc', v_1 : \tau'_1, \ldots, v_n : \tau'_n) \\ sub = (\tau_1/\alpha_1, \ldots, \tau_j/\alpha_j), cc = cc[sub] \text{ or } cc = \text{cc\_none} \\ LRT; \rho; \Phi \vdash v_i : \tau'_i[sub]\ \ \forall\ 1 \leq i \leq n \end{array}}{LRT \vdash (\rho; \hbar; \Phi; cc)\ \{\mathrm{call}(v, [\tau_1, \ldots, \tau_j])\}\ (\rho; \hbar; \Phi; cc)}$$

$$\frac{\begin{array}{l}\hbar = (n_1, n_2, \tau) \qquad n_1 \geq m \\ l \text{ is declared as } l[\alpha_1 : \kappa_1, \ldots, \alpha_j : \kappa_j](m, cc', v_1 : \tau_1', \ldots, v_n : \tau_n') = \iota_1; \ldots; \iota_k \\ sub = (\tau_1/\alpha_1, \ldots, \tau_j/\alpha_j), cc = cc[sub] \text{ or } cc = \text{cc\_none} \\ LRT; \rho; \Phi \vdash v_i : \tau_i'[sub] \;\; \forall\, 1 \leq i \leq n\end{array}}{LRT \vdash (\rho; \hbar; \Phi; cc)\ \{\text{calln}(l, [\tau_1, \ldots, \tau_j])\}\ (\rho; \hbar; \Phi; cc)}$$

$$\frac{\hbar = (\tau, \overline{0}, \text{boxed})}{LRT \vdash (\rho; \hbar; \Phi; cc)\ \{\text{testAvail}\}\ (\rho; \hbar; \Phi; \text{cc\_cmp}(\text{freem}, \overline{0}))}$$

$$\frac{LRT; \rho; \Phi \vdash v : \text{int}_= \tau_n}{\begin{array}{l}LRT \vdash (\rho; (\tau, \overline{0}, \text{boxed}); \Phi; \text{cc\_cmp}(\text{freem}, \overline{0}))\ \{\text{testFull(v)}\} \\ (\rho; (\tau, \overline{0}, \text{boxed}); \Phi; \text{cc\_cmp}(\text{freem}, \tau_n))\end{array}}$$

$$\frac{\begin{array}{l}LRT; \rho; \hbar; \Phi; cc \vdash_\ell l_1 \\ LRT; \rho; \hbar; \Phi; cc \vdash_\ell l_2\end{array}}{LRT \vdash (\rho; \hbar; \Phi; cc)\ \{\text{if}(\pi) \text{ then } l_1 \text{ else } l_2\}\ (\rho; \hbar; \Phi; cc)}$$

$$\frac{\begin{array}{l}cc = \text{cc\_cmp}(\text{freem}, \tau_n) \qquad LRT; \rho; (\tau, \overline{0}, \text{boxed}); \Phi; cc \vdash_\ell l_1 \\ LRT; \rho; (\tau_n + \overline{4096}, \overline{0}, \text{boxed}); \Phi; cc \vdash_\ell l_2\end{array}}{LRT \vdash (\rho; (\tau, \overline{0}, \text{boxed}); \Phi; cc)\ \{\text{iffull then } l_1 \text{ else } l_2\}\ (\rho; (\tau, \overline{0}, \text{boxed}); \Phi; cc)}$$

$$\frac{LRT; \rho; \Phi \vdash v' : \text{int}_= \overline{i} \qquad LRT; \rho; \Phi \vdash v : t_i \qquad t' = t \cap (\text{field } i\ t_i)}{LRT \vdash (\rho; (\tau_m, \tau_n, t); \Phi; cc)\ \{\text{store}(v', v)\}\ (\rho; (\tau_m, \tau_n + \overline{4}, t'); \Phi; cc)}$$

$$\frac{}{LRT \vdash (\rho; (\tau_m, \tau_n, \tau); \Phi; cc)\ \{v = \text{record}\}\ (\rho; (\tau_m, \tau_n, \tau); \Phi, v : \tau; cc)}$$

$$\frac{LRT; \rho; \Phi \vdash v : \mathrm{int}_{=}\,\tau_n}{LRT \vdash (\rho; (\tau_m, \tau_n, \tau); \Phi; \mathrm{cc\_cmp}(\mathrm{freem}, k))\ \{\mathrm{incAlloc}\ v\}\ (\rho; (\tau_m - \tau_n, \overline{0}, \mathrm{boxed}); \Phi; \mathrm{cc\_none})}$$

$$\frac{LRT; \rho; \Phi \vdash v : \mathrm{int}_{=}\,\tau_n}{LRT \vdash (\rho; (\tau_m, \tau_n, \tau); \Phi; cc)\ \{\mathrm{incAlloc}\ v\}\ (\rho; (\tau_m - \tau_n, \overline{0}, \mathrm{boxed}); \Phi; cc)}$$

$$\frac{}{LRT \vdash (\rho; (\tau_m, \overline{0}, \mathrm{boxed}); \Phi; cc)\ \{\mathrm{arrStart}[\tau]\}\ (\rho; (\tau_m, \overline{0}, \mathrm{array}(\overline{0}, \tau)); \Phi; cc)}$$

$$\frac{LRT; \rho; \Phi \vdash v_i : \mathrm{int}_{=}\,\tau_n \qquad \tau_n' = \tau_n + \overline{4} \qquad LRT; \rho; \Phi \vdash v : \tau}{LRT \vdash (\rho; (\tau_m, \tau_n, \mathrm{array}(\tau_n, \tau)); \Phi; cc)\ \{\mathrm{storeA}(v_i, v)\}\ (\rho; (\tau_m, \tau_n', \mathrm{array}(\tau_n', \tau)); \Phi; cc)}$$

$$\frac{LRT; \rho; \Phi \vdash v_1 : \mathrm{int} \quad LRT; \rho; \Phi \vdash v_2 : \mathrm{int}}{LRT \vdash (\rho; \hbar; \Phi; cc)\ \{\mathrm{cmp}(v_1, v_2)\}\ (\rho; \hbar; \Phi; \mathrm{cc\_none})}$$

$$\frac{LRT; \rho; \Phi \vdash v_1 : \mathrm{int}_{=}\,\tau_1 \quad LRT; \rho; \Phi \vdash v_2 : \mathrm{int}_{=}\,\tau_2}{LRT \vdash (\rho; \hbar; \Phi; cc)\ \{\mathrm{cmpr}(v_1, v_2)\}\ (\rho; \hbar; \Phi; \mathrm{cc\_cmp}(\tau_1, \tau_2))}$$

$$\frac{\begin{array}{l} cc = \mathrm{cc\_cmp}(\mathrm{int}_{=}\,\tau_n, \mathrm{int}_{=}\,\tau_l) \qquad LRT; \rho; (\tau_m, \tau_l, \mathrm{array}(\tau_l, \tau)); \Phi; cc \vdash_\ell l_1 \\ LRT; \rho; (\tau_m, \tau_n, \mathrm{array}(\tau_n, \tau)); \Phi; cc \vdash_\ell l_2 \end{array}}{LRT \vdash (\rho; (\tau_m, \tau_n, \mathrm{array}(\tau_n, \tau)); \Phi; cc)\ \{\mathrm{ifinitA\ then}\ l_1\ \mathrm{else}\ l_2\}\ (\rho; (\tau_m, \tau_n, \mathrm{array}(\tau_n, \tau)); \Phi; cc)}$$

$$\frac{LRT; \rho; \Phi \vdash v : \exists \alpha : \kappa . \tau}{LRT \vdash (\rho; \hbar; \Phi; cc)\ \{(\alpha, v_0) = \mathrm{open}(v)\}\ (\rho, \alpha : \kappa; \hbar; \Phi, v_0 : \tau; cc)}$$

$$\frac{LRT;\rho;\Phi \vdash v : \tau \quad cc' = \text{cc\_testbox}\ \alpha}{LRT \vdash (\rho;\hbar;\Phi;cc)\ \{(\alpha, v') = \text{testbox}(v)\}\ (\rho;\hbar;\Phi, v' : \text{int}_{=}\ \alpha \cap \tau; cc')}$$

$$\frac{\begin{array}{l} cc = \text{cc\_testbox}\ \tau_\alpha \\ LRT;\rho;\Phi \vdash v : \text{int}_{=}\ \tau_\alpha \cap (\text{int}_{=}\ \overline{0} \cup \text{int}_{=}\ \overline{1} \cup \ldots \cup \text{int}_{=}\ \overline{n-1} \cup \tau') \\ \tau' = \tau_1 \cup \tau_2 \cup \ldots \cup \tau_m \qquad \tau_i = (\text{field } 0\ \text{int}_{=}\ t_i) \cap \tau'_i \ \ \forall 1 \leq i \leq m \\ LRT;\rho;\hbar;\Phi, v_1 : \tau'; cc \vdash_\ell l_1 \qquad LRT;\rho;\hbar;\Phi, v_2 : \text{range}[0,n]; cc \vdash_\ell l_2 \end{array}}{LRT \vdash (\rho;\hbar;\Phi;cc)\ \{\text{ifboxed}\{v\}\ \text{then}\ (v_1,l_1)\ \text{else}\ (v_2,l_2)\}\ (\rho;\hbar;\Phi;cc)}$$

$$\frac{\begin{array}{l} cc = \text{cc\_cmp}(\tau_\alpha, \overline{i}) \qquad LRT;\rho;\Phi \vdash v : (\text{field } 0\ \text{int}_{=}\ \tau_\alpha) \cap \tau \\ \tau = \tau_1 \cup \tau_2 \cup \ldots \cup \tau_n \qquad \tau_i = \text{field } 0\ \text{int}_{=}\ t_i \cap \tau'_i \ \ \forall\ 1 \leq i \leq n \\ \tau_t = \cup\{\tau_j | 1 \leq j \leq n, t_j\ \pi\ i \neq \text{false}\} \\ \qquad \tau_f = \cup\{\tau_j | 1 \leq j \leq n, t_j\ \pi\ i \neq \text{true}\} \\ LRT;\rho;\hbar;\Phi, v_1 : (\text{field } 0\ \text{int}_{=}\ \tau_\alpha) \cap \tau_t; cc \vdash_\ell l_1 \\ LRT;\rho;\hbar;\Phi, v_2 : (\text{field } 0\ \text{int}_{=}\ \tau_\alpha) \cap \tau_f; cc \vdash_\ell l_2 \end{array}}{LRT \vdash (\rho;\hbar;\Phi;cc)\ \{\text{iftag}(\pi)\{v\}\ \text{then}\ (v_1,l_1)\ \text{else}\ (v_2,l_2)\}\ (\rho;\hbar;\Phi;cc)}$$

Appendix B

Tag Discrimination Example

The LTAL and Sparc instruction sequences for the example in Figure 5.1.2 is shown in Figure B.1. Variables $v, v_0, v_0', v_{CD}, v_{AB}, v_1, v_2, v_C, v_D$ are all assigned register r, and variable t is assigned r_t.

Block l tests whether v is a boxed value (a pointer). If v is a small integer (less than 256), then it is either A or B, otherwise it is C or D. Instruction (1) unfolds v to v_0. Instruction (2) performs the test for pointers, rebinds v_0 to v_0' and sets condition codes. Instruction (3) examines the condition codes and rebinds two fresh variables v_{CD} and v_{AB} with refined types for boxed and unboxed cases respectively.

The unboxed case l_{AB} tests whether v_{AB} is 0 or 1. Instruction (4) compares v_{AB} with 0. If v_{AB} is 0, the control goes to l_A after instruction (5). Otherwise, it falls through to l_B. Here instruction (5) is a normal conditional branch without type refinement, since type refinement is not necessary to guarantee safety in this case.

The boxed case l_{CD} tests the tag of v_{CD}. First v_{CD} is coerced to v_1 with an existential type by coercion csum2hastag. Variable v_1 hides the type of its tag within the existential type. Then instruction (7) opens v_1 to v_2 and binds a brand

	LTAL	Sparc
	$l:$	$l:$
(1)	$v_0 = \mathrm{cunfold}(v)$	
(2)	$(\beta, v_0') = \mathrm{testbox}(v_0)$	$\mathbf{subcc}\ r, 256$
(3)	$\mathrm{ifboxed}\{v_0'\}$ then (v_{CD}, l_{CD})	
	else (v_{AB}, l_{AB})	$\mathbf{bge}\ l_{CD}$
	$l_{AB}:$	$l_{AB}:$
(4)	$\mathrm{cmp}(v_{AB}, 0)$	$\mathbf{subcc}\ r, 0$
(5)	if $=$ then l_A else l_B	$\mathbf{be}\ l_A$
	$l_B: \ldots$	$l_B: \ldots$
	$l_A: \ldots$	$l_A: \ldots$
	$l_{CD}:$	$l_{CD}:$
(6)	$v_1 = \mathrm{csum2hastag}(v_{CD})$	
(7)	$(\alpha_1, v_2) = \mathit{open}(v_1)$	
(8)	$t = \mathrm{load}(v_2, 0)$	$\mathbf{ld}\ [r], r_t$
(9)	$\mathrm{cmpr}(t, 0)$	$\mathbf{subcc}\ r_t, 0, \%g0$
(10)	$\mathrm{iftag}(=)\{v_2\}$ then (v_C, l_C)	
	else (v_D, l_D)	$\mathbf{be}\ l_C$
	$l_D: \ldots$	$l_D: \ldots$
	$l_C: \ldots$	$l_C: \ldots$

Figure B.1: LTAL Instruction Sequence for Discriminating *Mylist*

new type variable α_1 for the unknown tag. Instruction (8) extracts the tag t (the first field) from v_2 . Then instruction (9) checks whether tag t is 0 and sets condition codes. Instruction (10) checks condition codes set by (9), rebinds two new variables v_C and v_D as aliases of v_2 and does conditional branch. The types of v_C and v_D are refined to indicate that v_C is tagged 0 and v_D is tagged 1. Instruction (10) is a specialized conditional branch that refines types according to the value of the tag.

Type-checking the LTAL sequence in Figure B.1 goes as follows:

At the beginning of block l, v has type

$$\mathit{mylist} = \mu\alpha : \Omega\,.\mathrm{int}_= \overline{0} \cup \mathrm{int}_= \overline{1} \cup [\mathrm{int}_= 0, \mathrm{int}] \cup [\mathrm{int}_= 1, \mathrm{int}, \alpha].$$

Variable v is unfolded to v_0 of type $\text{int}_= \overline{0} \cup \text{int}_= \overline{1} \cup [\text{int}_= 0, \text{int}] \cup [\text{int}_= 1, \text{int}, \textit{mylist}]$ by instruction (1). Instruction (2) rebinds v_0 to v'_0 and adds a conjunct $\text{int}_= \beta$ to v_0's type. v_0 has type $\text{int}_= \beta \cap (\text{int}_= \overline{0} \cup \text{int}_= \overline{1} \cup [\text{int}_= 0, \text{int}] \cup [\text{int}_= 1, \text{int}, \textit{mylist}])$. Instruction (2) also sets condition codes cc_testbox β. Instruction ifboxed rebinds v'_0 to v_{CD} in the boxed cases, and to v_{AB} in the unboxed cases. Variable v_{CD} has type $[\text{int}_= 0, \text{int}] \cup [\text{int}_= 1, \text{int}, \textit{mylist}]$, Variable v_{AB} has type $\text{range}(0, 2)$, which means integer 0 (case A) or 1 (case B).

In block l_{AB}, the type of v_{AB} is $\text{range}(0, 2)$, and is not refined further.

In block l_{CD}, instruction (6) first coerces v_{CD} to v_1, and uses a type variable to indicate the tag in an existential type. Then instruction (7) opens v_1 to v_2. Variable v_2 has type field 0 $\text{int}_= \alpha_1 \cap ([\text{int}_= 0, \text{int}] \cup [\text{int}_= 1, \text{int}, \textit{mylist}])$. Instruction (8) loads the tag of v_2 to t, which is assigned type $\text{int}_= \alpha_1$.

Instruction (9) compares t with 0 and sets condition code status $\text{cc\_cmp}(\alpha_1, \overline{0})$. Instruction (10) rebinds v_2 to v_C of type field 0 $\text{int}_= \alpha_1 \cap [\text{int}_= 0, \text{int}]$ in l_C, since the other disjunct shows tag 1 and thus is not compatible with the condition codes. And v_2 will be v_D of type field 0 $\text{int}_= \alpha_1 \cap [\text{int}_= 1, \text{int}, \textit{mylist}]$ in l_D for similar reasons.

$l : v : \textit{mylist}$

(1) $v_0 = \text{cunfold}(v)$

$\quad v_0 : \text{int}_= 0 \cup \text{int}_= 1 \cup [\text{int}_= 0, \text{int}] \cup [\text{int}_= 1, \text{int}, \textit{mylist}]$

(2) $(\beta, v_0') = \text{testbox}(v_0)$

$\quad v_0' : \text{int}_= \beta \cap (\text{int}_= 0 \cup \text{int}_= 1 \cup [\text{int}_= 0, \text{int}] \cup [\text{int}_= 1, \text{int}, \textit{mylist}])$

(3) $\text{ifboxed}\{v_0'\}$ then (v_{CD}, l_{CD})

else (v_{AB}, l_{AB})

$l_{AB} : v_{AB} : \text{int}_= 0 \cup \text{int}_= 1$

(4) $\text{cmpr}(v_{AB}, 0)$

(5) if $=$ then l_A

else l_B

$l_B : v_{AB} : \text{int}_= 0 \cup \text{int}_= 1$

. . .

$l_A : v_{AB} : \text{int}_= 0 \cup \text{int}_= 1$

. . .

$l_{CD} : v_{CD} : [\text{int}_= 0, \text{int}] \cup [\text{int}_= 1, \text{int}, \textit{mylist}]$

(6) $v_1 = \text{csum2hastag}(v_{CD})$

$\quad v_1 : \exists\alpha : \Omega_N.(\text{field } 0 \text{ int}_= \alpha) \cap ([\text{int}_= 0, \text{int}] \cup [\text{int}_= 1, \text{int}, \textit{mylist}])$

(7) $(\alpha_1, v_2) = \textit{open}(v_1)$

$\quad v_3 : (\text{field } 0 \text{ int}_= \alpha_1) \cap ([\text{int}_= 0, \text{int}] \cup [\text{int}_= 1, \text{int}, \textit{mylist}])$

(8) $t = \text{load}(v_2, 0)$

$\quad t : \text{int}_= \alpha_1$

(9) $\text{cmpr}(t, 0)$

condition codes: $\text{cc\_cmp}(\alpha_1, \overline{0})$

(10) $\text{iftag}(=)\{v_2\}$ then (v_C, l_C)

else (v_D, l_D)

$l_D : v_D : \text{field } 0 \text{ int}_= \alpha_1 \cap [\text{int}_= 1, \text{int}, \textit{mylist}]$

. . .

$l_C : v_C : \text{field } 0 \text{ int}_= \alpha_1 \cap [\text{int}_= 0, \text{int}]$

. . .

Figure B.2: Type-checking Discrimination of *Mylist*

Bibliography

[1] Authenticode. http://www.microsoft.com/ie/ie40/features/ sec-authenticode.htm.

[2] Smart firewall. http://govt.argreenhouse.com/SmartFirewalls/.

[3] Cert advisory ca-1999-04 melissa macro virus, Mar. 1999. http://www.cert.org/advisories/CA-1999-04.html.

[4] M. Abadi, L. Cardelli, and R. Viswanathan. An interpretation of objects and object types. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 396–409, St. Petersburg Beach, Florida, 1996.

[5] A. Ahmed, L. Jia, and D. Walker. Reasoning about hierarchical storage. In *IEEE Symposium on Logic in Computer Science*, pages 33–44, Ottawa, Canada, June 2003.

[6] A. J. Ahmed, A. W. Appel, and R. Virga. A stratified semantics of general references embeddable in higher-order logic. In *17th Annual IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 75–86. IEEE, June 2001.

[7] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, Cambridge, England, 1992.

[8] A. W. Appel. Foundational proof-carrying code. In *Symposium on Logic in Computer Science (LICS '01)*, pages 247–258. IEEE, 2001.

[9] A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *POPL '00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 243–253. ACM Press, Jan. 2000.

[10] A. W. Appel and D. B. MacQueen. Standard ML of new jersey. In J. Maluszyński and M. Wirsing, editors, *Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming*, number 528, pages 1–13. Springer Verlag, 1991.

[11] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Prog. Lang. Syst.*, 23(5):657–683, Sept. 2001.

[12] A. W. Appel, N. Michael, A. Stump, and R. Virga. A trustworthy proof checker. In I. Cervesato, editor, *Foundations of Computer Security workshop*, pages 37–48. DIKU, July 2002. diku.dk/publikationer/tekniske.rapporter/2002/02-12.pdf.

[13] A. W. Appel and D. C. Wang. JVM TCB: Measurements of the trusted computing base of Java virtual machines. Technical Report CS-TR-647-02, Princeton University, Apr. 2002.

[14] L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies. In *Foundations of Computer Security '02 (associated with LICS '02)*, Copenhagen, Denmark, July 2002.

[15] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. B. C. Chambers, and S. Eggers. Extensibility, safety and performance in the spin operating system. In *Proceedings of the 15th Symposium on Operating System Principles*, pages 267–284. ACM Press, Dec. 1995.

[16] M. Blume and A. 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.

[17] K. B. Bruce, L. Cardelli, and B. C. Pierce. Comparing object encodings. In *Theoretical Aspects of Computer Software*, pages 415–438, 1997.

[18] P. Canning, W. Cook, W. Hill, W. Olthoff, , and J. C. Mitchell. F-bounded quantification for object-oriented programming. In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, 1989.

[19] J. Chen, D. Wu, A. W. Appel, and H. Fang. A provably sound TAL for back end optimization. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '03)*. ACM Press, June 2003.

[20] C. Colby, P. Lee, G. C. Necula, F. Blau, K. Cline, and M. Plesko. A certifying compiler for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*. ACM Press, June 2000.

[21] K. Crary. Toward a foundational typed assembly language. In *POPL '03: The 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 198–212. ACM Press, Jan. 2003.

[22] D. Dean, E. W. Felten, and D. S. Wallach. Java security: From HotJava to Netscape and beyond. In *Proceedings of 1996 IEEE Symposium on Security and Privacy*, May 1996.

[23] S. Drossopoulou, S. Eisenbach, and S. Khurshid. Is the java type system sound? *Theory and Practice of Object Systems*, 5(1):3–24, 1999.

[24] Erlingsson and Schneider. SASI enforcement of security policies: A retrospective. In *WNSP: New Security Paradigms Workshop*. ACM Press, 2000.

[25] U. Erlingsson and F. B. Schneider. IRM enforcement of java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, 2000.

[26] L. George. MLRISC: Customizable and reusable code generators. Technical report, Bell Laboratories, May 1997.

[27] R. A. Gingell, M. Lee, X. T. Dang, and M. S. Weeks. Shared libraries in sunOS. *Proceedings of the USENIX 1987 Summer Conference*, pages 131–145, 1987.

[28] J.-Y. Girard. *Interpretation fonctionnelle et elimination des coupures dans l'arithmetique d'ordre superieur*. PhD thesis, Universite de Paris VII, 1972.

[29] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification (2nd edition)*. Addison-Wesley Pub Co, 2000.

[30] N. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. In *Proc. 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*, pages 89–100, July 2002.

[31] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, Jan. 1993.

[32] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, California, 1995.

[33] C. League, Z. Shao, and V. Trifonov. Representing Java classes in a typed intermediate language. In *Proc. Int'l Conf. Functional Programming*, pages 183–196, Paris, September 1999. ACM.

[34] C. League, V. Trifonov, and Z. Shao. Type-preserving compilation of Featherweight Java. In *Proc. Int'l Workshop on Foundations of Object-Oriented Languages*, London, January 2001.

[35] X. Leroy. Unboxed objects and polymorphic typing. In *19th Annual ACM Symp. on Principles of Prog. Languages*, pages 177–188, New York, January 1992. ACM Press.

[36] A. Leung and L. George. *MLRISC Annotations*. http://cm.bell-labs.com/cm/cs/what/smlnj/compiler-notes/annotations.ps.

[37] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 2003.

[38] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1999.

[39] A. Mayer, A. Wool, and E. Ziskind. Fang: A firewall analysis engine. In *2000 IEEE Symposium on Security and Privacy*, pages 177–187, California, May 2000.

[40] N. G. Michael and A. W. Appel. Machine instruction syntax and semantics in higher-order logic. In *CADE-17: 17th International Conference on Automated Deduction*, pages 7–24. Springer-Verlag, June 2000. LNAI 1831.

[41] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, 1997.

[42] Y. Minamide, J. G. Morrisett, and R. Harper. Typed closure conversion. In *Symposium on Principles of Programming Languages*, pages 271–283, 1996.

[43] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A realistic typed assembly language. In *Second ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, 1999. INRIA Technical Report 0288, March 1999.

[44] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *POPL '98: 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–97. ACM Press, Jan. 1998.

[45] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Trans. Prog. Lang. Syst.*, 21(3):527–568, May 1999.

[46] G. Necula. Proof-carrying code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, New York, Jan. 1997. ACM Press.

[47] G. Necula and P. Lee. Safe kernel extensions without runtime checking. In *2nd USENIX symposium on Operating System Design and Implementation*, Seattle, Oct. 1996.

[48] G. Nelson, editor. *Systems Programming with Modula-3*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[49] T. Nipkow and D. von Oheimb. Java-light is type-safe — definitely. In L. Cardelli, editor, *Conference Record of the 25th Symposium on Principles of Programming Languages (POPL'98)*, pages 161–170, San Diego, California, 1998. ACM Press.

[50] L. Petersen, R. Harper, K. Crary, and F. Pfenning. A type theory for memory allocation and data layout. In *POPL '03: The 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 172–184. ACM Press, Jan. 2003.

[51] J. C. Reynolds. Towards a theory of type structure. In *Coll. sur la Programmation, Lecture Notes in Computer Science*, volume 19. Springer-Verlag, 1974.

[52] F. B. Schneider. Enforceable security policies. *Information and System Security*, 3(1):30–50, 2000.

[53] F. B. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. *Lecture Notes in Computer Science*, 2000:86–101, 2001.

[54] Z. Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation*, June 1997.

[55] Z. Shao and A. W. Appel. Space-efficient closure representations. In *LISP and Functional Programming*, pages 150–161, 1994.

[56] Z. Shao and A. W. Appel. A type-based compiler for standard ML. In *Proc. ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 116–129, La Jolla, CA, 1995.

[57] F. Smith, D. Walker, and G. Morrisett. Alias types. *Lecture Notes in Computer Science*, 1782:366+, 2000.

[58] K. N. Swadi. *Typed Machine Language*. PhD thesis, Princeton University, 2003.

[59] D. Syme. Proving java type soundness. In *Formal Syntax and Semantics of Java*, pages 83–118, 1999.

[60] G. Tan, A. W. Appel, K. N. Swadi, and D. Wu. Construction of a semantic model for a typed assembly language. Aug. 2003.

[61] D. Tarditi and A. Diwan. The full cost of a generational copying garbage collection implementation. In *OOPSLA Workshop on Memory Management and Garbage Collection*, 1993.

[62] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 181–192, 1996.

[63] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 1997.

[64] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proc. 14th ACM Symposium on Operating System Principles*, pages 203–216, New York, 1993. ACM Press.

[65] D. C. Wang and A. W. Appel. Type-preserving garbage collectors. In *POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 166–178. ACM Press, Jan. 2001.

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

[67] A. K. Wright, S. Jagannathan, C. Ungureanu, and A. Hertzmann. Compiling Java to a typed lambda-calculus: A preliminary report. In *Types in Compilation*, pages 9–27, 1998.

[68] D. Wu, A. W. Appel, and A. Stump. Foundational proof checkers with small witnesses. In *PPDP '03: The Fifth ACM SIGPLAN Internation Conference on Principles and Practice of Declarative Programming*, Aug. 2003.

[69] H. Xi and R. Harper. A dependently typed assembly language. In *2001 ACM SIGPLAN International Conference on Functional Programming*, pages 169–180. ACM Press, Sept. 2001.