Typed Machine Language

Kedar N. Swadi

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

November 2003

Abstract

With high-speed networks, mobile code applications have become common. One of the important considerations for users (code consumers) of such applications is the guarantee that the downloaded programs are safe. Proof-carrying code is one of the frameworks that allow code consumers to independently verify the safety of untrusted code.

For any such framework to be usable, however, significant trust must be placed in the correctness of that framework itself. The Proof-Carrying Code (PCC) project at Princeton addresses this issue, and aims to build a foundational PCC system that has a trusted component of a very small size. This system consists of a certifying compiler that generates the executable code, and a safety prover that generates the proof of the safety of this code.

The certifying compiler also generates hints that help the prover in generating the safety proof. These hints are in the form of a typed assembly language program that corresponds to the output object-code program. Therefore, while the hints are specified as high-level type annotations, the prover must prove the safety of machine-level programs that operate on memory and register banks.

In this thesis, I present Typed Machine Language (TML), which is used to bridge this gap. TML is a calculus of type operators that can express constructs in high-level languages like core ML at a sufficiently low level to correspond directly to the concrete realisations of these type constructs at the machine level. These operators are expressive enough to be able to allow provably safe optimisations like array-bounds-check eliminations and sum-type discriminations. I shall first present a semantic model for these operators, and typing rules based on them. This model is then used to provide models for assembly-level instructions. The typing rules,

along with the instruction models provide an interface which allows typed assembly languages to be type checked without exposing the complex underlying semantic model.

Finally, I shall present a proof technique that shows how the typability of the typed assembly languages can be connected to the safety of corresponding machine-level programs using the foundational semantic models for types and instructions provided by TML.

Acknowledgments

Many people have contributed in helping me successfully finish my dissertation at Princeton. A large part of the credit is due to my adviser Andrew Appel. I could not have hoped to have a more patient, approachable, and understanding adviser. He shall always remain a model of being a great mentor, teacher, researcher, and a manager for me. My thesis readers Amy Felty and David Walker gave very helpful, detailed, and insightful comments which undoubtedly helped me improve the quality of the thesis.

I've also been privileged to have had great colleagues in my research group. Roberto Virga has been most helpful with many theoretical and implementation problems I faced. I have also benefited a lot from my technical discussions with Amal Ahmed, Juan Chen, Neophytos Michael, Gang Tan, and Dinghao Wu. Dan Wang was always helpful with any technical problem and was a great person to exchange programming-language related rants with.

I would also like to thank all the CS graduate students who have made my stay at the department so enjoyable, especially Lujo Bauer, Amit Chakrabarti, Daniel Dantas, Georg Essl, Yefim Shuf, Iannis Tourlakis, George Tzanetakis, and Brent Waters. Thanks also to my fellow Indian graduate students Hazra, Kama, Khot, ND, NPR, Ram, Sai, Sandeep, and SR for all their valuable friendship and support, and especially to Rahul for being the most accommodating and helpful roommate.

Many other persons have helped my research life in direct and indirect ways. In particular, I would like to thank my teacher Rustom Mody for getting me interested in programming languages. I'm also grateful to Ali-Reza Adl-Tabatabai for giving me the opportunity to work for a summer at MRL, a great research lab.

Thanks are also due to Melissa Lawson for taking care of so many things right

from my first day to my last day at Princeton. Thanks also to Chris Tengi and other technical staff members for keeping all the systems running so smoothly.

Most importantly, I thank my family for making all this possible. Shantanu has been the ideal big brother: his discipline, dedication, and perseverance have always been inspiring. Thanks to him and Madhura for providing me a home away from home in the US. Finally, I dedicate this thesis to the two persons whom I owe everything: my parents, Nandkumar and Sheela Swadi.

To Aai and Baba,

for your limitless love.

Contents

List of Figures

Chapter 1

Introduction

In the past decade, high-speed networks have resulted in the widespread use of programs that exchange information between computers. For example, peer-to-peer applications like instant messaging and file swapping involve the exchange of data between possibly unknown machines. Mobile computing, which involves transfer and execution of programs through web browsers and email clients, is now very common. Collaborative computing projects like SETI@Home [7, 31] and GIMPS [1] involve downloading a program from the project website into a host computer. Resources of the host computer are used in computing data which are sent back to the project website. This idea is carried even further in Grid Computing [50], which seeks to share and aggregate computing and data storage resources to provide a view of a single virtual supercomputer shared by all users.

In all such applications, a program user has to trust that the program maintains the integrity of his or her system. For example, a file-swapping program must be trusted only to share files from a particular location. Similarly, an instant messaging program must be trusted not to leak passwords out to other users. These

applications tend to be large pieces of software, and program errors often introduce security flaws. Most of the popular instant messaging programs have had errors [1] that compromise the security by leaking passwords, allowing unauthorised file creation and program execution on host machines, spoofing usernames, etc..

For infrastructures like Grid Computing to be truly successful, users must be able to compute and share resources without having to trust the source of the programs. Proof-Carrying Code (PCC)[2] [43] is one of the frameworks which allow users to independently certify the safety of third-party programs. For techniques like PCC to work, the framework itself needs to be trusted. Errors in the framework may allow malicious programs to be flagged as being safe. One of the solutions to this problem is to minimize the size of this trusted framework. In my thesis, I describe part of a PCC framework that requires minimum amount of trust from the user.

1.1 Safety, Security, and Correctness

Trusting a program has different implications depending on user requirements. Before addressing mechanisms for trustless computing, I shall list the various notions of trust, and clarify the one that is used in this thesis.

Correctness : In the traditional sense, trusting a program implies trusting that the program is *correct*, or that it does exactly what a formal specification might require it to do. Giving a formal specification to real application programs that typically have at least tens of thousands of lines of code is not easy, so program "correctness" is often informally expressed as a program doing what you expect it to do (or

[1] A software vulnerability database maintained by SecurityFocus [2] listed more than 20 critical flaws in popular instant messaging programs in 2002

[2] While there are many research projects focusing on proof-carrying code, I shall use the unqualified term PCC to refer to Necula's PCC system.

having no surprising effects). In practical settings, proving program correctness is extremely difficult if not impossible. In the scenarios listed above, though, it is often sufficient to ensure that the program does nothing "bad" or "illegal" rather than making sure that it gives the correct answer. We need much weaker guarantees than partial or total correctness.

Security : A software component may be trustworthy if it is *secure*. Security, however, has no one universally acceptable definition. "Not only is there no such thing as 100 % security, even figuring out what 'secure' means differs according to context" [62], "There is no single definition of security" [27], "... there is no harm in being liberal about what is considered a security policy" [55]. The term security encompasses many different considerations such as proper use of resources, maintenance of privacy, anonymity, and integrity, authentication, etc..

Schneider [55] narrowed the scope of security, and defined a security policy as a set of executions, specifying security policies by predicates on sets of executions. If P is a predicate specifying a safety policy, then a program is secure if there is an enforcement mechanism that verifies that any execution of that program satisfies P. As a result, ensuring program security may require the presence of a runtime environment which enforces the security policy.

Safety : Enforceable properties for program security can be classified [55] into *safety* and *liveness* properties. Lamport [32] defined a safety property as one which states that something (bad) *will not* happen, and liveness property as one which states that something (good) *must* happen. One safety property that can be statically determined is *type safety*, which is defined by Cardelli [16] as the property stating that programs do not cause untrapped errors (execution errors that are not immediately detected). Type safety can only guarantee memory safety and data abstraction,

but in practice, these two properties prevent the program from making most illegal operations.In this thesis, I restrict program safety to only mean type safety.

1.2 Techniques for Trusting Mobile Code

There are a wide variety of techniques used for ensuring that mobile code is trustworthy. Each of these give different kinds of guarantees, and incur different program runtime costs.

Digital Signatures : A common method to instill trust in mobile code is to authenticate its source. Digital signatures use cryptographic methods to allow a user to ensure that the code comes from a well-known (and hence "trustworthy") source. This system generally[3] allows the correct identification of the source, but does not guarantee the absence of programming errors by the software developers which may result in violation of safety. Furthermore, it also severely restricts the consumer to use only software from a limited number of producers.

Hardware-based methods : Most modern processors and operating systems provide support for multiple users. Privacy of user process data is ensured by restricting processes to access their own individual address spaces. Operating systems allow processes to access shared resources only through safe system-call interfaces. These interfaces provide a simple and effective way of ensuring system safety, but system calls involve expensive context switches into the system kernel, which result in a runtime penalty. This approach places trust in the correct implementation of the kernel, and places limitations on dynamic additions to the kernel code.

Software-based methods : Software-based methods seem to give more flexible ways

[3]A recent advisory from CIAC[3] involves erroneous certificates issues by Verisign to individuals fraudulently claiming to be Microsoft employees

of enforcing system security. Broadly, they can be classified into dynamic methods and static methods.

- *Interpretation* : This dynamic method involves using a trusted program which interprets every instruction, and ensures that only the safe instructions are allowed to execute. For example, interpreters for languages like Perl provide a safe environment in which to run untrusted programs. The interpreter checks for the validity and safety of all memory accesses in the program. Perl also has a set of predefined security mechanisms (e.g. the *taint* modes) that give some security guarantees even when a program is run with increased privileges.

- *Program Modification* : We can also guarantee safety by modifying a program to make it safe before execution. For example, Software Fault Isolation [64] is a technique that involves static analysis of a program to identify potentially unsafe operations. The program is then modified by enclosing these operations within additional *sandboxing* code that prevents any unsafe actions. This incurs slight runtime penalties, and also requires trust in the program analysis and modification tools. This technique can also be applied at the source level. For example, CCured [46] adds runtime checks to C code so that memory references are guaranteed to be safe. The Naccio system [26] is a generalisation of this approach, and allows safety policies such as those that seek to impose limits on writing to files or on network bandwidth usage. One of the shortcomings of techniques like Software Fault Isolation is that the interfaces between untrusted modules become complicated. To prevent one module from corrupting the data of another module, untrusted modules must run in separate address spaces. Communication between these modules

requires techniques such as remote procedure calls (RPC) which complicate code and also incur considerable runtime costs.

- *Reference monitors* To be able to express a wider set of security policies, *monitors* can be used. Monitors inspect the code at runtime, and edit the sequence of instructions so that potentially unsafe operations are identified and prevented. The SASI system [35] implements this approach by modifying the object code to merge the object and the monitor code. Other developments like Polymer [15, 14] allow environments that can be customised to many different expressive policies. However, the addition of the reference monitor code results in runtime overheads that may be impractical for real-time and limited-resource embedded applications. Monitors can also be implemented through interpreters. In this case, the interpreters, which are complicated pieces of software, must themselves be trusted to ensure complete system safety.

- *Type-preserving and certifying compilers* : Type preserving compilers rely on the type safety of the source languages, and the fact that the type safety is preserved in each intermediate representation that different phases of the compiler may need. For example, the TIL compiler [60, 59] can give guarantees that all optimisations are type-preserving and hence safe. The Flint[57] project has a similar architecture. TAL [41] carried the TIL work further by also giving types to the target languages. Assembly programs for architectures like the x86 are given type annotations [40]. Before assembling these programs, they can be type checked with respect to these annotations to ensure safety at very low levels. Though TAL does not result in any runtime penalties, the

assembler itself must be trusted. Proof-Carrying Code goes a step further by generating proof objects for the safety of machine-level programs. Figure 1.1 shows a simple form of the PCC architecture as shown by Necula [44].

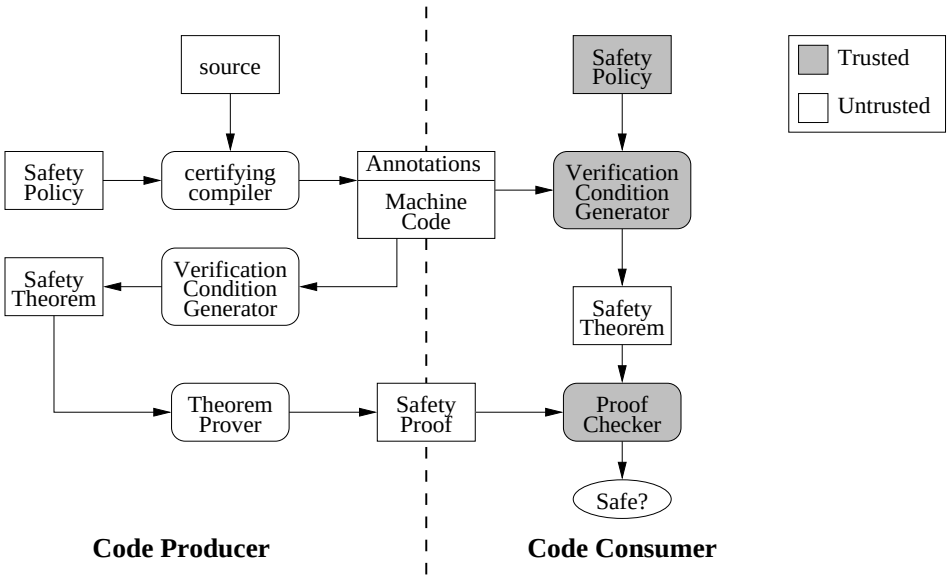

Figure 1.1: PCC Architecture

In this architecture, a code producer uses a *certifying* compiler which outputs machine code with type and dataflow annotations. These two outputs are sent to a "verification-condition generator" (VCGen) which infers the theorem of safety for the program. To ensure that this program is safe, the producer then uses a theorem prover which outputs a proof of the safety theorem. The producer then sends both the annotated machine code as well as the proof of safety to the consumer. The consumer uses the code annotations to independently generate his or her own safety theorem using the same VCGen (but now, trusted) as the producer. Since the annotations only serve as hints to the VCGen, these need not be trusted. This theorem is then checked against the proof supplied by the producer. Proof checking is computationally very

easy, and if the theorem is indeed proved by the supplied proof, the code is marked as safe, and can be executed. The consumer thus need not trust either the code annotations or the supplied proof. The Touchstone[45] certifying compiler implements this framework.

1.3 How much must a code consumer trust?

Programming-language-based techniques are extremely promising for ensuring program safety, but they still do have significantly large trusted components, or *trusted computing base* (TCB).

1.3.1 Bytecode Verification

Typed languages like Java are interpreted within a runtime environment. Appel and Wang [13] list components of a TCB for Java Virtual Machine systems like JDK [30], Kaffe[66], and Ginseng[18], etc.. Of this list, the items that make up the *safety TCB* [4] are

- the just-in-time compiler, bytecode verifier, and linker
- the garbage collector, and
- the core runtime system

Bytecode verification is performed by JVMs and just-in-time compilers to make sure that Java class files have the correct format and have valid class and class hierarchy descriptions. During linking the verifier ensures, among other things,

[4] Safety TCB measures only the core Java system, not the library base

that methods are called with appropriate arguments and fields are assigned values of appropriate types. The linker and verifier must thus be trusted to be correct. At runtime the garbage collector might relocate program data, and might therefore also change values of pointers to the new data location. Errors in the implementation of the garbage collector can lead to illegal data accesses and unsafe code execution. The garbage collector must thus be trusted for correctness. The core runtime system consists of the actual interpreter and core libraries. Errors in these components can easily compromise safety. Sun's JDK (Java 2 SDK 1.3) system has a safety TCB size of 54,200 lines of code [65].

1.3.2 TCB in type-preserving and certifying compilers

Compiler-based systems like TAL and PCC remove runtime penalties associated with interpretation, but also require TCBs of a large size.

PCC involves trusting a "Verification Condition Generator" program, which generates the safety theorem of the program. Any errors in this program would lead to an erroneous theorem that does not correspond to the actual target program. The encoding of the safety policy (which determines what conditions a safe program must satisfy) is also trusted for correctness, as is the proof checker. Errors in any of these can lead to the acceptance of unsafe programs by the framework.

The PCC-based Ginseng Java system takes x86 compiled certified binaries, and has about 25K lines of code making up the verification condition generator, logic axiom base, and the proof checker (as measured by Wang and Appel [13]). The core runtime system is another 6K lines of code. The code producer trusts and uses the Special J [19] compiler to generate the binaries, and trusts the the soundness of the type system that the compiler uses. As noted by League et al. [34], the Ginseng

system was susceptible to errors resulting from an unsound rule in the low-level type system it used.

TAL is also susceptible to the problems of a large TCB. TAL relies on a metatheorem that the type system used for annotating the assembly code is sound. Also, we must trust the type checker and the assembler to ensure that the final binary is indeed safe. The TAL system used in the Popcorn compiler [40] has a TCB consisting of 31K lines of code along with the standard C library and the OCaml and C compilers.

1.4 Contributions

I believe that the use of types imposes a natural discipline on computer programs and is by far the most promising approach to generation of safe programs. Since the PCC approach shifts the burden of safety guarantee to the code producer, it seems particularly suited for frameworks relying on trustless computing. The success of PCC depends both on the size of its TCB, as well as its practicality (in terms of safety proof sizes and proof checking times).

In this thesis, I concentrate on the first aspect, i.e. reduction of the TCB. I describe part of a *foundational* PCC (FPCC) [9] system where the TCB is reduced to the encoding of higher-order logic and basic arithmetic axioms, the description of the machine instruction semantics, and a small proof checker. The type system is removed from the TCB by expressing it in terms of a logical layer which relates it to the actual machine state. My work addresses the syntax and semantics of this layer, which is called the *Typed Machine Language* (TML).

TML has a rich set of type constructors and abstract instructions which will

allow the construction of semantic models for a wide variety of type systems. In particular, we target the LTAL type system that is used in the FPCC compiler by Chen et al. [17]. While type systems for compilers tend to be purely syntactic in nature, the semantic models for types used within them are complex, depending on many parameters like the machine semantics, the theory of recursion, the theory of mutation, and so on. TML provides its clients (like LTAL) foundational proofs for all their syntactic typing rules while still hiding all the complexities of the semantic model. This has the software engineering benefit of providing an interface between the syntactic and semantic aspects of the FPCC system.

Finally, the assembly language level at which soundness is established must also be connected to the actual machine words generated by the compiler. I shall show how TML makes this connection in an almost trustless manner (the TCB consisting of arithmetic and higher-order logic rules, along with the machine instruction specification must still be trusted), thereby removing semantic gaps that have been present in earlier approaches to language-based code certification techniques.

1.5 Organisation

This thesis is organised as follows :

- Chapter 2 gives a detailed look at Foundational PCC, and the role of TML in our FPCC system. I shall also discuss some of the other approaches towards FPCC and how they compare to the Princeton approach.
- Chapter 3 describes the syntax of type constructors in TML. I shall justify the choice of these operators, show how they can be used in proving compiler optimisations and data layout choices to be safe.

- Chapter 4 describes the semantic model for TML constructors. I shall briefly describe other semantic models on which TML is based. I shall also show how various syntactic categories such as types, type environments, and naturals are unified within the model and justify this model.

- Chapter 5 describes a kinding system over the TML type constructors. This system allows us to systematically reason about the well-formedness of complex type expressions resulting from the composition of TML constructors. This work was done with Andrew Appel and Christopher Richards.

- Chapter 6 introduces TML instructions and the semantic models for these instructions, and how they relate to assembly instructions for concrete architectures.

- Chapter 7 shows how TML is used as the layer which connects the syntactic LTAL rules to the semantics of machine-level instructions.This work was done with Andrew Appel, Gang Tan, and Dinghao Wu.

- Finally, Chapter 8 gives conclusion and future work.

Chapter 2

Foundational Proof-Carrying Code

As described in Chapter 1, the size of the trusted computing base is an important consideration for determining the effectiveness of language-based techniques for certified code. In this chapter, I shall describe in greater detail the problems associated with having large trusted components, and how a foundational approach to PCC minimizes the TCB. Finally, I shall outline the role of Typed Machine Language in this framework.

2.1 Limitations of traditional approaches

2.1.1 Type specialisation

Type systems are a crucial component of language-based safety certification techniques. For example, the safety policy in Necula's PCC system is expressed in terms of a progress and preservation theorem for programs written in a "safe assembly language" (SAL). A program in a concrete machine language is translated to an equivalent SAL program. The SAL program is then syntactically evaluated

using the static semantics given to each SAL instruction, and a safety theorem is generated for the program. Semantics for these instructions could have be given in terms of dataflow facts from which safety could be inferred. However, for large and scalable systems, it is more practical to give semantics in terms of types. Giving semantics in terms of types gives the programmer (or, in this case, the safety specifier) a more direct control over the desired properties for every instruction.[1] Therefore, the Touchstone compiler for a safe C subset (Safe-C), translates high-level C types and encodes the dataflow facts into a system of low-level types. SAL instruction are given semantics with respect to this type system. Figure 2.1 lists this low-level type system due to Necula [44].

$$\begin{aligned} \text{Type } \tau \quad ::= \quad & \texttt{int} \mid \texttt{bool} \mid \texttt{Mem} \mid \texttt{ptr}(\tau) \mid \texttt{ptropt}(\tau) \\ & \mid \texttt{array}(\tau_1, \tau_2) \mid \texttt{openarray}(\tau) \mid \texttt{pair}(\tau_1, \tau_2) \mid \mu\tau_1.\tau \end{aligned}$$

Figure 2.1: SAL types

While this list of types works very well for Safe-C, the provability of a program's safety will now depend not only on its safety, but also on the condition that the program only uses types that are a subset of those used in Figure 2.1. Consider, for example, certification of a program written in a language that allows functions with parametric polymorphism. The description of such functions would require having the universal quantification type as a primitive. Since the type system is not equipped with a "$\forall$" (universal quantifier) operator, it would not be possible to argue about the safety of programs that use this feature. Many programs written in ML, for example, cannot be certified safe in this type system.

[1] Palsberg and others have shown the equivalence of type systems having recursive, union, and intersection types with data flow analyses [48, 49], and so this does not lead to an overly conservative system.

TAL similarly has the notion of program safety tied to some particular type system. The very rich set of operators in TAL allow it to model most of the constructs found in modern programming languages. However, there are still some constructs like dependent types that TAL does not currently support. It would require an expansion to the system to be able to reason about optimisations like array-bounds check eliminations which require this feature.

In contrast, it would be ideal to have an infrastructure where the program provider could "explain" a new type system (or an extension to an already existing system) to the certification framework without having to modify it. By allowing the framework to be parametric over the type system that the programs use, it would be possible to certify programs written in new languages, and also to allow program to have components written in multiple source languages.

2.1.2 Axiomatic type systems

Normally, when constructing a typed language, the rules of the type system are not themselves machine checked for soundness, except perhaps through some semi-formal handwritten safety proof. The safety of a program that typechecks in TAL, for example, relies on the assumption that all the typing rules are sound. TAL has a sophisticated type system capable of encoding type abstractions found in many high-level type-safe languages. Using a rigorous handwritten syntactic induction proof, the soundness of this type system is proved. Manually writing proofs for soundness of large type systems is a formidable task, and prone to errors.

For example, though the soundness of the Standard ML system has a manually checked proof [38], to date, there is no machine-checked soundness proof for the complete ML type system. The proof (implemented) by Van Inwegen [61] is the

closest to a complete machine-checked proof. Furthermore, her work also uncovered problems in the type system with respect to the proof of type preservation. There has also been no type soundness proof for the complete Java system, though there has been encouraging research by Drossopoulou and others [25, 22, 23, 24] which gives soundness proofs to large subsets of Java. These proofs are very large[2] and had omissions[58] that were discovered and rectified only when the proofs were validated by means of a theorem prover. PCC and TAL used theorems such as type soundness that cannot be easily checked by a naïve code consumer, and therefore become a part of the TCB. It is necessary to be able to independently verify the soundness of the type system to have a trustworthy certifying framework. The PCC system must additionally also prove the correctness (a harder property) of the VCGen.

Systems specialised with respect to certain type systems are also not easily scalable. For example, the addition of a type constructor or modification of a typing rule requires the whole soundness proof to be reconstructed. For the PCC framework, the ability to modify the type system would require considerable effort on the part of the code consumer.

Furthermore, program safety can be guaranteed for only those programs written in type systems that are built into these frameworks. Changing or adding features to the system would also require a complete rework of the type soundness proofs.

2.2 FPCC

To address these shortcomings, Appel [9] introduced the notion of *Foundational* PCC (FPCC), where the emphasis is on building an end-to-end PCC framework

[2]Even for Featherweight Java, an extremely small Java subset, where the type system has only about 20 rules, the handwritten soundness proof[29] turns out to have a dense 10-page proof.

with the smallest TCB. I shall describe a high-level view of the FPCC certification process, with a brief description of each component of the framework.

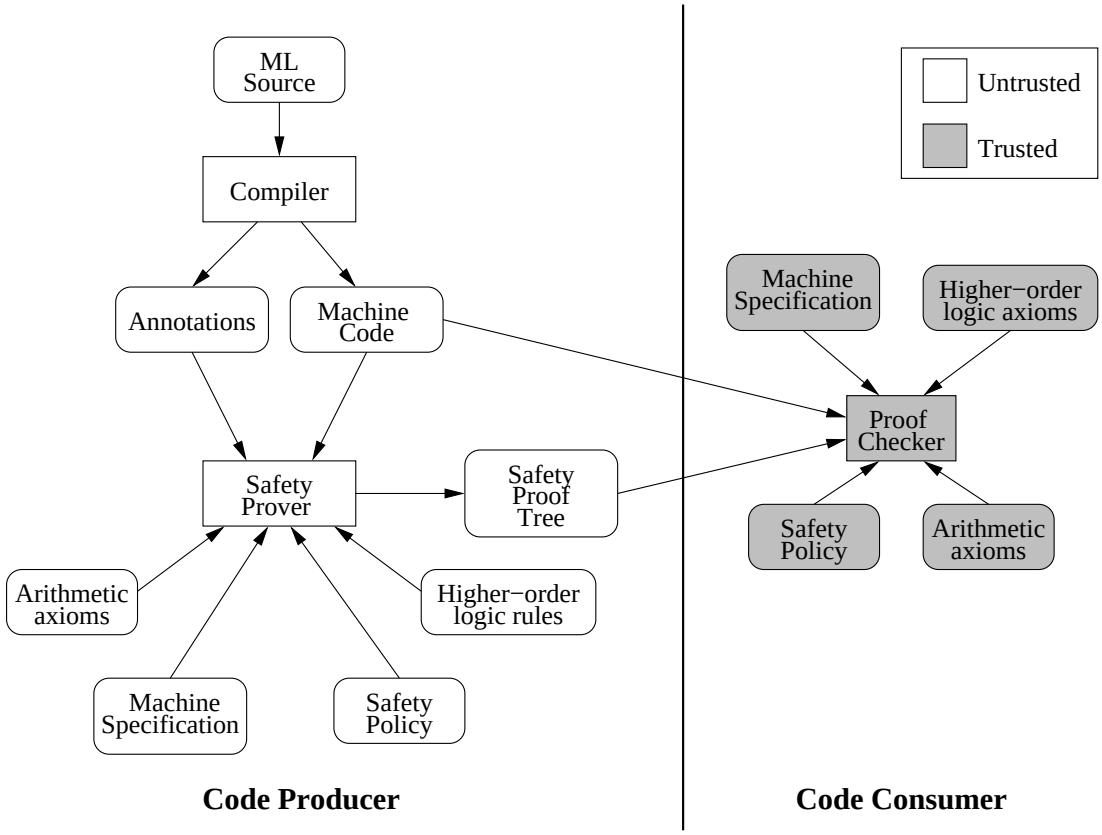

Figure 2.2: FPCC architecture

2.2.1 A semantic approach to PCC

As opposed to the syntactic-proof-based certification techniques, FPCC adopts a semantic approach to code certification. Appel and Felty[10] showed how to give semantics to types and instructions for proof-carrying code frameworks. I shall illustrate the essence of this technique using a small example.

Consider the set of type constructors in a type system shown below, with

$$\textit{Types} \quad \tau \quad ::= \quad \texttt{int} \mid \texttt{bool} \mid \texttt{pair}(\tau_1, \tau_2) \mid \texttt{box}(\tau_1)$$

base types `int` and `bool`, and the type constructors for pairing (`pair`) and memory references (`box`). The type safety rules for this language are given by :

$$\frac{v :_m \texttt{box}(\tau_1) \quad v+1 :_m \texttt{box}(\tau_2)}{v :_m \texttt{pair}(\tau_1, \tau_2)} \text{ PAIR} \qquad \frac{\texttt{readable}(m, v) \quad m[v] :_m \tau}{v :_m \texttt{box}(\tau)} \text{ BOX}$$

We write $v :_m \tau$ to mean that a value v has type τ in memory m, and readable(m, v) to mean that the location $m[v]$ is deemed readable according to the safety policy.

For this system, we assume values and locations to be natural numbers, and the memory to be a partial function from locations to values. We characterise types as predicates on values and the machine memory. The type of integers, `int`, places no restrictions on values, while a boolean value must equal either 0 or 1. Hence we define these types thus :

$$\begin{array}{rcl} \mathit{int} & = & \lambda(m, v).\texttt{true} \\ \mathit{bool} & = & \lambda(m, v).(v = 0) \vee (v = 1) \\ \mathit{box} & = & \lambda\tau.\ \lambda(m, v).\ \mathit{readable}(m, v) \wedge \tau(m, m[v]) \\ \mathit{pair} & = & \lambda(\tau_1, \tau_2).\ \lambda(m, v).\mathit{box}(\tau_1)(m, v) \wedge \mathit{box}(\tau_2)(m, (v+1)) \end{array}$$

Figure 2.3: Semantics for types and typing judgements

The semantic basis for the typing judgement that a value v has type τ in memory m is provided by a proof of the fact $\tau(m, v)$, and represented by pf $(\tau(m, v))$. The definitions above can be encoded in a logical framework such as Twelf [52] in terms of an encoding of higher-order logic as shown in Appendix A. The typing rules can then be encoded as lemmas based on these definitions, and given machine-checkable

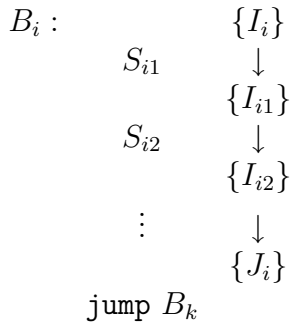

Figure 2.4: VCGen in PCC

proofs as shown in Appendix B. We similarly encode any other proofs required at various steps in proving the safety of a program, using a minimal axiom base consisting of and encoding of higher-order logic and arithmetic.

2.3 Reduced TCB in FPCC

Unlike Necula's PCC which has a large VCGen component, FPCC does not have a program to generate an explicit theorem of safety. The VCGen is a symbolic evaluator for the programs written in SAL. Every program is divided into basic blocks $(B_1, \ldots, B_n)$. Each B_i is annotated with a compiler-generated safety precondition, or invariant, I_i. This invariant is a logical predicate describing all the typing judgements that hold at the beginning of the block.

As shown in Figure 2.4, given a block B_i made up of the instruction list $\{S_{i1}, S_{i2}, \ldots, S_{in}, \texttt{jump}\ B_k\}$, the VCGen uses Hoare-logic style rules to generate the precondition J_i, just before the final control-transfer instruction of that block. If this condition is stronger than the precondition for the target block of the last instruction, i.e., if $J_i \Rightarrow I_k$, then the code satisfies all the preconditions necessary to jump to block B_k, and so it is safe to transfer control to that block.

In this process, we must trust the VCGen on two counts. First, we trust that the machine instruction decoder component of the VCGen obtains the correct assembly-level instructions from the machine code words. Second, we trust that each of the symbolic evaluation rules for assembly instruction are sound, and are correctly implemented by the VCGen.

As shown in Figure 2.2,the FPCC code producer outputs a program along with its safety proof in a way similar to Necula's PCC. However, in contrast to Necula's system, the safety theorem itself need not be generated by a program like the VCGen. Instead, in FPCC, the safety theorem is a predicate over machine words that make up the program. This predicate itself is constructed from components of the trusted computing base, and is independent of the program itself. The next few sections describe the form of this safety theorem and how it is constructed.

2.3.1 FPCC compiler

On the producer side, the source program first passes through the FPCC compiler. This prototype compiler due to Chen et al. [17] compiles core ML to Sparc machine code. In addition to this, the compiler also produces another program in LTAL [17], a low-level typed assembly language. While the LTAL program could itself be assembled and run, in our framework, it is not used with this intent. Instead, the LTAL program is really used only as a set of hints to help the prover generate the safety proof for the machine-language program. These hints take the form of type invariants (which also capture dataflow information) at various points corresponding to the generated Sparc machine code. These hints contain sufficient information for the safety prover to easily reason about any optimisations and data representation choices that the compiler might have made. Furthermore, the LTAL program is

organised in a way that allows the safety prover to be able to work in a syntax-directed way which avoids expensive backtracking proof search.

2.3.2 Safety Prover

The LTAL program hints serve as the interface between the program safety prover and the compiler. The top-level goal for the safety prover is to show that the machine program typechecks with respect to the LTAL hints. The prover then makes use of another rule which states that any program which typechecks with respect to its corresponding LTAL program is safe. Given a machine program P, and an LTAL program L, the high-level proof tree for safety is given in Figure 2.5. First, it must be shown that the bare machine words in P decode into a list of Sparc assembly instructions, A. This list represents the "untyped" view of the program. Second, it must be shown that the assembly instructions in A can be well typed. This is done by showing that they have a correspondence with the LTAL instructions in L, and respect the type annotations given for those instructions.

The safety prover itself need not be trusted to apply the correct rules. Each rule applied by the prover is justified by a lemma. These lemmas are based on the encodings of axioms of higher-order logic, axioms of arithmetic, the definition of the safety policy, and also the specification of Sparc machine instruction semantics. When the safety prover applies a rule, it essentially breaks down a proof obligation into smaller ones. The application of each such rule is justified by a lemma that shows how proofs for the smaller obligations can be combined. We can thus give an untrusted proof for the correspondence between code words in P and the assembly instructions in A, as well as the proof that the program A respects the typing annotations in L, and is thus itself well typed.

$$\dfrac{\dfrac{\begin{array}{c}P \text{ decodes to untyped assembly program } A \\ A \text{ respects typing annotations in } L\end{array}}{P \text{ typechecks with respect to } L}}{P \text{ is safe}}$$

Figure 2.5: High-level program safety proof

2.3.3 Proof Checker

The producer ships the consumer the programs L, P, and the safety proof derivation tree. The consumer must check that L and P are related as in Figure 2.5 using the derivation tree. This requires a trusted proof checker which also uses the same axioms and encodings (notice that these are trusted on the consumer side) as used by the program safety prover.

2.3.4 Connecting the compiler and the prover

The safety prover as described above must deal with two disparate views of the program. The first is the high-level view given by the LTAL program, which gives hints through a syntactic description of the typing judgements that must hold at various points in the program. The second view is the low-level instruction semantics given by the decode prover that describes concrete assembly instructions as relations on machine states.

These two views are connected using Typed Machine Language (TML). Using TML we can provide semantic models for all the LTAL operators in terms of predicates on machine states. Chapters 4 and 7 describe how LTAL programs are given foundational semantics using TML operators and instructions. Using TML also provides a layer of abstraction that allows the soundness proofs of the LTAL calculus

to be free from the complexity of the underlying semantic models.

2.4 Decreasing the size of trusted components

In addition to removing components from the TCB, the size of trusted components is also reduced in FPCC. One of the largest and most complex components of the TCB is the machine-instruction specification. Michael and Appel [37] carefully engineer the instruction decoder into a trusted "step" relation that associates each instruction with its semantics, and an untrusted "decode prover" which associates raw bit sequences with structured instructions (identifying opcodes, source and destination registers, immediate values, etc.). This allows a large part corresponding to the symbolic evaluator of the VCGen to be written as a set of lemmas based on the "step" relation, thereby reducing the size of the TCB.

On the consumer side, Appel et al. [12] also structure the proof and predicate representations in a way that would allow a very small proof checker to verify the safety proof. The size of their trusted proof checker is about 800 lines of C code. It is conceivable that a program of such a small size could be manually checked and tested for errors.

2.5 Related work

There has been some recent work in the area of foundational approaches to PCC. I shall describe some of the approaches that have been taken by others and their pros and cons vis-a-vis the approach taken by the Princeton FPCC group.

2.5.1 Syntactic approaches

As shown above, FPCC takes a *semantic* approach to proof-carrying code. As opposed to this, most other foundational PCC projects are taking the traditional syntactic approach to reason about program safety. The syntactic approaches make use of a two-stage technique to arrive at program safety proofs.

First, a type-annotated assembly-level language is specified. This language is designed with much the same design goals as TAL, so that many source languages can be expressed, and complex compiler optimisations can be accounted for using types. Expressions in this language are then given static and dynamic semantics through which a soundness theorem based on progress and preservation theorems is proved. This theorem says that a program in a well-typed state can always take an execution step (*progress*), and that the execution of a step results in a well-typed state (*preservation*).

This general assembly language is then connected to the real machine language program. For each machine instruction, it is shown that the evaluation used to express the execution step for the assembly maps correctly onto the operational semantics of the concrete machine. Figure 2.6 below due to Hamid et al. [28] illustrates this relation. If a well-typed assembly program P translates into machine state S, then program P' that results from evaluation of P translates into a machine state S' such that S' results from taking a machine execution step in state S.

In their FPCC project, Hamid et al. use a "Featherweight Typed Assembly Language" (FTAL) into which source languages are compiled. This language supports recursive types, mutable fields, and memory allocation, but does not support quantified types. The semantics of FTAL are encoded in the Calculus of Inductive

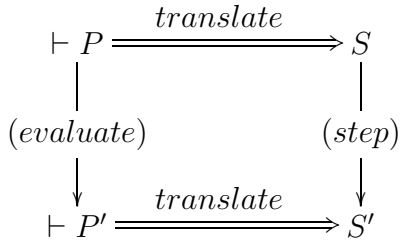

Figure 2.6: Connecting TAL evaluation to concrete machine steps

Constructions (CiC) [20, 51]. This has the advantage of being able to have inductive definitions which allows them to perform pattern matching on type terms and therefore construct syntactic proofs easily. FTAL, however, must correspond very closely to the target machine language so that the FTAL-to-machine correspondence can be easily obtained. Their system currently targets a toy machine architecture with a small instruction set.

Crary has a syntactic FPCC system that scales up to the Intel IA32 architecture. In his system, the assembly language used is "TAL Two" (TALT), which derives from the TALx86 language. While this affords a lot of expressive power, TALT must still use macro instructions like `malloc`. As a result, there is a gap between the actual machine instructions and the assembly instructions. This system has a trusted garbage collector with a formalised interface. The type safety of this system is proved using meta-theorems (in Twelf) of progress, preservation and of the program's adherence to the garbage collector interface. The Twelf metatheorem checker required to validate the safety proofs is a large and complex piece of software, and therefore using it to generate safety proofs increases the TCB size by a large factor.

Necula and Schneck [54, 47] are also developing a generic PCC system. This system has the intention of allowing the code producer to be able to specify a safety

policy for the program whose soundness can be verified by the consumer. This system, however, still must rely on a trusted VCGen, and is meant to be the first incremental step toward a truly foundational system.

Chapter 3

Typed Machine Language Syntax

3.1 Low-level types for FPCC

Chapter 2 gave an overview of a foundational PCC system. I also briefly mentioned how TML can be used as the interface between the prover and compiler. In this chapter, I shall describe the syntax of the TML type system, which is an extension of the type system described by Appel and Felty [10]. Their system lists low-level types that correspond directly to actual machine-level representations of high-level types found in general purpose programming languages. A low-level type system like this has the following benefits:

- It allows types to be described as logical predicates operating on concrete machine states. The typing rules for these types can be written and proved as lemmas over these type predicates. Therefore, the type system no longer needs to be a part of the TCB. Furthermore, a code producer need not be restricted to one particular source language; it is possible to "explain" a new type system to the code consumer without having to change the FPCC framework. In fact,

it also becomes possible to allow programs to have parts written in different type systems.

- Having low-level types allows us to describe and reason about multiple data representations for high-level source language types. A benefit of this is that programs compiled with data-representation optimisations can be proved safe.

- Small additions to the system will generally not affect any of the existing lemmas about the system and this makes it easy to have provably sound extensions to the type system[1].

The Appel-Felty type system lacked some of the type constructors required to reason about low-level optimisations and also did not deal with the construction and manipulation of type environments. The following sections of this chapter provide the syntax of TML type and environment constructors and justify the choice of low-level type constructors that make up the system. Also, the Appel-Felty system did not have an effective way to handle and reason about type expressions constructed using nested quantifiers.

3.2 Syntax

TML provides semantics to two main syntactic categories: types and instructions. This chapter deals only with type syntax. The semantics are discussed in Chap-

[1]During the development of the FPCC project, the models of types required significant changes to be able to model features such as mutation and contravariant recursive types. These changes have allowed us to model features for core ML, a realistic general-purpose language. However, the addition of objects and classes will require still more changes to the model. There were, however, many small additions to the type system that did not require a complete rework of the model. For these additions, we were able to retain all the existing lemmas.

ter 4, and instructions are discussed in Chapter 6. Figure 3.1 lists the types and environments in TML.

3.2.1 Types

While high-level languages refer to abstract expressions, which may be composed of smaller expressions, machine-level programs only refer to the realisation of these expressions in terms of pieces of code starting at a certain location, or data values. These values are held in either registers or spill locations, and are integers or memory references. Types in TML are modelled as predicates on such values. These predicates also take other arguments that are required by the semantic model, but I shall not consider them in this chapter.

Figure 3.1 gives a list of types in TML explained below:

- $\top$: Represents the most inclusive type; any value has type $\top$.
- $\bot$: Represents the "impossible" type; no value has type $\bot$.
- $\mathsf{box}\,(\tau)$: This is the type constructor for immutable memory references. Given a memory m, the value $v : \mathsf{box}\,(\tau)$ if the content of $m[v]$ satisfies the type τ.
- $\mathsf{ref}\,(\tau)$: The ref type constructor is for mutable memory references. ref is similar to box, but also offers limited mutation in the sense that any value of type τ may be stored into that memory location.
- $\mathsf{offset}(n, \tau)$: A value v has this type if $v + n$ has the type τ. This constructor is especially useful wherever address arithmetic is required. Consider, for example, a value held in a register r_1 that contains a pointer to a record with three fields $[a : \tau_1, b : \tau_2, c : \tau_3]$, each of size 4 bytes, as shown in Figure 3.2.

Kinds	κ	::=	$\mathsf{T} \mid \mathsf{N} \mid \mathsf{E}$
Types T	τ	::=	$\top \mid \bot$
		$\mid$	$\mathsf{offset}(n_1, \tau_1)$
		$\mid$	$\mathsf{box}\,(\tau)$
		$\mid$	$\mathsf{ref}\,(\tau)$
		$\mid$	$\mathsf{rec}\ \alpha.\ \tau$
		$\mid$	$\tau_1 \cup \tau_2$
		$\mid$	$\tau_1 \cap \tau_2$
		$\mid$	$\forall^{\mathsf{T}}\alpha.\ \tau$
		$\mid$	$\exists^{\mathsf{T}}\alpha.\ \tau$
		$\mid$	$\forall_B\alpha < n.\ \tau$
		$\mid$	$\exists_B\alpha < n.\ \tau$
		$\mid$	$\mathsf{codeptr}\,(\phi)$
		$\mid$	$\mathsf{int}_\pi(n)$
		$\mid$	$\mathsf{id}(\tau)$
		$\mid$	α_{T}
Environments E	ϕ	::=	$\{n : \tau\}$
		$\mid$	$\phi_1 \cap \phi_2$
		$\mid$	$\phi_1 \cup \phi_2$
		$\mid$	$\forall^{\mathsf{E}}\alpha.\ \phi$
		$\mid$	$\exists^{\mathsf{E}}\alpha.\ \phi$
		$\mid$	$\forall_{\mathrm{notindom}(n)}\alpha.\ \phi$
		$\mid$	α_{E}
Naturals N	n	::=	$\{0, 1, \ldots\}$
		$\mid$	$\mathsf{plus}(n_1, n_2)$
		$\mid$	$\mathsf{minus}(n_1, n_2)$
		$\mid$	$\mathsf{times}(n_1, n_2)$
		$\mid$	α_{N}
Relops	π	::=	$> \mid < \mid \leq \mid \geq \mid = \mid \neq$

Figure 3.1: TML Types and Environments

We could say that $r_x : \mathsf{offset}(8, \mathsf{box}\,(\tau_3))$. A convenient abbreviation, $\mathsf{field}(c, \tau)$ is the same as $\mathsf{offset}(c, \mathsf{box}\,(\tau))$.

- $\tau_1 \cap \tau_2$ is the type of a value satisfying both τ_1 and τ_2 types. The type of the register r_1 in Figure 3.2 can be given as $\mathsf{field}(0, \tau_1) \cap \mathsf{field}(4, \tau_2) \cap \mathsf{field}(8, \tau_3)$.

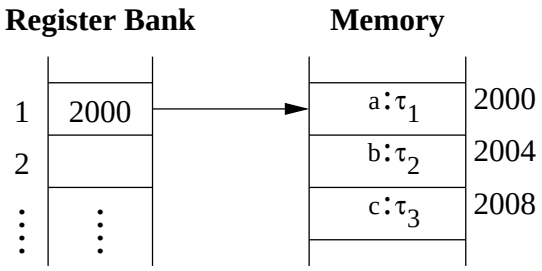

Figure 3.2: Using the offset type constructor

- $\tau_1 \cup \tau_2$ is the type of a value satisfying at least one of τ_1 or τ_2, and is useful for handling ML datatypes with multiple constructors.

- $\forall^{\mathsf{T}}, \exists^{\mathsf{T}}$: These are the universal and existential quantifiers that are essential for expressing polymorphic functions and abstract data types.

- $\forall_B \alpha < n.\ \tau, \exists_B \alpha < n.\ \tau$: These constructors give a limited form of bounded quantification. Within a type τ, the quantified variable (say, i) is a natural, and can take a value between $0 \leq i < n$. These constructors are mainly used for the purpose of encoding fixed-length arrays.

- $\mathsf{id}(\tau)$: This is the identity type constructor i.e., $\mathsf{id}(\tau) = \tau$.

- $\mathsf{rec}\ \alpha.\ \tau$: This is the constructor for general (covariant, contravariant) recursive types, where rec binds the variable α which may be free in τ (and is an alternative notation for the more familiar $\mu x.(Fx)$). For appropriate ("contractive"[2]) expressions in τ, we have that $\mathsf{rec}\ \alpha :: \kappa.\ \tau$ is a fixed point of τ. Section 3.3.2 describes how this type constructor can be used to encode linked lists.

- $\mathsf{codeptr}\,(\phi)$: This type constructor is necessary to express code pointers and

[2]As discussed in Chapter 4

function closures. codeptr takes as an argument a TML environment (described in Section 3.2.3). This environment describes all the typing invariants that must hold for the live variables so that it is safe to execute the code. Code pointers are associated with registers containing program locations (or labels). Therefore, if register x has type $\mathsf{codeptr}\,(\phi)$, then it is safe to transfer control to the program location given by the value in r_x if conditions in ϕ are satisfied by the other registers. In section 3.3.1, I shall show how function closures can be encoded using this type constructor.

- $\mathsf{int}_\pi(n)$: TML also supports a form of dependent types for integers. If v has type $\mathsf{int}_\pi(n)$, then we can say that the relation $v\ \pi\ n$ holds, where π is a relational operator. This allows us to capture dataflow facts, as discussed in Section 3.3. Using it with the $\cap$ type constructor allows us to express ranges. For example, unboxed values in SML are in the range $\{0 \ldots 255\}$, and this is expressed as $\mathsf{int}_{\geq}(0) \cap \mathsf{int}_{<}(256)$.

3.2.2 Integers

In TML, we can specify integers and primitive operations on them.

- n: This is an element in the set of natural numbers, and is used to construct offset and int_π types.

- plus, minus, times: Integers are required primarily for address arithmetic and for expressing dataflow information. For this purpose, it is sufficient to have only the basic operations like plus, minus and times. These operations are defined modulo 2^{32} due to the 4-byte size of machine memory word in our target Sparc architecture.

3.2.3 Environments

An environment is a collection of typing judgements associating each value in a list (for example, a register bank) with some type. Environments are used for expressing the codeptr constructor, and for TML instructions.

- $\{\ \}$: The empty environment places no constraints on any of the list values, i.e., all values are of type $\top$.
- $\{n : \tau\}$: In this environment, the n^{th} list value has type τ. All other list values are unconstrained.
- $\phi_1 \cap \phi_2$: We use this constructor to combine constraints on environments. For example, given a list with indices i of type τ_1 and j of type τ_2, (with i and j not necessarily distinct) we express this environment as $\{i : \tau_1\} \cap \{j : \tau_2\}$. In the presentation, we also use the syntax $\{i_1 : \tau_1,\ i_2 : \tau_2,\ \ldots,\ i_n : \tau_n\}$ for $\{i_1 : \tau_1\} \cap \{i_2 : \tau_2\} \cap \ldots \cap \{i_n : \tau_n\}$.
- $\forall^{\mathsf{E}}$, $\exists^{\mathsf{E}}$: These are the constructors for universal and existential quantifiers over environments respectively.
- $\forall_{\text{notindom}(n_1)}$: This constructor allows us to specify an environment that does not have binding for the ${n_1}^{\text{th}}$ variable. It is used for models of instructions, and is explained in detail in Chapter 6.

3.3 The choice of type constructors

The constructors shown in the earlier section are able to model the types that are commonly used in ML programs. I shall give some simple examples to illustrate this.

I shall also show how these constructors allow independence of data representation for compilers, and how they are able to encode dataflow information used in program optimisations.

3.3.1 Representing ML types using TML type constructors

The boolean type that encodes `true` and `false` values as 1 and 0 respectively may be specified simply as

$$\mathsf{bool} \stackrel{\text{def}}{=} \mathsf{int}_{=}(0) \cup \mathsf{int}_{=}(1).$$

A data structure that consists of two fields, an integer and a boolean, is represented as a boxed value. This value points to a memory location such that the it contains an integer, and the next memory location has a boolean value. This is encoded using TML types as

$$\mathsf{intBoolStruct} \stackrel{\text{def}}{=} \mathsf{field}(0, \mathsf{int}) \cap \mathsf{field}(4, \mathsf{bool}).$$

The option type for integers written in ML as

```
datatype int_option = None | Some of int
```

can be encoded in TML using the universal quantifier as

$$\mathsf{int}_{=}(0) \cup (\mathsf{int}_{\neq}(0) \cap \mathsf{box}\,(\mathsf{int}))$$

such that `Nil` is represented simply as a zero value, and `Some` is represented as a nonzero pointer to a boxed value of type `int`.

Function closures: To construct function closures, we first make a record of a code pointer and an abstracted environment. The abstraction is implemented using the $\exists$ operator, and the record is made using the field constructor. For example, given a function $f : \tau_1 \to \tau_2$ on base types τ_1 and τ_2, with a free variable environment γ, we first have a continuation-passing style[8] conversion to

$$f : (\tau_1 \times (\tau_2 \to Ans)) \to Ans$$

where Ans is the type of the return value. Then, this function is converted into a closure. The inner continuation is closure-converted into

$$\exists\, \gamma.\ ((\tau_2 \times \gamma) \to Ans, \gamma)$$

where the existential γ gives the type of the environment within which the function is evaluated. The closure is a structure with two components: the first is the CPS-converted function, and the second component is the environment. Using this closure, the entire expression is converted into the larger closure

$$\exists\, \beta.\ ((\tau_1 \times (\exists\, \gamma.\ ((\tau_2 \times \gamma) \to Ans, \gamma)) \times \beta) \to Ans, \beta).$$

We then use codeptr to model the function constructor $\to$, and field to model the structure constructor $\times$. The inner closure expression above can be expressed using TML constructors as

$$\tau_c \quad = \quad \exists^{\mathsf{T}} \alpha.\ \mathsf{field}(0, \mathsf{codeptr}\,(\{1 : \tau_2, 2 : \alpha\})) \cap \mathsf{field}(4, \alpha)$$

and the outer closure can be expressed as

$$\exists^{\top}\alpha.\ \mathsf{field}(0, \mathsf{codeptr}\,(\{1 : \tau_1, 2 : \alpha, 7 : \tau_c\})) \cap \mathsf{field}(4, \alpha)$$

3.3.2 Independence of data representation

The TML constructors allow us a great deal of flexibility for data representation. Consider, for example, a list-of-integers datatype corresponding to the SML declaration:

```
datatype List = Nil | Cons of int * List
```

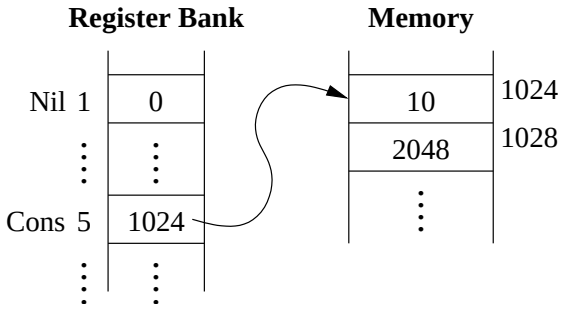

Figure 3.3: Untagged Data Representation

The compiler can choose to have either a tagged or an untagged representation for this type. In an untagged scheme (Figure 3.3) that assumes a 4-byte word size, if a register r_1 contains a list value, we can represent it as

$$r_1 : \mathsf{rec}\,\alpha.\ (\underbrace{\mathsf{int}_{=}(0)}_{\text{Nil}} \cup \underbrace{(\mathsf{int}_{\neq}(0) \cap \mathsf{field}(0, \mathsf{int}) \cap \mathsf{field}(4, \alpha))}_{\text{Cons}})$$

For the `Nil` case, the value 0 is used, and hence the left union type. For the `Cons` case, a pointer to the memory location containing a record of two fields, the

first being the data and the second being the pointer to the next cell. The right union type captures this layout.

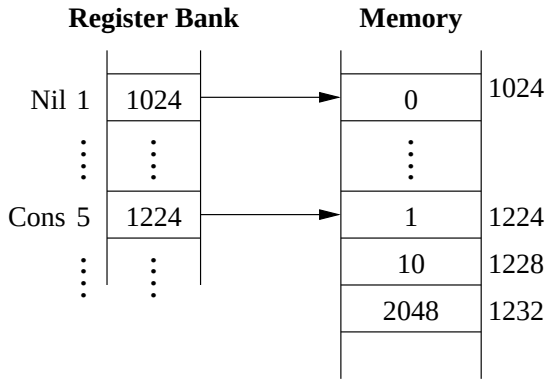

Figure 3.4: Data Representation

In a tagged representation, each variant in a sum type is uniquely associated with a *tag*, or a natural number. A machine-level representation of an instance of a sum type has this tag as its first field, followed by the particular fields for that variant.

$$r_1 : \mathsf{rec}\,\alpha.\ (\ \underbrace{\mathsf{field}(0, \mathsf{int}_{=}(0))}_{\text{Nil}}\ \cup\ \underbrace{\mathsf{field}(0, \mathsf{int}_{=}(1))\ \cap\ \mathsf{field}(4, \mathsf{int})\ \cap\ \mathsf{field}(8, \alpha)}_{\text{Cons}})$$

In the case for our list type in Figure 3.4, a nil cell has a tag of 0, and the cons cell has a nonzero (1) tag field followed by the integer data and the next-cell pointer.

3.3.3 Capturing dataflow information

An important consideration that motivated the choice of type constructors was the ability to express various kinds of invariants encountered in real programs in terms of types. Many of these invariants are required for performing provably safe

optimisations. I shall list examples of TML types that would capture information to allow provably safe optimisations. Existential, singleton, and intersection types allow us to capture dataflow information which can be used to relate two values. Consider an environment describing the contents of a register bank r used as a value list. If $r_i = r_j$, we can express this fact as

$$\exists^{\mathsf{E}}\alpha.\ \{i : \mathsf{int}_{=}(\alpha), j : \mathsf{int}_{=}(\alpha)\}$$

A more general environment constructor, relate, captures a wider class of such relations.

$$\mathsf{relate}_{\pi}(F, G)(i, j) \ \equiv \ \exists^{\mathsf{E}}\alpha.\ \{i : F\ (\mathsf{int}_{\pi}(\alpha))\} \cap \{j : G\ (\mathsf{int}_{=}(\alpha))\}$$

We use the following example to illustrate the use of relate. Let the environment $\phi_1 = \{i : \mathsf{box}\,(\tau)\}$ hold for register bank r. A load instruction $(r_j \leftarrow m[r_i])$ would fetch the contents of the memory (m) at r_i into register r_j. This operation results in a new environment, ϕ_2 for r. Trivially, $r_j = m[r_i]$ in ϕ_2. This fact is expressed using the relate constructor as $\mathsf{relate}_{=}(\mathsf{box}, \mathsf{id})(i, j)$. In the definition above, the existential is instantiated with $m[r_i]$. This instantiation can be shown to satisfy the two sub-environments, and we have $\phi_2 = \{i : \mathsf{box}\,(\tau)\} \cap \mathsf{relate}_{=}(\mathsf{box}, \mathsf{id})(i, j)$. From this environment, we can infer that the judgment $r_j : \tau$ holds in ϕ_2.

We can also use TML constructors to keep enough information to be able to allow safety proofs for programs using tagged representations for union types. Consider the list representation as shown earlier in Figure 3.3. A list of Sparc instructions as shown in Figure 3.5 might be used to load the data field of a list cell. This assembly program fragment assumes that the register %$o1$ points to

```
1              ld [%o1], %o2
2              tst %o2
3              bne cons_case ; nop
4  nil_case:   ....
5              ....
6  cons_case:     ld [%o1+4], %o3
7              ....
```

Figure 3.5: Program fragment: Safe sum-type discrimination

```
1        s = 0; i = 0;
2        while (i < N) {
3           s = s + A[i];
4           i = i + 1;
5        }
```

Figure 3.6: Program fragment: Sum of array elements

a list (possibly nil) cell. As a result we assume that the environment $\phi = \{1 : \mathsf{rec}\,\alpha.\ (\mathsf{field}(0, \mathsf{int}_{=}(0)) \cup \mathsf{field}(0, \mathsf{int}_{=}(1)) \cap \mathsf{field}(4, \mathsf{int}) \cap \mathsf{field}(8, \alpha))\}$ (where 1 corresponds to $\%o1$, 2 to $\%o2$, etc.) holds for the register bank. After the load instruction in line 1, we use the relate construct $\mathsf{relate}_{=}(\mathsf{box}, \mathsf{id})(1, 2)$ to express the fact that $m[\%o1] = \%o2$. For a cons cell, the test in line 2 would also allow us to infer that $\%o2 \neq 0$, expressed by the environment $\{2 : \mathsf{int}_{\neq}(0)\}$. Combining these two facts would allow us to infer $\{1 : \mathsf{field}(0, \mathsf{int}_{\neq}(0))\}$. Along with the environment ϕ, we can then infer that register `%o1` points to a cons cell, and prove the load instruction in line 6 to be safe.

Array bounds check eliminations: Consider the program fragment in Figure 3.6 that sums an integer array `A` into the variable `s`. In a safe language, this code would expand to include bounds checks as shown in line 3 of Figure 3.7. Using TML con-

```
1        s = 0; i = 0;
2        while (i < N) {
3          if (i >= 0 && i < N) {
4              s = s + A[i];
5              i = i + 1;
6          } else { error ("Array bounds violation"); }
7        }
```

Figure 3.7: Program in Figure 3.6 with bounds checks

structors, we can conveniently capture the dataflow information that allows us to remove these checks. After initialisation in line 1 (Figure 3.6), we have an environment with $\{i : \mathsf{int}_{\geq}(0)\}$. Combining this with the effect of the `while` condition check in line 2, we get a new environment in which $\{i : (\mathsf{int}_{\geq}(0) \cap \mathsf{int}_{<}(N))\}$. This mirrors the bounds check condition, and thus its elimination can be proved safe. The types for fixed-size arrays of length N can also be expressed as $\forall_B \alpha < N.\ \mathsf{offset}(\alpha, \mathsf{ref}\ (\tau))$. Similarly, arrays whose elements have size m can be expressed as
$\forall_B \alpha < N.\ \mathsf{offset}(\mathsf{times}(\alpha, m), \mathsf{ref}\ (\tau))$

3.4 Subtyping

One of the main purposes of TML is to act as the interface between the compiler and the prover. To be able to have this modularity, the compiler should be shielded from the complex model of types, environments, and values. In this section, I shall explain how subtyping on TML types and environments is used as a way of achieving this separation.

Informally, we say that type τ_1 is a subtype of type τ_2 ($\tau_1 \subseteq \tau_2$) if every value that has type τ_1 also has value τ_2. Equality is a two-way subtyping. The formal

definitions of subtyping and equality are presented in Chapter 4. Figures 3.9 and 3.10 give some of the subtyping and equality judgements for TML. The rules for rec also refer to the types being *wellfounded*. In Chapter 5, I shall give explanation about this requirement. In informal presentations of the subtyping rules, where convenient, I have listed rules for derived type constructors like offset.

3.4.1 Subtyping in program safety proofs

Our program safety proof technique (as will be shown in Chapter 6) uses Hoare-logic style rules to determine the safety of each basic block of a program. In traditional proofs of program correctness using Hoare logic, each statement S_i in a program is enclosed within a precondition P_i and a postcondition P_{i+1} as shown:

$$\{P_1\}$$
$$S_1$$
$$\{P_2\}$$
$$S_2$$
$$\{P_3\}$$
$$S_3$$
$$\vdots$$

The precondition P_i is the predicate that must hold for the program state before the execution of the statement S_i, and the postconditions P_{i+1} is the predicate that must hold for the program state resulting from the execution of S_i. Thus, these predicates essentially give (context-specific) semantics to the instruction S_i. For program safety proofs, where our notion of safety is given by type safety, we can

instructions	semantics	proof step
S_1	(ϕ_1, ϕ'_1)	$\phi'_1 \subseteq \phi_2$
S_2	(ϕ_2, ϕ'_2)	$\phi'_2 \subseteq \phi_3$
S_3	(ϕ_3, ϕ'_3)	$\phi'_3 \subseteq \phi_4$
$\vdots$	$\vdots$	
S_{n-1}	$(\phi_{n-1}, \phi'_{n-1})$	$\phi'_{n-1} \subseteq \phi_n$
S_n	(ϕ_n, ϕ'_n)	$\phi'_n \subseteq \phi_T$

Figure 3.8: Using subtyping rules in safety proofs

express these conditions as the typing judgements that should hold for the program states before and after the execution of an instruction.

Therefore, given a block B composed from statements $S_1, S_2, \ldots S_n$, we give each instruction semantics in terms of the type environments that should hold before and after the execution of that instruction. Consider, for example, the block in Figure 3.8. The safety proof first extracts the type semantics of each of the statements, S_i in the form of the pre- and postcondition tuple (ϕ_i, ϕ'_i). (The last statement of the block would refer to the precondition of the jump target ϕ_T.) Next, it must ensure that it is safe to execute the following instruction that has the precondition ϕ_{i+1}, i.e., the environment resulting form the execution of S_i should be compatible with ϕ_{i+1}. This check can be proved using the subtyping $\phi'_i \subseteq \phi_{i+1}$. Finally, we can ensure that it is safe to jump to the branch target using the subtyping $\phi'_{n-1} \subseteq \phi_T$. Thus, the checking of the safety of each block can be performed by the client of TML in a purely syntactic manner without having to know the underlying semantic model for types.

The rules in Figure 3.9 and Figure 3.10 can be roughly classified into the ordering rules (reflexive and transitive), rules that introduce constructors such as $\subseteq$ -BOX and $\subseteq$ -CODEPTR, rules that justify coercions such as $\subseteq$ -FOLD and $\subseteq$ -UNFOLD for

$\boxed{\tau_1 \subseteq \tau_2}$

$$\frac{}{\tau \subseteq \tau}\ \subseteq\text{-REFL} \qquad \frac{\tau \subseteq \tau'' \quad \tau'' \subseteq \tau'}{\tau \subseteq \tau'}\ \subseteq\text{-TRANS}$$

$$\frac{}{\tau \subseteq \top}\ \subseteq\text{-}\top \quad \frac{}{\bot \subseteq \tau}\ \subseteq\text{-}\bot$$

$$\frac{\tau_i \subseteq \tau_j \quad \tau_1 \subseteq \tau_2}{\mathsf{offset}(\tau_i, \tau_1) \subseteq \mathsf{offset}(\tau_j, \tau_2)}\ \subseteq\text{-OFFSET} \qquad \frac{}{\mathsf{offset}(\mathsf{int}_{=}(0), \tau) \approx \tau}\ \subseteq\text{-OFFSET\_0}$$

$$\frac{\tau_1 \subseteq \tau_2}{\mathsf{box}\,(\tau_1) \subseteq \mathsf{box}\,(\tau_2)}\ \subseteq\text{-BOX}$$

$$\frac{\text{“}\tau \text{ is wellfounded”}}{\tau\ \mathsf{rec}\,(\tau) \subseteq \mathsf{rec}\,(\tau)}\ \subseteq\text{-FOLD} \qquad \frac{\text{“}\tau \text{ is wellfounded”}}{\mathsf{rec}\,(\tau) \subseteq \tau\ \mathsf{rec}\,(\tau)}\ \subseteq\text{-UNFOLD}$$

$$\frac{}{\tau_1 \cap \tau_2 \subseteq \tau_1}\ \subseteq\text{-}\cap\text{L1} \qquad \frac{}{\tau_1 \cap \tau_2 \subseteq \tau_2}\ \subseteq\text{-}\cap\text{L2} \qquad \frac{\tau \subseteq \tau_1 \quad \tau \subseteq \tau_2}{\tau \subseteq \tau_1 \cap \tau_2}\ \subseteq\text{-}\cap\text{R}$$

$$\frac{}{\tau_1 \subseteq \tau_1 \cup \tau_2}\ \subseteq\text{-}\cup\text{R1} \qquad \frac{}{\tau_2 \subseteq \tau_1 \cup \tau_2}\ \subseteq\text{-}\cup\text{R2} \qquad \frac{\tau_1 \subseteq \tau \quad \tau_2 \subseteq \tau}{\tau_1 \cup \tau_2 \subseteq \tau}\ \subseteq\text{-}\cup\text{L}$$

$$\frac{\tau_1 \subseteq \tau_3}{\tau_1 \subseteq \tau_3 \cup \tau_4}\ \subseteq\text{-}\cup 11 \quad \frac{\tau_1 \subseteq \tau_4}{\tau_1 \subseteq \tau_3 \cup \tau_4}\ \subseteq\text{-}\cup 12$$

$$\frac{\tau_1 \subseteq \tau_3 \quad \tau_2 \subseteq \tau_4}{\tau_1 \cap \tau_2 \subseteq \tau_3 \cap \tau_4}\ \subseteq\text{-}\cap 1$$

$$\frac{\tau_1 \subseteq \tau_2}{\forall(\tau_1) \subseteq \forall(\tau_2)}\ \subseteq\text{-}\forall \qquad \frac{\tau_1 \subseteq \tau_2}{\exists(\tau_1) \subseteq \exists(\tau_2)}\ \subseteq\text{-}\exists$$

$$\frac{\phi_2 \subseteq \phi_1}{\mathsf{codeptr}\,(\phi_1) \subseteq \mathsf{codeptr}\,(\phi_2)}\ \subseteq\text{-CODEPTR}$$

Figure 3.9: Subtyping rules

$\boxed{\tau_1 \subseteq \tau_2}$

$$\frac{\tau_1 \subseteq \tau_2}{\{i : \tau_1\} \subseteq \{i : \tau_2\}} \subseteq\text{-SIN}$$

$$\frac{\mathsf{int}_<(i) \subseteq \mathsf{int}_<(k) \quad \mathsf{int}_<(k) \subseteq \mathsf{int}_<(j)}{\mathsf{int}_<(i) \subseteq \mathsf{int}_<(j)} \subseteq\text{-int\_<}$$

$$\frac{}{\mathsf{int}_=(i) \subseteq \mathsf{int}_\leq(i)} \subseteq\text{-int\_}\pi \qquad \frac{}{\mathsf{int}_=(i) \subseteq \mathsf{int}} \subseteq\text{-INT\_WEAK} \qquad \frac{\tau \subseteq \mathsf{int}_>(0)}{\tau \subseteq \mathsf{int}_{\neq}(0)'} \subseteq\text{->\_}\neq 0$$

$$\frac{\tau \subseteq \tau' \quad \tau' \subseteq \tau}{\tau \approx \tau'} \subseteq\text{-eq} \qquad \frac{\tau \approx \tau'}{\tau \subseteq \tau'} \subseteq\text{-eq\_e1} \qquad \frac{\tau \approx \tau'}{\tau' \subseteq \tau} \subseteq\text{-eq\_e2}$$

Figure 3.10: Subtyping rules (continued)

recursive types, and $\subseteq$ -$\cap$L1, $\subseteq$ -$\cap$R, $\subseteq$ -$\cup$R1, and $\subseteq$ -$\cup$L for intersection and union types. The rule $\subseteq$ -$\cap$SIN allows us to determine subtyping relations over single-value environments. Other rules such as $\subseteq$ -int-$<$ and $\subseteq$ -INT-WEAK are meant for integer (in)equalities that that arise in proving optimisations such as sum-type discriminations and array-bounds-check eliminations.

3.5 TML as a semantic basis

The type and environment constructors shown above include bounded quantifications and singleton types. Such constructors make type inference undecidable in general. However, TML is not intended to be used directly as a type system for a language, rather it is meant to provide a semantic basis for a higher-level language for which type checking would be decidable. In the FPCC project, we use the typed assembly language LTAL whose type constructors are translated into corresponding TML constructors. All the TML constructors have semantics in terms of the

machine state. Using these semantics, the typing rules in LTAL can be justified as higher-order logic lemmas, thus allowing the construction of foundational safety proofs.

3.5.1 Completeness of syntactic rules for the TML system

The TML rules (e.g. subtyping listed in Figure 3.9) are not meant to be exhaustive. There are other true subtyping facts that cannot be proved using the rules listed. Also, in the presence of union, intersection, singleton, and dependent types, we cannot expect to have completeness with respect to any set of rules.

Instead of trying to specify and semantically prove every type rule that holds in TML, for FPCC, we have proved and incorporated rules in the TML system on a by-need basis. The rules shown are those that were needed by LTAL, the immediate client of TML. Since these rules are targetted specifically for one particular language and one set of optimisations, additional rules may be needed for each new optimisation.

Chapter 4

Semantic Models for TML Types

The syntax for TML as shown in Chapter 3 described type constructors that would be used by the compiler for FPCC to interact with the prover. To generate foundational safety proofs of programs that use these type constructors, we must have semantic models for these constructors. In this chapter, I shall describe in detail the semantic model for all the described TML syntactic categories. The semantic model for TML types and environments is based on the indexed model of types by Appel and McAllester [11]. While there have been many models of types [56, 36, 39] they have not been particularly suitable for an application like PCC. Most of these models operate on lambda calculus, with abstract specifications of type constructors. On the other hand, FPCC requires semantic proofs based on machine-level representations of types, and models of types that work for Von Neumann machines (in our case, Sparc). Another model, the PER model by Mitchell and Viswanathan [39, 63] is not suitable for reasoning about programs running on concrete machine architectures because their theory is based on programs running on Turing machines. I shall now describe a semantic model of TML followed by some modifications that

$$
\begin{array}{rcl}
\mathsf{const}(n) & \equiv & \lambda(a,m).\,\lambda v.\,v = i \\
\mathsf{char} & \equiv & \lambda(a,m).\,\lambda v.\,0 \leq v < 256 \\
\mathsf{box} & \equiv & \lambda\tau.\,\lambda(a,m).\,\lambda v.\,v \in a \wedge \tau\,(a,m)\,(m(v)) \\
\mathsf{offset} & \equiv & \lambda i.\,\lambda(a,m).\,\lambda v.\,\tau\,(a,m)\,(v+i) \\
\mathsf{intersection} & \equiv & \lambda(\tau_1,\tau_2).\,\lambda(a,m).\,\lambda v.\,\tau_1\,(a,m)\,v \wedge \tau_2\,(a,m)\,v \\
\mathsf{exists} & \equiv & \lambda F.\,\lambda(a,m).\,\lambda v.\,\exists\tau.\mathrm{valid}(\tau) \wedge F\,\tau\,(a,m)\,v \\
\mathsf{forall} & \equiv & \lambda F.\,\lambda(a,m).\,\lambda v.\,\forall\tau.\mathrm{valid}(\tau) \implies F\,\tau\,(a,m)\,v \\
\mathsf{rec} & \equiv & \lambda F.\,\lambda(a,m).\,\lambda v. \\
 & & \quad \forall\tau.\mathrm{valid}(\tau) \implies \\
 & & \quad\quad (F\tau\,(a,m)\,v \implies \tau\,(a,m)\,v) \implies \tau(a,m)v
\end{array}
$$

Figure 4.1: Semantic definitions for types due to Appel and Felty

allow us to express TML environments.

4.1 The Appel-Felty Model

The Appel-Felty system [10] model supports basic types like integer constant (constty) and character (char). Immutable memory references are described using boxed type (box(τ)). Union (union(τ_1, τ_2)) and intersection intersection(τ_1, τ_2) types, along with the offset (offset i τ) type can be used to describe sum types and structures. The existential (existential(F)) and universal (universal(F)) quantifier types give the ability to describe abstract data types and polymorphic functions respectively. These types describe machine-level representations of values. Therefore, we may have machine-layout-dependent typing rules such as CHAR, BOX, and RECORD2 as

shown below.

$$\frac{0 \leq v < 256}{v :_m \text{char}} \text{ CHAR}$$

$$\frac{m[v] :_m \tau}{v :_m \text{box}(\tau)} \text{ BOX}$$

$$\frac{v :_m \text{box}(\tau_1) \qquad v :_m \text{offset}(1, \text{box}(\tau_2))}{v :_m \text{record2}(\tau_1, \tau_2)} \text{ RECORD2}$$

The semantic model of types by Appel and Felty which gives justification to the typing rules shown above describes types as predicates on values. Real-life programs store values in memory and rely on a model of memory management, which must keep track of locations that are already allocated and those that are free. The memory, denoted by m, is modelled as a total[1] function from numeric locations to numeric values, while the allocated set, denoted by a, is modelled as a set of locations. Types are defined as functions on a, m, and the value v; the semantics for the typing judgement $v :_{a,m} \tau$ is simply $\tau\,(a, m)\,v$. As an example, the type constructor $\mathsf{const}(n)$ does not depend on the contents of the memory, and is defined as

$$\mathsf{const}(n) \stackrel{\text{def}}{=} \lambda(a, m).\, \lambda v.\, v = n,$$

while the type constructor for references also refers to the memory and allocated set, and is defined as

$$\mathsf{box}(\tau) \stackrel{\text{def}}{=} \lambda(a, m).\, \lambda v.\, (v \in a) \ \wedge \ \tau\,(a, m)\,(m(v)),$$

where the first condition ensures that only allocated memory locations are accessed.

[1]Representations of total and partial functions are different when realised as logical definitions. We model the memory as a total function, and we assume invalid memory locations to hold some arbitrary value.

Figure 4.1 shows definitions of a larger set of type constructors in this model.

In this model, it is necessary to be able to prove that a typing judgement for an existing value continues to hold when new values are added or initialised. For this purpose, and also to be able to give correct semantics to type constructors like quantifiers which involve functions on types, this model also defined the notion of validity of types. A type τ is valid if any value that has type τ continues to have that type when additional values are allocated in the memory, i.e., a valid type must be invariant under an extended allocated set. Furthermore, any change to memory not in the allocated set should not affect any existing typing judgements. Validity is formally defined as

$$\begin{aligned}\text{valid}(\tau) \stackrel{\text{def}}{=}\ & \forall a, a', m, v.((a \subset a') \wedge \tau(a, m)v) \implies \tau(a', m)\, v \ \wedge \\ & \forall x, a, m', m, v.((x \in a \implies m(x) = m'(x)) \wedge \tau(a, m)\, v) \implies \tau(a, m')\, v\end{aligned}$$

In this model, the definition of the recursive type constructor rec was sufficient to model covariant recursive types. For example, it can model types such as

```
datatype expr = CONST of int
              | PLUS of expr * expr.
```

However, this model did not have the appropriate semantics for contravariant recursive types such as

```
datatype expr = CONST of int
              | PLUS of expr * expr
              | FUN of expr -> expr.
```

that has occurrences of `expr` on the left of the function arrow. As a result, general function closures (that realise the `FUN` variant) cannot be correctly modelled. The indexed model of types by Appel and McAllester [11] is able to solve this problem,

$$
\begin{array}{rcl}
\mathsf{decode} & \stackrel{\text{def}}{=} & \lambda\ m, l, \mathsf{instr}. \\
& & \text{“}\textit{bit pattern at}\ \ m[l]\ \ \textit{corresponds to instruction}\ \mathsf{instr}\text{”} \wedge \\
& & \big((\mathsf{instr} = \mathtt{add})\ \vee (\mathsf{instr} = \mathtt{addi})\ \vee (\mathsf{instr} = \mathtt{load}) \vee \\
& & (\mathsf{instr} = \mathtt{store})\ \vee (\mathsf{instr} = \mathtt{beq})\ \vee \ldots\big) \\
\mapsto & \stackrel{\text{def}}{=} & \lambda(r, m), (r', m'). \\
& & \exists\, i.\ \mathsf{decode}(m, r_l, i)\ \wedge i([l := l + 4]r, m, r', m')
\end{array}
$$

Figure 4.2: A simplified definition of the machine-step relation due to Michael and Appel[37]

and is described next.

4.2 Indexed Model of Types

The indexed model of types gives semantics to types in term of safety of future execution steps (this characterisation is most obvious in the definition of the code-pointer type constructor), and I shall start the description of this model with the formal definition of safety for FPCC.

A concrete machine state is the tuple (r, m) formed from the machine register bank and memory. A state is *stuck* if there is no next state that the machine can step to. A state is safe if it never leads to a stuck state and is formally defined as :

$$
\begin{array}{rcl}
\text{safe}(r, m) & \stackrel{\text{def}}{=} & \forall (r', m').\, (r, m) \mapsto^* (r', m')\ \Longrightarrow \\
& & \qquad \exists (r'', m'').\, (r', m') \mapsto (r'', m'')
\end{array}
$$

The machine step relation is architecture specific, and defines the result of execution of one machine instruction for any given state. The exact definition of this function for the Sparc architecture is given by Michael and Appel [37], and is, in part, a component of the FPCC trusted computing base. Figure 4.2 gives a flavour

of the actual definition for $\mapsto$. The derived notion of a multi-step ($\mapsto^*$) denotes the transitive closure of zero or more execution steps. It is often convenient (as will be seen later) to express safety for only k future program steps ($\mapsto^k$) as shown by the predicate safen defined as :

$$\begin{aligned}\mathsf{safen}(k, r, m) \stackrel{\text{def}}{=}\ & \forall\, j.\ (j < k) \implies \\ & \forall\, (r', m').\, (r, m) \mapsto^j (r', m') \implies \\ & \exists\, (r'', m'').\, (r', m') \mapsto (r'', m'')\end{aligned}$$

As opposed to the Appel-Felty model, in the indexed model a type τ is not just a set of values satisfying a particular predicate, but a family of sets of values indexed by k. In the judgement $v :_k \tau$, the index k represents how "close" v is to a value that actually satisfies τ, i.e. it represents how many machine steps can be taken using v as a value of type τ before the machine enters a "stuck" state. Consider, for example, a memory m, with register 1 supposed to point to a value of type $\tau = \mathsf{box}\,(\mathsf{box}\,(\mathsf{const}(42)))$. If $r_1 = 0$, register 1 does not satisfy this type at all (we assume zero to be an invalid address); nevertheless, it is safe to execute zero machine steps using this value, and hence the value 0 has the type τ to approximation $k = 0$.

Assuming that $r_1 = 1000$, register 1 may be attributed the type τ to various approximations as shown in Figure 4.3 In Figure 4.3(a), we may index the memory at location 1000, but not at location given by $m[1000]$, and hence register 1 has type τ to approximation 1. Figures 4.3(b) and 4.3(c) both allow two memory indirections (a load instruction) to be executed before any program can realise the error with the values (that are not equal to 42), and hence they both have register 1 having type τ to the approximation $k = 2$. Figure 4.3(d) does actually have a type-correct value, hence the register 1 has the type τ to approximation k for all values of the

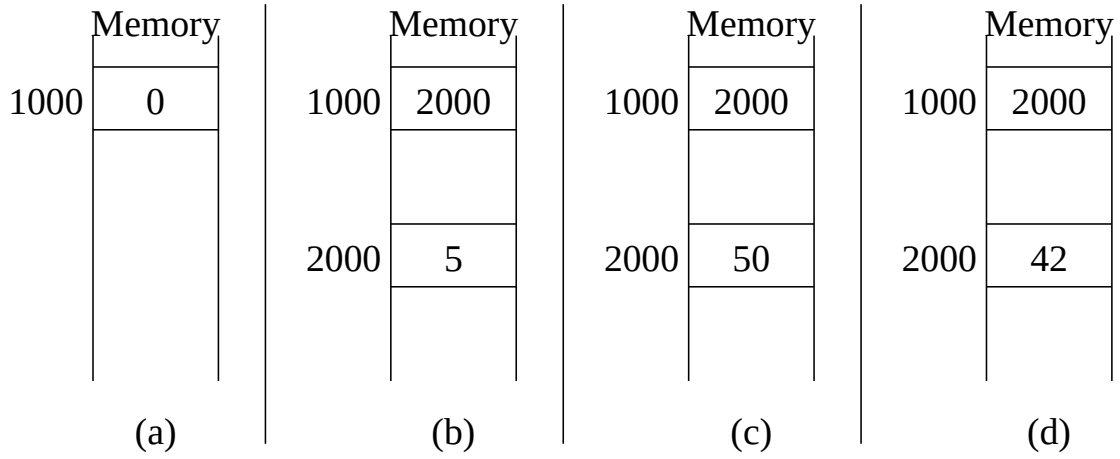

Figure 4.3: Values with approximations

index k.

The type definitions shown in Figure 4.1 are now modified to the indexed model definitions and are shown in Figure 4.4. The definition of codeptr, which connects the indices to the execution steps can be explained thus: if the program counter points to some program code location v (i.e. if the control has already passed to the code), such that the machine state satisfies the type environment ϕ to approximation $j < k$, then it would always be safe to execute at least $k - 1$ instructions starting at location given by the value v. The value v itself is in some register, and so it would further take at least another instruction to transfer control to location v (using a jump or branch instruction). Therefore, it would be safe to execute at least k instructions, i.e., v would satisfy the code pointer type to approximation k.

4.3 Semantic model for TML

I shall now present some of the modifications that were done to this model to be able to express some of the type constructors in TML.

$$
\begin{aligned}
\mathsf{const} &\equiv \lambda n.\, \lambda(a,m).\, \lambda k.\, \lambda v.\, v = n \\
\mathsf{char} &\equiv \lambda(a,m).\, \lambda k.\, \lambda v.\, 0 \le v < 256 \\
\mathsf{box} &\equiv \lambda \tau.\, \lambda(a,m).\, \lambda k.\, \lambda v.\, v \in a \wedge \tau\,(a,m)\,(k-1)\,(m(v)) \\
\mathsf{offset} &\equiv \lambda i.\, \lambda(a,m).\, \lambda k.\, \lambda v.\, \tau\,(a,m)\,k\,(v+i) \\
\mathsf{intersection} &\equiv \lambda(\tau_1,\tau_2).\, \lambda(a,m).\, \lambda v.\, \tau_1\,(a,m)\,k\,v \wedge \tau_2\,(a,m)\,k\,v \\
\mathsf{exists} &\equiv \lambda F.\, \lambda(a,m).\, \lambda k.\, \lambda v.\, \exists \tau.\mathrm{valid}(\tau) \wedge F\,\tau\,(a,m)\,k\,v \\
\mathsf{forall} &\equiv \lambda F.\, \lambda(a,m).\, \lambda k.\, \lambda v.\, \forall \tau.\mathrm{valid}(\tau) \implies F\,\tau\,(a,m)\,k\,v \\
\mathsf{rec} &\equiv \lambda F.\, \lambda(a,m).\, \lambda k.\, \lambda v.\, (F^{k+1}\bot)\,(a,m)\,k\,v \\
\mathsf{codeptr} &\equiv \lambda \phi.\, \lambda(a,m).\, \lambda k.\, \lambda v. \\
&\qquad \forall j, r', (a',m').\ \ j < k && (1) \\
&\qquad\quad \wedge\ \text{“}(a',m')\ \textit{extends}\ (a,m)\text{”} && (2) \\
&\qquad\quad \wedge\ r'_{pc} = v && (3) \\
&\qquad\quad \wedge\ \text{“}(r',m')\ \textit{satisfy environment}\ \phi\ \textit{to approximation}\ j\text{”} && (4) \\
&\qquad \implies \mathrm{safen}(j,r',m')
\end{aligned}
$$

Figure 4.4: Definitions for types in the indexed model[11]

In Chapter 3, the constructors for recursive and quantifier types were shown using explicit bound variables. In concrete implementations, however, it is more convenient to use de Bruijn indices to keep track of the quantified variables within type expressions. Using de Bruijn indices allows implementations to avoid the use of type functions for implementing the bound type expressions and hence avoids the requirement for higher-order kinds. The occurrences of de Bruijn indices inside type expressions are denoted by $\underline{n}$ for the n^{th} outermost bound variable. To map the de Bruijn indices to their types, the semantic model needs to have a type context, and this is denoted by ρ. Types in the context ρ themselves may not have any de Bruijn indices, i.e., they must be *closed*. We make use of the work on explicit substitutions [42, 4] to perform beta reduction manipulations on these type expressions.

For TML semantics, the root value v and the machine state is now combined into a single tuple $cv = (s, v)$, called a *concrete value*, where the first component s

encompasses the register bank contents, memory contents, and the allocation set. We use the functions state(cv) and root(cv) to extract the two components of a concrete value. The function $\langle s, v \rangle$ constructs the concrete value given the two components. A type τ, based on models for the context, index, and concrete values, is modelled as

$$\tau \quad = \quad \lambda\, \rho.\, \lambda\, k.\, \lambda\, cv. \ldots$$

where ρ is the type context, k is the approximation index, and cv is the concrete value.

While Figure 3.1 shows the existential and universal quantifiers to bind variables of any kind, in the semantic model, we require these quantifiers to constrain the kinds of the bound variables. The bound variables for type and environment quantifiers may only be of kind Type or kind Natural. This constraint may be implemented through different flavours for each quantifier such as $\exists^{\mathsf{T}}_{\mathsf{T}}$, $\exists^{\mathsf{T}}_{\mathsf{N}}$, $\exists^{\mathsf{E}}_{\mathsf{N}}$, and $\exists^{\mathsf{E}}_{\mathsf{T}}$, and $\forall^{\mathsf{T}}_{\mathsf{T}}$, $\forall^{\mathsf{T}}_{\mathsf{N}}$, $\forall^{\mathsf{E}}_{\mathsf{N}}$, and $\forall^{\mathsf{E}}_{\mathsf{T}}$, where the subscript denotes the kind of the bound type variable, and the superscript denotes the kind of the full type expression.

To be able to correctly implement these quantifiers in the explicit substitution calculus, we would require just as many kinds of de Bruijn indices, and therefore as many contexts ($\rho^{\mathsf{T}}_{\mathsf{T}}$, $\rho^{\mathsf{T}}_{\mathsf{N}}$, $\rho^{\mathsf{E}}_{\mathsf{N}}$, and $\rho^{\mathsf{E}}_{\mathsf{T}}$). The existence of multiple contexts is a problem, and therefore the model must be modified to avoid having more than one context.

One of the problems with multiple contexts is that for each kind of bound variable, we require a different version of every operator that is used to manipulate the de Bruijn indices of that kind in type expressions. Any lemmas based on these operators would similarly need to have different versions for each kind. Lemmas that deal with each quantifying operator (such as introduction and elimination lem-

mas) would also need to have different versions. It is not possible in our logic to have definitions and lemmas that can be cleanly abstracted over these kinds, and so efforts would have to be duplicated. For example, in the example below, a type and a number quantifier type would be modelled as the predicates shown below.

$$
\begin{aligned}
\forall^{\mathsf{T}}_{\mathsf{T}} &= \lambda\,\tau.\ \lambda\,(\rho_\tau, \rho_n).\,\lambda\,(m, a).\,\lambda\,k.\,\lambda\,cv. \\
&\qquad \forall\,\tau_2.\ \ldots\tau\ (\boxed{\tau_2 :: \rho_\tau}, \rho_n)\ (m, a)\ k\ cv\ \ldots \\
\forall^{\mathsf{T}}_{\mathsf{N}} &= \lambda\,\tau.\ \lambda\,(\rho_\tau, \rho_n).\,\lambda\,(m, a).\,\lambda\,k.\,\lambda\,cv. \\
&\qquad \forall\,n.\ \ldots\tau\ (\rho_\tau, \boxed{n :: \rho_n})\ (m, a)\ k\ cv\ \ldots
\end{aligned}
$$

The two predicates above add elements to different type contexts. To encode both these requirements in a single universal quantifier definition would involve making a case analysis over the kind of the variable. Since our implementation in Twelf does not allow us to write such intensional definitions, we are forced to have different quantifiers depending on the kind of the quantified variables. For each kind κ_1 that a quantified variable can have, we would require a different context ρ_{κ_1}, and for n such kinds, we would require n^2 quantifiers (indexed by the kind of the quantified variable, and the kind of the entire type expression).

In order to reduce this complexity, we made a design choice to have only one metalogical representation for all the kinds in TML that would encompass types for all machine-resident values. This is the kind for environments, and it subsumes all other kinds. The model for environments in TML can be given as shown in Figure 4.5.

With this scheme, we now have only one set of quantifiers, de Bruijn indices constructors, and related lemmas. I shall now describe how types of each kind previously defined are encoded as TML environments, (the kind for which is henceforth

Approximation Index	Index	=	Nat
Root Value	Root	=	Nat → Nat
Concrete Value	CV	=	(State, Root)
de Bruijn Context	Context	=	Nat → Index → CV → o
TML Environment	Tymap	=	Context → Index → CV → o

Figure 4.5: Semantic model for TML environments

referred to as Tymap). In later sections of this chapter, I shall discriminate Tymap terms into "TML types"(τ) and "TML environments"(ϕ) and "TML numbers" (n) wherever it aids explanation. All the type constructor semantics shown below are implemented in the Twelf system, but the explanation will not refer to the Twelf syntax.

Environments: To be able to correctly define environments, we must redefine our notion of values. In Appel-McAllester's semantics, a value was a number that represented contents of only *one* register or only *one* root pointer to some structure held in memory. While this characterisation was enough to express individual typing judgements, it wasn't enough to capture general dataflow information. Consider, for example, an environment in which two values v_1 and v_2 of type int are equal. Expressing this environment would require us to encode three facts : *i*) v_1 : int, *ii*) v_2 : int, and *iii*) $v_1 = v_2$. In TML, our intention is to be able to express all of these, especially the last one (and similarly other constraints of the form $f(v_1, v_2)$ for some relation f) as typing judgements. If each typing judgement relates to only one value, expressing these constraints becomes impossible. To be able to handle this problem, we change the model of the root of a value from a scalar to a vector, and represent roots as partial functions of type $Nat \rightarrow Nat$. Each index into this vector allows us to access one individual component of an environment. Using a vector, we can have another type, eq12 $\stackrel{\text{def}}{=} \lambda\,\rho.\lambda\,k.\ \lambda\,v.\ v(1) = v(2)$ and thus encode the third

fact. I shall now give the definitions for each of the environment constructors.

- The empty environment $\{\ \}$ is defined as

$$\{\ \} \stackrel{\text{def}}{=} \lambda\,\rho.\ \lambda\,k.\ \lambda cv.\ \text{true}$$

  Since an empty environment places no constraints on any of its components, and therefore any vector value satisfies this types. Hence, it is defined as just "true".

- The singleton environment $\{i : \tau\}$ defines typing for only one particular component of the vector. Due to the unification of kinds we must now write the singleton environment as $\{\tau_1 : \tau_2\}$, such that the first environment τ_1 effectively represents the number i and the second environment represents the type τ. For representing i, we use the function const-vector(n) that allows us to make a vector with every component equal to n.

$$\text{const-vector} \stackrel{\text{def}}{=} \lambda\,n.\ \lambda\,i.\ n$$

  Using this function, we define $\{\tau_1 : \tau_2\}$.

$$\begin{aligned}\{\tau_1 : \tau_2\} \stackrel{\text{def}}{=}\ & \lambda\,\rho.\ \lambda\,k.\ \lambda cv. \\ & \exists\, i, n.\ \text{root}(cv)(i) = n && (1)\\ & \quad \wedge\ \tau_1\ \rho\ k\ \langle \text{state}(cv), \text{const-vector}(i)\rangle && (2)\\ & \quad \wedge\ \tau_2\ \rho\ k\ \langle \text{state}(cv), \text{const-vector}(n)\rangle && (3)\end{aligned}$$

  The first conjunct in the definition requires the vector cv to have value n at some index i. The second conjunct ensures that $\tau_1 = i$. Finally, the

third conjunct ensures that every component of the type environment τ_2 is satisfied by the value n — in effect treating τ_2 like a simple type rather than an environment. The const type constructor (described later) allows a number i to be converted to the form of a type. To describe such a context, we normally make use of the $\mathsf{const}(i)$ constructor and encode the typing constraint about the i^{th} environment component as $\{\mathsf{const}(i) : \tau\}$.

- Intersection and union environment are defined to be the conjunction and disjunction of the two components respectively.

$$\begin{aligned} \tau_1 \cap \tau_2 &\stackrel{\text{def}}{=} \lambda\,\rho.\ \lambda\,k.\ \lambda cv.\ (\tau_1\ \rho\ k\ cv)\ \wedge\ (\tau_2\ \rho\ k\ cv) \\ \tau_1 \cup \tau_2 &\stackrel{\text{def}}{=} \lambda\,\rho.\ \lambda\,k.\ \lambda cv.\ (\tau_1\ \rho\ k\ cv)\ \vee\ (\tau_2\ \rho\ k\ cv) \end{aligned}$$

  We expect these constructors to be generally used with arguments made from singletons (e.g. $\{i : \tau_1\} \cap \{j : \tau_2\}$)

Numbers : By using the right concrete values, we convert numeric constants and operators into TML environments.

- const creates a type corresponding to a numeric constant. Being a constant, a value of this type should hold for any approximation, and should also not depend on the context ρ. For a constant value, all components of the value root vector, in particular, the 0^{th} component should have the the desired constant value. The constructor is therefore defined as

$$\begin{aligned} \mathsf{const} \stackrel{\text{def}}{=}\ & \lambda\,n.\ \lambda\,\rho.\ \lambda\,k.\ \lambda cv. \\ & \text{root}(cv)(0) = n \end{aligned}$$

  Using const-vector, we can easily make a value of const type.

- Instructions such as add may operate on source registers that have types such as $\tau_1 = \mathsf{int}_=(n)$ and $\tau_2 = \mathsf{int}_=(m)$. To correctly capture the type of the destination register, we require a type that expresses the addition of n and m. This is effected through arithmetic operations on types such as plus, minus, and times. For example, the result of the above addition can be expressed through $\mathsf{plus}(\tau_1, \tau_2)$. These operations are created using a more general type constructor, arith, defined as

$$\begin{aligned}
\mathsf{arith} \stackrel{\text{def}}{=}\ & \lambda\, aop.\ \lambda\, \tau_1.\ \lambda\, \tau_2.\ \lambda\, \rho.\ \lambda\, k.\ \lambda cv. \\
& \exists\, n_1, n_2.\ \tau_1\ \rho\ k\ \langle \text{state}(cv), \text{const-vector}(n_1)\rangle \quad (1) \\
& \qquad \wedge\ \tau_2\ \rho\ k\ \langle \text{state}(cv), \text{const-vector}(n_2)\rangle \quad (2) \\
& \qquad \wedge\ \mathsf{const}(aop(n_1, n_2))\ \rho\ k\ cv \quad (3)
\end{aligned}$$

  This definition is explained thus : condition (1) and (2) extract numbers out of environments, and condition (3) forces the concrete value cv to be equal to the constant resulting from the operation aop applied to the extracted numbers. We can get addition, subtraction, and multiplication by instantiating aop to $+$, $-$, and $\times$ respectively, where

$$\begin{aligned}
\mathsf{plus} &\stackrel{\text{def}}{=} \mathsf{arith}(\lambda\, n_1, n_2.\ n_1 + n_2) \\
\mathsf{minus} &\stackrel{\text{def}}{=} \mathsf{arith}(\lambda\, n_1, n_2.\ n_1 - n_2) \\
\mathsf{times} &\stackrel{\text{def}}{=} \mathsf{arith}(\lambda\, n_1, n_2.\ n_1 \times n_2)
\end{aligned}$$

Types : I shall now give semantic definitions of types in terms of the generalised type environments. Some of these definitions are simplified versions of the definitions used by FPCC. The actual definitions are complicated by the fact that the semantic

model must also support mutation of store. In this thesis I shall not be dealing with these complexities. Ahmed et al. [6] describe how to add to the type semantics to be able to handle mutation.

- $\top$, $\bot$: The top type imposes no constraints on values, and therefore the definitions for the top type and the empty environment coincide. $\bot$ is defined as

$$\bot \stackrel{\text{def}}{=} \lambda\,\rho.\ \lambda\,k.\ \lambda cv.\ \text{false}$$

  The $\bot$ type represents the "impossible" values, and would not actually have any machine-resident instances.

- offset : The unification of the three kinds allows us to express offset as a constructor derived from minus. The original definition by Appel and Felty[10] is

$$\mathsf{offset} = \lambda\,i.\lambda\,\tau.\lambda\,(a,m).\ \lambda\,v.\ \tau(a,m)(v+i)$$

  We can also equivalently express this definition as $\exists\,w.\ (v = w - i) \wedge \tau(a,m)w$. Considering the definitions of minus and arith, we see that the third condition in the definition of arith could express the equality $v = w - i$, the second condition enforces the equivalent of $\tau\ (a,m)\ w$ in the new model (with an approximation index), and the first condition extracts the number i from the type environment that would be supplied. We can therefore express offset in terms of minus and define it as

$$\mathsf{offset} \stackrel{\text{def}}{=} \lambda\,\tau_1.\ \lambda\,\tau_2.\ \mathsf{minus}(\tau_2,\tau_1)$$

- box : The simplified semantics for the type for immutable references, box, is

defined below. I shall not describe the semantics of mutable reference types. Please refer to description by Ahmed et al. [6] for a detailed description.

$$\begin{aligned}\mathsf{box} \stackrel{\text{def}}{=}\ & \lambda\,\tau.\ \lambda\,\rho.\ \lambda\,k.\ \lambda cv.\\ & \exists\, n.\ (\text{root}(cv)(0) = n)\ \wedge\ (n \in a)\\ & \qquad \wedge\ \tau\ \rho\ (k-1)\ \langle \text{state}(cv), \text{const-vector}(m[n])\rangle\\ & (\text{where } a \text{ is the allocation set derived from state of } cv, \text{ and}\\ & \qquad m \text{ is the memory derived from state of } cv)\end{aligned}$$

The first condition extracts the value of the first component of the root of the concrete value, cv. Using this value as an address to index into the memory, we construct a new concrete value, $cv' = \langle \text{state}(cv), \text{const-vector}(m[n])\rangle$. The second condition ensures that the extracted address n is in the allocated set extracted from cv, and that any subsequent memory load instructions would be legal. Since one instruction is required to perform the memory indirection, we expect cv' to approximate τ to one lesser degree, $k-1$, as given by the third condition in the definition.

- rec : The description of the recursive type is similar to that explained by the indexed model in Figure 4.4

$$\begin{aligned}\mathsf{rec} \stackrel{\text{def}}{=}\ & \lambda\,F.\ \lambda\,\rho.\ \lambda\,k.\ \lambda cv.\\ & (F^{k+1}\bot)\ \rho\ k\ cv\end{aligned}$$

The indexed model provides semantics for recursive types in a way similar to the metric spaces approach taken by MacQueen et al. [36], and it allows the rec operator to be proved as the fixed point necessary to be able to reason

about general recursion (covariant and contravariant). For this result to hold, it is necessary for the argument to rec to be *wellfounded* in the indexed model (or similarly, *contractive* in the metric spaces model). While using recursive types, we must therefore ensure this condition, and I shall explain the system we use to deal with this requirement in Chapter 5.

- $\cup$, $\cap$: The definitions of these constructors coincide with those for union and intersection operators on environments.

- $\forall$, $\exists$: Universal and existential quantifier definitions sufficient for interactions with immutable types are outlined below.

$$\forall \stackrel{\text{def}}{=} \lambda F.\ \lambda \rho.\ \lambda k.\ \lambda cv.\forall \breve{\tau}.\text{valid}(\breve{\tau}) \implies F(\breve{\tau} :: \rho)\ k\ cv$$
$$\exists \stackrel{\text{def}}{=} \lambda F.\ \lambda \rho.\ \lambda k.\ \lambda cv.\exists \breve{\tau}.\text{valid}(\breve{\tau}) \wedge F(\breve{\tau} :: \rho)\ k\ cv$$

In both these definitions, we quantify over a closed type ($\breve{\tau}$) (having no de Bruijn bindings), and we update the context ρ by adding $\breve{\tau}$ as the substitution for the zeroth de Bruijn index in F. A closed type, denoted by CTymap is obtained by applying a Tymap type expression to a context :

$$\mathsf{CTymap} = \mathsf{Index} \rightarrow \mathsf{CV} \rightarrow o$$

We also ensure that the closed type is valid (using a validity predicate for closed types, which is defined in a way analogous to the validity predicate on

open types defined in Section 4.1). The operator "::" is defined as

$$\begin{aligned} :: \stackrel{\text{def}}{=} \; & \lambda\, \breve{\tau}.\ \lambda\, \rho.\lambda\, i.\ \lambda\, k.\ \lambda\, cv\ (i = 0 \implies \quad \breve{\tau}\ k\ cv) \\ & \wedge (i > 0 \implies \quad \rho\ (i-1)\ k\ cv) \end{aligned}$$

- codeptr : The code pointer constructor definition is similar to that shown for the indexed model in Section 4.2 for a system with immutable references.

$$\begin{aligned} \mathsf{codeptr} \stackrel{\text{def}}{=} \; & \lambda\, \phi.\ \lambda\, \rho.\ \lambda\, k.\ \lambda cv. \\ & \forall l, j, s.\ \ (j < k && (1) \\ & \quad \wedge\ \text{root}(cv)[0] = l && (2) \\ & \quad \wedge\ \text{“state}(cv)\ \textit{extends to}\ s\text{”} && (3) \\ & \quad \wedge\ r_{pc} = l && (4) \\ & \quad \wedge\ \phi\ \rho\ j\ \langle s, \text{reg-vector}(s)\rangle) && (5) \\ & \implies \text{safen}(j, cv) \end{aligned}$$

Consider a state where register r_i contains a program location address l. To be able to execute the instructions starting at location l, we would require one jump instruction to first transfer control to that location. If the $(j+1)^{\text{th}}$ instruction starting at l were were unsafe to execute, then we could still execute one control transfer instruction to jump to location l, and safely take j further execution steps starting at l. In other words, if the control were at l and the machine state satisfied the environment ϕ to degree k, then it would be possible to jump to l and execute $k-1$ steps (thus k steps in total) safely. In the definition of codeptr above, condition (2) gets l, the value of the first component of cv, and condition (4) ensures that the program counter r_{pc}

points to the code beginning at memory location l. We then wish to state that for any register bank (considered as a value vector) that satisfies the type environment given by argument ϕ, it should be safe to execute $j < k$ machine steps starting at location l. We do so by first making a root value vector from the register bank in state s using a function reg-vector(s)[2]. Then, in condition (5) we ensure that ϕ holds for a concrete value made from state s and root value vector reg-vector(s) to any approximation $j < k$.

- int_π : The constructor for types that capture integer ranges relies on an auxiliary definition for a general relational constructor, rel.

$$
\begin{aligned}
\mathsf{rel} \stackrel{\text{def}}{=}\ & \lambda\, rop.\ \lambda\, \tau.\ \lambda\, \rho.\ \lambda\, k.\ \lambda cv. \\
& \exists\, n_1, n_2.\ \tau_1\ \rho\ k\ \langle \text{state}(cv), \text{const-vector}(n_1)\rangle && (1) \\
& \qquad \wedge\ \mathsf{const}(n_2)\ \rho\ k\ cv && (2) \\
& \qquad \wedge\ rop(n_1, n_2) && (3)
\end{aligned}
$$

Condition (1) extracts the number n_1 from the type-environment representation τ, condition (2) ensure that the cv has numeric value n_2, and condition (3) ensures the relation between the two. We use instantiations of *rop* to get all the dependent type constructors.

$$
\begin{aligned}
\mathsf{int}_{>}(\tau) &\stackrel{\text{def}}{=} \mathsf{rel}(\lambda\, n_1, n_2.\ n_1 > n_2)\ \tau \\
\mathsf{int}_{<}(\tau) &\stackrel{\text{def}}{=} \mathsf{rel}(\lambda\, n_1, n_2.\ n_1 < n_2)\ \tau \\
\mathsf{int}_{\leq}(\tau) &\stackrel{\text{def}}{=} \mathsf{rel}(\lambda\, n_1, n_2.\ n_1 \leq n_2)\ \tau \\
\vdots\ &\ \vdots\ \ \vdots
\end{aligned}
$$

[2]The model for machine values which includes values held in registers and spill locations is fairly complex and is not dealt with in this thesis.

- $\underline{n}$: We use this operator to extract the binding for the n^{th} de Bruijn index from the context ρ, and the semantics for this operator is

$$\underline{n} \stackrel{\text{def}}{=} \lambda\,\rho.\ \lambda\,k.\ \lambda cv.\rho(n)\ k\ cv$$

4.3.1 Bounded quantification

In Chapter 3, I described bounded quantification that allowed us to describe fixed-length arrays. The quantifiers described earlier are not powerful enough to capture bounded quantification that is necessary to be able to describe arrays. A general encoding of bounded quantification (for example, universal) could be written as

$$\begin{aligned}\forall_B \stackrel{\text{def}}{=}\ & \lambda\,\mathit{bound}.\ \lambda\,F.\ \lambda\,\rho.\ \lambda\,k.\ \lambda cv.\\ & \forall\check{\tau}.\ \text{valid}(\check{\tau}) \wedge \mathit{bound}(\check{\tau}) \implies F(\check{\tau} :: \rho)\ k\ cv\end{aligned}$$

where the argument *bound* represents the function that imposes the necessary restrictions on the quantifying bound variable. This generality however allows us to encode quantifiers that turn out not to be valid in the indexed model of types. Since our main use for bounded quantification in FPCC was to be able to deal with contiguous memory locations as in arrays, we use a constrained form of boundedness, which only allows us to express numeric ranges. We therefore have the bounded

quantifiers defined as

$$
\begin{aligned}
\forall_B \stackrel{\text{def}}{=}\ & \lambda\,\tau_{\mathrm{B}}\ \lambda\,F.\ \lambda\,\rho.\ \lambda\,k.\ \lambda cv. \\
& \exists\, n_2.\ \forall\,\breve{\tau}.\ \exists\, n_1.\ (\text{valid}(\breve{\tau}) && (1)\\
& \quad \wedge\ (\tau_B\ \rho\ k\ \langle \text{state}(cv), \text{const-vector}(n_2)\rangle) && (2)\\
& \quad \wedge\ (\breve{\tau}\ k\ \langle \text{state}(cv), \text{const-vector}(n_1)\rangle) && (3)\\
& \quad \wedge\ n_1 \geq 0 \wedge n_1 < n_2) && (4)\\
& \quad \Longrightarrow\ (F\ (\breve{\tau} :: \rho)\ k\ cv)
\end{aligned}
$$

In this definition, the argument τ_B encodes the number that serves as the upper bound for the range. The lower (inclusive) bound is taken to be zero. Condition (2) and (3) ensure that τ_B and the quantified $\breve{\tau}$ represent numbers, and condition (4) ensures that the quantified variable respects the bounds. The definition of existential bound variable is similar.

$$
\begin{aligned}
\exists_B \stackrel{\text{def}}{=}\ & \lambda\,F.\ \lambda\,\tau_{\mathrm{B}}\ \lambda\,\rho.\ \lambda\,k.\ \lambda cv. \\
& \exists_B\, n_2.\ \exists\,\breve{\tau}.\ \exists\, n_1.\ \text{valid}(\breve{\tau}) \\
& \quad \wedge\ (\tau_B\ \rho\ k\ \langle \text{state}(cv), \text{const-vector}(n_2)\rangle) \\
& \quad \wedge\ (\breve{\tau}\ k\ \langle \text{state}(cv), \text{const-vector}(n_1)\rangle) \\
& \quad \wedge\ n_1 \geq 0 \wedge n_1 < n_2 \\
& \quad \wedge\ (F\ (\breve{\tau} :: \rho)\ k\ cv)
\end{aligned}
$$

Using these definitions, we can now express the array types (as shown in Section 3.3) as

$$
\begin{aligned}
\text{array} \stackrel{\text{def}}{=}\ & \lambda\,\tau.\ \lambda\, up\_bound.\forall_B(\mathsf{offset}(I, \mathsf{box}\,(\tau)), \mathsf{const}(up\_bound)) \\
& \text{where } I = \mathsf{times}(\underline{0}, \mathsf{const}(4))
\end{aligned}
$$

The array size is specified by instantiating *up_bound* to $\mathsf{const}(n)$ for the desired upper bound n. For the Sparc architecture, each machine word has size 4. We would therefore require all offsets to have a separation of 4 bytes. To enforce this condition, we multiply the indexing variable by 4 using the times type constructor. Sometimes, it might also be necessary to consider array ranges starting at an index $m \neq 0$. This can be encoded by using the plus operator to get $\mathsf{plus}(\mathsf{const}(m \times 4), \mathsf{times}(\underline{0}, \mathsf{const}(4)))$, which adds an additional $m \times 4$ bytes to all offsets.

4.3.2 Structural rules for Tymaps

Introduction and elimination rules for all Tymap type constructors are implemented in the Twelf system as machine-checked lemmas. For example, the intersection environment has the following lemmas :

$$\frac{(\tau_1 \cap \tau_2)\ \rho\ k\ cv}{\tau_1\ \rho\ k\ cv}\ \cap\_\mathrm{E}1 \qquad \frac{(\tau_1 \cap \tau_2)\ \rho\ k\ cv}{\tau_2\ \rho\ k\ cv}\ \cap\_\mathrm{E}2$$

$$\frac{\tau_1\ \rho\ k\ cv \quad \tau_2\ \rho\ k\ cv}{(\tau_1 \cap \tau_2)\ \rho\ k\ cv}\ \cap\_\mathrm{I}$$

One of the main benefits of these rules is to provide an abstraction between the definitions of these constructors and the lemmas that used them. These rules also provide semantics to coercions (see Sections 4.6) like `pack` and `unpack` that are necessary to typecheck LTAL programs.

Having shown the semantic definitions of the type constructors in TML, I shall now describe some of the issues related to the static semantics.

4.4 Properties of TML types

For a type to be valid (i.e., for a predicate on $\rho,\ k,\ cv$ to be a "type") in the indexed model, they need to satisfy three conditions :

indexclosed If a value has a type τ to an approximation index k, then, it has that type for all approximation indices lower than k, i.e.,

$$\text{indexclosed}(\tau) \stackrel{\text{def}}{=} \forall\, j, k.\, \forall\, \rho, cv.(j < k \ \wedge\ \tau\ \rho\ k\ cv) \implies \tau\ \rho\ j\ cv$$

extensible Typing judgements continue to hold under extended state as described by Ahmed et al. [6]. This condition is necessary to ensure that the allocation and initialisation of new objects in the memory does not affect existing typing judgements on other reference values.

$$\begin{aligned}\text{extensible}(\tau) \stackrel{\text{def}}{=}\ & \forall\, s.\, \forall\, \rho, cv. \\ & (\text{“}s \text{ extends to state}(cv)\text{”} \ \wedge\ \tau\ \rho\ k\ cv) \\ & \implies \tau\ \rho\ j\ \langle s, \text{root}(cv)\rangle\end{aligned}$$

extensional Since we model types as function predicates on contexts, we require these function predicates to be extensional with respect to their arguments, i.e., if a value has a type in some context ρ_1, then it should continue to have this type for an equivalent context, ρ_2. In the concrete Twelf implementation our logic does not have an extensionality axiom. Therefore, this property

must be proved individually for every type that is constructed in TML.

$$\begin{aligned}\text{extensional}(\tau) \stackrel{\text{def}}{=}\ & \forall\ \rho_1, \rho_2, k,\ cv. \\ & (\forall\, i, k, cv.\ \rho_1\ i\ k\ cv\ \equiv \rho_2\ i\ k\ cv) \\ & \Longrightarrow\ (\tau\ \rho_1\ k\ cv\ \equiv \tau\ \rho_2\ k\ cv)\end{aligned}$$

This property is required for many rules relating to subtyping, explicit substitutions, and wellformedness (described in Chapter 5) of Tymap expressions.

Theorem 1 *All types described in Section 4.3 are valid.*

Proof. Machine-checked proofs of validity of the type constructors are implemented in the Twelf system. As an example, consider the proof for the extensionality of $\mathsf{const}(n)$ shown below presented in the traditional style. In this proof, we assume rules such as implication introduction ($\Longrightarrow$_I), equivalence introduction ($\equiv$_I), and extensionality introduction (EXTENSIONAL_I).

$$\frac{}{\text{extensional}(\mathsf{const}(n))}\ \text{CONST\_EXTENS}$$

	ρ_1, ρ_2, k, cv	
1.	[(const(n) ρ_1 k cv)] 1.1. $cv[0] = n$ "By CONST_E" 1.2. (const(n) ρ_2 k cv) "By CONST_I"	
2.	(const(n) ρ_1 k cv) $\Longrightarrow$ (const(n) ρ_2 k cv)	"By $\Rightarrow$_I(1.1, 1.2)"
3.	[(const(n) ρ_2 k cv)] 3.1. $cv[0] = n$ "By CONST_E" 3.2. (const(n) ρ_1 k cv) "By CONST_I"	
4.	(const(n) ρ_2 k cv) $\Longrightarrow$ (const(n) ρ_1 k cv)	"By $\Rightarrow$_I(3.1, 3.2)"
5.	(const(n) ρ_2 k cv) $\equiv$ (const(n) ρ_1 k cv)	"By $\equiv$_I(2,4)"
6.	$(\forall i.\ \rho_1\ i\ k\ cv\ \equiv \rho_2\ i\ k\ cv)$ $\Longrightarrow$ ((const(n) ρ_1 k cv) $\equiv$ (const(n) ρ_2 k cv))	"By $\Rightarrow$_I(5)"
7.	extensional(const(n))	"By EXTENSIONAL_I(6)"

4.4.1 Semantics of subtyping

Subtyping for all Tymap constructors ("$\subseteq$") forms a partial order and is defined as

$$\tau_1 \subseteq \tau_2 \ \stackrel{\text{def}}{=}\ \forall \rho, k, cv.\ (\tau_1\ \rho\ k\ cv)\ \Longrightarrow\ (\tau_2\ \rho\ k\ cv)$$

Equality on types, $\approx$, is defined in terms of two-way subtyping as

$$\tau_1\ \approx\ \tau_2\ \stackrel{\text{def}}{=}\ \tau_1 \subseteq \tau_2\ \wedge\ \tau_2 \subseteq \tau_1$$

We can similarly define notions of subtyping and equality on closed types $(\subseteq, \approx)$ and equality on contexts $(\approx)$:

$$\begin{aligned}
\breve{\tau_1} \subseteq \breve{\tau_2} &\stackrel{\text{def}}{=} \forall k, cv.\ (\breve{\tau_1}\ k\ cv) \implies (\breve{\tau_1}\ k\ cv) \\
\breve{\tau_1} \approx \breve{\tau_2} &\stackrel{\text{def}}{=} \breve{\tau_1} \subseteq \breve{\tau_2} \wedge \breve{\tau_2} \subseteq \breve{\tau_1} \\
\rho_1 \approx \rho_2 &\stackrel{\text{def}}{=} \forall i, (\rho_1\ i) \approx (\rho_2\ i)
\end{aligned}$$

For the indexed model, it is sometimes necessary to prove the subtyping relation to approximation k, and this is given by

$$\tau_1 \subseteq_k \tau_2 \stackrel{\text{def}}{=} \forall \rho, j, cv.\ (j < k \wedge (\tau_1\ \rho\ j\ cv)) \implies (\tau_2\ \rho\ j\ cv)$$

4.5 Semantics for explicit substitutions

We use the standard explicit substitutions calculus [4] for manipulation of TML expressions with respect to de Bruijn contexts. The substitution operators are

$$\text{Substitutions}\quad \mathsf{Subst} ::= \mathtt{id} \mid \mathtt{shift}(i) \mid \mathtt{cons}(\tau, s) \mid \mathtt{compose}(s_1, s_2)$$

The foundational semantics for substitution operators are given as :

Subst	s	:	$\mathsf{Context} \to \mathsf{Context}$
Identity substitution	$\mathtt{id}$	$\stackrel{\text{def}}{=}$	$\lambda\, \rho.\ \rho$
Index shift	$\mathtt{shift}$	$\stackrel{\text{def}}{=}$	$\lambda\, i.\ \lambda\, \rho.\ \rho(i+1)$
Term append	$\mathtt{cons}$	$\stackrel{\text{def}}{=}$	$\lambda\, (\tau, s).\ \lambda\, \rho.\ (\tau\ \rho) :: (s\rho)$
Substitution composition	$\mathtt{compose}$	$\stackrel{\text{def}}{=}$	$\lambda\, s_1.\ \lambda\, s_2 . \lambda\, \rho.\ s_1\ (s_2\ \rho)$

$$\frac{}{\rho \approx \mathtt{id}(\rho)}\ \text{SUB-ID}$$

$$\frac{}{((\mathtt{cons}(\tau, s))\ \rho)(0) \approx \tau\ \rho}\ \text{SUB-}\rho\text{0}$$

$$\frac{}{\tau\ (s_1\ (s_2\ \rho)) \approx \tau\ (\mathtt{compose}(s_1, s_2)\ \rho)}\ \text{SUB-}\rho\text{\_COMPOSE}$$

$$\frac{}{\tau\ (\mathtt{compose}(\mathtt{id}, s)\ \rho) \approx \tau\ (s\ \rho)}\ \text{SUB-ID\_COMPOSE}$$

$$\frac{}{\tau\ (\mathtt{compose}(\mathtt{shift}, id)\ \rho) \approx \tau\ (\mathtt{shift}\ \rho)}\ \text{SUB-SHIFT\_ID}$$

$$\frac{\text{extensional}(\tau)}{\tau\ (\mathtt{compose}(\mathtt{shift}, (\mathtt{cons}(\tau_\rho, s)))\ \rho) \approx \tau\ (s\ \rho)}\ \text{SUB-SHIFT\_CONS}$$

Figure 4.6: Explicit substitution rules

Figure 4.6 gives the identities that hold for expressions involving substitutions. The substitution operator semantics have been encoded in Twelf and these identities are encoded as lemmas with machine-checked proofs.

4.6 TML as a semantic model for the LTAL type systems

In our FPCC project, core ML is compiled into machine code, and also into LTAL, a "low-level typed assembly language". The machine code program is typechecked with respect to annotations in the LTAL program and the LTAL typing rules. The LTAL types are given foundational semantics in terms of TML constructors, and LTAL typing judgements are given semantics in terms of machine-checked lemmas about TML constructors. LTAL has four syntactic categories: values, types, coercions, and instructions. Of these, TML is used to provide semantics to all but

$$
\begin{array}{llll}
\textsf{Types} & \tau & ::= & \textsc{t\_v}(n) \mid \textsc{t\_int} \mid \textsc{t\_code}(n, m, \phi) \mid \textsc{t\_sum}(n, \tau) \mid \textsc{t\_fix}(\tau) \\
 & & & \mid \textsc{t\_exists}(\tau) \mid \textsc{t\_i}(n) \mid \textsc{t\_nat}(\pi, n) \mid \textsc{t\_bottom} \\
 & & & \mid \textsc{t\_inters}(\tau_1, \tau_2) \mid \textsc{t\_union}(\tau_1, \tau_2) \mid \textsc{t\_range}(n_1, n_2) \\
 & & & \mid \textsc{t\_field}(n, \tau) \\
\textsf{Values} & v & ::= & v \mid n \mid l \mid c(v) \\
\textsf{Coercions} & c & ::= & \mid \textsc{c\_fold} \mid \textsc{c\_unfold} \mid \textsc{c\_pack} \mid \textsc{c\_union1} \mid \textsc{c\_union2} \\
 & & & \mid \textsc{c\_inters1} \mid \textsc{c\_inters2} \mid \textsc{c\_offset0}
\end{array}
$$

Figure 4.7: LTAL core syntax (without instructions)

LTAL values. TML is therefore more of a logic than a language.

A part of the core LTAL type system due to Chen et al. [17] is reproduced in Figure 4.7, and the TML semantics for constructors in LTAL are listed for each syntactic category.

Types : LTAL has integer (T_INT) and refined integer types (T_I) (to capture dataflow facts), intersection and union types (used to encode sum types and structures), fixed-length array types, existential and code pointer types (to create closures), reference types, and recursive types. TML types can model each of them as shown in Figure 4.8. While most of these types are standard, some of them require explanation. The type T_V is for type variables, and is modelled by de Bruijn indices. The code pointer type takes two extra arguments (n and m) which are concerned with the number of type variables and memory availability requirements. The type T_SUM allows us to express the number of constant constructors in a sum type. The range type T_RANGE allows LTAL to express integer ranges. As shown, LTAL also has other types that are used more for type checking than for specifying actual values that reside in registers or memory. These types do have models, but the semantics for these types is given directly in terms of predicates based on higher-order logic, rather than through composition of TML constructors. Since

LTAL constructors with TML models			
variable	T_V(n)	$=\!\!\mid$	$\underline{n}$
integer	T_INT	$=\!\!\mid$	$\mathsf{int}_{\geq}(0) \cap \mathsf{int}_{<}(maxint32)$
code pointer	T_CODE(n, m, ϕ)	$=\!\!\mid$	$\mathsf{codeptr}\,(\phi)$
sum	T_SUM(n, τ)	$=\!\!\mid$	$\mathsf{int}_{\geq}(0) \cup \mathsf{int}_{<}(n) \cup \tau$
recursive	T_FIX(τ)	$=\!\!\mid$	$\mathsf{rec}\,(\tau)$
existential	T_EXISTS(τ)	$=\!\!\mid$	$\exists(\tau)$
constant integer	T_I(n)	$=\!\!\mid$	$\mathsf{int}_{=}(n)$
refined integer	T_NAT(π, n)	$=\!\!\mid$	$\mathsf{int}_{\pi}(n)$
bottom	T_BOTTOM	$=\!\!\mid$	$\bot$
intersection	T_INTERS(τ_1, τ_2)	$=\!\!\mid$	$\tau_1 \cap \tau_2$
union	T_UNION(τ_1, τ_2)	$=\!\!\mid$	$\tau_1 \cup \tau_2$
integer range	T_RANGE(n_1, n_2)	$=\!\!\mid$	$\forall_B(n_2, \mathsf{int}_{=}(\mathsf{plus}(n_1, \underline{0})))$
field	T_FIELD(n, τ)	$=\!\!\mid$	$\mathsf{field}(n, \tau)$
Some LTAL constructors without TML models			
address offset	T_DIFF		
condition code	T_CMPCC(τ_1, τ_2)		
memory check	T_TESTFULL(n)		
address check	T_ADDR(n)		

Figure 4.8: TML type constructors to model LTAL types

there would be no machine-resident value of these types, they need not have to be proved valid.

Coercions : Instructions in a program often need to view values as having different types depending on how they are accessed. For example, a value of type integer may also be seen as a variant of a sum type having an integer and boolean. LTAL has coercions such as FOLD and UNFOLD for recursive types, PACK for existential types, UNION1, UNION2 and INTERS1, INTERS2 for union and intersection types. Figure 4.9 lists some of the coercions used in LTAL and the TML subtyping rules that are used to provide semantic bases to them. The full LTAL calculus has many

Recursive type folding	C_FOLD	$\models$	$\subseteq$-FOLD
Recursive type unfolding	C_UNFOLD	$\models$	$\subseteq$-UNFOLD
Existential type packing	C_PACK	$\models$	$\subseteq$-PACK
union type introduction	C_UNION1	$\models$	$\subseteq$-$\cup$ R1
union type introduction	C_UNION2	$\models$	$\subseteq$-$\cup$ R1
intersection elimination	C_INTERS1	$\models$	$\subseteq$-$\cap$ L1
intersection elimination	C_INTERS2	$\models$	$\subseteq$-$\cap$ L2
sum to range	C_INTERS2	$\models$	$\subseteq$-$\cap$ L2
zero offset elimination	C_OFFSET0	$\models$	$\subseteq$-OFFSET_0

Figure 4.9: TML subtyping rules for to model LTAL coercions

other coercions; most of these can be easily given semantics by the composition of two or more of the subtyping rules shown in Figure 3.9 and Figure 3.10.

Instructions : The semantics for LTAL instructions in terms of TML instructions shall be dealt with in Chapter 6.

Chapter 5

Managing TML Types With Kinds

5.1 Introduction

Conventional type systems classify types using a kind system. In Chapter 3, I described various TML kinds (types, environments, and naturals), which were unified in Chapter 4 to simplify the model for types for our FPCC project. That is, constructors for types, environments, and naturals were all encoded using the same metalogical kind. This approach helped us reduce the complexity of the semantic model, and also reduce the number of lemmas in the system.

This unification of different TML kinds, however, comes at a price: it now becomes possible for illformed types to be composed. This chapter deals with a kinding system that not only allows types to be checked for valid compositions of type constructors, but also allows us to reason about the effect of such compositions on some of the important properties of type expressions. Unlike conventional kinding systems, we do not have the kind of type functions in TML. This is because it is not possible for us to have a single metalogical type (in our higher-order logic)

that would handle arbitrarily higher-order kinds. Core ML programs do not need this feature and it is possible to reason about these programs using only first order kinds.

5.2 Illformed expressions

Our main motivation for combining the three kinds (vectors, scalars, and numbers) into a single kind was to reduce the number of quantification operators. For example, in Chapter 3, there were different quantifier for each quantified-variable and expression combination such as $\forall^{\mathsf{T}}_{\mathsf{T}}$, $\forall^{\mathsf{E}}_{\mathsf{N}}$, $\forall^{\mathsf{E}}_{\mathsf{T}}$, etc.. By having a single metalogical kind to represent all the TML kinds, we were able to implement variable bindings through a single de Bruijn context. As a result, it was sufficient to have only a single set of quantifiers ($\forall$ and $\exists$).

However, removing the kind restrictions allows us to construct type expressions such as the one given below :

$$\exists(\{0 : \mathsf{codeptr}\,(\underline{0}),\ 1 : \mathsf{plus}(\underline{0}, \mathsf{const}(1))\})$$

In this expression, the quantified variable $\underline{0}$ is used as an environment in its first occurrence and as a number in its second occurrence. This inconsistency would not lead to any safety problems since any use of such type constructors would have to be backed by machine-checked lemmas. While syntactically valid, this expression would just not be useful for any real programs. In constructing syntax-directed proofs of safety, however, it is often useful to ensure that the quantifier variables are used in a consistent way at all occurrences in the binding type expressions.

The kinding system presented in this chapter allows these types of checks to be performed.

5.3 Composition of type constructors

The semantic models for reasoning about recursive types (such as the indexed model that we use for FPCC and the metric spaces model [36, 5]) require that the recursive type functions be *contractive* in their arguments. This requirement in turn necessitates certain type constructors (functions) to be either *contractive* or *nonexpansive* in their arguments. The above mentioned works on these models explain how we can prove contractiveness for single-argument type functions, and also how to reason about the composition of single-argument contractive and nonexpansive type functions. As an example, in the type expression

$$\mathsf{rec}\,(\mathsf{box}\,(\underline{0}) \cup \mathsf{field}(\mathsf{const}(4), \underline{0}))$$

both occurrences of $\underline{0}$ referred to the same variable (the one bound by the rec constructor), and subexpressions occurring inside the rec constructor (for example, the inner union type constructor) can also be considered as type functions on a single variable. This simplifies the presentation of lemmas such as the fold-unfold lemma in the Appel-McAllester model. However, in FPCC, we commonly use type expressions (functions) that are compositions of multiple-argument type functions. For example, in the expression below, the union type constructor inside the existential refers to two variables, one bound by itself, and the other bound by the inner rec constructor.

$$\mathsf{rec}\,(\mathsf{rec}\,(\underline{1} \cap \exists(\underline{0} \cup \underline{1})))$$

This chapter describes a kinding system that allows us to systematically keep track of properties such as contractiveness in the presence of type expressions in several variables. Though these properties relate to semantic well formedness of type expressions, the kinding system allows for a purely syntactic presentation, so that the safety prover does not need to know the underlying semantics of either the types or the kinds. Any semantic reasoning about type constructors for languages like core ML would require the properties that are considered in this system, and such a system might be useful even outside the current scope of the FPCC project. TML provides semantics to only first-order kinds, and this kinding system is also restricted to the management of first-order kinds.

5.3.1 Contractive and nonexpansive types

To reason about recursive types using the two-way subtyping $\mathsf{rec}\,(\tau) \approx \tau(\mathsf{rec}\,(\tau))$ that is used for the FOLD and UNFOLD rules, we must know that τ is *contractive* in its first argument.

In terms of execution steps for a concrete machine, a type constructor F is contractive if it takes more machine instructions to reach the bottom of a value of type $F(\tau)$ than it does to reach the bottom of a value of type τ. For example, if it takes n instructions to completely traverse the value v of type τ, then it takes at least $n+1$ instructions to traverse a value v' of type $\mathsf{box}\,(\tau)$ since one more memory load instruction is needed to access the boxed value in v'. In terms of approximation indices, if value v has type τ to approximation k, then the value v' has type $F(\tau)$ to approximation $k+1$ at least.

Type constructors such as box and codeptr are contractive. However, the constructors offset, $\cap$, and $\cup$ are not. For example, it would take only as many

instructions to completely traverse a value of type $\tau_1 \cap \tau_2$ as it would take to traverse a value of one of τ_1 or τ_2. These operators are, however, *nonexpansive*, i.e., if value v has type τ to approximation k, then the value v' has type $F(\tau)$ (for some nonexpansive operator F) to approximation k. An important property of these nonexpansive operators is that the result of composing them with contractive operators is contractive.

Formally, we define contractiveness of a closed type expression as follows. Let $\lfloor \breve{\tau} \rfloor_k$ be a closed type representing the k^{th} approximation to the closed type $\breve{\tau}$,

$$\lfloor \breve{\tau} \rfloor_k = \lambda v. \forall j < k. \, \breve{\tau}(j, v)$$

and let the replacement of the i^{th} binding in the type context ρ by its j^{th} approximation be given by $\rho[i \mapsto \lfloor \rho(i) \rfloor_j]$. Then we say a type function f is contractive in its i^{th} type argument iff

$$\forall \rho, k, v. \quad \lfloor f(\rho) \rfloor_{k+1} \; v \equiv \lfloor f(\rho[i \mapsto \lfloor \rho(i) \rfloor_k]) \rfloor_{k+1} \; v$$

Similarly, a type function is nonexpansive with respect to its i^{th} argument iff :

$$\forall \rho, k, v. \quad \lfloor f(\rho) \rfloor_k \; v \equiv \lfloor f(\rho[i \mapsto \lfloor \rho(i) \rfloor_k]) \rfloor_k \; v$$

If f is contractive, judging $f \;\; \rho \;\; (k+1) \;\; v$ requires requires judging $\rho(i)$ at a lesser approximation k. Due to this, we can have a well-founded induction (over the approximation index) when we construct and reason about recursive types. The contractiveness property is therefore referred to as *wellfoundedness* by Appel and McAllester in the indexed model.

Appel and McAllester show rules for deciding the wellformedness of compositions of single-argument type functions like rec in the indexed model. While techniques in MacQueen et al., and Appel and McAllester both can be extended to work for multiple argument functions, they are too unwieldy for writing machine-checkable implementations.

Consider, for example, the ML type

```
datatype 'a List = Cons of 'a * 'a List | Nil
```

Assuming a untagged representation, this type can be represented using TML operators as :

$$\mathsf{rec}\,(\ \underbrace{\mathsf{int}_{\neq}(0)\ \cap\ \mathsf{offset}(0, \mathsf{box}\,(\underline{1}))\ \cap\ \mathsf{offset}(1, \mathsf{box}\,(\underline{0}))}_{\text{Cons}}\quad \cup\quad \underbrace{\mathsf{int}_{=}(0)}_{\text{Nil}}\ )$$

In this type expression, the de Bruijn index $\underline{1}$ refers to the type instantiation of the polymorphic list type, and the de Bruijn index $\underline{0}$, the parameter to rec, appears deep inside the type expression within a box operator. The rec constructor requires its argument to be contractive in order for the whole type expression to be well formed. It is not always the first argument (de Bruijn index $\underline{0}$) that requires the contractiveness to hold. Consider an ML datatype `t` built using Standard ML first-class continuations

```
datatype t = A of t cont
```

Note that this recursive type has an additional existential for the context that the continuation is evaluated in, and as explained in Section 3.2.1, it can be described

as :

$$\mathsf{rec}\,(\exists(\ \underbrace{\mathsf{field}(0,\mathsf{codeptr}\,(\{0:\underline{1},1:\underline{0}\}))}_{\text{Function Pointer}}\ \cap\ \underbrace{\mathsf{field}(4,\underline{0})}_{\text{Environment}}\ ))$$

In the boxed type, we require contractiveness to hold for the de Bruijn index $\underline{1}$ (the rec parameter). A kinding system allows us to keep track of the contractive and nonexpansive properties of type variables in expressions that might have such wide separations of the binding of the type variables and their uses. One of the aims of our kinding system is to make this calculus easier for writing and proving lemmas, especially in an automated or semiautomated framework.

5.3.2 Representable types

The semantic model described in Chapter 4 does not address the issue of mutable references. Ahmed et al. show how the model for types and machine state can be enhanced to allow the inclusion of mutable references in the presence of impredicative polymorphism, and it is this model that is used for FPCC. For this mutable-references model to work, it is necessary for types to be *representable*. Informally, if a type is *representable* then a value with that type can be stored in a mutable cell in memory. We give a brief explanation of what it means for types to be representable.

The Appel-Felty model [10] for machine state and memory allocation allows only immutable references. As described in Chapter 4, in this model, a type is a predicate on memory m, the allocated set a, and a value v where a is the set of allocated locations. In conventional syntactic calculi with mutable references, it is not enough to just know that a location is allocated, we must also know what types

of values may be safely stored to that location. Hence, the allocated set a must be modelled as a finite map from locations to types. These syntactic calculi have a judgment $m : a$ saying that each allocated element in memory m has an appropriate type that is given by the allocated set a.

A naïve extension of the Appel-Felty model where a is a function from locations to types would yield the following problematic model:

Type	τ	:	$(m, a) \rightarrow v \rightarrow o$
Memory	m	:	$l \rightarrow v$
Allocated Set	a	:	$l \rightarrow \tau$
Values	v	:	$\{0, 1, ...\}$
Locations	l	:	$\{0, 1, ...\}$

In this semantic model, there is a circularity; types are predicates on memory allocation sets, and allocation sets are predicates on types. To break this circularity, Ahmed et al. [6] model the allocation set as a partial map from locations to a *type syntax* that represents the type. This representation takes form of a tree (or can be equivalently enumerated as a Gödel number) which gives a syntactic description of the type expression. (In their scheme, each node in the type expression syntax tree has a number which is uniquely associated with a type constructor). This removes the circularity, but it requires another map from the type syntax to the semantic type. This map is the representation function repr. Given a type-syntax encoding,

repr gives the corresponding semantic type. The new model of types is given as :

Type syntax	τ_{syn}	:	$\{0, 1, 2, ...\}$
Allocated Set	a	:	$l \rightarrow \tau_{syn}$.
RepresentationFunction	repr	:	$\tau_{syn} \rightarrow \tau$.

The predicate "representable" is defined over type expressions as

$$\text{representable} \stackrel{\text{def}}{=} \lambda\tau.\ \exists\tau_{syn}.\ \mathsf{repr}(\tau_{syn}) = \tau$$

In the FPCC system, in addition to the types shown in Figure 3.1, there are also some types (as given in Chapter 4) that never need to be put inside mutable references. These types therefore do not need to be representable.

Since the type system itself allows the construction of type expressions that have references to these types, we must have a systematic way of ensuring that all the types that are actually put in mutable references are valid and representable. In the semantic model, the base types shown in Figure 3.1 can be shown to be representable by constructing explicit syntax trees for them. Most of the non-base type constructors (such as box and offset) can also be shown to be representable if their components are representable. Quantified types are representable only if they are quantified over representable types. To keep track of representability of complex type expressions in an organized way, we encode representability as a kind in our system. Types are then ensured to be meaningful with respect to representability through the kinding system.

5.4 Kinding system

Previous sections have motivated the need for a kinding system which is able to track type expressions properties such as validity, representability, contractiveness, and constancy. I shall now describe a set of kinds suited for this purpose, and give these kinds a semantic model that allows us to prove kinding rules as lemmas. Given these lemmas, it is easy to implement a kinding prover that assists the program-safety prover in ensuring the wellformedness of TML type expressions.

5.4.1 Kinding hierarchy

The kinding system for TML consists of predicates on closed types. There are four basic kinds in the system :

- Ω_V : This is the most inclusive kind, requiring only that its members be valid (as explained in Chapter 4).
- Ω_R : All the types belonging to this kind should be representable.
- Ω_C : Types belonging to this kind should be contractive.
- Ω_S : Since we unify all the kinds into that of a TML environment, we use this kind to qualify those type expressions that are intended to represent types for single values rather than environments over a vector of values.
- Ω_N : This is the kind of integer singleton types (those constructed from const).

Often, it is necessary to have types that respect more than one of these kinds. We have a list of derived kinds that are constructed from the combinations of these

kinds. These derived kinds are :

$$\begin{aligned}
\Omega_{VR} &\stackrel{\text{def}}{=} \lambda f.\ \Omega_V(f)\ \wedge\ \Omega_R(f) \\
&\quad (\text{“valid, representable”}) \\
\Omega_{VRC} &\stackrel{\text{def}}{=} \lambda f.\ \Omega_{VR}(f)\ \wedge\ \Omega_C(f) \\
&\quad (\text{“valid, representable, contractive”}) \\
\Omega_{VRCS} &\stackrel{\text{def}}{=} \lambda f.\ \Omega_{VRC}(f)\ \wedge\ \Omega_S(f) \\
&\quad (\text{“valid, representable, contractive, scalar”}) \\
\Omega_{VRCSN} &\stackrel{\text{def}}{=} \lambda f.\ \Omega_{VRCS}(f)\ \wedge\ \Omega_N(f) \\
&\quad (\text{“valid, representable, contractive, scalar, constant”})
\end{aligned}$$

With these derived kinds, we can define a subkinding judgment

$$\kappa_1 \subseteq_\kappa \kappa_2 \stackrel{\text{def}}{=} \forall f.\ (\kappa_1\ f)\ \Longrightarrow\ (\kappa_2\ f)$$

The order of compositions is intended to capture a flat list of kinds rather than impose a hierarchy on derived kinds. One of the main reasons for having a total order is to be able to write a nonbacktracking prover for the kinding system where each derived kind can be easily broken down into its constituent basic kinds.

5.4.2 Checking types for wellformedness

Figure 5.1 shows the kinding rules for TML type constructors (Figure 3.1). We use the kinding judgments described above to check that the types we use are well formed. As an example, consider the type expression $\tau = \mathsf{rec}\,(\forall(\mathsf{box}\,(\underline{0})\ \cap\ \underline{1}))$. Note that $\underline{0}$ refers to the variable bound by “$\forall$”, and $\underline{1}$ to the variable bound by “rec”. The argument of rec is not contractive. Consider, for example, a value v which has

$$\frac{\Gamma_i \subseteq_\kappa \kappa}{\Gamma \vdash \ \underline{i} :: \kappa}\ \text{WF-}VAR \qquad \frac{}{\Gamma \vdash \ \top :: \Omega_{VRCS}}\ \text{WF-}\top \qquad \frac{}{\Gamma \vdash \ \bot :: \Omega_{VRCS}}\ \text{WF-}\bot$$

$$\frac{\Gamma \vdash \ \tau :: \Omega_{VRC}}{\Gamma \vdash \ \mathsf{codeptr}\,(\tau) :: \Omega_{VRCS}}\ \text{WF-}CPTR$$

$$\frac{\kappa \in \{\Omega_V, \Omega_R, \Omega_S\} \quad \Gamma \vdash \ \tau :: \kappa}{\Gamma \vdash \ \mathsf{box}\,(\tau) :: \kappa}\ \text{WF-}BOX\_R0 \qquad \frac{\Gamma \vdash \ \tau :: \Omega_{VR}}{\Gamma \vdash \ \mathsf{box}\,(\tau) :: \Omega_{VRC}}\ \text{WF-}BOX\_RC$$

$$\frac{\Gamma \vdash \ \tau :: \Omega_{VR}}{\Gamma \vdash \ \mathsf{ref}\,(\tau) :: \Omega_{VRC}}\ \text{WF-}REF \qquad \frac{\Gamma \vdash \ \tau :: \Omega_{VR} \quad \Gamma \vdash \ \tau :: \Omega_S}{\Gamma \vdash \ \mathsf{ref}\,(\tau) :: \Omega_{VRCS}}\ \text{WF-}REF$$

$$\frac{\Omega_{VR}, \Gamma \vdash \ \tau :: \Omega_{VRC}}{\Gamma \vdash \ \mathsf{rec}\ \tau :: \Omega_{VR}}\ \text{WF-}REC$$

$$\frac{\Gamma \vdash \ \tau_1 :: \kappa \quad \Gamma \vdash \ \tau_2 :: \kappa}{\Gamma \vdash \ \tau_1 \cap \tau_2 :: \kappa}\ \text{WF-}\cap \qquad \frac{\Gamma \vdash \ \tau_1 :: \kappa \quad \Gamma \vdash \ \tau_2 :: \kappa}{\Gamma \vdash \ \tau_1 \cup \tau_2 :: \kappa}\ \text{WF-}\cup$$

$$\frac{\Omega_{VRC}, \Gamma \vdash \ \tau :: \Omega_{VRC(SN)}}{\Gamma \vdash \ \forall\, \tau :: \Omega_{VRC(SN)}}\ \text{WF-}\forall \qquad \frac{\Omega_{VRC}, \Gamma \vdash \ \tau :: \Omega_{VRC(SN)}}{\Gamma \vdash \ \exists\, \tau :: \Omega_{VRC(SN)}}\ \text{WF-}\exists$$

$$\frac{}{\Gamma \vdash \ \mathsf{const}(n) :: \Omega_{VRCSN}}\ \text{WF-}N \qquad \frac{\Gamma \vdash \ \tau :: \Omega_{VRCSN}}{\Gamma \vdash \ \mathsf{geq}(\tau) :: \Omega_{VRCSN}}\ \text{WF-}\geq$$

$$\frac{\Gamma \vdash \ \tau_1 :: \Omega_{VRCSN} \quad \Gamma \vdash \ \tau_2 :: \Omega_{VRCSN}}{\Gamma \vdash \ \mathsf{plus}(\tau_1, \tau_2) :: \Omega_{VRCSN}}\ \text{WF-}+$$

$$\frac{\Gamma \vdash \tau_1 :: \Omega_{VRCSN} \quad \Gamma \vdash \tau_2 :: \Omega_{VRCS}}{\Gamma \vdash \{\tau_1 : \tau_2\} :: \Omega_{VRC}}\ \text{WF-}\{\}$$

$$\frac{}{\Omega_V \subseteq_\kappa \Omega_{VR}}\ \Omega\text{\_vr1} \qquad \frac{}{\Omega_R \subseteq_\kappa \Omega_{VR}}\ \Omega\text{\_vr2} \qquad \frac{}{\Omega_{VRC} \subseteq_\kappa \Omega_{VR}}\ \Omega\text{\_vrc1} \qquad \frac{}{\Omega_{VRC} \subseteq_\kappa \Omega_C}\ \Omega\text{\_vrc2}$$

$$\frac{}{\Omega_{VRCS} \subseteq_\kappa \Omega_{VRC}}\ \Omega\text{\_vrcs1} \qquad \frac{}{\Omega_{VRCS} \subseteq_\kappa \Omega_S}\ \Omega\text{\_vrcs2}$$

$$\frac{}{\Omega_{VRCSN} \subseteq_\kappa \Omega_{VRCS}}\ \Omega\text{\_vrcsn1} \qquad \frac{}{\Omega_{VRCS} \subseteq_\kappa \Omega_N}\ \Omega\text{\_vrcs2}$$

$$\frac{\kappa_1 \subseteq_\kappa \kappa_2 \quad \kappa_2 \subseteq_\kappa \kappa_3}{\kappa_1 \subseteq_\kappa \kappa_3}\ \Omega\text{\_TRANS} \qquad \frac{\Gamma \vdash \ \tau :: \kappa_1 \quad \kappa_1 \subseteq_\kappa \kappa_2}{\Gamma \vdash \ \tau :: \kappa_2}\ \text{WF-}<:$$

Figure 5.1: Kinding (well-formedness and subkinding) rules for TML

a recursive type τ shown below to approximation k.

$$v \quad :_k \quad \underbrace{\mathsf{rec}\,(\forall(\mathsf{box}\,(\underline{0}) \cap \underline{1}))}_{\text{approximation } k}$$

Unfolding the recursive type would involve the application of this type to itself to give

$$\forall(\mathsf{box}\,(\underline{0}) \cap \underbrace{\mathsf{rec}\,(\forall(\mathsf{box}\,(\underline{0}) \cap \underline{1}))}_{\text{approximation } k})$$

The intersection type constructor is only nonexpansive, and the right component of intersection only has approximation k. Hence the complete type expression $(\tau\ \tau)$ has approximation k (and not $k+1$), as a result of which it is not contractive. This type is therefore not well formed, and the kinding derivation tree to prove this fails, as shown in Figure 5.2.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\{\Omega_{VRC}, \Omega_{VR}, \Gamma\}(0) \subseteq_\kappa \Omega_{VR}}{\Omega_{VRC}, \Omega_{VR}, \Gamma \vdash \underline{0} :: \Omega_{VR}}\ \text{WF-}VAR}{\Omega_{VRC}, \Omega_{VR}, \Gamma \vdash \mathsf{box}\,(\underline{0}) :: \Omega_{VRC}}\ \text{WF-}BOX\_RC \qquad \dfrac{\boxed{?}}{\Omega_{VRC}, \Omega_{VR}, \Gamma \vdash \underline{1} :: \Omega_{VRC}}}{\Omega_{VRC}, \Omega_{VR}, \Gamma \vdash \mathsf{box}\,(\underline{0}) \cap \underline{1} :: \Omega_{VRC}}\ \text{WF-}\cup}{\Omega_{VR}, \Gamma \vdash \forall(\mathsf{box}\,(\underline{0}) \cap \underline{1}) :: \Omega_{VRC}}\ \text{WF-}\forall}{\Gamma \vdash \mathsf{rec}\,(\forall(\mathsf{box}\,(\underline{0}) \cap \underline{1})) :: \Omega_{VR}}\ \text{WF-}REC$$

Figure 5.2: Failed kinding derivation tree for an illformed type

5.4.3 Semantics of the kinding judgment

In a conventional kinding system the syntax of the kinding judgment takes the following form:

$$x_0 :: \kappa_0, \ldots, x_n :: \kappa_n \vdash \tau :: \kappa$$

where the kind of a type expression τ may depend on the kinds of the type variables found within that expression. Since we use de Bruijn indices in type expressions in TML instead of explicit named variables, our kinding judgment mentions kinds for the indices occurring in the expression. We assume that the index $\underline{i}$ has kind κ_i. The kinding judgment now takes the form

$$\kappa_0, \ldots, \kappa_n \ \vdash \tau :: \kappa$$

where the kind of the i^{th} binding variable is implicit in the ordering of the list of kinds to the left of the turnstile. We refer to this list of kinds using Γ, and denote by Γ_i the i^{th} kind in the array

Giving semantics to the kinding judgments is slightly complicated due to the difference between the semantics for the bound variables in the expressions and the expressions themselves. In the judgment shown above, τ is an open term, i.e. it has de Bruijn indices occurring inside. The expressions that the indices represent, however, are assumed to be closed. This difference is removed by applying τ to a context ρ which would map each occurring de Bruijn in τ to some closed type $\rho(i)$.

Given this setup of associating each type variable to a kind, the natural model for kinds is predicates on closed types. This approach is good enough to model properties like validity, representability and constancy. However, we are unable to

correctly model contractiveness.

Contractiveness poses a problem because unlike the other kinds, it is a property of functions from types to types rather than types themselves. Thus we must formulate our kinding calculus so that kinds (Ω) are predicates on type functions. In this setup, the base kinds can defined as predicates on a type function argument f as follows :

$$
\begin{array}{lcl}
\Omega & : & (\mathsf{CTymap} \to \mathsf{CTymap}) \to o \\
\Omega_V & \stackrel{\text{def}}{=} & \lambda f.\ \forall\, \breve{\tau}.\ \text{valid}(f\ \breve{\tau}) \\
\Omega_R & \stackrel{\text{def}}{=} & \lambda f.\ \forall\, \breve{\tau}.\ \text{representable}\ (f\ \breve{\tau}) \\
\Omega_C & \stackrel{\text{def}}{=} & \lambda f.\ \forall\, \breve{\tau}, k, v.\ \ (\lfloor f\ \breve{\tau} \rfloor_{k+1})\ v\ \equiv\ (\lfloor f\ \lfloor \breve{\tau} \rfloor_k \rfloor_{k+1})\ v \\
\Omega_S & \stackrel{\text{def}}{=} & \lambda f.\ \forall\, \breve{\tau}.\ \ \forall\, k, v_1, v_2. \\
 & & (\text{state}(v_1) = \text{state}(v_2)\ \wedge \text{root}(v_1)[0] = \text{root}(v_2)[0]) \\
 & & \implies\ (f\ \breve{\tau})\ k\ v_1\ \equiv\ (f\ \breve{\tau})\ k\ v_2 \\
\Omega_N & \stackrel{\text{def}}{=} & \lambda f.\ \forall \breve{\tau}.\ \exists\ n.\ \forall\, \rho.\ \ (f\ \breve{\tau})\ \approx\ (\mathsf{const}(n)\ \rho)
\end{array}
$$

The definitions of Ω_V and Ω_R are obvious, and since we raise the level from types to type functions, we expect that the kind properties hold no matter what (closed) type arguments the (closed) type functions are applied to. The definition of Ω_C follows the definition of contractiveness. When determining that an expression is a TML scalar type (rather than the default TML environment), we first extract a closed type expression from f by applying it to some closed type argument $\breve{\tau}$. Then, we ensure that for any two concrete values v_1 and v_2 that agree on their state components and their first root vector components, one of them satisfies $(f\ \breve{\tau})$ iff the other does. For the last kind, Ω_N, we simply ensure that there exists a number n such that the type $(f\ \breve{\tau})$ for any closed type $\breve{\tau}$ is equivalent to a constant type for

n.

The added complexity of the kinding system due to type functions should be hidden from the users of this system, and therefore, we would like to keep the simple syntax of the kinding judgment unchanged. The semantics of the kinding judgment can be given as

$$\begin{array}{lcl} \Gamma & : & Nat \to \Omega \\ R & : & Nat \to (\mathsf{CTymap} \to \mathsf{CTymap}) \\ \Gamma \vdash \tau :: \kappa & \stackrel{\text{def}}{=} & \forall R.\ (\forall i.\ \Gamma_i(R_i)) \Rightarrow \kappa(\lambda\breve{\tau}.\ \tau(\lambda i.\ R_i(\breve{\tau}))) \end{array}$$

In this definition, we use a context R which is modelled as a partial map from naturals to closed-type functions. R_i denotes the i_{th} component of R. For any R such that each of its components satisfies the kinds given in Γ (i.e. $\forall\, i.\, \Gamma_i(R_i)$), the kinding judgment for τ should hold. Since τ itself is an open type, we cannot directly apply the kind κ to it. The type τ is therefore raised to the level of a closed-type function thus : first, we close the type τ with some closed type τ_c and enclose it in a function (taking a closed-type argument)

$$\lambda\breve{\tau}.\ (\tau\ \tau_c)$$

where the argument τ_c is itself constructed from the closed-type function context R and the argument $\breve{\tau}$ by applying each component R_i of the context to $\breve{\tau}$ and thus lowering it to the level of a closed type.

$$(\lambda i.\ R_i(\breve{\tau}))$$

The syntactic rules become machine-checked lemmas in the semantic models described above, and so soundness follows immediately. With the syntactic rules, however, rules, however, we do not have semantic completeness. i.e. there are contractive expressions that would not be found by our system. Consider, for example, the type expression

$$\mathsf{rec}\,(\mathsf{leq}(2)\ \cap\ \mathsf{gt}(2)\ \cap\ \underline{0})$$

The first intersection results in a null range, and thus there cannot be a value that can satisfy this type. Therefore, this type is semantically equivalent to the bottom type "$\bot$", and hence is contractive. However, our kinding rules would not be able to prove this type as being contractive. However, a higher-level client of TML like LTAL is expected to use only a restricted class of type expressions that would not be susceptible to this problem.

Chapter 6

Modelling Instructions In TML

Chapter 4 gave semantics to types and environments in TML. In this chapter, I shall give semantics to high-level typed instructions. As in traditional Hoare logic, we can model instructions as predicate transformers, i.e., an instruction may be characterised by the manner in which it transforms preconditions to give the strongest postconditions (or transforms postconditions to give the weakest preconditions). Since our safety policy for FPCC is type safety, the scope of these predicates is limited to typing judgments on the values of variables occurring in programs.

6.1 Arithmetic instructions

We can model an instruction as a relation between two type environments, ϕ, and ϕ', where ϕ gives the precondition for the instruction to be safely executed, and ϕ' gives the postcondition resulting from the execution of the instruction. For the arithmetic `add` instruction, shown as (`add` $v_i \leftarrow v_j + v_k$), one of the conditions requires the source values to be integers. So we have the precondition $v_j : \text{T\_INT}, v_k : \text{T\_INT}$.

This condition is captured by the TML type environment given by

$$\phi_1 = \phi \cap \{j : \mathsf{int}_{32}\} \cap \{k : \mathsf{int}_{32}\}$$

where ϕ may impose additional constraints on other values. The environment ϕ_2 denotes the type environment that should hold after the addition. In this environment, all previous typing judgements about v_i are invalidated. Additionally, in ϕ_2, v_i gets the integer type, and this is encoded[1] in TML as

$$\phi_2 = \phi_1[i \mapsto \mathsf{int}_{32}]$$

We can have a much more general formulation for `add` where we quantify over the values that the arguments take. For example, for any values n and m that v_j and v_k may contain, i.e.,

$$\phi_1 \subseteq \{j : \mathsf{int}_{=}(n), k : \mathsf{int}_{=}(m)\}$$

we may encode the resultant environment as

$$\phi_2 = \phi_1[i \mapsto \mathsf{int}_{=}(\mathsf{plus}(n, m))]$$

In real programs, however, additions occur in contexts other than those shown above above, most notably for address arithmetic needed for subsequent loads and stores. In these situations, we have one variable v_j holding the base address b, and another variable v_k holding the offset l. To safely load from the address obtained from

[1]The update operator ($[\mapsto]$) in the given postcondition is difficult to model as a Tymap constructor. Since the update operator is only required in the semantics for instructions, it is internalised in semantics associated with instructions and the model is discussed in Section 6.4.

$v_j + v_k$, the typing precondition can be given as :

$$\phi_1 = \phi \cap \{j : \mathsf{field}(\mathsf{const}(c), \tau)\} \cap \{k : \mathsf{int}_=(\mathsf{const}(c))\}$$

The result of the addition would be given by

$$\phi_2 = \phi_1[i \mapsto \mathsf{field}(\mathsf{const}(0), \tau)] \quad \text{or, equivalently,}$$
$$\phi_2 = \phi_1[i \mapsto \mathsf{box}\,(\tau)]$$

For the move instruction, ($\texttt{mov}\ i \ \leftarrow \ j$), the pre- and postconditions, ϕ_1 and ϕ_2 respectively can be given simply by

$$\phi_1 = \phi \cap \{j : \tau\} \ \text{ and } \ \phi_2 = \phi_1[i \mapsto \tau].$$

6.2 Control-flow instructions

Unlike traditional Hoare logic which reasons about structured control-flow constructs, in FPCC we must reason about explicit control transfer instructions like (conditional) jumps. To ensure that a jump is safe, the typing judgement holding at the jump instruction should be at least as strong as the typing judgement at the jump target. Therefore, to model such instructions, we also require Γ, a function that maps memory locations to the typing judgements that must hold at those addresses. For a set of labels $l_1, l_2, \ldots, l_n$, we would have

$$\Gamma = \{l_1 \mapsto \mathsf{codeptr}\,(\phi_{l_1}), l_2 \mapsto \mathsf{codeptr}\,(\phi_{l_2}), \ldots, l_n \mapsto \mathsf{codeptr}\,(\phi_{l_n})\}$$

Given that v_l contains the value n (the branch target location) and $\Gamma[n] = \mathsf{codeptr}\,(\phi')$, the precondition for the instruction `jump v_l` is

$$\phi_1 = \phi \cap \{l : \mathsf{codeptr}\,(\phi')\}$$

where some ϕ may give the typing judgements for other variables. Due to the control transfer, we do not reach the (statically) next instruction, and as a result, the postcondition for this instruction is simply $\bot$.

A general model for an instruction may thus be given by the TML instruction constructor instr that takes the label map Γ, the precondition ϕ_1, and the postcondition ϕ_2. Using this constructor, we give a TML instruction model for the machine-level `add` instruction as :

$$\begin{aligned} \texttt{tml-add i} \leftarrow \texttt{j} + \texttt{k} \quad &\stackrel{\text{def}}{=} \quad \forall m, n, \Gamma, \phi.\ \mathsf{instr}(\Gamma, \phi_1, \phi_1[i \mapsto \mathsf{int}_=(\mathsf{plus}(n, m))]) \\ &\qquad \text{where } \phi_1 = \phi \cap \{j : \mathsf{int}_=(n), k : \mathsf{int}_=(m)\} \end{aligned}$$

With this general form for instructions, we can also define subtyping on instructions that would allow us to relate a principal form of an instruction to a specialised form such as the integer- and pointer-arithmetic forms of the `add` instruction. Subtyping is denoted by $\subset_i$, and defined as

$$\mathsf{instr}(\Gamma_1, \phi_1, \phi_1') \subset_i \mathsf{instr}(\Gamma_2, \phi_2, \phi_2') \stackrel{\text{def}}{=} \Gamma_2 \subseteq \Gamma_1 \ \wedge\ \phi_2 \subseteq \phi_1 \ \wedge\ \phi_1' \subseteq \phi_2'$$

6.3 Memory access instructions

In FPCC we must also be able to reason about the concrete instruction sequences that implement macro instructions like `malloc`, so that optimised code that rear-

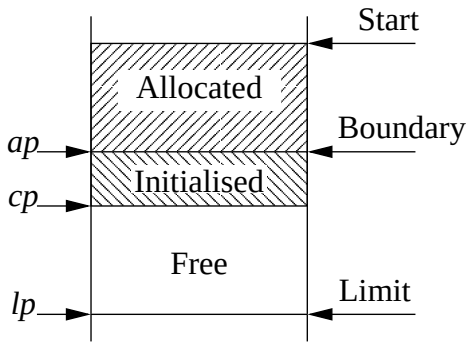

Figure 6.1: Memory Allocation

ranges these sequences can also be proved safe. For `malloc`, in particular, it is necessary to be able to argue about the safety of store instructions that initialise new values in memory. (Ahmed et al. [6] describe how to reason about store instructions for mutation of existing values). To explain the semantics we give to the store instruction, I shall outline the particular memory management scheme we use in this project. Semantics of the store instructions are dependent on this scheme.

I assume a restricted memory model where allocations are performed in a contiguous region in order i.e., every location is allocated after all preceding locations have been allocated. As shown in figure 6.1, the heap is a region in memory that begins at "Start" and ends at "Limit", and all locations till "Boundary" are allocated. We have two special purpose variables, v_{ap}, the allocation pointer which keeps track of the boundary, and, v_{lp} the limit pointer, which keeps track of the allocation limit. When we store multiple words, it is efficient to first store all words (initialise the locations) and then bump the allocation pointer (mark the locations as allocated). Furthermore, an optimising compiler might also reschedule other instructions before incrementing the allocation pointer. However, such optimisations make it difficult to reason about the machine state for all the intermediate instructions before the

allocation pointer is incremented. To allow us to reason about the safety of these optimisations, we make use of the "current boundary pointer" variable v_{cp}, a *virtual* variable which points to the next word to store in. A *virtual* variable is one that would not correspond to any real machine register or memory location; it is merely used as a bookkeeping variable in the semantic model. Virtual variables are similar to *logical* or *ghost* variables used in Hoare logic.

We always start out with the initial condition $v_{cp} = v_{ap}$ for all programs, and since the variable v_{cp} is virtual, it is never an argument to any instruction.

For an initialising store $(m[v_a + c] \leftarrow v_b)$, we must ensure that *i*) we store at the first unwritten position $(v_a + c = v_{cp})$, and *ii*) we have enough space to store $(v_{cp} < v_{lp})$. When performing multiple writes, we bump the allocation pointer after all the writes. We use the virtual v_{cp} variable to keep track of the next place to write to. This allows the program to interlace allocation with other operations and also to handle code sequences that safely break apart an opaque `malloc` instruction. The precondition for the store is therefore given by ϕ' such that

$$\begin{array}{rcll} \phi'(a,b,c) & \equiv & \phi \cap \mathsf{relate}_{\leq}(\mathsf{id},\mathsf{id})(ap,cp) & \text{“}v_{cp} \geq v_{ap}\text{”} \\ & & \cap \{cp : \mathsf{int}_{=}(l)\} & \text{“}v_{cp} = l\text{”} \\ & & \cap \mathsf{relate}_{=}(\mathsf{offset}\ c, \mathsf{id})(a,cp) & \text{“}v_a + c = v_{cp}\text{”} \\ & & \cap \{b : \tau\} & \\ & & \cap \mathsf{relate}_{<}(\mathsf{id},\mathsf{id})(cp,lp) & \text{“}v_{cp} < v_{lp}\text{”} \end{array}$$

The postcondition, ϕ'' updates v_{cp} to point to the next location, and gives the type $\mathsf{field}(c,\tau)$ to variable a.

$$\phi''(a,b,c) \equiv \phi'(a,b,c)[cp \mapsto \mathsf{int}_{=}(l+1), a \mapsto \mathsf{field}(c,\tau)]$$

Therefore, the semantics for the store instructions is

$$\texttt{tml-store}(\texttt{a},\texttt{b},\texttt{c}) = \quad \forall \Gamma, \phi, l, \tau.\ \mathsf{instr}(\Gamma, \phi'(a,b,c), \phi''(a,b,c))$$

Note that the allocation pointer has not been affected by this store, and so the locations that have been just written to are not well typed with respect to the allocation set. It is therefore not possible to reason about the safety of accessing these values (via load instructions) until the allocation pointer is bumped to be equal to the virtual current pointer in some future machine instruction.

As an example, consider a program that stores two words to memory, written in pseudocode (and corresponding Sparc assembly, assuming that register `w1`, `w2`, and the allocation pointer, `ap` respectively) as:

(1) `store w1`	`(st %o1, [%o5+0])`
(2) `update allocation pointer`	`(add %o5, 1, %o5)`
(3) `store w2`	`(st %o2, [%o5+0])`

Figure 6.2 illustrates the changes to the memory contents after each store. At the beginning, we have v_{cp} and v_{ap} both pointing to the start of the allocation area. After storing a word, the virtual v_{cp} is incremented using the TML store instruction semantics, thus leaving a 1-word area that is initialised but not allocated. The second instruction explicitly updates the allocation pointer v_{ap}, thereby turning this area into an allocated area, and updating the allocation boundary. The third instruction stores the second word and adds another 1-word initialised area.

We also use the notion of virtual variables to give semantics to conditional branch instructions. Conditional branches require references to condition codes. For example, in the branch-if-equal instruction (`beq l`) we jump to location l if the zero flag is set, else go to the next instruction. We model each condition code as a separate virtual variables. For example, the "zero" condition is modelled as a variable v_z. We assume $\{z : \mathsf{const}(1)\}$ if the zero flag is set, and $\{z : \mathsf{const}(0)\}$

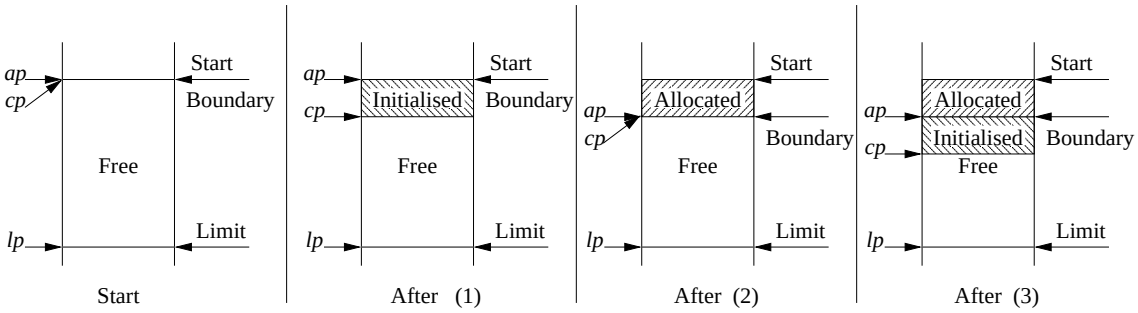

Figure 6.2: Memory Allocation

otherwise. The corresponding TML instruction can be written as

$$\texttt{tml-beq l} \stackrel{\text{def}}{=} \forall n, \Gamma, \phi.\ \mathsf{instr}(\Gamma, \phi_1, \phi_2)$$
$$\text{“ where } \phi_1 = \phi \cap \{l : \mathsf{codeptr}\,(\Gamma[n])\} \cap \{z : \mathsf{int}_{=}(\mathsf{const}(1))\}$$
$$\text{and } \phi_2 = \phi \cap \{l : \mathsf{codeptr}\,(\Gamma[n])\} \cap \{z : \mathsf{int}_{=}(\mathsf{const}(0))\}\text{”}$$

The compare instruction `cmp` allows us to relate the compared value with the condition codes. For example, (considering only the equal flag), after the instruction (`cmp a b`), we may assert that the equal flag is set if $a = b$ and not set otherwise. This may be encoded using TML constructors as

$$\begin{aligned}\phi_1 \quad = \quad & (\mathsf{relate}_{=}(\mathsf{id}, \mathsf{id})(a, b) \cap \{z : \mathsf{int}_{=}(\mathsf{const}(1))\}) \\ & \cup (\mathsf{relate}_{\neq}(\mathsf{id}, \mathsf{id})(a, b) \cap \{z : \mathsf{int}_{=}(\mathsf{const}(0))\})\end{aligned}$$

Using ϕ_1, the `cmp` instruction may be encoded as the TML instruction

$$\texttt{tml-cmp a b} \stackrel{\text{def}}{=} \forall \Gamma, \phi.\ \mathsf{instr}(\Gamma, \phi, \phi \cap \phi_1)$$

This encoding captures the dependency of the condition codes on the two values being compared, and allows an optimising compiler to insert other instructions

between the comparison and the branch instruction.

Using the TML instructions described above, we can provide semantic models for LTAL instructions. Figure 6.3 lists the TML instruction models for some of the main LTAL instructions. These models are used in the construction of program safety proofs as will be discussed in Chapter 7.

6.4 The semantic model for instructions

The TML instructions just described are now given a semantic model in terms of the machine state. The pre- and postconditions for the instr constructor are TML environments and therefore have the same semantic model as that described in Chapter 4 for TML environments. Similarly, the label-to-environment map Γ can also be described as a TML environment of the form :

$$\{l_1 : \mathsf{codeptr}\,(\phi_1),\ l_2 : \mathsf{codeptr}\,(\phi_2),\ \ldots,\ l_n : \mathsf{codeptr}\,(\phi_n)\}$$

where the set $\{l_1, l_2, \ldots, l_n\}$ contains all the branch targets. A major difference between the two environments, however, is the set of values that they will be applied to. The ϕs are applied to values that reflect the contents of the registers, while Γ is applied to the list of program locations. The register bank can be modelled as a function from the register number (a natural) to its contents (a natural), while the set of locations can be modelled simply as idvec, the identity function on natural numbers. Therefore, we can define a TML instruction as a relation on two concrete values cv and cv' where the root vectors represent the contents of the machine

Ltal Instructions	TML models	Remarks
LTAL-MOV	`tml-mov`	
LTAL-MOVHI	`tml-mov`	(for Sparc `sethi`)
LTAL-WRY	`tml-mov`	(for Sparc `wry`)
LTAL-LOAD	`tml-load`	
LTAL-STORE	`tml-store`	
LTAL-LBLADD	`tml-add`	(Address arithmetic)
LTAL-RECORD-OR	`tml-mov`	(store allocation pointer in register)
LTAL-RECORD-STORE	`tml-store`	(store allocation pointer in memory)
LTAL-ALLOCA	*macro instruction*	
LTAL-SEL	`tml-load`	(select field of record)
LTAL-SUB	`tml-load`	(select array element)
LTAL-ARITH	`tml-add`	(arithmetic instructions, e.g. add)
LTAL-INJN-NULL		Type coercion
LTAL-INJN-OR	`tml-mov`	(Inject value into a union)
LTAL-GETTAG	`tml-load`	(Get tag of boxed value)
LTAL-AARITH	`tml-add`	(Address arithmetic, e.g. add)
LTAL-IARITH	`tml-add`	(Immediate arithmetic, e.g. add)
LTAL-CMPCC	`tml-cmp`	(Comparisons)
LTAL-TESTFULL	`tml-cmp`	(Test if memory full)
LTAL-TESTBOX	`tml-cmp`	(Test if value is boxed)
LTAL-CMPCCI	`tml-cmp`	(Comparison)
LTAL-OPEN	`tml-mov`	(Move and open an existential value)
LTAL-UPDATE	`tml-store`	(Update a memory location)
LTAL-ASSIGN	`tml-store`	(Update a array index)
LTAL-IFFULL	`tml-bge`	(Branch if memory full)
LTAL-IF	`tml-beq`	(Generic branch, e.g. branch-if-equal)
LTAL-IFTAG	`tml-beq`	(Branch on tag of a boxed value)
LTAL-IFBOXED	`tml-bge`	(Branch if value is boxed)
LTAL-CALL-LBL	`tml-jump`	(Transfer control to label)
LTAL-CALLN		(Fall through)

Figure 6.3: TML models for LTAL instruction

register banks. The TML instruction constructor instr is defined as :

$$
\begin{aligned}
\mathsf{instr} \stackrel{\text{def}}{=}\ & \lambda(\Gamma, \phi_1, \phi_2).\ \lambda\rho, k, cv, cv'. \\
& (\ (k > 0)\ \wedge\ (\phi_1\ \rho\ k\ cv)\ \wedge\ (\Gamma\ \rho\ k\ \langle \text{state}(cv), \mathsf{idvec}\rangle)\ )\ \Longrightarrow \\
& ((\text{reg-vector}(\text{state}(cv))(pc) = \text{reg-vector}(\text{state}(cv'))(pc) \\
& \Longrightarrow\ (\phi_2\ \rho\ (k-1)\ cv')) \\
& \wedge\ (\forall \phi_3.\ (\text{reg-vector}(\text{state}(cv))(pc) \neq \text{reg-vector}(\text{state}(cv'))(pc)) \\
& \wedge\ (\Gamma \subseteq \{\text{reg-vector}(\text{state}(cv'))(pc) : \mathsf{codeptr}\,(\phi_3)\}) \\
& \Longrightarrow\ (\phi_3\ \rho\ (k-1)\ cv')))
\end{aligned}
$$

Intuitively, this definition says that if we start in a machine state that satisfies the precondition ϕ_1 to degree k (i.e. if k machine steps can be safely taken starting in this state), and if the label environment Γ also holds to approximation k (i.e. k steps may be safely taken starting at any label location), then it should be possible to execute the current instruction and end in a state (given by cv') from which $k-1$ more steps may be safely taken. This requires an analysis of two cases: in the first case, the instruction is not a control-transfer instruction (so that the program counter remains unchanged), and as a result, the second environment ϕ_2 must be satisfied by the resulting concrete value cv' to approximation $k-1$. If it is a control-transfer instruction (and the program counter is changed), then cv' must instead satisfy the target label environment to approximation $k-1$. This allows a total of k steps to be executed from the current instruction regardless of whether it is a control-transfer instruction. This definition reflects the notion of type preservation for the indexed model of types.

6.4.1 Semantic model for quantifiers and the update operator

The descriptions of instructions also required us to quantify over types. These were shown simply using the forall quantifier, $\forall$. However, we must make a new instruction constructor for such quantification, $\forall_i$, and its semantic model is given as given as

$$\begin{aligned}\forall_i \stackrel{\text{def}}{=}\ & \lambda(\Gamma, \phi_1, \phi_2).\ \lambda\rho, k, cv, cv'. \\ & \forall\check{\tau}.\ \text{valid}(\check{\tau}) \implies \mathsf{instr}\ (\Gamma, \phi_1, \phi_2)\ (\check{\tau} :: \rho)\ k\ cv\ cv'\end{aligned}$$

which adds the quantified type to the context if it is valid, in a way similar to the quantifier over Tymap constructors.

To give a semantic model for the update operator used in the descriptions of instructions, I shall outline a technique due to Gang Tan. In this technique, we first define a predicate notindom.

$$\begin{aligned}\text{notindom} \stackrel{\text{def}}{=}\ & \lambda i, \lambda\phi.\ \ \forall k, v_1, v_2. \\ & \qquad (\forall\ j.\ (j \neq i) \implies ((\text{root}(v_1))(j) = (\text{root}(v_2))(j))) \\ & \implies \phi\ k\ v_1 \equiv \phi\ \rho\ k\ v_2\end{aligned}$$

This predicate allows us to express the fact that the type environment ϕ does not have any constraints for the i^{th} component of the root vector of a concrete value. It can be extended to account for multiple components (e.g. i and j), which we abbreviate as notindom(i, j). Using this predicate, we define a new form of bounded quantification, $\forall_{\text{notindom}(i)}$, which quantifies over all environments which have no binding for the i^{th} component of the root vector in the way similar to the range

bounded quantification described in Section 4.3.1.

$$\begin{aligned}\forall_{\text{notindom}(i)} \stackrel{\text{def}}{=}\ & \lambda(\Gamma, \phi_1, \phi_2).\ \lambda\rho, k, cv, cv'. \\ & \forall\phi.\ \text{valid}(\phi)\ \wedge\ \text{notindom}(i, \phi)\ \Longrightarrow \\ & \qquad \mathsf{instr}\ (\Gamma, \phi_1, \phi_2)\ (\phi :: \rho)\ k\ cv\ cv'\end{aligned}$$

Using this quantification, we can model an instruction such as `add` that requires updates as

$$\begin{aligned}\mathtt{add}(v_i \leftarrow v_j + v_k) \stackrel{\text{def}}{=}\ & \forall\Gamma, n, m.\ \forall_{\text{notindom}(i)}\ \phi. \\ & \mathsf{instr}(\Gamma, \phi \cap \{j : \mathsf{int}_=(n)\} \cap \{k : \mathsf{int}_=(m)\}, \\ & \qquad \phi \cap \{i : \mathsf{int}_=(\mathsf{plus}(n, m))\})\end{aligned}$$

The `add` instruction can be used in two ways, in the first one, the source and destination variables are distinct. In this case, the instantiation of ϕ, the type environment has bindings for both the source variables. For example given an add instruction `add v1 <- v3 + v4`, ϕ has bindings given by $\{3 : \mathsf{int}_=(n)\} \cap \{4 : \mathsf{int}_=(m)\}$. The result of the addition removes the binding for `v1`, but retains that for `v3` and `v4`. On the other hand, we could also use `add` to overwrite one of the source registers. For example, given the instruction `add v3 <- v3 + v4` we wish to retain the original binding only for `v4`. In this case, we would instantiate ϕ with $\{4 : \mathsf{int}_=(m)\}$. This would give an environment with the new binding for variable `v3`, but would not retain the earlier binding.

Thus, the list of instruction types in TML can be given as:

$$\begin{array}{lll} \textit{Instructions} & \iota & ::= \quad \texttt{tml-add}\ (\texttt{n}_1, \texttt{n}_2, \texttt{n}_3) \mid \texttt{tml-load}\ (\texttt{n}_1, \texttt{n}_2, \texttt{n}_3) \mid \texttt{tml-jump}\ (\texttt{n}) \\ & & \mid \texttt{tml-mov}\ (\texttt{n}_1, \texttt{n}_2) \mid \texttt{tml-store}\ (\texttt{n}_1, \texttt{n}_2, \texttt{n}_3) \mid \texttt{tml-beq}\ (\texttt{n}) \\ & & \mid \texttt{tml-cmp}\ (\texttt{n}_1\ \texttt{n}_2) \end{array}$$

where the TML instruction types are defined as:

$$\begin{array}{lcl}
\texttt{tml-add}\ (\texttt{i}, \texttt{j}, \texttt{k}) & \stackrel{\text{def}}{=} & \forall m, n, \Gamma. \forall_{\text{notindom}(i)} \phi. \\
& & \mathsf{instr}(\Gamma, \phi_1, \phi \cap \{i : \mathsf{int}_=(\mathsf{plus}(n, m))\}) \\
& & \text{where } \phi_1 = \phi \cap \{j : \mathsf{int}_=(n), k : \mathsf{int}_=(m)\} \\
\texttt{tml-load}\ (\texttt{i}, \texttt{j}, \texttt{c}) & \stackrel{\text{def}}{=} & \forall n, \Gamma, \forall_{\text{notindom}(i)} \phi. \mathsf{instr}(\Gamma, \phi_1, \phi \cap \{i : \tau\}) \\
& & \text{where } \phi_1 = \phi \cap \{j : \mathsf{field}(c, \tau)\} \\
\texttt{tml-jump}\ (\texttt{l}) & \stackrel{\text{def}}{=} & \forall n, \Gamma, \phi. \mathsf{instr}(\Gamma, \phi \cap \{l : \mathsf{codeptr}\,(\Gamma[n])\}, \bot) \\
\texttt{tml-mov}\ (\texttt{i}, \texttt{j}) & \stackrel{\text{def}}{=} & \forall \Gamma, \forall_{\text{notindom}(i)} \phi. \mathsf{instr}(\Gamma, \phi \cap \{j : \tau\}, \phi \cap \{i : \tau\}) \\
\texttt{tml-store}\ (\texttt{i}, \texttt{j}, \texttt{c}) & \stackrel{\text{def}}{=} & \forall \Gamma, \forall_{\text{notindom}((i,cp))} \phi. \\
& & \mathsf{instr}(\Gamma, \phi', \phi \cap \{cp : \mathsf{int}_=(l+1)\} \cap \{i : \mathsf{field}(c, \tau)\}) \\
& & \text{where } \phi' = \phi \ \cap\ \mathsf{relate}_{\leq}(\mathsf{id}, \mathsf{id})(ap, cp) \ \cap\ \{cp : \mathsf{int}_=(l)\} \\
& & \qquad \cap\ \mathsf{relate}_=(\mathsf{offset}\ c, \mathsf{id})(i, cp) \ \cap\ \{j : \tau\} \\
& & \qquad \cap\ \mathsf{relate}_<(\mathsf{id}, \mathsf{id})(cp, lp) \\
\texttt{tml-beq l} & \stackrel{\text{def}}{=} & \forall n, \Gamma, \phi. \\
& & \mathsf{instr}(\Gamma, \phi \cap \{l : \mathsf{codeptr}\,(\Gamma[n])\} \cap \{z : \mathsf{int}_=(\mathsf{const}(1))\}, \\
& & \qquad \phi \cap \{l : \mathsf{codeptr}\,(\Gamma[n])\} \cap \{z : \mathsf{int}_=(\mathsf{const}(0))\}) \\
\texttt{tml-cmp i j} & \stackrel{\text{def}}{=} & \forall \Gamma, \forall_{\text{notindom}(z)} \phi. \mathsf{instr}(\Gamma, \phi, \phi \cap \phi_1) \\
& & \text{where } \phi_1 = (\mathsf{relate}_=(\mathsf{id}, \mathsf{id})(i, j) \cap \{z : \mathsf{int}_=(\mathsf{const}(1))\}) \\
& & \qquad \cup\ (\mathsf{relate}_{\neq}(\mathsf{id}, \mathsf{id})(i, j) \cap \{z : \mathsf{int}_=(\mathsf{const}(0))\})
\end{array}$$

Chapter 7

Generating Program Safety Proofs

Having shown the semantics for types, environments, and instructions, I shall now discuss the important steps in constructing safety proofs of machine code using these semantic models. One of the challenges in producing proofs of machine code is that there must be no semantic gaps in the safety proofs. Earlier approaches to PCC and language-based security have been prone to such gaps. For example, in Necula's PCC system, the safety proof is generated by the component called the Verification Condition Generator (VCGen) with respect to a verification condition (or a safety theorem) derived from a program. The system assumes that this theorem corresponds correctly to the program. An error in the VCGen would allow unsafe programs to be executed. The concrete-machine-specific TAL system TALx86 [40] is very close to the x86 assembly-level language. However, TALx86 still reasons about safety at the assembly level, and then uses a trusted assembler[1]

[1]In FPCC, we do not trust an assembler, however we do trust the component of the TCB that captures the syntax and semantics of machine instructions. Michael and Appel [37] have shown how to have a minimal amount of trust in this component by carefully engineering it to be highly factored and to have a very small axiomatic base. This component consists of predicates over the machine state, and the disassembler is a logic program whose correctness is proved from the axioms in the TCB, in a way similar to the rules of TML being proved correct using the semantic

to produce the machine-level program. The operational semantics for TAL involves a relation between abstract machine states given by a triple of the heap, register bank, and instruction sequences rather than purely based on the raw machine memory and register bank. The soundness of the TALx86 type system has been shown by a rigorous manually constructed proof. This soundness is still with respect to an abstract machine, and the relation between the type system of TALx86 and the concrete machine operational semantics does not have a machine-checked proof. For a foundational approach, the absence of this proof results in a semantic gap in the system.

The FPCC system avoids semantic gaps in safety proofs by directly reasoning about the safety of the actual machine code. In this chapter, I shall describe various judgements and rules involved in the generation of syntactic safety proofs, and how we can use TML to give semantic models for them to yield truly foundational proofs.

7.1 High-level structure of semantic safety proofs

7.1.1 Syntactic proof technique

The program safety proof in FPCC does not use techniques used in syntactic presentations of program safety proofs. I shall first outline the syntactic program safety proof technique, and then describe the proof technique that we use in FPCC.

Syntactic presentations of program safety proofs involve subject reduction and progress theorems based on the operational semantics of typed assembly language instructions. To relate the typed assembly language (LTAL, in our case) to an assembly language of a concrete architecture (Sparc, in our case), a simulation theorem

model. As a result, the logic program itself does not need to be trusted.

is proved which shows that one step in the typed assembly language corresponds exactly to one step in the concrete architecture assembly language.

Consider, for example, the tiny typed assembly language, T, given in Figure 7.1, along with fragments of its semantics enough only to illustrate this technique. To determine the safety of a program in T, we first prove the two lemmas :

- *Progress* : If P is well formed ($\vdash P$), then there exists a P' such that P evaluates to P' in one step ($P \longmapsto P'$).
- *Preservation* : If P is well formed, and $P \longmapsto P'$, then P' is wellformed.

This allows us to prove that the typed assembly language always steps to a well typed (and hence, safe) state. Next, this abstract semantics is related to the actual machine state in terms of the memory contents and the program counter given by the translation relation ($\Longrightarrow$) :

$$\frac{\begin{array}{c}\text{“Heap } \mathsf{H} \text{ corresponds to memory state } M\text{”} \\ \text{“}\mathsf{R} \text{ corresponds to machine register bank } R\text{”} \\ \text{“Instruction sequence } I \text{ resides in memory at location } l\text{”} \\ \text{“Program counter } (pc) \text{ points to } l\text{”}\end{array}}{(\mathsf{H}, \mathsf{R}, I) \Longrightarrow (M, R, pc)} \text{ (TRANSLATE)}$$

Finally, we need a simulation relation between the assembly step and the machine step, i.e. we show that if assembly state P steps to P' ($P \overset{T}{\longmapsto} P'$), and P translates to machine state S ($P \Longrightarrow S$), and if machine state S steps to S', then $P' \Longrightarrow S'$. This simulation relation connects the safety of the assembly to the safety of the machine code : since $P \overset{T}{\longmapsto} P'$ is a relation on only safe states, the machine states corresponding to P and P' must be safe.

Syntax

Type	τ	::=	$\mathsf{int} \mid \tau \times \tau$
Register Environment	Γ	::=	$\{r_1 : \tau_1, r_2 : \tau_2, \ldots, r_n : \tau_{31}\}$
Register Value	v_r	::=	N
Heap Values	v_h	::=	$\langle N_0, N_1, \ldots, N_{n-1}\rangle$
Register Bank	R	::=	$\{r0 \mapsto v_{r0}, r1 \mapsto v_{r1}, \ldots, r31 \mapsto v_{r31}\}$
Heap	H	::=	$\{0 \mapsto v_{h0},\ 1 \mapsto v_{h1},\ \ldots,\ n \mapsto v_{hn}\}$
Instruction	ι	::=	$\mathsf{load}\ r_d, r_s, c$
Instruction Sequence	I	::=	$\iota; I \mid \blacksquare$
Natural	N	::=	$\{0, 1, 2, \ldots\}$

Operational Semantics

$$(\mathsf{H}, \mathsf{R}, \mathsf{load}\ r_d, r_s, c; I) \xrightarrow{T} (\mathsf{H}, \mathsf{R}\{r_d \mapsto v_i\}, I)$$
where $0 \le c < n$,
$\mathsf{H}(\mathsf{R}(r_s)) = \langle v_0, \ldots, v_c, \ldots, v_{n-1}\rangle$

$$(\mathsf{H}, \mathsf{R}, \blacksquare) \xrightarrow{T} (\mathsf{H}, \mathsf{R}, \blacksquare)$$

Static Semantics

$\vdash P$ (Program wellformed)

$$\frac{(0 \le i < 32 \quad \mathsf{H} \vdash \mathsf{R}(i) : \Gamma(i)) \quad \Gamma \vdash I\ \mathit{safe}}{\vdash\ (\mathsf{H}, \mathsf{R}, I)}\ (\textsc{prog})$$

$\Gamma \vdash I\ \mathit{safe}$ (Instructions wellformed)

$$\frac{\Gamma(r_s) : \tau_0 \times \ldots \times \tau_i \times \ldots \times \tau_n \quad \Gamma[r_d \mapsto \tau_i] \vdash I\ \mathit{safe}}{\Gamma \vdash \mathsf{load}\ r_d, r_s, c;\ I\ \mathit{safe}}\ (\textsc{load})$$

$$\frac{}{\Gamma \vdash \blacksquare\ \mathit{safe}}\ (\blacksquare)$$

Figure 7.1: Syntax of T

7.1.2 Semantic proof technique

In the FPCC project, the semantic approach has influenced the program safety proof technique. As opposed to the syntactic approach, in FPCC all typing judgements related to proving safety are given semantic models in terms of TML constructors. The well-typedness of a program now becomes a theorem based on this semantic model. Then, another theorem that states that well-typed programs are safe is also proved with respect to this semantic model. The LTAL program instructions are given operational semantics in terms of the trusted Sparc step relation, hence we also do not need an explicit simulation proof. The main predicate we prove in this approach outlined by Appel and McAllester[11] is safe(r, m). This predicate is defined over machine register bank r and memory m as

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

and says that a machine will not step into a stuck state. The step relation ($\mapsto$) used in this definition is that defined by Michael and Appel [37] and is shown in Figure 4.2. It captures the decoding and execution of all program instruction. For example, the machine word corresponding to the load instruction `load rd, rs, c` would be decoded into a predicate *load* on a pair of "before" and "after" machine states.

$$\begin{aligned} \mathit{load}(rd, rs, c) \stackrel{\text{def}}{=} \; & \lambda(r, m), (r', m'). \\ & \text{readable}(r_s + c) \;\wedge\; r' = r[r_d \mapsto m(r_s + c)] \;\wedge\; m' = m \end{aligned}$$

Using this notion of machine-step semantics based safety, the main lemma that

allows us to prove the program safe is

$$\frac{\begin{array}{rr}\text{“Program } p \text{ loaded in memory } m \text{ at location } l\text{”} & (1)\\ \text{“State } (r, m) \text{ satisfies program preconditions”} & (2)\\ \text{“All program locations satisfy compiler-generated typing invariants”} & (3)\end{array}}{\text{safe}(r, m)}\ (\textsc{safe})$$

The compiler generates invariants of the form

$$\Gamma \equiv \{l_1 : \mathsf{codeptr}\,(\phi_1), l_2 : \mathsf{codeptr}\,(\phi_2), \ldots, l_n : \mathsf{codeptr}\,(\phi_n)\}$$

Condition (1) ensures that the program counter pc points to the beginning of the program and condition (2) ensures that the the initial state satisfies the preconditions for the program, which is given by $\Gamma(r_{pc})$. Condition (3) involves proving that all locations are well-typed, or alternatively, that the typing judgement $l_i : \Gamma(l_i)$ holds for each location $i \in dom(\Gamma)$. In the indexed model, this involves showing that the typing judgement $l_i :_k \Gamma(i)$, or, $l_i :_k \mathsf{codeptr}\,(\phi_i)$ holds to all approximations k. The semantic definition of $\mathsf{codeptr}$ given in Section 4.3, along with the semantics of instructions (as shown for load) then allows us to prove that any number of machine steps can be safely taken from each location i, thus proving the safety of the program.

The construction of this proof requires various components of FPCC such as LTAL to communicate the typing hints given by Γ, and the decode prover to communicate the decoding and execution semantics of machine words to the safety prover. In the remaining sections I shall give the details of how the program safety proof is constructed using TML as an interface between the prover and these components.

To illustrate the proof technique, as an example, consider the core ML program to find the length of a list shown below :

```
datatype myList = Cons of myList | Nil
fun length (Nil) = 0
  | length (Cons rest) = 1 + length (rest)
```

A simple translation of this program into a low-level program is

```
length (l) =
              len = 0;
       loop : if (l is a cons cell)  goto cons;
              return len;
       cons : len = len + 1;
              l = tail(l);
              goto loop;
```

After passing this program through the LTAL compiler, we get the LTAL program[2] (P) that is listed in Figure 7.4 and explained using Figure 7.2 and Figure 7.3. The corresponding machine code (C) generated by the LTAL compiler is shown in Figure 7.3 (in column 2) with annotations.

The LTAL program can be divided into three parts : the first part consists of LTAL instructions which operate on LTAL variables. Each instruction either corresponds to one machine-level instruction, or to a coercion (which requires no machine-level instructions). As shown below, the LTAL instructions follow the

[2]The LTAL compiler actually generates programs corresponding to the continuation-passing-style and closure-converted form, which complicates the function entry and exit code. For the ease of presentation, however, I shall use an LTAL program without these complications.

control flow shown in the pseudocode earlier:

Labels	LTAL instructions	Corresponding pseudocode
	$\mathbf{var(2)} = \mathbf{C(NAT2INT, 0)}$	`len = 0`
2\$:	$\mathbf{var(3)} = \mathbf{C(UNFOLD,\ var(1))}$	
	$\mathbf{var(4)} = \mathbf{TESTBOX(var3)}$	
	IFBOXED $\{\mathbf{var(4)}\}$	`if l is cons cell`
	THEN CALL$(\mathbf{label(4)})$	`then goto 4`
	ELSE CALL$(\mathbf{label(3)})$	`else goto 3`
3\$:	**CALL** $(\mathbf{var(99)})$	`return len`
4\$:	$\mathbf{var(5)} = \mathbf{var(4)}$	`len = len + 1`
	$\mathbf{var(7)} = \mathbf{var(2)} + \mathbf{C(NAT2INT, 1)}$	`len = len + 1`
	$\mathbf{var(8)} = \mathbf{var(5)[0]}$	`l = tail(l)`
	$\mathbf{var(2)} = \mathbf{var(7)}$	
	$\mathbf{var(1)} = \mathbf{var(8)}$	
	CALL $(\mathbf{label(2)})$	`goto 2`

In Figure 7.4, the LTAL instructions are shown enclosed in '$\langle\ \rangle$' brackets. The second part consisting of lines starting with '#' give a description of the environment typing at all the branch targets[3]. Environment typings are given to LTAL variables. The mapping of LTAL variables to machine registers is given by the "Register Map". For the pseudocode, we have the type environments as shown in Figure 7.2, which can be seen to roughly correspond to the LTAL program output.

[3]In this example, I have given environment descriptions for every instruction. From the environment typings at branch targets, the program-safety prover can automatically generate environment typings at all other program locations.

$\langle \Gamma(0) = \{\texttt{l} : \text{T\_FIX}(\text{tvar}(1), \text{T\_SUM}(1, \text{T\_FIELD}(0, \text{T\_V tvar}(1)))),$
$\quad$ return env : $\text{T\_CODE}([], [\text{var}(2) : \text{T\_INT}])\}\rangle$
$\quad$ `len` $= 0$;
$\langle \Gamma(1) = \Gamma(0) \cap \{\texttt{len} : \text{T\_INT}\}\rangle$
`loop` : `if (l is a cons cell ) goto cons`;
$\langle \Gamma(3) = \{\texttt{l} : \text{T\_NAT}(=, 0), \texttt{len} : \text{T\_INT}\}$
$\quad$ `return len`
$\langle \Gamma(4) = \{\texttt{l} : \text{T\_FIELD}(0, \text{T\_FIX}(\text{tvar}(1), \text{T\_SUM}(1, \text{T\_FIELD}(0, \text{T\_V tvar}(1))))),$
$\quad$ `len` : T_INT, return env : $\text{T\_CODE}([], [\text{var}(2) : \text{T\_INT}])\}\rangle$
`cons` : `len` $=$ `len` $+ 1$;
$\langle \Gamma(5) = \{\texttt{l} : \text{T\_FIELD}(0, \text{T\_FIX}(\text{tvar}(1), \text{T\_SUM}(1, \text{T\_FIELD}(0, \text{T\_V tvar}(1))))),$
$\quad$ `len` : T_INT, return env : $\text{T\_CODE}([], [\text{var}(2) : \text{T\_INT}])\}\rangle$
$\quad$ `l` $=$ `tail(l)`;
$\langle \Gamma(6) = \{\texttt{l} : \text{T\_FIX}(\text{tvar}(1), \text{T\_SUM}(1, \text{T\_FIELD}(0, \text{T\_V tvar}(1)))),$
$\quad$ `len` : T_INT, return env : $\text{T\_CODE}([], [\text{var}(2) : \text{T\_INT}])\}\rangle$
$\quad$ `gotoloop`;

Figure 7.2: List-length program annotated with LTAL type environments

Address	Machine word	Sparc Assembly	Annotation
0	0x94102000	`start :` `mov 0, %o2`	clear counter
4	0x80a26100	`loop :` `cmp %o1, 256`	if cons cell
8	0x36800004	`bge cons`	goto `cons`
12	0x01000000	`nop`	
16	0x81c3c000	`jmp %o7`	else done
20	0x01000000	`nop`	
24	0x9402a001	`cons :` `add %o2, 1, %o2`	increment counter
28	0xd2024000	`ld [%o1], %o1`	get next list element
32	0x10bffff9	`ba loop`	repeat `loop`
36	0x01000000	`nop`	

Figure 7.3: Sparc translation of the list length program

```
1   # Type Environment : {
2   #       var(1)  : T_FIX(tvar(1), T_SUM(1, T_FIELD(0, T_V tvar(1)))),
3   #       var(10) : T_CODE[ ](var(2) : T_INT)}
4   # Register Map :{1 : %o1, 10 : %o7}
5   1$ :  mov 0, %o2          ⟨var(2) = C(NAT2INT, 0)⟩
6   # Type Environment : {
7   #       { var(1) :T_FIX(tvar(1), T_SUM(1, T_FIELD(0, T_V tvar(1))))
8   #         var(2) :T_INT
9   #         var(10) :T_CODE[ ](var(2) : T_INT)}
10  # Register Map :{1 : %o1, 2 : %o2, 10 : %o7}
11  2$ :                      ⟨var(3) = C(UNFOLD, var(1))⟩
12  #Register Map :{1 : %o1, 2 : %o2, 3 : %o1, 10 : %o7}
13        cmp %o1, 256        ⟨var(4) = TESTBOX(var3)⟩
14  # Register Map :{1 : %o1, 2 : %o2, 3 : %o1, 4 : %o1, 10 : %o7}
15        bge 4$              ⟨IFBOXED {var(4)} THEN
16                            var(5) ⇒ CALL(label(4), [])
17                            ELSE var(6) ⇒ CALL(label(3), [])⟩
18        nop
19  # Type Environment : {
20  #       { var(2) :T_INT
21  #         var(10) :T_CODE[ ](var(2) : T_INT)}
22  # Register Map :{2 : %o2, 10 : %o7}
23  3$ :  jmp %o7             ⟨{CALL var(99)[]()}⟩
24        nop
25  # Type Environment : {
26  #       { var(2) :T_INT
27  #         var(5) :T_FIELD(0, T_FIX(tvar(1), T_SUM(1, T_FIELD(0, T_V tvar(1)))))
28  #         var(10) :T_CODE[ ](var(2) : T_INT)}
29  # Register Map :{2 : %o2, 5 : %o1, 10 : %o7}
30  4$ :  add %o2, 1, %o2     ⟨var(7) = var(2) + C(NAT2INT, 1)⟩
31        ld [%o1], %o1       ⟨var(8) = var(5)[0]⟩
32                            ⟨var(2) = var(7)⟩
33                            ⟨var(1) = var(8)⟩
34        ba 2$               ⟨CALL (label(2), [])⟩
35        nop
```

Figure 7.4: Entire LTAL program for the list length program

Finally, the third part consists of the actual Sparc assembly instructions that the core ML program compiles into. It is this part that must be proved safe without having to trust the annotations given by the other two parts.

Given such a tuple, $\mathsf{L} = (\mathsf{P}, \mathsf{C})$, our primary aim is to show that C is safe. The safety of C is determined in the following way: since typability implies safety for FPCC, we first ensure that the LTAL program type checks with respect to the specified environment typing judgements. Then, using semantics for LTAL instructions, we show that each untyped machine-level instruction in the program C respects the typing judgments for a corresponding typed LTAL program instruction in P. The next section describes the proof technique we use to generate the foundational safety proof for C.

7.2 Program safety proofs

I shall now present the syntactic judgements necessary to prove the safety of a program C obtained from the output of an LTAL compiler. (This was joint work with Gang Tan and Dinghao Wu.) The LTAL compiler[17] also outputs the LTAL program that provides, among other pieces of information, a label environment (which is denoted by Γ) that gives the typing invariants for all branch targets. Program safety is syntactically expressed through the following wellformedness judgements on the tuple C and Γ.

- **Program** : The judgement $\vdash_{\mathrm{p}} (\mathsf{C}, \Gamma)$ means that the program C is wellformed with respect to label environment Γ.

- **Block** : The judgment $\Gamma; l \vdash_{\mathrm{b}} (\mathsf{C}, \Gamma')$ means that machine code C, that starts at a basic block beginning at address l and goes to the end of the remaining code, is well formed. The block C may transfer control to any label within Γ. The environment Γ' is a subenvironment of Γ, and its significance will be made clear in the semantics for the block rules given in Section 7.3.2.

- **Decode** : The judgement $\mathsf{C} \vdash_{\mathrm{d}} \mathsf{I}$ means that the code sequence C decodes to a list of TML instructions I. This judgement uses the machine instruction semantics specification that is a part of the TCB.

- **Instruction** : The judgement $\Gamma; l \vdash_{\mathrm{i}} \{\phi\}\,\mathsf{I}\,\{\phi'\}$ means that assembly code instruction sequence I, starting at address l, is wellformed with precondition ϕ and postcondition ϕ'. The purpose of location l is to be able to compute the destination address for a pc-relative jump instructions.

To check that an annotated program (C with Γ) is wellformed ($\vdash_{\mathrm{p}} (\mathsf{C}, \Gamma)$), the PROG rule (in Figure 7.5) will call ($\Gamma; 0 \vdash_{\mathrm{b}} (\mathsf{C}, \Gamma)$), which would call rules BLOCK-1 and BLOCK-2 to check that each basic block in C is well-formed under the assumption that the first block in C starts at address 0. For the list-length program example in Figure 7.3, the domain of Γ is $\{0, 4, 24\}$, C has three pieces:

$$
\begin{aligned}
&\mathit{start}: && \texttt{0x94102000}\\
&\mathit{loop}: && \texttt{0x80a26100}; \ldots; \texttt{0x81c3c000}; \texttt{0x01000000}\\
&\mathit{cons}: && \texttt{0x94021001}; \ldots; \texttt{0x10bffff9}; \texttt{0x01000000}
\end{aligned}
$$

To check the wellformedness of each piece of C, we first decode the machine words into an assembly language instruction sequence given by I. This is done using block wellformedness ($\vdash_{\mathrm{b}}$) rules, DEC-1 and DEC-2. Then, each instruction

$$\frac{\Gamma; 0 \vdash_{\mathrm{b}} (\mathsf{C}, \Gamma)}{\vdash_{\mathrm{p}} (\mathsf{C}, \Gamma)}\ \textsc{prog}$$

$$\frac{\begin{array}{c}\Gamma(l) = \mathsf{codeptr}\,(\phi_1) \\ \Gamma(l + |\mathsf{C}_1|) = \mathsf{codeptr}\,(\phi_2) \\ \mathsf{C}_1 \vdash_{\mathrm{d}} \mathsf{I}_1 \quad \Gamma; l \vdash_{\mathrm{i}} \{\phi_1\}\,\mathsf{I}_1\,\{\phi_2\} \\ \Gamma; l + |\mathsf{C}_1| \vdash_{\mathrm{b}} (\mathsf{C}_2, \Gamma')\end{array}}{\Gamma; l \vdash_{\mathrm{b}} (\mathsf{C}_1; \mathsf{C}_2, \{l : \mathsf{codeptr}\,(\phi_1)\} \cap \Gamma')}\ \textsc{block\_1}$$

$$\frac{\begin{array}{c}\mathsf{C} \vdash_{\mathrm{d}} \mathsf{I} \\ \Gamma \subseteq \{l : \mathsf{codeptr}\,(\phi)\} \quad \Gamma; l \vdash_{\mathrm{i}} \{\phi\}\,\mathsf{I}\,\{\bot\}\end{array}}{\Gamma; l \vdash_{\mathrm{b}} (\mathsf{C}, \{l : \mathsf{codeptr}\,(\phi)\})}\ \textsc{block\_2}$$

$$\frac{\mathsf{C}_1 \vdash_{\mathrm{d}} \mathsf{I}_1 \quad \mathsf{C}_2 \vdash_{\mathrm{d}} \mathsf{I}_2}{\mathsf{C}_1; \mathsf{C}_2 \vdash_{\mathrm{d}} \mathsf{I}_1; \mathsf{I}_2}\ \textsc{dec\_1} \qquad \frac{\mathrm{decode}(n, i)}{n \vdash_{\mathrm{d}} i}\ \textsc{dec\_2}$$

$$\frac{\Gamma; l \vdash_{\mathrm{i}} \{\phi_1\}\,\mathsf{I}_1\,\{\phi_2\} \quad \Gamma; l + |\mathsf{I}_1| \vdash_{\mathrm{i}} \{\phi_2\}\,\mathsf{I}_2\,\{\phi_3\}}{\Gamma; l \vdash_{\mathrm{i}} \{\phi_1\}\,\mathsf{I}_1; \mathsf{I}_2\,\{\phi_3\}}\ \textsc{comp}$$

$$\frac{\phi_1' \subseteq \phi_1 \quad \phi_2 \subseteq \phi_2' \quad \Gamma; \mathsf{I} \vdash_{\mathrm{i}} \{\phi_1\}\,\mathsf{I}\,\{\phi_2\}}{\Gamma; l \vdash_{\mathrm{i}} \{\phi_1'\}\,\mathsf{I}\,\{\phi_2'\}}\ \textsc{weaken}$$

$$\frac{\phi \subseteq \phi_1 \quad \phi_2 \subseteq \phi'}{\Gamma; l \vdash_{\mathrm{i}} \{\phi\}\,\mathsf{instr}(\Gamma, \phi_1, \phi_2)\,\{\phi'\}}\ \textsc{ins}$$

Figure 7.5: Syntax : Type checking rules

LTAL instructions	Corresponding TML operations
$\mathbf{var(2)} = \mathbf{C(NAT2INT, 0)}$	`tml-mov`
$\mathbf{var(3)} = \mathbf{C(UNFOLD,\ var(1))}$	$\subseteq -$UNFOLD
$\mathbf{var(4)} = \mathbf{TESTBOX(var3)}$	`tml-cmp`
$\mathbf{IFBOXED}\ \{\mathbf{var(4)}\}\ldots$	`tml-bge`
$\mathbf{CALL\ var(99)}()$	`tml-jump`
$\mathbf{var(7) = var(2) + C(NAT2INT, 1)}$	`tml-add`
$\mathbf{var(8) = var(5)[0]}$	`tml-load`
$\mathbf{CALL\ label(2)}()$	`tml-jump`

Figure 7.6: TML instruction models for list-length LTAL program instructions

sequence must be checked to be wellformed (or type safe), and this is done using the judgements for instruction sequences ($\vdash_{\mathrm{i}}$), where the sequence is broken down into individual instructions using the rule COMP. The rule INS is then used for individual instructions to determine their wellformedness. For our example program, we have the LTAL instructions and their TML instr models given in Figure 7.6.

Each application of the INS rule ensures that the pre- and postconditions for TML instructions are compatible with the typing judgements that hold at the locations where the corresponding machine instructions are located. LTAL coercion instructions such as UNFOLD help guide the prover to generate the correct typing judgements, and require a subtyping rule (e.g. $\subseteq$-UNFOLD) rather than a TML instruction. As an example, we begin the list-length program (Figure 7.4) with the environment

$$\{\%o1 : \mathsf{rec}\,(\mathsf{const}(0) \cup \mathsf{field}(0, \underline{0})), \%o7 : \mathsf{codeptr}\,(\{2 : \mathsf{int}\})\}$$

where the Sparc *output* registers 1 and 7 are denoted using the Sparc assembly-language convention as $\%o1$ and $\%o7$. The register map given by the LTAL program is used to map the LTAL variables into actual Sparc registers, but I shall not illustrate this intermediate step for this example. Using the `tml-mov` instruction,

we get the postcondition to be

$$\{\%o1 : \mathsf{rec}\,(\mathsf{const}(0) \cup \mathsf{field}(0, \underline{0})), \%o7 : \mathsf{codeptr}\,(\{2 : \mathsf{int}\}), \%o2 : \mathsf{int}\}$$

The type environment at label \$2 in Figure 7.4 is a subtype of the type environment shown above, and therefore it is safe to execute the `tml-mov` instruction. Next, the unfold coercion results in using the $\subseteq$-UNFOLD subtyping rule to obtain the type environment

$$\begin{aligned}&\{\%o1 : \mathsf{const}(0) \cup \mathsf{field}(0, \mathsf{rec}\,(\mathsf{const}(0) \cup \mathsf{field}(0, \underline{0}))),\\ &\ \%o7 : \mathsf{codeptr}\,(\{2 : \mathsf{int}\}), \%o2 : \mathsf{int}\}\end{aligned}$$

The comparison instruction (Figure 7.4 line 13) is modelled by the `tml-cmp` TML instruction, and results in a zero flag being tied to the comparison of `%o1` and the value[4] 256. The postcondition of the comparison instruction is

$$\begin{aligned}&\{\%o1 : \mathsf{const}(0) \cup \mathsf{field}(0, \mathsf{rec}\,(\mathsf{const}(0) \cup \mathsf{field}(0, \underline{0}))),\\ &\ \%o7 : \mathsf{codeptr}\,(\{\%o2 : \mathsf{int}\}), \%o2 : \mathsf{int}\} \cap\\ &\ (\mathsf{relate}_{>}(\mathsf{id}, \mathsf{id})(\%o1, \mathsf{const}(256)) \cap \{g : \mathsf{int}_{=}(\mathsf{const}(1))\}\\ &\qquad \cup\ \mathsf{relate}_{\leq}(\mathsf{id}, \mathsf{id})(\%o1, \mathsf{const}(256)) \cap \{g : \mathsf{int}_{=}(\mathsf{const}(0))\})\end{aligned}$$

The comparison instruction is followed by a `bge` branch-if-greatereq instruction. To ensure the safety of the `bge` instruction, we must ensure that the precondition at the branch target is a subtype of the precondition at the `bge` instruction. The branch target precondition as shown at line 25 in Figure 7.4 is translated into a TML type environment as

$$\begin{aligned}&\{\%o1 : \mathsf{rec}\,(\mathsf{const}(0) \cup \mathsf{field}(0, \underline{0})),\\ &\ \%o7 : \mathsf{codeptr}\,(\{\%o2 : \mathsf{int}\}), \%o2 : \mathsf{int}\}\end{aligned}$$

[4]Datatypes that have constant constructors (such as the `Nil` variant for lists) do not need to be boxed. ML implementations allow multiple constant constructors by representing them as unboxed constant numbers in the range $(0 \ldots 255)$. The comparison with 256 allows the differentiation between unboxed and boxed constructors.

and we can show that the precondition for the `bge` instruction is a subtype of the precondition for the branch target instruction. If the branch is not taken, then we execute a jump to the contents of register $\%o7$. The type of this register (and hence the precondition for a safe jump) is $\mathsf{codeptr}\,(\{\%o2 : \mathsf{int}\})$. The current `bge` precondition enforces this condition, and therefore the branch-if-greatereq instruction is safe to execute in either case. If the branch were taken (if there are more elements in the list), we have the branch target precondition

$$\{\%o1 : \mathsf{field}(0, \mathsf{rec}\,(\mathsf{const}(0) \cup \mathsf{field}(0, \underline{0}))),$$
$$\%o7 : \mathsf{codeptr}\,(\{\%o2 : \mathsf{int}\}), \%o2 : \mathsf{int}\}$$

We use the TML instruction `tml-add` to argue about the safety of the add instruction to increment the list-length variable in register `%o2` (line 30 Figure 7.4). This addition is safe since we have $\{\%o2 : \mathsf{int}\}$ as a part of the type environment, and this satisfies the precondition for the add instruction. The precondition for the subsequent instruction to load the next list element (line 31 in Figure 7.4) is also satisfied, since register `%o1` is of a field type. After the load instruction, as modelled by the TML instruction `tml-load`, we are left with the postcondition

$$\{\%o1 : \mathsf{rec}\,(\mathsf{const}(0) \cup \mathsf{field}(0, \underline{0})), \%o7 : \mathsf{codeptr}\,(\{\%o2 : \mathsf{int}\}), \%o2 : \mathsf{int}\}$$

which satisfies the precondition for the target (label \$2 at line 11 in Figure 7.4) of the subsequent unconditional jump instruction. Therefore this program is well formed.

The next section gives models for all the syntactic rules which allow us to prove that a wellformed machine program obeys the safety policy, and is therefore safe to execute.

7.3 Semantics for judgements of safety proofs

To make these syntactic proofs foundational, I shall present a semantic model for each of these judgements and show the role of TML instructions in the construction of proofs. From these models, each of the typing or wellformedness rules given in Figure 7.5 can be proved as a derived lemma. To correctly model the program wellformedness judgement $\vdash_{\mathrm{p}}$, we first model the machine program words in C as a TML environment. This is done as follows : If a program code word n is loaded in memory at location l, we encode this using the singleton environment as $\{l : \mathsf{num}\,(n)\}$, where $\mathsf{num}\,(n) \stackrel{\mathrm{def}}{=} \mathsf{box}\,(\mathsf{int}_{=}(n))$. The entire program can thus be described as

$$\Delta(n_0; n_1; \ldots) \stackrel{\mathrm{def}}{=} \{0 : \mathsf{num}\,(n_0),\ 4 : \mathsf{num}\,(n_1), \ldots\}$$

where the constructor Δ converts a sequence of machine code words into a type environment. For the list-length program, we have

$$\Delta(\mathsf{C}) \quad = \quad \{0 : \mathsf{num}\,(\mathtt{0x94102000}), 4 : \mathsf{num}\,(\mathtt{0x80a26100}), \ldots, 36 : \mathsf{num}\,(\mathtt{0x01000000})\}\}$$

7.3.1 Program wellformedness

The model for the program wellformedness judgement $\vdash_{\mathrm{p}}$ is given as

$$\vdash_{\mathrm{p}}\ (\mathsf{C}, \Gamma) \stackrel{\mathrm{def}}{=} \text{safe-code}(\mathsf{C})$$

where we have

$$\begin{aligned}
\mathsf{C} \sqsubseteq m &\stackrel{\text{def}}{=} \forall x \in \text{dom}(\mathsf{C}).\ x \in \text{dom}(m) \wedge \mathsf{C}(x) = m(x) \\
\text{safe-code}(\mathsf{C}) &\stackrel{\text{def}}{=} \forall \rho, k, s. \\
&\quad (\mathsf{C} \sqsubseteq \text{mem}(s) \wedge\ \text{reg-vector}(s)(\text{pc}) = 0 \wedge (\phi_0\ \rho\ k\ \text{reg-vector}(s))) \\
&\quad \Longrightarrow\quad \text{safe}(\text{mem}(s), \text{reg-vector}(s))
\end{aligned}$$

using the definition of safe given in Section 4.2. This definition says that a machine code program C is safe in any state s having memory m and registers r if *i*) m extends C, *ii*) the program counter, $r(\text{pc})$ initially points to address 0, and *iii*) when the program begins executing, some precondition ϕ_0 holds.[5] The semantics of $\vdash_{\text{p}}\ (\mathsf{C}, \Gamma)$ is safe-code(C) so that "typability implies safety" follows directly from this definition. Notice that Γ is not used in this definition. This highlights the fact the the LTAL program P need not be trusted; any Γ given by the compiler is acceptable since it is only a *hint* to prove that C is safe.

7.3.2 Block wellformedness

The judgment $\Gamma; l\ \vdash_{\text{b}}\ (\mathsf{C}, \Gamma')$ makes a connection between the program and the typing invariants provided by the LTAL program. We must ensure that C respects the invariants Γ. Using Δ to show that $\Delta(\mathsf{C}) \subseteq \Gamma$ directly is hard. The difficulty

[5] We want our semantic model of types to be entirely in the *proof* of the safety theorem, not in the *statement* of the theorem; this way, the choice of type system is up to the compiler and prover, and is not constrained by the checker. But this makes the definition of the initial precondition ϕ_0 tricky : we must not use types. In our actual implementation, the machine-code program is called with simple integer arguments, which can be specified easily without using our semantic model. The example program that computes the length of a linked list is not quite realistic (as a complete program) because it takes a linked-list argument; this program is more representative of what might occur as a subroutine of some larger chunk of proof-carrying code. In summary, we assume that the initial precondition ϕ_0 is simple enough that it can be described directly in our underlying logic, and is also easy to describe using our semantic type operators.

is explained informally thus: On one hand, to judge that a jump instruction is safe we need to assume that the destination is safe to jump as long as the precondition is met, i.e., we need to *assume* Γ correctly specifies the label preconditions. On the other hand, the point of $\Delta(\mathsf{C}) \subseteq \Gamma$ is to *prove* that Γ has the correct label invariants.

Traditional domain-theoretic models[21] prove that a (recursive) program C matches invariants Γ by induction over approximations of C. In the indexed model, this proof instead uses induction over the approximation indices of Γ. Since types in our model are sequences of k-approximations, we prove $\Delta\mathsf{C} \subseteq \Gamma$ by induction over k.

If done naïvely, this approach would be inconvenient because we would be continually manipulating explicit indices $(k, k+1)$ throughout a very large proof. Instead, we make the proof manageable by abstracting away from the indices: we define the operator $\vdash_0 \Gamma$ which says that Γ is true at index zero, and use the *subtype-plus* predicate $\Gamma_1 \Subset \Gamma_2$, which says that if Γ_1 holds to approximation k, then Γ_2 should hold to one degree more accurately.

$$\begin{aligned} \vdash_0 \Gamma &\stackrel{\text{def}}{=} \forall \rho\ k\ v.\ \Gamma\ \rho\ 0\ v \\ \Gamma_1 \Subset \Gamma_2 &\stackrel{\text{def}}{=} \forall \rho\ k\ v.\ \Gamma_1\ \rho\ k\ v \implies \Gamma_2\ \rho\ (k+1)\ v \end{aligned}$$

Our obligation for any given program is to prove $\vdash_0 \Gamma$ and to prove $\Delta(\mathsf{C}) \cap \Gamma \Subset \Gamma$; then we can combine these two with the following induction theorem:

Theorem 2 *(Subtype Induction)*

$$\frac{\vdash_0 \Gamma \quad \Delta(\mathsf{C}) \cap \Gamma \Subset \Gamma}{\Delta(\mathsf{C}) \subseteq \Gamma}$$

Proof. The base case is given by $\vdash_0 \Gamma$. For the inductive case, we assume

$\Delta(\mathsf{C})\ \rho\ k\ v \implies \Gamma\ \rho\ k\ v$ and show that $\Delta(\mathsf{C})\ \rho\ (k+1)\ v \implies \Gamma\ \rho\ (k+1)\ v$. It can be easily verified that $\Delta(\mathsf{C})$ is a valid environment and thus closed under decreasing index. Hence, given $\Delta(\mathsf{C})\ \rho\ (k+1)\ v$, we also have $\Delta(\mathsf{C})\ \rho\ k\ v$. From the induction hypothesis, we get $\Gamma\ \rho\ k\ v$.

Therefore, we have $\Delta(\mathsf{C}) \cap \Gamma\ \rho\ k\ v$, and by the premise $\Delta(\mathsf{C}) \cap \Gamma \Subset \Gamma$, we have $\Gamma\ \rho\ (k+1)\ v$. □

Proving the first premise $\vdash_0 \Gamma$ for the subtype induction rule is easy: Γ is the intersection of code pointer types, which can be verified to be always true under index zero (informally, after jumping to any address it's always safe to execute zero steps).

To prove $\Delta(\mathsf{C}) \cap \Gamma \Subset \Gamma$, we need to show that $\forall \rho, k, v.$ the judgement $\Gamma\ \rho\ k\ v$ holds under the assumption that $(\Delta(\mathsf{C}) \cap \Gamma)\ \rho\ k\ v$ holds. In other words, we need to prove all the branch targets are of code pointers to approximation $k+1$ under the assumption that they are of code pointers to approximation k.

In the block wellformedness judgement $\Gamma; l \vdash_{\mathrm{b}} (\mathsf{C}', \Gamma')$, C' is a portion of the program code, from address l to the end of the program, and Γ' is the collection of preconditions of labels inside C' and is a part of global label environment Γ. If we gave $\Gamma; l \vdash_{\mathrm{b}} (\mathsf{C}', \Gamma')$ the simple model $\mathsf{offset}(l, \Delta(\mathsf{C})) \subseteq \Gamma'$, we would be in trouble if C were to contain a jump instruction to a label outside C; since this model didn't give us any assumptions about such labels. Therefore, we give it the model :

$$\Gamma; l \vdash_{\mathrm{b}} (\mathsf{C}, \Gamma') \quad \dashv\vDash \quad (\mathsf{offset}(l, \Delta(\mathsf{C})) \cap \Gamma \Subset \Gamma') \ \wedge (\vdash_0 \Gamma')$$

This model would guarantee that every label in C is safe for at least k steps even in the presence of jumps.

With these semantic models, we can prove the PROG rule in Figure 7.5 as a derived lemma.

Theorem 3 *(Safe Code)*

$$\frac{\Gamma; 0 \vdash_{\mathrm{b}} (\mathsf{C}, \Gamma)}{\vdash_{\mathrm{p}} (\mathsf{C}, \Gamma)} \text{ PROG}$$

Proof. From the definition of $\Gamma; 0 \vdash_{\mathrm{b}} (\mathsf{C}, \Gamma)$, we have

$$\mathsf{offset}(0, \Delta(\mathsf{C})) \cap \Gamma \Subset \Gamma \quad \wedge \quad \vdash_0 \Gamma$$

Since $\mathsf{offset}(0, \Delta(\mathsf{C})) = \Delta(\mathsf{C})$, we have $\Delta(\mathsf{C}) \subseteq \Gamma$ by Theorem 2.

The goal $\vdash_{\mathrm{p}} (\mathsf{C}, \Gamma)$ is modeled by safe-code(C), from whose definition we have the following assumptions for state s and want to show that safe(reg-vector(s), mem(s)).

$$i)\ \ \mathsf{C} \sqsubseteq \mathrm{mem}(s) \qquad ii)\ \ \phi_0\ \rho\ k\ \langle s, \text{reg-vector}(s)\rangle \qquad iii)\ \ \text{reg-vector}(s)(pc) = 0$$

The deduction steps from $\Delta(\mathsf{C}) \subseteq \Gamma$ to safe(reg-vector(s), mem(s)) are summarized by the proof tree in Figure 7.7.

Step (1) says to prove s is safe, it suffices to prove that s is safe for an arbitrary k steps. Step (2) is justified by the definition of the code pointer constructor codeptr in Section 4.3. along with assumptions *ii*) and *iii*). Step (3) are by instantiation of the universal quantifier. Step (4) is by the definition of the singleton environment constructor. Finally, step (5) is by the definition of '$\subseteq$' (Section 3.4), and transitivity of subtyping (Figure 3.9). □

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Delta(\mathsf{C})\ \rho\ k\ \langle s, \mathsf{idvec}\rangle \quad \Delta(\mathsf{C}) \subseteq \Gamma \quad \Gamma \subseteq \{0 : \mathsf{codeptr}\,(\phi_0)\}}{\{0 : \mathsf{codeptr}\,(\phi_0)\}\ \rho\ k\ \langle s, \mathsf{idvec}\rangle}\ (5)}{\forall \rho, k.\mathsf{codeptr}\,(\phi_0)\ \rho\ k\ \langle s, \text{const-vector}(0)\rangle}\ (4)}{\forall \rho, k.\mathsf{codeptr}\,(\phi_0)\ \rho\ (k+1)\ \langle s, \text{const-vector}(0)\rangle}\ (3) \qquad ii \quad iii}{\forall k.\ \text{safen}(k, \text{reg-vector}(s), \text{mem}(s))}\ (2)}{\text{safe}(\text{reg-vector}(s), \text{mem}(s))}\ (1)$$

Figure 7.7: Semantics : Proof tree for $\Delta(\mathsf{C}) \subseteq \Gamma$

7.3.3 Connecting axiomatic instruction semantics to TML instructions

The decode judgement uses the decode relation due to Michael and Appel [37] to relate instruction words to relations on states in form of register bank and memory pairs. For example, the `add` instruction is given an axiomatic characterisation in the FPCC system as

$$\begin{aligned} \text{add} \ \stackrel{\text{def}}{=}\ & \lambda(i,j,k).\lambda(r,m),(r',m'). \\ & (r' = r[i \mapsto r(i) + r(j)]) \ \wedge\ (m = m') \end{aligned}$$

In Chapter 6, I described high-level TML instructions that operate on type environments rather than on raw machine state. For example, the TML instruction corresponding to the addition instruction above is:

$$\begin{aligned} \texttt{tml-add i} \leftarrow \texttt{j} + \texttt{k} \ \stackrel{\text{def}}{=}\ & \forall m, n, \Gamma, \phi.\ \mathsf{instr}(\Gamma, \phi_1, \phi_1[i \mapsto \mathsf{int}_=(\mathsf{plus}(n,m))]) \\ & \text{where } \phi_1 = \phi \cap \{j : \mathsf{int}_=(n), k : \mathsf{int}_=(m)\} \end{aligned}$$

It is much more convenient to consider TML instructions rather than the axiomatic characterisations of instructions for the generation of safety proofs. To connect a low-level instruction such as add to the TML instruction `tml-add`, we

use the constructor $\mathsf{instr@}$, which allows us to express the fact that some location l contains a machine word that decodes to the low-level instruction add, and add implements the TML instruction `tml-add`. We make this connection with rules such as:

$$\frac{\text{decode}(n, \text{add}(r_i, r_j, r_k))}{\mathsf{num}\,(n) \subseteq \mathsf{instr@}(\forall \phi, m, n'.\mathsf{instr}(\Gamma, \phi_1, \phi_1[i \mapsto \mathsf{int}_=(\mathsf{plus}(n', m))]))}\ add$$

$$\text{where}\quad \phi_1 = \phi \cap \{j : \mathsf{int}_=(n'), k : \mathsf{int}_=(m)\}$$

This rule allows us to express instructions as predicate transformers on type environments rather than on raw machine states. Some of the other important low-level axiomatic definitions of instructions[6] are listed in Figure 7.8, and rules connecting them to corresponding TML instructions using $\mathsf{instr@}$ are listed in Figure 7.9. The precise definition of $\mathsf{instr@}$ is due to Gang Tan and is not discussed in this thesis, rather, the rules below give the interface that the safety prover can use to be able to link machine words to TML instructions. The foundational proofs for the rules in Figure 7.9 are currently under construction.

Using the $\mathsf{instr@}$ type constructor, we construct a TML code environment Π defined as

$$\Pi(\iota_0, \iota_1, \ldots) \equiv \{0 : \mathsf{instr@}(\iota_0), 2 : \mathsf{instr@}(\iota_1), \ldots\}$$

This construction allows us to compare the machine code environment and the high-level TML code environment via the subtyping operator, i.e., if $\Delta(\mathsf{C})(i)$ contains a machine word, then $\Pi(\mathsf{I})(i)$ must contain a TML instruction such that $\Delta(\mathsf{C})(i) \subseteq \Pi(\mathsf{I})(i)$ holds. Therefore, the semantic model for $\mathsf{C} \vdash_{\mathrm{d}} \mathsf{I}$ is simply

[6]The instruction cmp only considers the zero condition, and cmp and beq use z to stand for the zero flag in the condition code register.

$$
\begin{array}{rcl}
\text{load} & \stackrel{\text{def}}{=} & \lambda(i,j,c).\lambda(r,m),(r',m'). \\
& & (r' = r[i \mapsto m[r(j)+c]]) \ \wedge \ (m = m') \\
\text{store} & \stackrel{\text{def}}{=} & \lambda(i,j,c).\lambda(r,m),(r',m'). \\
& & (r = r') \ \wedge \ (m' = m[(r(j)+c) \mapsto r(i)]) \\
\text{jump} & \stackrel{\text{def}}{=} & \lambda(l).\lambda(r,m),(r',m'). \\
& & (r' = r[pc \mapsto l]) \ \wedge \ (m = m') \\
\text{cmp} & \stackrel{\text{def}}{=} & \lambda(i,c).\lambda(r,m),(r',m'). \\
& & \quad ((r[i] = c \implies (r' = r[z \mapsto 1])) \vee (r[i] \neq c \implies (r' = r[z \mapsto 0]))) \\
& & \wedge \ (m = m') \\
\text{beq} & \stackrel{\text{def}}{=} & \lambda(l).\lambda(r,m),(r',m'). \\
& & \quad ((r[z] = 1 \implies (r' = r[pc \mapsto l])) \ \vee \ (r[z] = 0 \implies (r' = r))) \\
& & \wedge \ (m = m')
\end{array}
$$

Figure 7.8: Definitions of instructions in terms of machine state

$$
\frac{\text{decode}(n, \text{load}(r_i, r_j, c))}{\mathsf{num}\,(n) \subseteq \mathsf{instr@}(\forall \phi.\mathsf{instr}(\Gamma, \phi_1, \phi_1[i \mapsto \tau]))} \ \textit{load}
$$

where where $\phi_1 = \phi \cap \{j : \mathsf{field}(c, \tau)\}$

$$
\frac{\text{decode}(n, \text{store}(r_i, r_j, c))}{\mathsf{num}\,(n) \subseteq \mathsf{instr@}(\forall \phi.\mathsf{instr}(\Gamma, \phi', \phi''))} \ \textit{store}
$$

where ϕ' and ϕ'' are as described in Section 6.3

$$
\frac{\text{decode}(n, \text{jump}(l))}{\mathsf{num}\,(n) \subseteq \mathsf{instr@}(\forall m, \phi.\mathsf{instr}(\Gamma, \phi \cap \{l : \mathsf{codeptr}\,(\Gamma[m])\}, \bot))} \ \textit{jump}
$$

$$
\frac{\text{decode}(n, \text{cmp}(r_i, c))}{\mathsf{num}\,(n) \subseteq \mathsf{instr@}(\forall \Gamma, \phi.\ \mathsf{instr}(\Gamma, \phi, \phi \cap \phi_1))} \ \textit{cmp}
$$

where $\phi_1 = (\mathsf{relate}_{=}(\mathsf{id}, \mathsf{id})(i, \mathsf{const}(c)) \cap \{z : \mathsf{int}_{=}(\mathsf{const}(1))\})$
$\cup\ (\mathsf{relate}_{\neq}(\mathsf{id}, \mathsf{id})(i, \mathsf{const}(c)) \cap \{z : \mathsf{int}_{=}(\mathsf{const}(0))\})$

$$
\frac{\text{decode}(n, \text{beq}(l))}{\mathsf{num}\,(n) \subseteq \mathsf{instr@}(\forall m, \phi.\ \mathsf{instr}(\Gamma, \phi_1, \phi_2))} \ \textit{beq}
$$

where $\phi_1 = \phi \cap \{l : \mathsf{codeptr}\,(\Gamma[m])\} \cap \{z : \mathsf{int}_{=}(\mathsf{const}(1))\}$
$\phi_2 = \phi \cap \{l : \mathsf{codeptr}\,(\Gamma[m])\} \cap \{z : \mathsf{int}_{=}(\mathsf{const}(0))\}$

Figure 7.9: Rules connecting low-level instructions to TML instructions

$\Delta(\mathsf{C}) \subseteq \Pi(\mathsf{I})$.

With these semantics, we can now prove the rules BLOCK-1 and BLOCK-2 as lemmas.

Theorem 4

$$\frac{\begin{array}{c}\Gamma(l) = \mathsf{codeptr}\,(\phi_1) \\ \Gamma(l + |\mathsf{C}_1|) = \mathsf{codeptr}\,(\phi_2) \\ \mathsf{C}_1 \vdash_{\mathrm{d}} \mathsf{I}_1 \quad \Gamma; l \vdash_{\mathrm{i}} \{\phi_1\}\,\mathsf{I}_1\,\{\phi_2\} \\ \Gamma; l + |\mathsf{C}_1| \vdash_{\mathrm{b}} (\mathsf{C}_2, \Gamma')\end{array}}{\Gamma; l \vdash_{\mathrm{b}} (\mathsf{C}_1; \mathsf{C}_2, \{l : \mathsf{codeptr}\,(\phi_1)\} \cap \Gamma')}\ \text{BLOCK\_1}$$

Proof. By the semantic model $\vdash_{\mathrm{b}}$, we need to show

$$\mathsf{offset}(l, \Delta(\mathsf{C}_1; \mathsf{C}_2)) \cap \Gamma \Subset \{l : \mathsf{codeptr}\,(\phi_1)\} \cap \Gamma'$$

and

$$\vdash_0 \{l : \mathsf{codeptr}\,(\phi_1)\} \cap \Gamma'$$

First, we have (a)

$$\begin{array}{lll} & \mathsf{offset}(l, \Delta(\mathsf{C}_1; \mathsf{C}_2)) \cap \Gamma & \\ \subseteq & \mathsf{offset}(l, \Delta(\mathsf{C}_1)) \cap \Gamma & (1) \\ \subseteq & \mathsf{offset}(l, \Pi(\mathsf{I}_1)) \cap \Gamma & (2) \\ \subseteq & \mathsf{offset}(l, \Pi(\mathsf{I}_1)) \cap \Gamma \cap \{l + |\mathsf{I}_1| : \mathsf{codeptr}\,(\phi_2)\} & (3) \\ \Subset & \{l : \mathsf{codeptr}\,(\phi_1)\} & (4) \end{array}$$

Step (1) uses the rule (from the definition of Δ):

$$\frac{}{\Delta(\mathsf{C}_1;\mathsf{C}_2) \subseteq \Delta(\mathsf{C}_1) \cap \mathsf{offset}(|\mathsf{C}_1|, \Delta(\mathsf{C}_2))}$$

the weakening rule of intersection: $\phi_1 \cap \phi_2 \subseteq \phi_1$ and the rule:

$$\frac{\phi_1 \subseteq \phi_2}{\mathsf{offset}(l, \phi_1) \subseteq \mathsf{offset}(l, \phi_2)}$$

Step (2) uses the definition of $\mathsf{C}_1 \vdash_{\mathrm{d}} \mathsf{I}_1$; step (3) uses the fact that $\{l + |\mathsf{I}_1| : \mathsf{codeptr}\,(\phi_2)\}$ is a part of Γ, which is based on the premise that $\Gamma(l + |\mathsf{C}_1|) = \mathsf{codeptr}\,(\phi_2)$ and $|\mathsf{C}_1| = |\mathsf{I}_1|$; step (4) uses the definition of $\Gamma; l \vdash_{\mathrm{i}} \{\phi_1\}\,\mathsf{I}_1\,\{\phi_2\}$.

We also have (b)

$$\begin{array}{lll} & \mathsf{offset}(l, \Delta(\mathsf{C}_1;\mathsf{C}_2)) \cap \Gamma & \\ \subseteq & \mathsf{offset}(l + |\mathsf{C}_1|, \Delta(\mathsf{C}_2)) \cap \Gamma & (1) \\ \Subset & \Gamma' & (2) \end{array}$$

Step (2) uses the definition of $\Gamma; l + |\mathsf{C}_1| \vdash_{\mathrm{b}} (\mathsf{C}_2, \Gamma')$.

Combining (a) and (b) use the lemma

$$\frac{\Gamma \Subset \Gamma_1 \quad \Gamma \Subset \Gamma_2}{\Gamma \Subset \Gamma_1 \cap \Gamma_2}$$

we get

$$\mathsf{offset}(l, \Delta(\mathsf{C}_1;\mathsf{C}_2)) \cap \Gamma \Subset \{l : \mathsf{codeptr}\,(\phi_1)\} \cap \Gamma'$$

By $\Gamma; l + |\mathsf{C}_1| \vdash_{\mathrm{b}} (\mathsf{C}_2, \Gamma')$, we also get $\vdash_0 \Gamma'$. Combining it with the easily verified

lemma $\vdash_0 \{l : \mathsf{codeptr}\,(\phi_1)\}$ using the lemma

$$\frac{\vdash_0 \Gamma_1 \quad \vdash_0 \Gamma_2}{\vdash_0 \Gamma_1 \cap \Gamma_2}$$

we get $\vdash_0 \{l : \mathsf{codeptr}\,(\phi_1)\} \cap \Gamma'$. □

Theorem 5

$$\frac{\mathsf{C} \vdash_{\mathrm{d}} \mathsf{I} \qquad \Gamma(l) = \mathsf{codeptr}\,(\phi) \quad \Gamma; l \vdash_{\mathrm{i}} \{\phi\}\,\mathsf{I}\,\{\bot\}}{\Gamma; l \vdash_{\mathrm{b}} (\mathsf{C}, \{l : \mathsf{codeptr}\,(\phi)\})}\ \text{BLOCK\_2}$$

Proof. By the semantic model $\vdash_{\mathrm{b}}$, we need to show

$$\mathsf{offset}(l, \Delta(\mathsf{C})) \cap \Gamma \Subset \{l : \mathsf{codeptr}\,(\phi)\}$$

and

$$\vdash_0 \{l : \mathsf{codeptr}\,(\phi)\}$$

The second goal can be proved from the definition of type $\mathsf{codeptr}\,(\phi)$. For the first goal, we have

$$\begin{array}{lll} & \mathsf{offset}(l, \Delta(\mathsf{C})) \cap \Gamma & \\ \subseteq & \mathsf{offset}(l, \Pi(\mathsf{I})) \cap \Gamma & (1) \\ \subseteq & \mathsf{offset}(l, \Pi(\mathsf{I})) \cap \Gamma \cap \{l + |\mathsf{I}| : \mathsf{codeptr}\,(\bot)\} & (2) \\ \Subset & \{l : \mathsf{codeptr}\,(\phi)\} & (3) \end{array}$$

Step (1) uses the model for $\mathsf{C} \vdash_{\mathrm{d}} \mathsf{I}$; step (2) uses the lemma that $\{l : \mathsf{codeptr}\,(\bot)\} = \top$; step (3) uses the model for $\Gamma; l \vdash_{\mathrm{i}} \{\phi\}\,\mathsf{I}\,\{\bot\}$. □

7.3.4 Instruction wellformedness

To prove that an instruction is wellformed we must show that it respects the typing hints given by Γ. That is, given the *assumption* location l has an instruction given by $\mathsf{instr}(\Gamma, \phi_1, \phi_2)$, we should prove i) that $\Gamma(l)$ is $\mathsf{codeptr}\,(\phi')$ such that $\phi' \subseteq \phi_1$. This ensures that the environment at l satisfied all the preconditions for safely executing the instruction at location l. Furthermore, we should prove ii) that the postcondition ϕ_2 should satisfy all the requirements for the next instruction at location $l+4$. This is expressed as $\phi_2 \subseteq \phi''$. In the indexed model, we wish to assert that at location l, the type constraint ϕ_1 should hold for one index higher than the constraint ϕ_2 at location $l+4$ to account for the execution of one instruction at l. We encode all these constraints and get the following model for $\vdash_{\mathrm{i}}$:

$$\vdash_{\mathrm{i}} \stackrel{\text{def}}{=} \lambda\Gamma, l, \phi, \mathsf{I}, \phi'.$$
$$\underbrace{\mathsf{offset}(l, \Pi(\mathsf{I}))}_{assumption} \cap \Gamma \cap \underbrace{\{l + |\mathsf{I}| : \mathsf{codeptr}\,(\phi')\}}_{condition(ii)} \Subset \underbrace{\{l : \mathsf{codeptr}\,(\phi)\}}_{condition(i)}$$

With this model, the proof of rule INS becomes straightforward.

Theorem 6

$$\frac{\phi \subseteq \phi_1 \quad \phi_2 \subseteq \phi'}{\Gamma; l \vdash_{\mathrm{i}} \{\phi\}\,\mathsf{instr}(\Gamma, \phi_1, \phi_2)\,\{\phi'\}}\ \text{INS}$$

Proof. By the semantic model $\vdash_{\mathrm{i}}$, we have the assumptions

$$\mathsf{offset}(l, \Pi(\mathsf{instr}(\Gamma, \phi_1, \phi_2))) \quad (1)$$
$$\{l + 4 : \mathsf{codeptr}\,(\phi')\} \quad (2)$$

By the hypotheses in the INS rule, we also have $\phi \subseteq \phi_1$ and $\phi_2 \subseteq \phi'$. Using the definition of instruction subtyping, we have

$$\mathsf{offset}(l, \Pi(\mathsf{instr}(\Gamma, \phi, \phi')))$$

Therefore, it is safe to execute one step at location l so that the resultant state satisfies the type environment ϕ'. By (2), it is safe to execute k instructions from the location $l + 4$. Therefore, $k + 1$ steps may be safely taken from location l, thus completing the proof. Note that this proof remains the same in case the instruction at l is a jump, since by assumption, the environment Γ is also satisfied to approximation k. $\square$

Chapter 8

Conclusion

The FPCC project aims to provide a PCC framework with the minimal-size TCB. Implementing an FPCC framework requires a compiler that generates machine code along with hints to prove safety of that code. In our framework, these hints are provided through a typed assembly language (LTAL) program. The type annotations in the LTAL program are used as hints by the safety prover and it uses the typing rules for LTAL to prove the safety of the machine code. To get a foundational safety proof, each of these rules must be given a semantic basis. In this thesis, I have shown how to provide semantic bases for some of the aspects of the FPCC Infrastructure project through the Typed Machine Language. More importantly, TML has provided us with an interface between the certifying compiler and the safety prover. A client of TML such as LTAL is able to use TML as a syntactic calculus without having to deal with the complex underlying semantic model. TML extends the semantic models due to Appel and Felty [10] and Appel and McAllester [11], and provides semantics for types, type environments, and typed instructions. I have also described a kinding system that gives semantic models for a first-order kinding

system required for the types used by TML. Finally, I have described a semantic program-safety-proof technique used in the FPCC framework, which allows TML to be used as an interface between the program safety prover and other parts of the FPCC framework such as the LTAL compiler. There is more work to be done in this area, and I shall outline some of the interesting directions for future work.

8.1 Models for more language features

The type constructors described in earlier chapters are sufficient to encode core ML. However, the encoding of our semantic models in Twelf is not sufficiently powerful to be able to encode features such as classes, which is essential for compiling modern object-oriented languages such as Java. There has been some recent work related to encoding of classes in low-level types by League et al. [33] that uses higher-order and row kinds [53] to be able to encode inheritance. Incorporating the ML module system into the FPCC semantic framework would also require us to have higher-order kinds. Adding semantic models for more expressive kinds therefore seems to be essential to be able to compile languages that support such features.

As seen in Chapter 4, LTAL also has some type constructors that are given semantic models directly in terms of higher-order logic predicates rather than through a composition of TML type constructors. Having TML constructors for these types would allow us to make TML an even more well-defined interface between LTAL and the program safety prover.

8.2 Flexibility of models and proof techniques

Our current definition of safety is type safety. There are many other notions of safety such as bounded resource usage. Our current definitions for types and safety proof techniques are closely related to the notion of safety. Changing the definitions to cater to different safety requirements seems to be difficult. I would like to explore how semantic frameworks such as FPCC could be parameterised over different notions of safety, which would allow easy extensions and combinations of different safety policies.

8.3 Scaling implementation to a realistic system

The current implementation is still under progress, and there are many details that remain to be implemented, especially the proofs discussed in Chapters 6 and 7. The rules and definitions that are discussed in this thesis are targetted for a FPCC system for core ML source programs. For FPCC to be truly useful, the TML type and instruction definitions and rules would have to be scaled up to account for the constructs used in other languages as well as the complex optimisations that are performed by modern compilers. This would allow us to discover shortcomings in the current system and would also present us with new interesting research questions.

Appendix A

Twelf encoding of higher-order logic and related lemmas

The encoding of higher-order logic that is a part of the FPCC trusted computing base is listed below. The listing below uses an encoding with implicit parameters. This requires trusting the Twelf type reconstruction and unification algorithm which increases the size of the TCB. In order to reduce this size, the actual implementation has a trusted layer of all definitions with explicit parameters. On top of this layer, the FPCC system has an untrusted layer of implicit definitions that look similar to the ones listed below. This layering is described in detail by Appel et al. [12]. For simplicity of presentation, the explicit encoding is not listed.

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

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

beta_e   : {P : tm T2 -> tm form}
           pf (P (lam (F : tm T1 -> tm T2) @ X)) ->  pf (P (F X)).
beta_i   : {P : tm T2 -> tm form}
           pf (P (F X)) -> pf (P (lam (F : tm T1 -> tm T2) @ X)).
imp_i    : (pf A -> pf B) -> pf (A imp B).
imp_e    : pf (A imp B) -> pf A -> pf B.
forall_i : ({X:tm T}pf (A X)) -> pf (forall A).
forall_e : pf(forall A) -> {X : tm T} pf (A X).

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

or       : tm form -> tm form -> tm form =
           [A : tm form][B : tm form]
           forall [c : tm form] (A imp c) imp (B imp c) imp c.
%infix right 11 or.

exists   : (tm T -> tm form) -> tm form =
           [F] forall [b] (forall [x] F x imp b) imp b.

eq       : tm T -> tm T -> tm form =
           [A : tm T][B : tm T] forall [p] p @ B imp p @ A.
false    : tm form =
           forall [a : tm form] a.
not      : tm form -> tm form =
           [A : tm form] A imp false.
true     : tm form =
           not false.
```

```
and_i    : pf A -> pf B -> pf (A and B) =
           [P1][P2] forall_i [C] imp_i [P3]
  imp_e (imp_e P3 P1) P2.

and_e1   : pf (A and B) -> pf A =
           [P1] imp_e (forall_e P1 A) (imp_i [P2] imp_i [P3] P2).

and_e2   : pf (A and B) -> pf B =
           [P1] imp_e (forall_e P1 B) (imp_i [P2] imp_i [P3] P3).

or_i1    : pf A -> pf (A or B) =
           [P] forall_i [C] imp_i [P1] imp_i [P2] imp_e P1 P.

or_i2    : pf B -> pf (A or B) =
           [P] forall_i [C] imp_i [P1] imp_i [P2] imp_e P2 P.

or_e     : pf (A or B) -> (pf A -> pf C) -> (pf B -> pf C) ->
           pf C =
           [P1][P2][P3] imp_e (imp_e (forall_e P1 C) (imp_i P2))
  (imp_i P3).

false_e  : pf false -> pf A =
           [P: pf (forall [A] A)] forall_e P A.

or_e1    : pf (A or B) -> pf (not B) -> pf A =
           [p1: pf (A or B)] [p2: pf (not B)]
           or_e p1 ([p3: pf A] p3) ([p4: pf B] false_e
    (imp_e p2 p4)).

or_e2    : pf (A or B) -> pf (not A) -> pf B =
           [p1: pf (A or B)] [p2: pf (not A)]
           or_e p1 ([p3: pf A] false_e (imp_e p2 p3))
   ([p4: pf B] p4).

exists_i : {X : tm T}pf ((A : tm T -> tm form) X) ->
            pf (exists A) =
           [X : tm T][P1 : pf (A X)]
           forall_i [B : tm form]
           imp_i [P2 : pf (forall [X] A X imp B)]
           imp_e (forall_e P2 X) P1.
```

```
exists_e : pf (exists A) -> ({X:tm T} pf (A X) -> pf B) ->
           pf B =
           [P1: pf (forall [B] (forall [X] A X imp B) imp B)]
           [P2: ({X:tm T} pf (A X) -> pf B)]
           imp_e (forall_e P1 B)
     (forall_i [X] imp_i [P3: pf (A X)] P2 X P3).

not_i    : (pf A -> pf false) -> pf (not A) =
           [FQ] (imp_i ([Z] (FQ Z))).

not_e    : pf (not A) -> pf A -> pf false = imp_e.

beta     : pf (eq ((lam F) @ X) (F X)) =
            forall_i [G]
            imp_i [P1: pf (G @ (F X))]
            beta_i ([C] G @ C) P1.

congr    : {H: tm T -> tm form} pf (eq X Z) -> pf (H Z) ->
           pf (H X) =
           [H] [P1: pf (eq X Z)][P2: pf (H Z)]
           beta_e ([C] C) (imp_e (forall_e P1 (lam H))
     (beta_i ([C] C) P2)).

refl     : pf (eq X X) =
           forall_i [A] imp_i [P] P.

symm     : pf (eq X Y) -> pf (eq Y X) =
           [q] (congr ([z] (eq Y z)) q refl).

gdef_i   : pf (eq X Z) -> {Pattern: tm T -> tm form}
           pf (Pattern Z) -> pf (Pattern X) =
           [Name][Pattern] congr Pattern Name.
gdef_e   : pf (eq X Z) -> {Pattern: tm T -> tm form}
           pf (Pattern X) -> pf (Pattern Z) =
           [Name][Pattern] (congr Pattern (symm Name)).

def_i    : pf (eq X Z) -> pf Z -> pf X = [Name] gdef_i Name ([A]A).
def_e    : pf (eq X Z) -> pf X -> pf Z = [Name] gdef_e Name ([A]A).

defarg1  : pf (eq (lam B @ X) (B X)) =
           congr ([S] eq S (B X)) beta refl.
```

```
def1_i   : pf (B X) -> pf (lam B @ X) = def_i defarg1.
def1_e   : pf (lam B @ X) -> pf (B X) = def_e defarg1.


lam2     : (tm A -> tm B -> tm C) -> tm (A arrow B arrow C) =
            [f] lam [x] lam (f x).
defarg2  : pf (eq (lam2 B @ Y @ Z) (B Y Z)) =
           congr ([S] eq (S @ Z) (B Y Z)) beta
           (congr ([S] eq S (B Y Z)) beta refl).
def2_i   : pf (B X Y) -> pf (lam2 B @ X @ Y) = def_i defarg2.
def2_e   : pf (lam2 B @ X @ Y) -> pf (B X Y) = def_e defarg2.

lam3     : (tm A -> tm B -> tm C -> tm D) ->
           tm (A arrow B arrow C arrow D) =
           [f] lam [x] lam2 (f x).
defarg3  : pf (eq (lam3 B @ X @ Y @ Z) (B X Y Z)) =
           congr ([s] eq (s @ Y @ Z) (B X Y Z)) defarg1 defarg2.

def3_i   : pf (B X Y Z) -> pf (lam3 B @ X @ Y @ Z) = def_i defarg3.
def3_e   : pf (lam3 B @ X @ Y @ Z) -> pf (B X Y Z) = def_e defarg3.

lam4     : (tm A -> tm B -> tm C -> tm D -> E) ->
           tm (A arrow B arrow C arrow D arrow E) =
           [f] lam [x] lam3 (f x).
defarg4  : pf (eq (lam4 B @ W @ X @ Y @ Z) (B W X Y Z)) =
           congr ([s] eq (s @ Y @ Z) (B W X Y Z)) defarg2 defarg2.

def4_i   : pf (B W X Y Z) -> pf (lam4 B @ W @ X @ Y @ Z) =
           def_i defarg4.
def4_e   : pf (lam4 B @ W @ X @ Y @ Z) -> pf (B W X Y Z) =
           def_e defarg4.

memory   : tp = (num arrow num).
readable : tm (memory arrow num arrow form).
```

Appendix B

Type and typing rule representations in Twelf

```
% type definitions
int_type  = lam[m] lam[v] true.
bool_type = lam[m] lam[v] (eq v zero) or (eq v one).
box_type  = lam[tau]
              lam[m] lam[v] readable @ m @ v and tau @ m @ v.
pair_type = lam[tau1] lam[tau2]
              lam[m] lam[v] box_type @ tau_1 @ m @ v
                        and box_type @ tau_2 @ m @ (succ v).

% typing rules
pair_i : pf (box_type @ Tau_1 @ M @ V) ->
         pf (box_type @ Tau_2 @ M @ (succ V)) ->
```

```
          pf (pair_type @ Tau_1 @ Tau_2 @ M @ V)
      = [p1][p2] def4_i (and_i p1 p2).

box_i : pf (readable @ M @ V) ->
        pf (Tau @ M @ (M @ V)) ->
        pf (box_type @ Tau @ M @ V)
      = [p1][p2] def3_i (and_i p1 p2).
```

Bibliography

[1] The Great Internet Mersenne Prime Search. http://www.mersenne.org/prime.htm.

[2] Security Focus software vulnerabilities database. http://www.securityfocus.com/bid.

[3] Computer incident advisory capability information bulletin. http://www.ciac.org/ciac/bulletins/l-062.shtml, Mar. 2003. web page fetched Tue Mar 4 2003.

[4] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Levy. Explicit substitutions. In *Seventeenth Annual ACM Symp. on Principles of Prog. Languages*, pages 31–46. ACM Press, Jan 1990.

[5] M. Abadi, B. C. Pierce, and G. D. Plotkin. Faithful ideal models for recursive polymorphic types. *International Journal of Foundations of Computer Science*, 2(1):1–21, 1991.

[6] A. Ahmed, A. W. Appel, and R. Virga. A stratified semantics of general references embeddable in higher-order logic. Jan. 2002. Proceedings of 17th IEEE Symposium on Logic in Computer Science LICS 2002.

[7] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

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

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

[10] 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.

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

[12] A. W. Appel, N. G. Michael, A. Stump, and R. Virga. A trustworthy proof checker. In *Foundations of Computer Security - FCS 2002 Copenhagen, Denmark*, July 2002.

[13] A. W. Appel and D. C. Wang. JVM TCB: Measurements of the Trusted Computing Base of Java Virtual Machines. Technical Report TR-647-02, Princeton University, Apr. 2002.

[14] L. Bauer, J. Ligatti, and D. Walker. A calculus for composing security policies. Technical Report TR-655-02, Princeton University, Aug. 2002.

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

[16] L. Cardelli. Type systems. In A. B. Tucker, editor, *The Computer Science and Engineering Handbook*, Boca Raton, FL, 1997. CRC Press.

[17] J. Chen, A. Appel, D. Wu, 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.

[18] 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.

[19] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. *ACM SIGPLAN Notices*, 35(5):95–107, 2000.

[20] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[21] A. de Bruin. Goto statements: Semantics and deduction systems. *Acta Informatica*, (15):385–424, 1981.

[22] S. Drossopoulou and S. Eisenbach. Java is type safe — probably. *Lecture Notes in Computer Science*, 1241, 1997.

[23] S. Drossopoulou and S. Eisenbach. *Towards an Operations Semantics and Proof of Type Soundness for Java.* Springer-Verlag, Berlin Germany, 1998.

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

[25] S. Drossopoulou, T. Valkevych, and S. Eisenbach. Java type soundness revisited.

[26] D. Evans and A. Twyman. Flexible policy-directed code safety. In *1999 IEEE Symposium on Security and Privacy Oakland, California*, May 1999.

[27] D. Gollmann. *Computer Security.* Worldwide Series in Computer Science. John Wiley & Sons Ltd., 1998.

[28] N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. In *Proceedings of 17th IEEE Symposium on Logic in Computer Science LICS 2002*, jan 2002.

[29] A. Igarashi, B. Pierce, and P. Wadler. Featherweight java: A minimal core calculus for java and gj. In L. Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.

[30] Java object serialization specification—JDK 1.2. ftp://ftp.javasoft.com/docs/jdk1.2/serial-spec-JDK1.2.ps, Nov. 1998.

[31] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Lebofsky. SETI@home : Massively distributed computing for SETI. In *IEEE Computing in Science and Engineering*, volume 3, pages 78–83, Jan. 2001.

[32] L. Lamport. Proving the correctness of multiprocess programs. In *IEEE Transactions on Software Engineering SE-3 (March) 125-143*, Mar. 1977.

[33] C. League, Z. Shao, and V. Trifonov. Encoding Java classes in a typed intermediate language. Unpublished FLINT Memo, September 1998.

[34] C. League, Z. Shao, and V. Trifonov. Precision in practice: A type-preserving Java compiler. Technical Report YALEU/DCS/TR-1223, Dept. of Computer Science, Yale University, New Haven, CT, Mar. 2002.

[35] lfar Erlingsson and F. B. Schneider. Sasi enforcement of security policies: A retrospective. Sept. 1999.

[36] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymophic types. *Information and Computation*, 71(1/2):95–130, 1986.

[37] N. G. Michael and A. W. Appel. Machine instruction syntax and semantics in higher-order logic. In *17th International Conference on Automated Deduction*, June 2000.

[38] R. Milner and M. Tofte. *Commentary on Standard ML*. MIT Press, Cambridge, Massachusetts, 1991.

[39] J. C. Mitchell and R. Viswanathan. Effective models of polymorphism, subtyping and recursion. In *23rd International Colloquium on Automata, Languages, and Programming*. Springer-Verlag, 1996.

[40] 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. 1999.

[41] 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.

[42] G. Nadathur. An explicit substitution notation in a lambdaProlog implementation. http://www.cs.uchicago.edu/~gopalan, Dec. 1997.

[43] 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.

[44] G. C. Necula. *Compiling with Proofs*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Sept. 1998.

[45] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proceedings of the 1998 ACM SIGPLAN Conference on Prgramming Language Design and Implementation (PLDI)*, pages 333–344, 1998.

[46] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages*, pages 128–139, 2002.

[47] G. C. Necula and R. R. Schneck. Proof-carrying code with untrusted proof rules. In *The 2nd International Software Security Symposium*, Nov. 2002.

[48] J. Palsberg and P. O'Keefe. A type system equivalent to flow analysis. *ACM Transactions on Programming Languages and Systems*, 17(4):576–599, July 1995.

[49] J. Palsberg and C. Pavlopoulou. From polyvariant flow information to intersection and union types. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, New York, pages, 1998.

[50] M. Parashar, editor. *Grid Computing - GRID 2002, Third International Workshop, Baltimore, MD, USA, November 18, 2002, Proceedings*, volume 2536 of *Lecture Notes in Computer Science*. Springer, 2002.

[51] C. Paulin-Mohring. Inductive definitions in the system Coq; rules and properties. In M. Bezem and J. F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, volume 664, pages 328–345. Springer Verlag Lecture Notes in Computer Science, 1993.

[52] F. Pfenning and C. Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *The 16th International Conference on Automated Deduction*. Springer-Verlag, July 1999.

[53] D. Rémy. Syntactic theories and the algebra of record terms. Research Report 1869, Institut National de Recherche en Informatique et Automatisme, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, 1993.

[54] R. R. Schneck and G. C. Necula. A gradual approach to a more trustworthy, yet scalable, proof-carrying code. In *Conference on Automated Deduction (CADE'02) Copenhagen*, July 2002.

[55] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1), Feb. 2000.

[56] D. S. Scott and C. Strachey. Towards a mathematical semantics for computer languages. In *Proc. Symp. Computers and Automata*, pages 19–46, Brooklyn, New York, 1971. Polytechnic Press.

[57] Z. Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation*, June 1997.

[58] D. Syme. Proving java type soundness. Technical Report 427, University of Cambridge Computer Science Laboratory, June 1997.

[59] D. Tarditi. *Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1997.

[60] 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.

[61] M. VanInwegen. *The Machine-Assisted Proof of Programming Language Properties.* PhD thesis, University of Pennsylvania, 1996. MS-CIS-96-31.

[62] J. Viega and G. McGraw. *Building Secure Software.* Professional Computing Series. Addison Wesley, 2002.

[63] R. Viswanathan. *Recursion Theoretic Semantics, Fully Abstract Term Models, and Imperative Constructs.* PhD thesis, Stanford University, 1995.

[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] T. J. Wilkinson. Kaffe–a virtual machine to compile and interpret java bytecodes.