Abstract

A Syntactic Approach to Foundational Proof-Carrying Code

Nadeem Abdul Hamid

2005

Certified code technology and type systems research has reached a point where it is now possible to certify advanced safety and security properties of low-level systems code. Type systems are a common programming language feature today, allowing fast and easy verification of basic safety properties for application developers. Although verifiable bytecode and typed common intermediate languages have made significant contributions in the area of secure computing, a considerable amount of further (unverified) compilation and optimization is required before programs written in such languages can run on actual hardware. To address this issue, research in the past decade has turned to the use of type systems and logic to verify properties of low-level code. Much of this work focuses only on carrying through the compilation of high-level languages down to verifiable machine code. There has not been significant progress in combining such compilation with verification and certification of that code which is only written at the systems programming and assembly code level. Indeed, providing security guarantees for such infrastructure code has become a dire need.

This thesis aims at a framework to certify "the whole code." It presents an approach by which well-typed, high-level programs are compiled to certified machine code. In the same framework, the runtime library and operating systems components such as memory management can be certified safe at the level of assembly code. Now, the complete combination of compiled high-level code and low-level system libraries can be verified for safe operation according to a user's safety policy. To enable this development, I develop a new alternative for producing foundational proof-carrying code (FPCC), utilizing a syntactic encoding of the high-level type system along with syntactic soundness proofs.

The first part of this thesis describes a monolithic compilation scheme from a high-level type system to FPCC, utilizing the syntactic method. In the second part, I refine the framework to produce localized invariants, allowing for interoperation between different source languages. Finally, I demonstrate an application of the framework to a typed assembly language with a region-based memory management library, where the library is certified using low-level Hoare logic reasoning.

A Syntactic Approach to Foundational Proof-Carrying Code

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Nadeem Abdul Hamid

Dissertation Director: Zhong Shao

May 2005

A Syntactic Approach to Foundational Proof-Carrying Code

Contents

بسم الله الرحمن الرحيم

Acknowledgments

I have little hope of being able to adequately acknowledge all those who have helped me make this dissertation possible. Nonetheless, I start with my advisor Zhong Shao, who, for the last five years, has given me constant encouragement and more to explore and develop this area of research. Without his guidance and an occassional push, I would not have achieved these results. Towards developing my understanding of formal methods, Valery Trifonov has been a valuable resource. I have turned to him many times to unravel the intricacies of formal systems. My officemate, Dachuan Yu, has traveled much of the same journey with me over these past several years, and I benefitted mutually from many discussions with him and others in the Flint group at Yale, including Chris League, Stefan Monnier, and Zhaozhong Ni. Albeit perhaps to a lesser degree, I have found the other faculty in the Computer Science department supportive in many ways. And I am indebted to the secretarial staff for helping relieve administrative and bureaucratic burdens. I am grateful towards Arvind Krishnamurthy, Carsten Schürmann, and Peter Lee for agreeing to serve on my thesis committee.

Going beyond the immediate past, I truly appreciate all the support and encouragement received from my undergraduate professors and even secondary and primary school instructors. Without their efforts I would not have attained this level of achievement. Of special mention are Alice Fischer, Barun Chandra, and, of course, Jim Sweet. Many others contributed no less but were I to list them all, this would become a very long section.

Much is due to my family and community for their constant prayers and, well, for just being there. And, finally, no words can express what my parents, Abdul and Bonnie Lynn Hamid, have done for me.

All praise, then, to Allah, the Lord of the heavens and the Lord of the earth,
the Lord of the worlds. His alone is the Majesty in the heavens and the earth,
and He is the Mighty, the Wise.
(Qur'an 45:37-38)

[0]This research is based on work supported in part by DARPA OASIS grant F30602-99-1-0519, NSF grant CCR-9901011, NSF ITR grant CCR-0081590, and NSF grant CCR-0208618. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

Chapter 1

Introduction

At 11:30 p.m. EST on January 3, 2004, the Mars Exploration Rover "Spirit" sent its first radio signal to NASA scientists following its initial impact on the surface of the Red Planet. Then, after two weeks of exciting image gathering and exploration activities, a breakdown in communication occurred during which scientists would receive no more than a simple communication tone, indicating a serious anomaly aboard the spacecraft. It took 10 days of long-distance diagnostics and debugging to finally restore the rover to health again. Scientists determined that thousands of old files had accumulated in Spirit's flash memory and the onboard software was having difficulty managing the memory shortage situation, resulting in a complete reboot of the rover's computer about once every hour [53].

A certain amount of Spirit's onboard memory is dedicated to its real-time operating system and assorted science applications, the rest is used to store acquired data. As the mission progressed, technicians were supposed to periodically download and delete old data files to free up data memory for reuse, but apparently this step had not been performed fast enough in the excitement of the Mars landing [87].

Despite claims that it was a "system constraint" and not a hardware or application bug, it seems very strange that the computer software embedded in such a precious machine had not taken into account the occurence of such a memory shortage scenario. Per-

haps the possibility had risen in the minds of the programmers or system designers, but the proper checks to ensure that it did not happen, and to handle the situation automatically if a critical memory shortage was imminent, were not put into the code.

It would certainly be ideal that a set of appropriate safety specifications be provided with embedded operating systems code, such as that on the Mars rover, along with a guarantee that the actual code satisfies and conforms to those specifications. In fact, a framework to handle such system safety and integrity concerns using techniques from mathematical logic and programming language semantics has already been developed in the form of *proof-carrying code* [55]. In this dissertation, I will be contributing a new design to the proof-carrying code (PCC) framework, allowing for certification of safety properties for code written in different high- and low-level programming languages and integrated together to form a complete runtime system.

1.1 Proof-Carrying Code

Proof-Carrying Code (PCC) was initially developed in the context of network packet filters [56]. The basic idea of PCC, as the name implies, is that a piece of executable code comes packaged with a proof of its safety. A diagram of a PCC system is given in Figure 1.1. In a PCC system, there are typically two main entities, (1) a code producer, who transforms a source code program into compiled machine code along with its safety proof, and (2) a code consumer, who wishes to run the compiled code as long as it satisfies the safety policy.

While the main subject of this dissertation is the mechanism by which safety proofs are generated and checked, I will briefly review here what a safety policy is. Although a code consumer may ideally wish to specify that the code should do exactly what it is supposed to do, this is usually impractical for any programs other than short, trivial ones. The task of (formally) specifying everything that a large, complex piece of code is expected to accomplish is much too hard and open to error anyway. Instead, the code consumer could

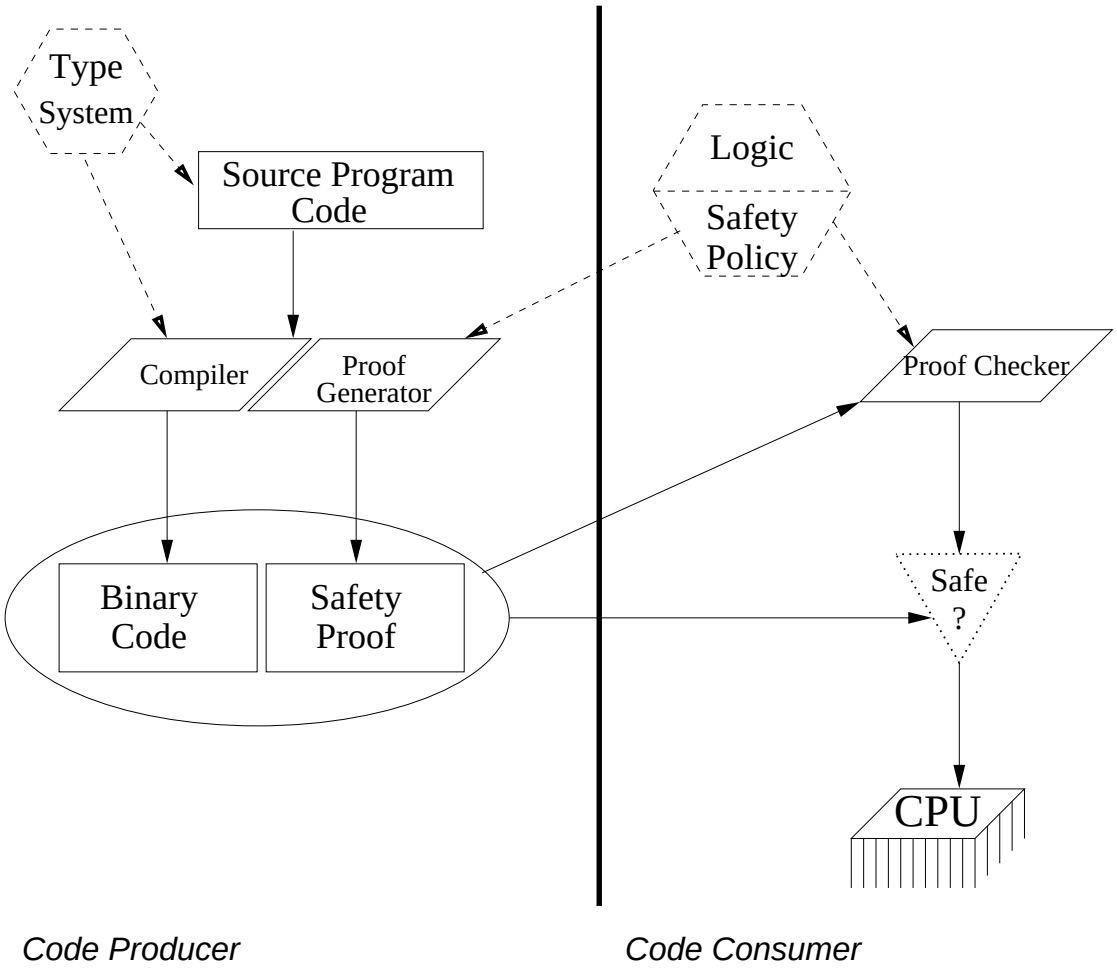


Figure 1.1: A (foundational) proof-carrying code system.

define a set of restrictions on execution– a *security policy* [71] defining what code behavior is unacceptable. There are then a variety of types of security policies that may be applied – *e.g.* access control, information flow, and availability. For this thesis, I will be using only access control policies, which restrict the operations that principals (*e.g.* blocks of code) can perform on objects (*e.g.* computer memory). In the terminology of [44], this means to specify *safety properties* of code – policies stating that no "bad thing" must happen during execution.

In the first part of this dissertation, I will be using a simple safety policy that only requires code never reach an illegal or non-decodable instruction. Later on, in Chapter 5, I will extend the policy to restrict memory accesses. Throughout, I assume a system in which the safety policy is fixed ahead of time and known to all parties. The issue of generating PCC packages where the code producer does not know the consumer's safety policy ahead of time is a topic of ongoing and future research. Furthermore, it should be noted that some part of the framework described in this thesis, developed jointly with Yu [97, 98], has been used to specify and prove more advanced security and correctness properties of low-level system code [97, 98, 99].

Returning to Figure 1.1, note that as usual with PCC, the code component that accompanies the safety proof is actual machine code. More accurately, the code binary is a representation of the machine state – memory and registers – at program startup. Although the ideas of PCC can be applied to code at higher levels, we will concentrate on producing proofs at the lowest level of the machine.[1] The motivation for this is based on a desire to reasonably minimize the trusted computing base (TCB) of the PCC system. The TCB is the totality of protection mechanisms within a computer system, including hardware, firmware, and software, the combination of which is responsible for enforcing a security policy. The smaller the mechanisms in this set are, the easier it is to verify them and the more confident one may be that they will operate correctly.

[1] In fact, the development of PCC followed a progression of research in the areas of higher level type systems and type-preserving compilation. See Chapter 7 for related works in these areas.

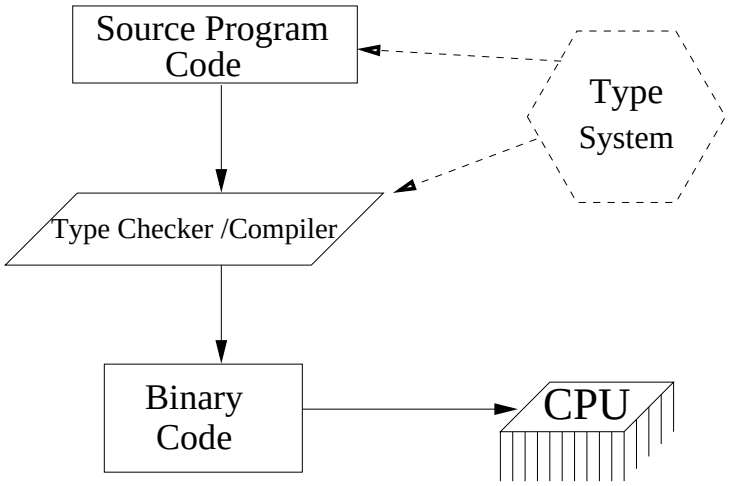


Figure 1.2: A conventional compilation system.

To understand this, consider the diagram of a "conventional" compilation system in Figure 1.2. If we are using a strongly typed language, like Java or ML, then all the components in the figure with which the original program source code interacts are in the TCB. We must trust that the type system is correct in that well-typed programs will not get stuck when compiled and run on the CPU. Then, we must trust that the type checker correctly implements the type system, and that the compiler will not introduce any bugs in the compiled binary. Finally, we must trust that the machine hardware will actually work according to specifications and execute the binary code as expected.

In Figure 1.2, one of the most error-prone components is the compiler. Compilers are large, complicated pieces of code themselves and are extremely likely to produce erroneous output at some point of transforming and optimizing program code. One approach to dealing with this is to somehow verify the compiler to obtain a formal guarantee that it will *always* work correctly. PCC takes quite a different approach, which is to remove the compiler from the TCB entirely. Hence, the situation in Figure 1.1, where the compiler is not part of the code consumer's TCB. Ideally, the TCB of such a system is composed of the following:

1. *Hardware* - My research concentrates entirely on certifying software components so

we take it for granted that the actual machine hardware will operate as expected.

2. *Logic* - The safety policy and proofs of safety must be expressed in some formal logic. The logic should have appropriately "good" qualities – for instance, consistency: that one cannot prove a statement and its negation at the same time. Proving such desirable properties of a logic is done by hand and we must trust that such proofs are correct. The smaller and simpler the logic we use, the more confidence we may have in its soundness properties. This is a very important issue for PCC, and we will later cite an example of using an unsound logic, which completely undermines the safety properties of the entire system.

3. *Proof checker* - The proof checker is a software implementation of the PCC logic. It allows one to express statements in the syntax of the logic and mechanically check proofs, which are a series of applications of the logical inference rules. Assuming a logic has all the desirable properties, we must trust that the proof checker implementation is correct. The checker is, however, a much smaller program than even a simple compiler, so it is feasible for a human to inspect and verify. Though I have not focused on an absolutely minimal proof checker for my current work, Appel and Michael [6] have aggressively pursued this aspect of the TCB in related PCC work.

4. *Safety Policy* - The final trusted component of the ideal PCC system is the definition of safety. This is expressed in the syntax of the logic and must be satisfied before code will be allowed to execute. The safety policy component may actually be made up in turn of several pieces:

   (a) *Machine encoding* - Within the PCC logic, one must provide a specification of the physical machine hardware and its operational semantics. This encoding must match the actual behavior of the machine.

   (b) *Decoder* - Code to convert between the actual machine state and the encoding in the logic.

(c) *Specification* of the actual policy, in terms of the machine encoding (a). I have already given above an overview of the type of policies I will work with. For more concrete definitions see Sections 2.2.3 or 5.3.2.

Thus, the components to the right of the vertical line in Figure 1.1 make up the ideal PCC trusted code base. On the left side, we do not care what the source program is, or how the compiler and proof generator produce a code package, as long as our proof checker verifies that the safety proof for the binary code satisfies the established policy.

I have been refering to Figure 1.1 as an "ideal" PCC diagram. The reason is that historically the first developments of PCC included more components than these in the TCB. I will now give an overview of "traditional" PCC, followed by the development of "foundational" PCC, which corresponds more to Figure 1.1.

1.2 Traditional Proof-Carrying Code

The concept of proof-carrying code was first introduced by Necula and Lee [56, 54, 58]. Subsequent developments resulted in PCC-generating compilers for Java programs [15], a subset of C, and a minimal subset of ML. A comprehensive overview of the initial system can be found in Necula's thesis [55]. The basic framework of Necula, *et al.* is diagrammed in Figure 1.3.

Although it looks considerably more complicated than Figure 1.1, it must be noted that this pioneering work focused very much on the practical aspects of implementing a realistic framework. Thus, the Java system cited above could handle "real world" Java programs of up to half a million lines of code [70]. Nonetheless, the emphasis on engineering and scalability meant a larger TCB – and the possibility, and even discovered presence, of security holes.

In the original PCC system, a certifying compiler takes a source program and produces binary code along with annotations (such as loop invariants) to various instructions in the code. These annotations are fed to a verification condition generator (VCGen) which

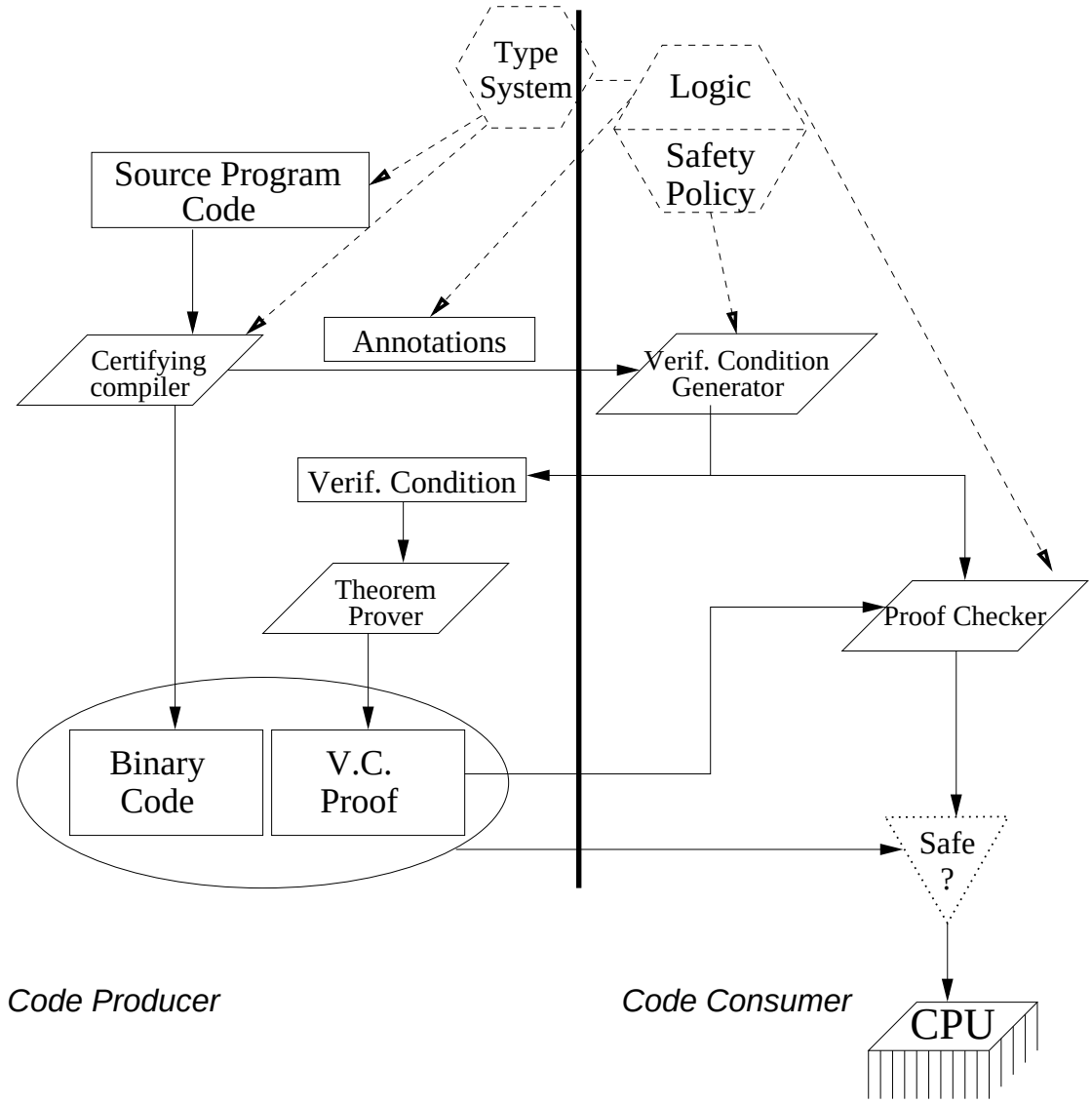


Figure 1.3: Original proof-carrying code system.

produces a set of conditions that must be shown to hold for the safety policy to be satisfied. A proof of these verification conditions (VC) is then produced by an automatic theorem prover (or even by hand). The binary code and the VC proof make up the PCC package. In this setup, note that the proof provided does not directly show the safety policy is satisfied. Instead, as the safety policy is implicitly defined by the way the VCGen works, we must trust that the VCs produced imply the safety policy. But this is only proved "by hand," not formally (mechanically) as part of the framework. The VCGen itself is quite a large and complex piece of code (23,000 lines of C code in [15]) to include in the TCB.

Finally, in the traditional PCC setup, the logic used is typically a standard mathematical logic extended with a number of type-system specific primitives. For example, [15] uses first-order predicate logic enriched with Java-specific predicates and rules for objects, interfaces, methods, *etc.* The implication of this is that it is a much more difficult and error-prone task to show that the logic satisfies the desirable properties alluded to on page 6. Introducing type system primitives into a logic, along with associated rules of inference, can lead to subtle interactions with the base logic and unsoundness. In [15] a bug was found in the safety conditions for virtual method calls, documented by League, *et al.* [45]. A malicious code producer could thus exploit such a security hole to get the consumer to run completely unsafe code on his or her system.

Besides increasing the size of the TCB, the original framework for PCC limits the flexibility of the system. In the previous paragraph I mention the use of a logic and VCGen customized with Java-specific extensions. This means that the safety policy is expressed in terms of Java constructs and it requires that the program code be annotated with Java typing information. Thus, for compiling another source language, it is necessary to use a completely different VCGen (and probably proof checker), which would result practically in a proliferation of incompatible PCC compiler systems.

In order to alleviate the drawbacks discussed in the foregoing paragraphs, Appel *et al.* [3] introduced the notion of *foundational* proof-carrying code (FPCC). This, in a sense, purist approach corresponds more directly to Figure 1.1. Emphasizing a minimal TCB

– as opposed to rapid engineering of a full-scale, realistic system – FPCC removed the VCGen and any type system-specific primitives from the right hand side of Figure 1.3. However, developing safety proofs for such a framework was not straightforward– or so it seemed at first. In the next section, I describe the development of the FPCC framework by Appel *et al.* and then concluded the chapter with an overview of my alternative FPCC framework and an outline of the rest of this thesis.

1.3 Foundational Proof-Carrying Code (Semantic Approach)

Foundational proof-carrying code (FPCC) [4, 3] focuses on constructing and verifying proofs using strictly the foundations of mathematical logic, with no type system-specific axioms or primitives. FPCC, in principle, is immediately more flexible and secure than traditional PCC because it is not tied to any particular type system and has a smaller trusted base (the VCGen is gone). For FPCC, the operational semantics of machine code as well as the concept of safety are defined in a suitably expressive logic. The code producer must provide both the executable code and a proof in the foundational logic that the code satisfies the safety condition. Both the machine description and the proof must explicitly define, down to the foundations of mathematics, all required concepts and must prove any needed properties of these concepts.

Foundational proofs, however, are much harder to construct. Previous efforts on FPCC [4, 25, 5, 2] all required constructing sophisticated semantic models to reason about types. For example, to support contravariant recursive types, Appel and Felty [25] initially decided to model each type as a partial equivalence relation, but later found that building the actual foundational proofs would "require years of effort implementing machine-checked proofs of basic results in computability theory" [5, page 2]. Appel and McAllester [5] later proposed an indexed model which significantly simplified the proofs but still involves tedious reasoning of computation steps with each type being defined as a complex set of indexed values. More seriously, none of these approaches could be easily extended

to support mutable fields. A solution for handling mutable fields was proposed later by Ahmed *et al.* [2] – it involves building a hierarchy of Gödel numberings and making extensive changes to semantic models used in existing FPCC systems [4, 5].

In the remainder of this section, I will compare the method of generating proofs in the traditional VCGen-based framework and a semantic FPCC-based system. A concrete example will hopefully clarify for the reader some aspects of the VCGen and PCC discussed in the previous section, while also demonstrating complications that arise in the semantic approach to FPCC.

Let us consider a hypothetical source code snippet that reads an element from a tuple:

```
(l1)        val t : int × int := (3, 4);
            ⋮
(l2)        val y : int := snd(t);
```

I will only concentrate on line (l2) here. A compiler might compile the assignment operation on this line to a machine load instruction (or more accurately, the binary encoding) such as $\texttt{ld}\ \texttt{r}_\texttt{y},\ \texttt{r}_\texttt{t}(1)$.

Now let us see how traditional PCC would produce a safety proof for this piece of code. When this instruction is examined by the VCGen, the following obligation will be produced as part of the VC [55, page 71]:

$$\texttt{readable(M, R(r}_t\texttt{) + 1)}$$

That is, in the current memory, M, the address computed by the contents of register $\texttt{r}_t$ incremented by 1 must be readable according to the safety policy. This VC term goes to the PCC theorem prover. The PCC theorem prover implements a logic extended with type system-specific primitives and inference rules. So, for example, the following inference rule might be built into the theorem prover [54]:

$$\frac{\texttt{M} \vdash \texttt{v} : \tau_1 \times \tau_2}{\texttt{readable(M, v)} \ \wedge\ \texttt{readable(M, v + 1)} \ \wedge\ \texttt{M} \vdash \texttt{M(v)} : \tau_1 \ \wedge\ \texttt{M} \vdash \texttt{M(v + 1)} : \tau_2} \tag{1.1}$$

At the point that line (l2) is being processed, the prover will have the information (*e.g.* from type annotations) that the content of register $\mathtt{r}_t$, $\mathtt{R}(\mathtt{r}_t)$, does have a tuple type, $\mathtt{int} \times \mathtt{int}$, with relation to the current memory, M. It then directly applies the inference rule (1.1) to satisfy the necessary VC obligation.

In this example, the rule (1.1) is quite simple and we can imagine that adding tuple type primitives to a logic will not be that dangerous. However, to handle more realistic type systems, like Java's, one would have to start including much more complex inference rules to the logic, VCGen, and theorem prover, resulting in the dangerous situation described in Section 1.2.

Now let us see how (semantic) FPCC works. Instead of extending the logic with type primitives, FPCC interprets types as predicates in the base logic:

$$\begin{aligned} \lceil \mathtt{M} \vdash \mathtt{v} : \tau_1 \times \tau_2 \rceil =_{def}\ & \mathtt{readable}(\mathtt{M}, \mathtt{v}) \ \wedge\ \mathtt{readable}(\mathtt{M}, \mathtt{v}+1) \\ & \wedge \lceil \mathtt{M} \vdash \mathtt{M}(\mathtt{v}) : \tau_1 \rceil \ \wedge\ \lceil \mathtt{M} \vdash \mathtt{M}(\mathtt{v}+1) : \tau_2 \rceil \end{aligned} \tag{1.2}$$

That is, the fact that data in memory has a tuple type in the source language is interpreted as the proposition on the right side of the definition above. A base case predicate may be defined for the int type:

$$\lceil \mathtt{M} \vdash \mathtt{v} : \mathtt{int} \rceil =_{def}\ \mathtt{true} \tag{1.3}$$

And, the readable predicate itself may be defined using more primitive constructs of the logic, such as requiring all readable addresses be less than 50:

$$\mathtt{readable}(\mathtt{M}, \mathtt{v}) =_{def}\ \mathtt{v} < 50 \tag{1.4}$$

Thus, for the code compiled from line (l2) in our example, we have the source typing information,

$$\mathtt{M} \vdash \mathtt{R}(\mathtt{r_t}) : \mathtt{int} \times \mathtt{int}$$

FPCC then interprets this as the following proposition (expanding all definitions, including `readable`):

$$\mathtt{R}(\mathtt{r_t}) < 50 \ \wedge \ \mathtt{R}(\mathtt{r_t}) + 1 < 50 \ \wedge \ \mathtt{true} \ \wedge \ \mathtt{true} \tag{1.5}$$

The FPCC consumer may specify a safety policy that all memory loads must only be from addresses less than 50 (*i.e.* readable). Thus, the compiled binary code, $\mathtt{ld}\ \mathtt{r_y}, \mathtt{r_t}(1)$, would clearly satisfy this requirement in the context of (1.5). Notice now that the FPCC reasoning only involves normal arithmetic and logic operators. The logic is much smaller and simpler, as will be the corresponding proof checker; and there is no VCGen at all. In more proper notation, source code types are formalized as functions on sets. So, (1.2) and (1.3) would be written as [3]:

$$\begin{aligned} &\lceil \tau_1 \times \tau_2 \rceil(\mathtt{M})(\mathtt{v}) =_{def} \ \mathtt{readable}(\mathtt{M}, \mathtt{v}) \ \wedge \ \mathtt{readable}(\mathtt{M}, \mathtt{v}+1) \ldots \\ &\lceil \mathtt{int} \rceil(\mathtt{M})(\mathtt{v}) =_{def} \ \mathtt{true} \end{aligned} \tag{1.6}$$

The logical type then of semantic interpretations of source code types such as `int` is a predicate on a memory and integer value:

$$type = mem \rightarrow val \rightarrow prop$$

Now let us suppose tuple elements are mutable and consider a third source code line:

```
(l3)            snd(t) := y;
```

which compiles to a machine store instruction, $\mathtt{st}\ \mathtt{r_t}(1),\ \mathtt{r_y}$. Based on the source code, we may assign types to the registers as follows:

$$\mathtt{M} \vdash \mathtt{R}(\mathtt{r_y}) : \mathtt{int} \tag{1.7}$$

$$\mathtt{M} \vdash \mathtt{R}(\mathtt{r_t}) : \mathtt{int} \times \mathtt{int} \tag{1.8}$$

Expanding the semantic interpretation of (1.8) we again have the proposition (1.5) above, while (1.7) simply expands to the `true` proposition by (1.3). Locally, then, the store instruction will not cause any problems because we are dealing with base `true` propositions at the location $\mathtt{R}(\mathtt{r_t}) + 1$.

However, what if there is aliasing in the register file? That is, the content of $\mathtt{r_s}$ is the same as that of $\mathtt{r_t}$ and somehow $\mathtt{r_s}$ has the type:

$$\mathtt{M} \vdash \mathtt{R}(\mathtt{r_s}) : \mathtt{int} \times (\mathtt{int} \times \mathtt{int})$$

Now the second element of $\mathtt{R}(\mathtt{r_s})$ is supposed to be a tuple pointer but we are writing a plain integer to the second element of $\mathtt{R}(\mathtt{r_t})$, where $\mathtt{R}(\mathtt{r_t}) = \mathtt{R}(\mathtt{r_s})$. In this situation, we would no longer be able to satisfy the typing requirement of $\mathtt{R}(\mathtt{r_s})$ after the store instruction is executed.

Most source level type systems eliminate this problem of aliasing by fixing a single type for each memory location and only allowing writes of that type to the location. In such a context, the situation above where two aliases have different types would not arise. But how can this be achieved in the FPCC semantic model of types? We cannot specify a type mapping directly because all types are being interpreted as predicates. The FPCC approach was to enhance the meaning of types by adding an "allocset" mapping to definition (1.6)– something like:

$$\begin{aligned} &\lceil \tau_1 \times \tau_2 \rceil(\mathtt{M})(\mathtt{a})(\mathtt{v}) =_{def} \ \ldots \wedge \mathtt{a}(\mathtt{M}(\mathtt{v})) = \lceil \tau_1 \rceil(\ldots) \wedge \mathtt{a}(\mathtt{M}(\mathtt{v}+1)) = \lceil \tau_2 \rceil(\ldots) \\ &\lceil \mathtt{int} \rceil(\mathtt{M})(\mathtt{a})(\mathtt{v}) =_{def} \ \mathtt{true} \end{aligned} \tag{1.9}$$

The idea being to enforce that all predicates on memory addresses be consistent with the mapping defined by the allocset, `a`. However, the definition above is not well-founded. Examing the logical types of $\lceil \tau_1 \times \tau_2 \rceil$ and `a`, we now have:

$$type = mem \rightarrow allocset \rightarrow val \rightarrow prop$$

$$allocset = val \rightarrow type$$

with an inconsistent cardinality in the metalogical type (note the circularity in the definition). This problem "stumped" the FPCC developers for over a year [3] until a solution was produced by Ahmed, *et al.* [2], which forms the basis of Ahmed's thesis dissertation.

The semantic modeling of types hit other similar difficulties when dealing with first-class functions (code pointers) and recursive types. As a result, the semantic models became much more complex in order to handle the additional type system features.

1.4 Dissertation Contributions and Outline

In contrast to the developments discussed above, I have worked on a "syntactic" approach to FPCC which is the main topic of this dissertation. In my approach, I have avoided using a logic with type system primitives as PCC, while at the same time have been immediately able to handle type system features that took years of work for the semantic FPCC approach to model. In addition, I have developed a framework allowing the interaction between code compiled from different type systems, or allowing the interoperation of code a portion of which is automatically proven safe based on compilation from a high level type-safe language, and the remaining portion of which (the "runtime library") has been certified manually or semi-automatically using a proof assistant. The potential, then, is a framework where certified user programs can run on, and interact with, a similarly certified runtime library and operating system. Though I have not yet progressed to actually programming a full-scale runtime system or operating system, this dissertation lays a viable foundation for such an effort.

In the next chapter, I describe the machine, logic, and safety policy that will be used in the rest of the work. Then, in Chapter 3, I introduce the syntactic approach to FPCC. The chapter is based mostly on previously published papers by Hamid, Shao, Trifonov,

et al. [33, 34]. Chapter 3 presents a somewhat monolithic approach to building certified syntactic FPCC packages and does not address the issue of interfacing with a runtime system or other type systems. Thus, I extend the framework for that purpose in Chapter 4, which is based on a recent paper by Hamid and Shao [32]. In Chapter 5, I demonstrate an application of the framework for an assembly language with a region type system and the corresponding region management runtime library. Finally, I discuss in Chapter 6 a variety of issues that I have encountered in my work on syntactic FPCC, and conclude with related works, a summary, and future work in Chapters 7 and 8.

Scope of the Dissertation

Before progressing onto the bulk of this thesis, I would note that the framework constructed so far is a prototype for an idealized machine, as will be described in more detail in the next chapter. Thus, I will not be presenting here a solution to the Mars rover problem – *i.e.*, an operating system kernel or full-scale memory management runtime library. As discussed in the earlier sections of this introduction, my prototype development shows that common type system features that have not been handled adequately or easily in previous frameworks can be supported in a straightforward manner using a syntactic approach to proof-carrying code. This thesis is supported by other researchers' recent and ongoing developments of more complete PCC frameworks for realistic machines (*e.g.* [19, 18]) based on the syntactic approach I have presented in the first half of this dissertation. For my own research, I have not immediately concentrated on technical details such as targeting a real machine, but have instead focused on the aspect of code interoperation for PCC. The latter part of this dissertation deals with my extension of the prototype framework to handle the issue of interfacing type system code with a certified library. In Chapter 8, I discuss some further limitations and future work in the context of my current prototype before concluding.

Coq implementation

All the proofs described in this paper have been formalized and mechanically proven in the Coq proof assistant, unless otherwise noted. (In particular, proofs for the runtime system of Chapter 5 are still in progress.) For each chapter, then, I provide a link to the downloadable Coq code and in each section I will mention the corresponding Coq file in the proof development. Also, the main portion of the Coq developments has been included in the Appendices.

Chapter 2

A Machine and Logic for Foundational Proof-Carrying Code

In this chapter, I present the target machine on which programs will run and the logic that I use to reason about the safety of the code being run. Throughout the thesis I will use an idealized machine to present my FPCC framework. A "real" machine introduces many engineering details– fixed-size integers, overflow, addressing modes, memory model, variable length instructions, relative addressing, speculative execution, *etc.*– which I would rather avoid while presenting my central contributions. Although some progress has been made towards an implementation upon the IA-32 (Intel x86) architecture, I leave that as future work for now. The primary issues that I have worked to solve for this thesis– an FPCC framework that easily handles advanced type system features such as mutable records and recursive types, and interfacing between code written at different levels of abstraction or type systems– are orthogonal to the technical details of a real architecture, although there are indeed many important issues to be dealt with there.

Following the presentation of the machine in Section 2.1, I present the logical system that I have used for reasoning about safety. The logic must be suitably expressive enough to encode the operational semantics of machine code as well as the concept of safety. A code producer will provide both the executable code and a proof in the foundational logic

$$
\begin{array}{llll}
\textit{Word} & \ni & w, i, pc & ::= 0 \mid 1 \mid \ldots \\
\textit{Regt} & \ni & r & ::= \mathtt{r0} \mid \mathtt{r1} \mid \ldots \mid \mathtt{r15} \\
\textit{Cmd} & \ni & c & ::= \mathtt{add}\ r_d, r_s, r_t \mid \mathtt{addi}\ r_d, r_s, i \mid \mathtt{sub}\ r_d, r_s, r_t \mid \mathtt{subi}\ r_d, r_s, i \\
 & & & \quad \mid \mathtt{mov}\ r_d, r_s \mid \mathtt{movi}\ r_d, i \mid \mathtt{bgt}\ r_s, r_t, w \mid \mathtt{bgti}\ r_s, i, w \\
 & & & \quad \mid \mathtt{ld}\ r_d, r_s(i) \mid \mathtt{st}\ r_d(i), r_s \mid \mathtt{jd}\ w \mid \mathtt{jmp}\ r \mid \mathtt{illegal}
\end{array}
$$

$$
\begin{aligned}
\mathbb{M} &\in \textit{Mem} = \textit{Word} \to \textit{Word} \\
\mathbb{R} &\in \textit{RFile} = \textit{Regt} \to \textit{Word} \\
\mathbb{S} &\in \textit{State} = \textit{Mem} \times \textit{RFile} \times \textit{Word}
\end{aligned}
$$

Figure 2.1: Machine state: memory, registers, and instructions (commands).

that the code satisfies the safety condition. In Section 2.2, I also define a safety condition that I will use throughout the first part of this thesis and discuss how proofs will be generated.

The Coq files corresponding to this chapter may be downloaded at:

`http://flint.cs.yale.edu/flint/publications/safpccjar.html`

The Coq developments for this chapter and the next are also included in Appendix A.

2.1 An Idealized Machine

The idealized machine I use is defined by a *machine state* and a *step function* describing the deterministic transition from one machine state to the next. The state consists of the hardware components of the machine: a memory, register file, and a special register containing the current program counter (pc), defined in Figure 2.1. I use a 16-register word-addressed machine with an unbounded memory of words of unlimited size. The figure also shows the instruction set (which I will refer to as *commands* to distinguish from assembly language *instructions* presented in a later chapter). Informally, the commands have the following effects:

`add` r_d, r_s, r_t	set register r_d to the sum of the contents of r_s and r_t;
`addi` r_d, r_s, i	set r_d to the sum of the contents of r_s and i;
`sub` r_d, r_s, r_t	set r_d to the difference between the contents of r_s and r_t;
`subi` r_d, r_s, i	set r_d to the difference between the contents of r_s and i;
`mov` r_d, r_s	copy the contents of r_s into r_d;
`movi` r_d, i	move an immediate value, i, into r_d;
`bgt` r_s, r_t, w	branch to location w if $r_s > r_t$;
`bgti` r_s, i, w	branch to location w if $r_s > i$;
`ld` $r_d, r_s(i)$	load the contents of memory location $r_s + i$ into r_d;
`st` $r_d(i), r_s$	store the contents of r_s into memory location $r_d + i$;
`jd` w	direct jump (transfer execution) to location w;
`jmp` r	indirect jump to the address in register r;
`illegal`	put the machine in an infinite loop.

Of course, these commands are actually encoded as words (integers) in the machine state. I define *Cmd* as an inductive type because its constructors are much easier to manipulate that the encoded words. Now, in order to specify the operational semantics of the machine, I define a decoding function (Dc) and a Step function. Dc (of type *Word* $\rightarrow$ *Cmd*) decodes integers words from memory into the appropriate structured representation of commands shown in Figure 2.1. Non-decodable words will result in an `illegal` instruction (indicating the program counter is at a non-code address and the machine has "crashed").

The Step function describes the deterministic transition from one machine state to the next, depending on the command at the current pc. Its definition is given in Figure 2.2. The commands' effects are as they have been informally explained above. In our idealized machine, the load and store commands have no side conditions because memory is infinite (readable and writable limitations on areas of memory will be specified later in

if $\mathsf{Dc}(\mathbb{M}(pc)) =$	then $\mathsf{Step}(\mathbb{M}, \mathbb{R}, pc) =$
`add` r_d, r_s, r_t	$(\mathbb{M}, \mathbb{R}\{r_d \mapsto \mathbb{R}(r_s) + \mathbb{R}(r_t)\}, pc{+}1)$
`addi` r_d, r_s, i	$(\mathbb{M}, \mathbb{R}\{r_d \mapsto \mathbb{R}(r_s) + i\}, pc{+}1)$
`sub` r_d, r_s, r_t	$(\mathbb{M}, \mathbb{R}\{r_d \mapsto \mathbb{R}(r_s) - \mathbb{R}(r_t)\}, pc{+}1)$
`subi` r_d, r_s, i	$(\mathbb{M}, \mathbb{R}\{r_d \mapsto \mathbb{R}(r_s) - i\}, pc{+}1)$
`mov` r_d, r_s	$(\mathbb{M}, \mathbb{R}\{r_d \mapsto r_s\}, pc{+}1)$
`movi` r_d, i	$(\mathbb{M}, \mathbb{R}\{r_d \mapsto i\}, pc{+}1)$
`ld` $r_d, r_s(i)$	$(\mathbb{M}, \mathbb{R}\{r_d \mapsto \mathbb{M}(\mathbb{R}(r_s) + i)\}, pc{+}1)$
`st` $r_d(i), r_s$	$(\mathbb{M}\{\mathbb{R}(r_d) + i \mapsto \mathbb{R}(r_s)\}, \mathbb{R}, pc{+}1)$
`bgt` r_s, r_t, w	$(\mathbb{M}, \mathbb{R}, pc{+}1)$ when $\mathbb{R}(r_s) \leq \mathbb{R}(r_t)$ $(\mathbb{M}, \mathbb{R}, w)$ when $\mathbb{R}(r_s) > \mathbb{R}(r_t)$
`bgti` r_s, i, w	$(\mathbb{M}, \mathbb{R}, pc{+}1)$ when $\mathbb{R}(r_s) \leq i$ $(\mathbb{M}, \mathbb{R}, w)$ when $\mathbb{R}(r_s) > i$
`jd` w	$(\mathbb{M}, \mathbb{R}, w)$
`jmp` r	$(\mathbb{M}, \mathbb{R}, \mathbb{R}(r))$
`illegal`	$(\mathbb{M}, \mathbb{R}, pc)$

Figure 2.2: Machine semantics.

my safety policy– see Chapter 5).

Coq code: `tis.v` formalizes Figure 2.1 and the decode and step functions of this section.

2.2 The Logic and Safety Condition

2.2.1 The Calculus of Inductive Constructions

In order to produce FPCC packages, we need a logic in which we can express (encode) the operational semantics of the machine defined above, as well as define the concept and criteria of safety. A code producer must then provide a code executable (initial machine state) along with a proof that the initial state and all future transitions therefrom satisfy the safety condition.

The foundational logic I use is the calculus of inductive constructions (CiC) [78, 62]. CiC is an extension of the calculus of constructions (CC) [16], which is a higher-order typed lambda calculus. CC corresponds to a variant of higher-order predicate logic via the formulae-as-types principle (Curry-Howard correspondence [40]). The syntax of CC

is:

$$A, B ::= \mathsf{Set} \mid \mathsf{Type} \mid X \mid \lambda X\!:\!A.\, B \mid A\, B \mid \Pi X\!:\!A.\, B$$

The Coq implementation (discussed more below) adds another sort, Prop, to the calculus, along with Set. Under the *proposition-as-types*, *proofs-as-terms* paradigm, if A has sort Prop then it represents a logical proposition. A term M that inhabits A (*i.e.* has type A) is a proof of the proposition. On the other hand, terms of the sort Set are used as the types of data types, such as the natural numbers, lists, trees, booleans, *etc.*

CiC, as its name implies, extends the calculus of constructions with inductive definitions [17, 64, 62]. An inductive definition can be written in a syntax similar to that of ML datatypes. For example, the following introduces an inductive definition of natural numbers of kind Set with two constructors of the specified types:

$$\mathsf{Inductive\ Nat\ :\ Set\ :=\ zero : Nat \mid succ : Nat \to Nat}$$

Inductive definitions may be parameterized as in the following definition of polymorphic lists:

$$\begin{aligned}\mathsf{Inductive\ List}\ [t\!:\!\mathsf{Set}]\ :\ \mathsf{Set}\ :=\ \mathsf{nil}\quad &: \mathsf{List}\, t \\ \mid\ \mathsf{cons} &: t \to \mathsf{List}\, t \to \mathsf{List}\, t\end{aligned}$$

The logic also provides elimination constructs for inductive definitions, which combine case analysis with a fix-point operation. Objects of an inductive type can thus be iterated over using these constructs. In order for the induction to be well-founded and for iterators to terminate, a few constraints are imposed on the shape of inductive definitions; most importantly, the defined type can only occur positively in the arguments of its constructors. Mutually inductive types are also supported.

The calculus of inductive constructions has been shown to be strongly normalizing [88], hence the corresponding logic is consistent. It is supported by the Coq proof assistant [78], which I use to implement a prototype system of the results presented in this thesis.

In the remainder of this thesis, I will often use more familiar mathematical notation (as in Figure 2.1) to present definitions and the statement of propositions, rather than the strict definition of CiC syntax given in this section. For example, the application of two terms will be written as $A(B)$ and inductive definitions will be presented in BNF format. I will often, however, retain the Π notation, which can generally be read as a universal quantifier.

2.2.2 Representing Proofs and Coq Syntax

Before discussing safety proofs, I first give a taste of the Curry-Howard correspondence between proofs and programs in action – that is, how one uses a calculus like CiC to state propositions and represent proofs.

Let us say we wish to produce some proofs about properties of the natural numbers, defined above (Nat). First, I define the less-than-equal predicate using an inductive definition:

$$\begin{aligned}\mathsf{Inductive\ le}\ [n:\mathsf{Nat}]\ :\ \mathsf{Nat}\rightarrow\mathsf{Prop}\ :=\ &\mathsf{le\_n}\ :\ \mathsf{le}\,n\,n\\ &\mid\ \mathsf{le\_s}\ :\ \Pi m:\mathsf{Nat}.\,\mathsf{le}\,n\,m\rightarrow\mathsf{le}\,n\,(\mathsf{succ}\,m)\end{aligned}$$

This definition corresponds to the pair of inference rules:

$$\frac{}{n \leq n}\ (\textsc{le\_n}) \qquad \frac{n \leq m}{n \leq m+1}\ (\textsc{le\_s})$$

Now, suppose we wish to prove the following theorem:

Theorem 2.1 (n_le_zero) For all natural numbers n, $0 \leq n$.

Proof By induction on n. Base case $n = 0$: apply the (LE_N) rule. Inductive case $n = m + 1$: by the inductive hypothesis, we know $0 \leq m$. Then use this with the (LE_S)

rule to show $0 \leq m + 1$. □

In the CiC calculus, we would represent this proof as a function taking a natural number argument n and producing a proof term that $0 \leq n$. The proof term is built by recursive case analysis on the argument, corresponding to the induction in our written proof above:

```
zero_le_n : Πn:Nat. le 0 n
  := λn:Nat. Cases n of
               zero ⇒ (le_n zero)
             | succ m ⇒ (le_s zero m (zero_le_n m))
```

The recursive call corresponds to use of the inductive hypothesis.

Thus, the process of formalizing and mechanizing proofs in CiC is very similar to writing a program, although there are constraints on what "programs" can be written. In practice, such proofs are developed using the Coq proof assistant [79], an implementation of the CiC calculus. Coq concrete syntax is somewhat different than the mathematical syntax used above. Additionally, over the course of my dissertation work, the Coq tools have undergone a major upgrade with a revised syntax. Unfortunately, therefore, some of my proof developments for this thesis are in the older (version 7.3) syntax while I have done the latest developments using the new (8.0) syntax.

The table below compares Coq notations to the CiC syntax presented earlier in this section.

CiC	Coq 7.3	Coq 8.0
Set	Set, Prop	Set[1], Prop
Type	Type	Type
X	x	x
$\lambda X : A.\ B$	[x:A] B	fun x:A => B
$A\ B$	A B	A B
$\Pi X : A.\ B$	(x:A) B	forall x:A, B

The notation for inductive definitions and case analysis in Coq is essentially similar to the example above with natural numbers and less-than-equal. Coq provides a number of primitive tactics used to develop proofs interactively. Though powerful, however, the level of automation is not as much as one might hope for, as I will discuss in Section 6.1.

For now, let us resume the discussion of proofs and safety in the FPCC framework.

2.2.3 Defining Safety

The safety condition is a predicate expressing the fact that code will not "go wrong." I say that a machine state $\mathbb{S}$ is safe if every state it can ever reach satisfies the safety policy SP:

$$\mathsf{Safe}\,(\mathbb{S}, \mathsf{SP}) = \Pi n : \mathsf{Nat}.\ \mathsf{SP}\,(\mathsf{Step}^n\,(\mathbb{S}))$$

A typical safety policy may require such things as the program counter must point to a valid instruction address in the code area and that any writes (reads) to (from) memory must be from a properly accessible area of the data space. For the next chapter of this thesis, I will be using a very simple safety policy, requiring only that the machine is never at an invalid instruction:

$$\mathsf{BasicSP}\,(\mathbb{M}, \mathbb{R}, pc) = \quad (\mathsf{Dc}\,(\mathbb{M}(pc)) \neq \texttt{illegal})$$

We can easily define access controls on memory reads and writes by including another predicate in the safety policy, $\mathsf{SafeRdWr}(\mathbb{M}, \mathbb{R}, pc)$ – I will do this in Chapter 5. By reasoning over the number of steps of computation more complex safety policies including temporal constraints can potentially be expressed. However, I will not be dealing with such policies here.[2]

The FPCC code producer has to provide an encoding of the initial state $\mathbb{S}_0$ along with a proof A that this state satisfies the safety condition BasicSP, specified by the code con-

[1] Predicative universe.

[2] Yu [99] has recently shown some results in this.

sumer. The final FPCC package is thus a pair:

$$F = (\mathbb{S}_0 : \mathit{State},\ A : \mathsf{Safe}\,(\mathbb{S}_0, \mathsf{BasicSP})).$$

2.2.4 Generating Proofs

In the first iteration, the actual proof of safety is organized following the approach used by Appel *et al.* [4, 5]. I construct an induction hypothesis Inv, also known as the *global invariant*, which holds for all states reachable from the initial state and is strong enough to imply safety. Then, to show that the initial state $\mathbb{S}_0$ is safe, I provide proofs for the propositions:

FPCC Initial Condition:	$\mathsf{Inv}\,(\mathbb{S}_0)$
FPCC Preservation:	$\Pi\mathbb{S} : \mathit{State}.\ \mathsf{Inv}\,(\mathbb{S}) \rightarrow \mathsf{Inv}\,(\mathsf{Step}\,(\mathbb{S}))$
FPCC Progress:	$\Pi\mathbb{S} : \mathit{State}.\ \mathsf{Inv}\,(\mathbb{S}) \rightarrow \mathsf{SP}\,(\mathbb{S})$

These propositions intuitively state that the invariant holds for the initial state, and for every subsequent state during the execution. FPCC Progress establishes that whenever the invariant holds, the safety policy of the machine is also satisfied. Together, these imply that during the execution of the program the safety policy will never be violated. To prove the initial state is safe, first I use the Initial Condition and Preservation, and show by induction that

$$\Pi n : \mathsf{Nat}.\ \mathsf{Inv}\,(\mathsf{Step}^n\,(\mathbb{S}_0)).$$

Then $\mathsf{Safe}\,(\mathbb{S}_0)$ follows directly by Progress.

Unlike Appel *et al.*, who construct the invariant by means of a semantic model of types at the machine level, my approach is based on the use of type soundness [93]: I define $\mathsf{Inv}\,(\mathbb{S})$ to mean that $\mathbb{S}$ is "well-formed" syntactically. The well-formedness property must be preserved by the step function, and must imply safety; the proofs of these properties are encoded in the FPCC logic as proof terms for Preservation and Progress.

In the following chapter I show how to derive the notion of well-formedness for a machine state by relating the state to a type-correct *program* in a typed assembly language. The type system of the language defines a set of inference rules for judgments of the form $\vdash \mathcal{P}$, meaning that the program $\mathcal{P}$ is well-formed (type-correct). The dynamic semantics of the language specifies an evaluation relation $\longmapsto$ on programs; I use here the term "program" to denote not only code but a more general configuration fully representing a stage of the evaluation. The syntactic approach to proving soundness of a type system involves proving progress[3] (if $\vdash \mathcal{P}$, then $\mathcal{P}$ is not stuck, *i.e.* there exists $\mathcal{P}'$ such that $\mathcal{P} \longmapsto \mathcal{P}'$) and preservation (if $\vdash \mathcal{P}$ and $\mathcal{P} \longmapsto \mathcal{P}'$, then $\vdash \mathcal{P}'$).

The central idea of my approach to FPCC is to find a typed assembly language and a translation relation $\Rightarrow$ between its programs and machine states, such that type-correct programs are mapped to well-formed states, and the evaluation relation is related to the machine step function – that is, if $\mathcal{P} \Rightarrow \mathbb{S}$ and $\mathcal{P} \longmapsto \mathcal{P}'$, then $\mathcal{P}' \Rightarrow \mathsf{Step}(\mathbb{S})$. If these properties hold, I can define the invariant $\mathsf{Inv}(\mathbb{S})$ as simply stating that there exists a type-correct program $\mathcal{P}$ such that $\mathcal{P} \Rightarrow \mathbb{S}$. Then the proofs of progress and preservation for the type system (completely formalized in the FPCC logic) can be used to construct straightforward proofs of the corresponding propositions needed for the safety proof for $\mathbb{S}_0$. Further details of the construction of proof terms are provided in Section 3.2.3.

The method of generating FPCC proofs above appears somewhat monolithic in its use of the global invariant as I have described above. It imposes requirements on the design of the typed assembly language other than just having a sound type system. If the assembly language has "macro" instructions (*e.g.* `malloc` [52, 51] and `newarray` [95], which "expand" into sequences of several machine instructions), the well-formedness of the assembly program alone will be insufficient for the construction of the global invariant. This is because Inv must hold for all machine states reachable from $\mathbb{S}_0$. For the intermediate states of the execution of a macro instruction there are no corresponding well-formed assembly

[3]This refers to progress in the source type system, not the FPCC Progress proposition defined earlier in this section.

programs. Hence, each one of the assembly instructions must correspond to exactly one machine instruction. Note, however, that this exact correspondence of instructions is not necessary in general for the syntactic approach to work, although it facilitates the definition of the invariant and allows for a simpler presentation.

In fact, in Chapter 4, I will refine the approach described above by inserting a generic layer of reasoning above the machine code which can (1) be a target for the compilation of typed assembly languages, (2) certify low-level runtime system components using assertions as in Hoare logic, and (3) "glue" together these pieces by reasoning about the compatibility of the interfaces specified by the various types of source code. In this context, the global invariant is broken down into a disjunction of local invariants on each individual machine instruction. The local invariants still use the syntactic FPCC approach to relate machine instructions to the typed assembly language source code, but by breaking apart the global invariant, I can easily handle macro instructions and interactions between different type systems.

Chapter 3

A Syntactic Approach to Foundational Proof-Carrying Code

An overview of the syntactic approach to FPCC has been given in the previous chapter. In this chapter, I define a sample source language and type system, including its static and dynamic semantics and proof of soundness. Then I show how to compile programs from this language to the machine defined in Chapter 2, producing the required FPCC proofs in the process. I also give an overview of the Coq implementation (Appendix A), the complete files of which are available for download at:
`http://flint.cs.yale.edu/flint/publications/safpccjar.html`

3.1 Featherweight Typed Assembly Language

The source language that I will be compiling to FPCC is a version of the typed assembly language (TAL) by Morrisett *et al.* [52]. The approach developed in this thesis can be applied to a TAL-like language extended with higher-order kinds and recursive types. For simplicity, I only introduce here a subset of such a language, called Featherweight Typed Assembly Language (FTAL). It does not include polymorphism or existential types, which can be easily added but would complicate the presentation (Chapter 5 presents another

$$
\begin{array}{lll}
(\textit{type}) & \tau & ::= \alpha \mid \mathsf{int} \mid \forall[\Gamma] \mid \langle \tau_1^{\,\varphi_1}, \ldots, \tau_n^{\,\varphi_n} \rangle \mid \mu\alpha.\tau \\
(\textit{init flag}) & \varphi & ::= \mathsf{0} \mid \mathsf{1} \\
(\textit{heap ty}) & \Psi & ::= \{\mathbf{0}:\tau_0, \ldots, \mathbf{n}:\tau_n\} \\
(\textit{alloc pt ty}) & \rho & ::= \mathsf{fresh} \mid \mathsf{used}(n) \\
(\textit{regfile ty}) & \Gamma & ::= \{r_0:\tau_0, \ldots, r_n:\tau_n, \mathrm{r15}:\rho\} \\
(\textit{label}) & l & ::= \mathbf{0} \mid \mathbf{1} \mid \ldots \\
(\textit{user reg}) & r & ::= \mathsf{r0} \mid \mathsf{r1} \mid \ldots \mid \mathrm{r14} \\
(\textit{all reg}) & \hat{r} & ::= r \mid \mathrm{r15} \\
(\textit{word val}) & v & ::= l \mid i \mid ?\tau \mid \mathsf{fold}\, v\, \mathsf{as}\, \tau \\
(\textit{heap val}) & h & ::= \langle v_1, \ldots, v_n \rangle \mid \mathsf{code}\, [\Gamma].I \\
(\textit{heap}) & H & ::= \{\mathbf{0} \mapsto h_0, \ldots, \mathbf{n} \mapsto h_n\} \\
(\textit{regfile}) & R & ::= \{r_0 \mapsto v_0, \ldots, r_{15} \mapsto v_{15}\} \\
(\textit{instr}) & \iota & ::= \mathsf{add}\ r_d, r_s, r_t \mid \mathsf{addi}\ r_d, r_s, i \mid \mathsf{alloc}\ r_d[\vec{\tau}] \mid \mathsf{bgt}\ r_s, r_t, l \\
 & & \quad \mid \mathsf{bump}\ i \mid \mathsf{fold}\ r_d[\tau], r_s \mid \mathsf{ld}\ r_d, r_s(i) \mid \mathsf{mov}\ r_d, r_s \\
 & & \quad \mid \mathsf{movi}\ r_d, i \mid \mathsf{movl}\ r_d, l \mid \mathsf{st}\ r_d(i), r_s \mid \mathsf{unfold}\ r_d, r_s \\
(\textit{instr seq}) & I & ::= \iota; I \mid \mathsf{jd}\ l \mid \mathsf{jmp}\ r \\
(\textit{program}) & \mathcal{P} & ::= (H, R, I)
\end{array}
$$

Figure 3.1: Syntax of FTAL.

TAL with these features). However, it does support first-class code pointers, recursive types, memory allocation, and mutable records (tuples).

The syntactic approach to FPCC as presented here requires that for each machine state and each state transition, there be a corresponding FTAL program and transition. For most FTAL instructions it is easy to see there is a one-to-one mapping to the machine instructions of Section 2.1. However, having a malloc "macro instruction" in FTAL (as in TAL) will not work because it cannot be mapped to a single machine instruction and will not satisfy our requirements for generating FPCC proofs, since there would be no corresponding FTAL state between the expanded machine instructions. (See Section 3.1.6 for details on this issue.) My approach here is to make the memory allocation model explicit and split the malloc instruction into, in this case, two individual instructions.

3.1.1 Syntax

The syntax of FTAL is presented in Figure 3.1. As in TAL, the abstract machine state (which I will call a *program* to distinguish from the machine state of Section 2.1) consists of a heap H, a register file R, and a sequence of instructions I. The heap maps labels l to heap values h, and the register file maps registers $\hat{r}$ to word values v. I use $\{\}$ for an empty heap. The notation $H\{l \mapsto h\}$ represents a heap which extends H with a label l mapped to h. Similar notation is used for heap types, register files, and register file types (except that for the latter two, I also use the same notation to indicate an update to the mapping). When extending the heap, the type system will implicitly enforce a constraint that the labels in the heap be in consecutive ascending order starting at **0**. In the register file type (Γ), r15 is a special allocation pointer register and of the remaining user registers, not all of them need appear in the type. The notation $|H|$ and $|\Psi|$ is used to represent the number of labels in the heap and heap type, respectively. Notice, that this number will also correspond to the next unused label in the heap or heap type.

Only tuples and code blocks are stored in the heap and thus these are the heap values. Word values include labels (pointers to heap values), integers, recursive data, and junk values ($?\tau$), which are used by the operational semantics to represent uninitialized tuple elements annotated with a type. The distinction between word values and small values in TAL is eliminated in FTAL by specializing the instruction set. Thus, for example, there are now two instructions for addition, one (add) taking a register and the other (addi) using an immediate value as the third operand.

The memory model is a simple linear unbounded heap with an allocation pointer pointing to the heap top, initially set to the bottom of the heap space. Memory allocation consists of copying the current allocation pointer to a register using alloc and then adjusting the allocation pointer with bump. In Section 3.2.1 we will see how these two instructions can be directly translated into one FPCC machine instruction each. One of the general registers, r15, is reserved as the allocation pointer register, tracking the amount

of allocated memory. FTAL instructions will only explicitly refer to the first 15 "user" registers (r).

After an alloc instruction, a corresponding bump must be executed, to adjust the allocation pointer, before alloc can be used again. To statically enforce this, I give the allocation pointer register a special *allocation status type*, ρ, rather than a normal type. The possible types for this register, fresh and used(i), reflect the two states of allocation. Keeping track of the allocation status allows other instructions to be interleaved between an alloc and bump pair.

To meaningfully implement linear allocation, we need an ordering on memory labels, so I have defined labels as natural numbers. To determine whether a label has been allocated in the heap, it is compared with the heap size, $|H|$.

The types of FTAL are integers, code, tuple types annotated with initialization flags (φ), and recursive types. The initialization flags indicate whether there is valid data at each position of the tuple (when a tuple is first allocated, all the flags are 0). Other than fold and unfold, the remaining instructions (add, addi, bgt, mov, movi, movl, ld, and st) are equivalent or similar to those in the original TAL. A code block is a sequence of instructions, annotated with a register file type (essentially specifying the preconditions on the data expected in the registers when the code block begins executing). Code blocks always end with a jmp or jd instruction, though they may also be exited in the middle by bgt.

Operations on recursive types in FTAL are supported by the fold and unfold instructions. Dynamically, these are no different than a simple mov. Statically, however, their purpose is to "cast" the type of a word, by either "rolling up" or "unrolling" the recursive type. (See the relevant rules of the static semantics in Section 3.1.3.)

3.1.2 Dynamic Semantics

The operational semantics of FTAL is presented in Figure 3.2. Most of the instructions have an intuitively clear meaning. The ld and st instructions load from and store to a

$(H, R, I) \longmapsto \mathcal{P}$ where	
if $I =$	then $\mathcal{P} =$
add $r_d, r_s, r_t; I'$	$(H, R\{r_d \mapsto R(r_s) + R(r_t)\}, I')$
addi $r_d, r_s, i; I'$	$(H, R\{r_d \mapsto R(r_s) + i\}, I')$
alloc $r_d[\vec{\tau}]; I'$	$(H', R\{r_d \mapsto l\}, I')$ where $\vec{\tau} = \tau_1, \ldots, \tau_n$, $R(\mathsf{r15}) = l$, and $H' = H\{l \mapsto \langle ?\tau_1, \ldots, ?\tau_n \rangle\}$
bgt $r_s, r_t, l; I'$	(H, R, I') when $R(r_s) \leq R(r_t)$; and (H, R, I'') when $R(r_s) > R(r_t)$ where $H(l) = \mathsf{code}\,[\Gamma].I''$
bump $i; I'$	$(H, R\{\mathsf{r15} \mapsto \lvert H \rvert\}, I')$
fold $r_d[r_s], \tau; I'$	$(H, R\{r_d \mapsto \mathsf{fold}\, R(r_s)\, \mathsf{as}\, \tau\}, I')$
jd l	(H, R, I') where $H(l) = \mathsf{code}\,[\Gamma].I'$
jmp r	(H, R, I') where $H(R(r)) = \mathsf{code}\,[\Gamma].I'$
ld $r_d, r_s(i); I'$	$(H, R\{r_d \mapsto v_i\}, I')$ where $0 \leq i < n$ $H(R(r_s)) = \langle v_0, \ldots, v_{n-1} \rangle$
mov $r_d, r_s; I'$	$(H, R\{r_d \mapsto R(r_s)\}, I')$
movi $r_d, i; I'$	$(H, R\{r_d \mapsto i\}, I')$
movl $r_d, l; I'$	$(H, R\{r_d \mapsto l\}, I')$
st $r_d(i), r_s; I'$	$(H\{l \mapsto h\}, R, I')$ where $0 \leq i < n$ $R(r_d) = l$, $H(l) = \langle v_0, \ldots, v_{n-1} \rangle$, and $h = \langle v_0, \ldots, v_{i-1}, R(r_s), v_{i+1}, \ldots, v_{n-1} \rangle$
unfold $r_d, r_s; I'$	$(H, R\{r_d \mapsto v\}, I')$ where $R(r_s) = \mathsf{fold}\, v\, \mathsf{as}\, \tau$

Figure 3.2: Operational semantics of FTAL.

tuple in the heap using the specified index. The instruction bgt r_s, r_t, l tests whether the value in r_s is larger than that in r_t, and, if so, transfers control to the code block at l.

In order to allocate a tuple in the heap, first the alloc instruction is used to copy the current heap allocation pointer to r_d and allocate the desired size in the heap. Before the next allocation, the allocation pointer needs to be adjusted. This is achieved using the bump instruction, which sets the allocation pointer to the next unused region of the heap, as described earlier. (The i argument is not used by the operational semantics.) Since we assume a linear allocation method, unused regions of the heap are simply all those beyond the currently allocated data.

The fold instruction annotates the value of r_s with the recursive type and moves it into r_d, while unfold extracts the value from the recursive package in r_s into r_d. Note that the

Judgment	Meaning
$\vdash \tau$	τ is a well-formed type
$\vdash \Psi$	Ψ is a well-formed heap type
$\vdash \Gamma$	Γ is a well-formed regfile type
$\vdash \tau_1 \leq \tau_2$	τ_1 is a subtype of τ_2
$\vdash \Gamma_1 \subseteq \Gamma_2$	Γ_1 is a regfile subtype of Γ_2
$\vdash \mathcal{P}$	$\mathcal{P}$ is a well-formed program
$\vdash H : \Psi$	H is a well-formed heap of type Ψ
$\Psi \vdash R : \Gamma$	R is a well-formed regfile of type Γ
$\Psi \vdash l : \rho$	l is a label of allocation status ρ
$\Psi \vdash h : \tau$ hval	h is a well-formed heap value of type τ
$\Psi \vdash v : \tau$	v is a well-formed word value of type τ
$\Psi \vdash v : \tau^{\varphi}$	v is a well-formed word value of type τ^{φ}
$\Psi; \Gamma \vdash I$	I is a well-formed instruction sequence

Figure 3.3: Static judgments of FTAL.

fold and unfold instructions of FTAL (as well as TAL) are not no-ops but copy a value from one register to another.

3.1.3 Static Semantics

The primary judgment of the static semantics is that of the well-formedness of a program. That in turn depends on judgments of the well-formedness of the heap, heap type, register file, register file type, and instruction sequence. The various typing judgments are summarized in Figure 3.3. The complete rules of the FTAL static semantics are given in Figures 3.4 to 3.6.

The top-level well-formedness rules are shown in Figure 3.4. To have a well-formed program, the heap and register file must be well-formed in some appropriate environments, as must be the current instruction sequence. Additionally, the current instruction sequence must be present in the heap. The notation $I \subseteq_{\mathit{tail}} I'$ means that I is a suffix of I'. For a heap to be well-formed the domain of the heap type must be the same as that of the heap and each heap value must be well-formed. However, the type of a well-formed register file need only specify a subset of the registers in its domain. The premise on the second line of the (REG) rule is not needed for FTAL type soundness but it is necessary to

$$\boxed{\vdash \mathcal{P} \quad \vdash H : \Psi \quad \Psi \vdash R : \Gamma}$$

$$\frac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \Psi; \Gamma \vdash I \quad \exists l \in Dom(H).H(l) = \mathsf{code}\ [\Gamma'].I' \textit{ and } I \subseteq_{\textit{tail}} I'}{\vdash (H, R, I)}\ (\textsc{prog})$$

$$\frac{\vdash \Psi \quad |\Psi| = |H| \quad \Psi \vdash H(l) : \Psi(l)\ \mathsf{hval} \quad (\mathbf{0} \leq l < |H|)}{\vdash H : \Psi}\ (\textsc{heap})$$

$$\frac{\Psi \vdash R(r_i) : \tau_i \quad (0 \leq i \leq n) \quad \Psi \vdash R(\mathsf{r15}) : \rho \quad \forall r \in Dom(R) - \{\mathsf{r15}\}.\textit{if } R(r) = l \textit{ then } l < |\Psi|}{\Psi \vdash R : \{r_0 : \tau_0, \ldots, r_n : \tau_n, \mathsf{r15} : \rho\}}\ (\textsc{reg})$$

Figure 3.4: Well-formedness of FTAL programs, heaps, and register files.

enforce some invariants during the translation to FPCC, as will be discussed later.

Subtyping is used for two purposes: one to allow a code block to be called when the current register file type is more detailed than needed, and the other to be able to type-check the initialization of an uninitialized tuple element as described below.

To type-check heap allocation and the load and store operations, we follow TAL by introducing initialization flags in the type of tuples. When a tuple is newly allocated on the heap, all the elements are flagged with 0. A store operation will set the flag of the appropriate element to 1. Thus, a load operation is only well-formed if the flagged type of the element being accessed is set to 1. Because the type system only approximately tracks the initialization of tuple elements, we use subtyping to allow initialized tuple elements to be treated as if they were not initialized – see rules (0-1) and (LABEL) in Figure 3.5. In this way, if a tuple is updated through one register, aliased pointers (labels) in other registers or in the heap will still be well-typed (although they may be treated as still being uninitialized).

$$\boxed{\vdash \tau \quad \vdash \Psi \quad \vdash \Gamma \quad \vdash \tau_1 \leq \tau_2 \quad \vdash \Gamma_1 \subseteq \Gamma_2}$$

$$\frac{\mathsf{FTV}(\tau) = \emptyset}{\vdash \tau}\ (\textsc{type}) \qquad \frac{\vdash \tau_i \quad (1 \leq i \leq n)}{\vdash \{l_0 : \tau_0, \ldots, l_n : \tau_n\}}\ (\textsc{htype})$$

$$\frac{\vdash \tau_i \quad (0 \leq i \leq n)}{\vdash \{r_0 \mapsto v_0, \ldots, r_n \mapsto v_n\}}\ (\textsc{rftype})$$

$$\frac{\vdash \tau}{\vdash \tau \leq \tau}\ (\textsc{reflex}) \qquad \frac{\vdash \tau_1 \leq \tau_2 \quad \vdash \tau_2 \leq \tau_3}{\vdash \tau_1 \leq \tau_3}\ (\textsc{trans})$$

$$\frac{\vdash \tau_i \quad (1 \leq i \leq n)}{\vdash \langle \tau_1{}^{\varphi_1}, \ldots, \tau_{i-1}{}^{\varphi_{i-1}}, \tau_1{}^{i}, \tau_{i+1}{}^{\varphi_{i+1}}, \ldots, \tau_n{}^{\varphi_n} \rangle \leq \langle \tau_1{}^{\varphi_1}, \ldots, \tau_{i-1}{}^{\varphi_{i-1}}, \tau_0{}^{i}, \tau_{i+1}{}^{\varphi_{i+1}}, \ldots, \tau_n{}^{\varphi_n} \rangle}\ (\text{0-1})$$

$$\frac{\vdash \tau_i \quad (0 \leq i \leq m) \quad (m \geq n)}{\vdash \{r_0 \mapsto v_0, \ldots, r_m \mapsto v_m\} \subseteq \{r_0 \mapsto v_0, \ldots, r_n \mapsto v_n\}}\ (\textsc{weaken})$$

$$\boxed{\Psi \vdash h : \tau\ \mathsf{hval} \quad \Psi \vdash v : \tau \quad \Psi \vdash l : \rho \quad \Psi \vdash v : \tau^{\varphi}}$$

$$\frac{\Psi \vdash v_i : \tau_i{}^{\varphi_i} \quad (1 \leq i \leq n)}{\Psi \vdash \langle v_1, \ldots, v_n \rangle : \langle \tau_1{}^{\varphi_1}, \ldots, \tau_n{}^{\varphi_n} \rangle\ \mathsf{hval}}\ (\textsc{tuple}) \qquad \frac{\vdash \Gamma \quad \Psi; \Gamma \vdash I}{\Psi \vdash \mathsf{code}\ [\Gamma].I : \forall[\Gamma]\ \mathsf{hval}}\ (\textsc{code})$$

$$\frac{}{\Psi \vdash i : \mathsf{int}}\ (\textsc{int}) \qquad \frac{\Psi \vdash v : \tau[\mu\alpha.\tau/\alpha]}{\Psi \vdash \mathsf{fold}\ v\ \mathsf{as}\ \mu\alpha.\tau : \mu\alpha.\tau}\ (\textsc{fold}) \qquad \frac{\vdash \Psi(l) \leq \tau}{\Psi \vdash l : \tau}\ (\textsc{label})$$

$$\frac{l = |\Psi|}{\Psi \vdash l : \mathsf{fresh}}\ (\textsc{fresh}) \qquad \frac{l = |\Psi| - \mathbf{1} \quad \Psi \vdash l : \langle \tau_1{}^{\varphi_1}, \ldots, \tau_n{}^{\varphi_n} \rangle}{\Psi \vdash l : \mathsf{used}(n)}\ (\textsc{used})$$

$$\frac{\Psi \vdash v : \tau}{\Psi \vdash v : \tau^{\varphi}}\ (\textsc{init}) \qquad \frac{\vdash \tau}{\Psi \vdash ?\tau : \tau^{0}}\ (\textsc{uninit})$$

Figure 3.5: Well-formedness of FTAL types, heap and word values.

$\boxed{\Psi; \Gamma \vdash I}$

$$\frac{\Gamma(r_s) = \mathsf{int} \quad \Gamma(r_t) = \mathsf{int} \quad \Psi; \Gamma\{r_d : \mathsf{int}\} \vdash I}{\Psi; \Gamma \vdash \mathsf{add}\ r_d, r_s, r_t; I}\ (\textsc{add})$$

$$\frac{\Gamma(r_s) = \mathsf{int} \quad \Psi; \Gamma\{r_d : \mathsf{int}\} \vdash I}{\Psi; \Gamma \vdash \mathsf{addi}\ r_d, r_s, i; I}\ (\textsc{addi})$$

$$\frac{\vdash \tau_i \quad \Psi; \Gamma\{r_d : \langle {\tau_1}^0, \ldots, {\tau_n}^0 \rangle\}\{\mathsf{r15} : \mathsf{used}(n)\} \vdash I}{\Psi; \Gamma\{\mathsf{r15} : \mathsf{fresh}\} \vdash \mathsf{alloc}\ r_d[\tau_1, \ldots, \tau_n]; I}\ (\textsc{alloc})$$

$$\frac{\Psi; \Gamma\{\mathsf{r15} : \mathsf{fresh}\} \vdash I}{\Psi; \Gamma\{\mathsf{r15} : \mathsf{used}(n)\} \vdash \mathsf{bump}\ n; I}\ (\textsc{bump})$$

$$\frac{\Gamma(r_s) = \mathsf{int} \quad \Gamma(r_t) = \mathsf{int} \quad \Psi(l) = \forall[\Gamma'] \quad \vdash \Gamma \subseteq \Gamma' \quad \Psi; \Gamma \vdash I}{\Psi; \Gamma \vdash \mathsf{bgt}\ r_s, r_t, l; I}\ (\textsc{bgt})$$

$$\frac{\Psi; \Gamma\{r_d : \Gamma(r_s)\} \vdash I}{\Psi; \Gamma \vdash \mathsf{mov}\ r_d, r_s; I}\ (\textsc{mov}) \qquad \frac{\Psi; \Gamma\{r_d : \mathsf{int}\} \vdash I}{\Psi; \Gamma \vdash \mathsf{movi}\ r_d, i; I}\ (\textsc{movi})$$

$$\frac{\Psi; \Gamma\{r_d : \tau\} \vdash I \quad \vdash \Psi(l) \leq \tau}{\Psi; \Gamma \vdash \mathsf{movl}\ r_d, l; I}\ (\textsc{movl})$$

$$\frac{\begin{array}{c}\Gamma(r_s) = \langle {\tau_0}^{\varphi_0}, \ldots, {\tau_{i-1}}^{\varphi_{i-1}}, {\tau_1}^{i}, {\tau_{i+1}}^{\varphi_{i+1}}, \ldots, {\tau_{n-1}}^{\varphi_{n-1}} \rangle \\ \Psi; \Gamma\{r_d : \tau_i\} \vdash I \quad (0 \leq i < n)\end{array}}{\Psi; \Gamma \vdash \mathsf{ld}\ r_d, r_s(i); I}\ (\textsc{ld})$$

$$\frac{\begin{array}{c}\Gamma(r_s) = \tau_i \quad \Gamma(r_d) = \langle {\tau_0}^{\varphi_0}, \ldots, {\tau_{n-1}}^{\varphi_{n-1}} \rangle \\ \Psi; \Gamma\{r_d : \langle {\tau_0}^{\varphi_0}, \ldots, {\tau_{i-1}}^{\varphi_{i-1}}, {\tau_1}^{i}, {\tau_{i+1}}^{\varphi_{i+1}}, \ldots, {\tau_{n-1}}^{\varphi_{n-1}} \rangle\} \vdash I \qquad (0 \leq i < n)\end{array}}{\Psi; \Gamma \vdash \mathsf{st}\ r_d(i), r_s; I}\ (\textsc{st})$$

$$\frac{\Gamma(r_s) = \tau[\mu\alpha.\tau/\alpha] \quad \Psi; \Gamma\{r_d : \mu\alpha.\tau\} \vdash I}{\Psi; \Gamma \vdash \mathsf{fold}\ r_d[r_s], \mu\alpha.\tau; I}\ (\textsc{fold-i})$$

$$\frac{\Gamma(r_s) = \mu\alpha.\tau \quad \Psi; \Gamma\{r_d : \tau[\mu\alpha.\tau/\alpha]\} \vdash I}{\Psi; \Gamma \vdash \mathsf{unfold}\ r_d, r_s; I}\ (\textsc{unfold})$$

$$\frac{\Psi(l) = \forall[\Gamma'] \quad \vdash \Gamma \subseteq \Gamma'}{\Psi; \Gamma \vdash \mathsf{jd}\ l}\ (\textsc{jd}) \qquad \frac{\Gamma(r) = \forall[\Gamma'] \quad \vdash \Gamma \subseteq \Gamma'}{\Psi; \Gamma \vdash \mathsf{jmp}\ r}\ (\textsc{jmp})$$

Figure 3.6: Well-formedness of FTAL instruction sequences.

The special allocation register is typed using a new judgment of allocation status:

$$\frac{l = |\Psi|}{\Psi \vdash l : \mathsf{fresh}} \text{ (FRESH)}$$

$$\frac{l = |\Psi| - \mathbf{1} \quad \Psi \vdash l : \langle {\tau_1}^{\varphi_1}, \ldots, {\tau_n}^{\varphi_n} \rangle}{\Psi \vdash l : \mathsf{used}(n)} \text{ (USED)}$$

In the first typing rule, a label whose value is equivalent to the size of the heap type must necessarily be unallocated, *i.e.* fresh. When allocation takes place, then the allocation register temporarily points to the newly allocated memory, and thus will have allocation status $\mathsf{used}(n)$ where n is the length of the allocated tuple. The assignment of allocation status interacts with the two novel FTAL instructions, alloc and bump, as shown in their typing rules:

$$\frac{\vdash \tau_i \quad \Psi; \Gamma\{r_d : \langle {\tau_1}^0, \ldots, {\tau_n}^0 \rangle\}\{\mathsf{r15} : \mathsf{used}(n)\} \vdash I}{\Psi; \Gamma\{\mathsf{r15} : \mathsf{fresh}\} \vdash \mathsf{alloc}\ r_d[\tau_1, \ldots, \tau_n]; I} \text{ (ALLOC)}$$

$$\frac{\Psi; \Gamma\{\mathsf{r15} : \mathsf{fresh}\} \vdash I}{\Psi; \Gamma\{\mathsf{r15} : \mathsf{used}(n)\} \vdash \mathsf{bump}\ n; I} \text{ (BUMP)}$$

For an alloc instruction to be well-typed, the allocation register, r15, must be in the fresh status, since otherwise, as can be seen from the operational semantics, the previously allocated data will be overwritten. After the alloc instruction, the remainder of the instruction sequence is checked with the status of r15 changed to $\mathsf{used}(n)$. No further allocation can take place until a bump instruction is encountered, which resets the status to fresh, corresponding again to the update in the operational semantics.

Coq code: `ftal.v` contains the encoding of FTAL syntax and semantics.

3.1.4 Examples

In this section, I give a couple of examples of FTAL programs to demonstrate that such a language (eventually extended with polymorphism and existentials, of course) provides

features which make it suitable for compiling high-level languages such as Java, ML, or Safe C.

The first example is the calculation of a Fibonacci number in Figure 3.7. The C-like program at the top of the figure can be compiled to the FTAL code below it. The code segments `fib`, `fib_loop`, and `fib_return` form a function, written in continuation passing style (CPS), which calculates the Fibonacci number with index given in r1, and then passes control to the continuation function given in r14. The `main` block calls `fib` to calculate F_{10} and passes the address of the `halt` block as its continuation. `fib` initializes the loop variables and then jumps into the loop code segment `fib_loop`, which jumps to `fib_return` when the calculation is done. As the body of `main` appears twice in the figure (once in the initial program state), it has been factored out as I to save space.

The second example, in Figure 3.8, demonstrates how to use recursive types and memory allocation to handle classes and objects. Class `c` has no data fields and only one method `f`, which takes an object of class `c` and invokes its method `f`. In the main program, an object of class `c` is created and its method `f` is called with the object itself as argument. The program will end up in an infinite recursive call to `c.f`. In FTAL, an object of class `c` is represented as a recursive tuple type whose only element is a code block with an only argument of the object type `c`. The code block at label `c_f` uses the `unfold` and `ld` instructions to extract the argument object's own method `f`, and then jumps to it. The constructor for `c`, inlined in the `main` code block, uses the `alloc` and `bump` instructions to allocate heap space for a tuple, then initializes its method `f` with the label `c_f`, and folds the tuple into an object using the `fold` instruction. Similarly to `c_f`, the `main` code block then extracts method `f` from the newly created object and jumps to it.

3.1.5 Soundness

In order to produce the necessary FPCC proofs as described in Section 2.2.4, we must encode the complete semantics of FTAL in CiC along with its proof of soundness, which

```
int fib (n:int) {                    // "Safe C" code
  int a=1, b=1;
  for (int i=2; i++; i<=n)
    { int c = a + b; a = b; b = c; }
  return a;
}
int main () {
  return fib(10);
}
```

$\mathcal{P} =$ $(H,$ {}, $I)$ // FTAL code

$H =$
```
fib: code[{ r1:int, r14:∀[{r1:int}] }].
       mov r3, r1;
       movi r1, 1;
       movi r2, 1;
       movi r4, 2;
       jd fib_loop
     fib_loop: code[{ r1:int, r2:int, r3:int, r4:int,
                      r14:∀[{r1:int}] }].
       bgt r4, r3, fib_return;
       add r5, r1, r2;
       mov r1, r2;
       mov r2, r5;
       addi r4, r4, 1;
       jd fib_loop
     fib_return: code[{ r1:int, r14:∀[{r1:int}] }].
       jmp r14
     halt: code[{r1:int}].
       jd halt
     main: code[{}].
       I
```

$I =$
```
       movi r1, 10;
       movl r14, halt;
       jd fib
```

Figure 3.7: FTAL Example: Fibonacci Numbers

```
class c {                              // "Safe C++" code
  void f (c x) { x.f(x); }
}
void main () {
  c x = new c;
  x.f(x);
}
```

$\mathcal{P} = (H,\ \{\},\ I)$ // FTAL code

```
c = μα.<∀[{r1:α}]>
H = c_f: code[{r1:c}].
      unfold r2, r1;
      ld r2, r2(0);
      jmp r2
    main: code[{}].
      I

I =   alloc r1 [∀[{r1:c}]];
      bump 1;
      movl r2, c_f;
      st r1(0), r2;
      fold r1[c], r1;
      unfold r2, r1;
      ld r2, r2(0);
      jmp r2
```

Figure 3.8: FTAL Example: Mini-Object

will be used in defining and proving the FPCC propositions. The critical theorems for the soundness of FTAL are the usual progress and preservation lemmas:

Theorem 3.1 (FTAL Progress)

If $\vdash \mathcal{P}$, then there exists $\mathcal{P}'$ such that $\mathcal{P} \longmapsto \mathcal{P}'$.

Theorem 3.2 (FTAL Preservation)

If $\vdash \mathcal{P}$ and $\mathcal{P} \longmapsto \mathcal{P}'$, then $\vdash \mathcal{P}'$.

As usual, several intermediate lemmas are used to prove these two theorems, all of which can be formally encoded and proved in the Coq proof assistant. The most important of these lemmas are given below. Their encoding in Coq is described in Section 3.3. As the proofs of these lemmas and the two theorems above are extremely similar to those of the original TAL [52], I omit any discussion of them here. In this regard, my main contribution has been to mechanize proofs that had previously only been done on paper.

Lemma 3.3 (FTAL Register File Update)

1. If $\Psi \vdash R : \Gamma$ and $\Psi \vdash v : \tau$ then $\Psi \vdash R\{r \mapsto v\} : \Gamma\{r : \tau\}$.
2. If $\Psi \vdash R : \Gamma$ and $\Psi \vdash l : \rho$ then $\Psi \vdash R\{\mathsf{r15} \mapsto l\} : \Gamma\{\mathsf{r15} : \rho\}$.

Lemma 3.4 (FTAL Canonical Word Forms) If $\vdash H : \Psi$ and $\Psi \vdash v : \tau$ then:

1. if $\tau = \mathsf{int}$ then $v = i$;
2. if $\tau = \forall[\Gamma]$ then $v = l$ and $H(l) = \mathsf{code}\ [\Gamma].I$;
3. if $\tau = \langle {\tau_1}^{\varphi_1}, \ldots, {\tau_n}^{\varphi_n} \rangle$ then $v = l$;
4. if $\tau = \mu\alpha.\tau'$ then $v = \mathsf{fold}\, v'\, \mathsf{as}\, \tau$.

Lemma 3.5 (FTAL Canonical Register Word Forms) If $\Psi \vdash R : \Gamma$ and $\Gamma(r) = \tau$ then:

1. $R(r) = v$;
2. if $\tau = \mathsf{int}$ then $R(r) = i$;

3. if $\tau = \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle$ then $R(r) = l$.

Lemma 3.6 (FTAL Canonical Heap Forms) If $\Psi \vdash h : \tau$ hval then:

1. if $\tau = \forall[\Gamma]$ then $h = \mathsf{code}\ [\Gamma].I$ and $\Psi; \Gamma \vdash I$;
2. if $\tau = \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle$ then $h = \langle v_1, \ldots, v_n \rangle$ and $\Psi \vdash v_i : \tau_i^{\varphi_i}$

Lemma 3.7 (FTAL Register File Weakening) If $\vdash \Gamma_1 \subseteq \Gamma_2$ and $\Psi \vdash R : \Gamma_1$ then $\Psi \vdash R : \Gamma_2$.

Lemma 3.8 (FTAL Heap Extension) If $\vdash H : \Psi$, $l = |H|$ (thus, $l \notin Dom(H)$), and $\vdash \tau$, then:

1. $\vdash \Psi\{l : \tau\}$;
2. if $\Psi \vdash v : \tau'$ then $\Psi\{l : \tau\} \vdash v : \tau'$;
3. if $\Psi \vdash v : \tau^{\varphi}$ then $\Psi\{l : \tau\} \vdash v : \tau^{\varphi}$;
4. if $\Psi; \Gamma \vdash I$ then $\Psi\{l : \tau\}; \Gamma \vdash I$;
5. if $\Psi \vdash R : \Gamma\{\mathsf{r15} : \mathsf{fresh}\}$ then $\Psi\{l : \tau\} \vdash R : \Gamma\{\mathsf{r15} : \mathsf{used}(n)\}$;
6. if $\Psi \vdash h : \tau'$ hval then $\Psi\{l : \tau\} \vdash h : \tau'$ hval;
7. if $\Psi\{l : \tau\} \vdash h : \tau$ hval then $\vdash H\{l \mapsto h\} : \Psi\{l : \tau\}$.

Lemma 3.9 (FTAL Heap Update) If $\vdash H : \Psi$ and $\vdash \tau \leq \Psi(l)$ then:

1. $\vdash \Psi\{l : \tau\}$;
2. if $\Psi \vdash v : \tau'$ then $\Psi\{l : \tau\} \vdash v : \tau'$;
3. if $\Psi \vdash v : \tau^{\varphi}$ then $\Psi\{l : \tau\} \vdash v : \tau^{\varphi}$;
4. if $\Psi; \Gamma \vdash I$ then $\Psi\{l : \tau\}; \Gamma \vdash I$;
5. if $\Psi \vdash R : \Gamma$ then $\Psi\{l : \tau\} \vdash R : \Gamma$;
6. if $\Psi \vdash h : \tau'$ hval then $\Psi\{l : \tau\} \vdash h : \tau'$ hval;
7. if $\Psi\{l : \tau\} \vdash h : \tau$ hval then $\vdash H\{l \mapsto h\} : \Psi\{l : \tau\}$.

Now that we have an assembly language with a sound type system, we are ready to generate proof-carrying code from a well-typed FTAL program.

Coq code: `lemmas_ftal.v` contains these lemmas and their complete proofs.

3.1.6 Designing TAL for FPCC

I have designed a novel FTAL language for the presentation in this thesis which corresponds closely to the underlying machine defined in Section 2.1. As will become clear in the next section, every well-formed FTAL state can be mapped to a safe machine state, and this property is used to produce a safety proof for the machine state.

For safety policies which need to enforce complex constraints on every machine state or step, such a one-to-one mapping can be very convenient. In general, however, this strict correspondence is not necessary for the syntactic approach to work. For example, if we wished to retain "macro" instructions in the FTAL language, our definition of FPCC Preservation might be modified to

$$\Pi \mathbb{S} : \mathit{State}.\, \mathsf{Inv}\,(\mathbb{S}) \rightarrow \exists n : \mathsf{Nat}.\, \mathsf{Inv}\,(\mathsf{Step}^{(n+1)}\,(\mathbb{S}))$$

stating that starting from a state satisfying the global invariant, the machine will eventually (after one or more steps) reach another state satisfying the invariant. Better yet, I will show in Chapter 4 a more general way to handle macro instructions, and even those which do not have any runtime effect. When introducing polymorphism or existentials into the FTAL language, there will be certain FTAL operations (*e.g.* type application) which do not correspond to any run-time machine instructions at all. In this case, the FTAL operation would correspond to a "cast" in the FPCC proof for the machine state.

Another reason why naïvely using existing typed assembly languages will not necessarily help in producing FPCC is that the type system must be designed to enforce appropriate invariants. There are requirements in the typing rules of FTAL which are not critical for FTAL soundness but are necessary when translating FTAL to FPCC as described in the

next section. An example of this is the requirement in the (REG) rule (Figure 3.4) that all labels in registers be within the domain of the heap (including those registers that are not specified in the type of the register file and hence not accessible by well-formed code anyway). This condition is crucial in proving the properties discussed in Section 3.2.3.

3.2 Translating FTAL to FPCC

As outlined in Section 2.2.3, an FPCC package provides an initial state, $\mathbb{S}_0$, and a proof that the state satisfies the safety policy. In the next few subsections, I show how to translate an FTAL program into a machine state and how to use the FTAL type system to generate proofs of the FPCC Preservation and Progress propositions, which imply safety.

3.2.1 From FTAL to Machine State

FTAL programs are compiled to machine code by (1) defining a layout for the memory which maps heap values of the program to memory addresses, (2) translating FTAL instructions to machine instructions, and (3) choosing the appropriate program counter and register values. The layout must ensure that there are no overlaps between the images of tuples and code sequences in the memory. Our choice of the FTAL instruction set allows us to translate every FTAL instruction into one machine instruction word.

I will express the correspondence between an FTAL program and a machine state by a family of translation relations upon the various syntactic categories. The forms of these relations are:

Relation	Correspondence
$(H, R, I) \Rightarrow (\mathbb{M}, \mathbb{R}, pc)$	FTAL program to machine state
$L \vdash H \Rightarrow \mathbb{M}$	FTAL heap to memory
$L \vdash R \Rightarrow \mathbb{R}$	register files
$L \vdash I \Rightarrow_{\mathsf{s}} \mathbb{M}_{i..j}$	sequence of instructions to memory layout
$L \vdash c \Rightarrow_{\mathsf{i}} w$	instruction translation
$L \vdash h \Rightarrow_{\mathsf{h}} \mathbb{M}_{i..j}$	heap value to memory layout
$L \vdash v \Rightarrow_{\mathsf{w}} w$	word value to machine word

Recall that the machine memory is modeled as a function, *Word* $\rightarrow$ *Word*, so $\mathbb{M}(w)$ denotes the memory word at address w. The judgments $L \vdash I \Rightarrow_{\mathsf{s}} \mathbb{M}_{i..j}$ and $L \vdash h \Rightarrow_{\mathsf{h}} \mathbb{M}_{i..j}$ state that a sequence of instructions and a heap value (either a tuple or a code block), respectively, translate to a series of consecutive words in memory $\mathbb{M}$ from address i to address j.

An important step in the translation is flattening the FTAL heap into the machine memory. To achieve this, I define a *Layout* function of type *Heap* $\rightarrow$ *Label* $\rightarrow$ *Word* which, given an FTAL heap, returns a mapping from labels to memory addresses. (In the relations above, L is this *Layout* function applied to the heap.) For our current purpose, we define

$$\begin{aligned}
\mathit{Layout}\,(\{\})\,(l') &= 0 \\
\mathit{Layout}\,(H\{l \mapsto h\})\,(l') &= \begin{cases} w + \mathit{size}\,(h), & \text{if } l < l' \\ w, & \text{otherwise,} \end{cases} \\
&\quad \text{where } w = \mathit{Layout}\,(H)\,(l')
\end{aligned}$$

where *size* (h) is the size of the heap value h (n for an n-tuple, and the length of the instruction sequence for a code block). This *Layout* function maps labels to addresses starting at 0 and forces the translation $\Rightarrow$ to lay out FTAL heap values compactly, consecutively, and with no overlapping (due to the implicit type system constraint that the labels in a well-

formed heap appear in descending order). Additionally, the first unused label (whose value equals the size of the heap) is mapped to the first unused address. These properties of the *Layout* function are useful later on in proving FPCC Preservation and Progress.

The translation relations are defined by a set of inference rules, given in Figure 3.9. The rules are straightforward and operate purely on the syntax of FTAL programs. Note that FTAL type annotations are discarded in the translation (for example, in the fold instruction), and label word values are mapped to memory words using the layout function. Each FTAL heap value corresponds to a sequence of words in memory. A heap translates to a memory if every heap value in the heap translates to the appropriate sequence of memory words. Registers translate directly between FTAL and the machine ($\hat{r}$ is defined in Figure 3.1 and r in Figure 2.1). An FTAL program corresponds to a machine state if the translation relation holds on the heap and register file, and if the current instruction sequence is at some location in the memory. Since in a well-typed FTAL program the current instruction sequence must also be present in the heap, we can always translate it to a known program counter. Notice that the FTAL alloc and bump instructions correspond to machine move and addition instructions, respectively, using the register reserved for allocation, r15. (It is for this purpose that bump has an i argument.)

The translation relation as presented in Figure 3.9 is not deterministic with respect to the unused and uninitialized parts of the memory and to the positioning of the program counter. However, it is straightforward on the basis of its definition to develop a deterministic function (*i.e.* a compiler) which translates an FTAL program into a machine state for which the translation relation described above holds. In the next section, I will show how this initial translation is used to provide the Initial Condition FPCC proof.

3.2.2 The Global Invariant

As discussed in Section 2.2.4, in addition to translating the FTAL program to an initial machine state $\mathbb{S}_0$, we must define the invariant Inv, which holds during the execution of a

WORD VALUES

$$L \vdash l \Rightarrow_{\mathsf{w}} L(l) \qquad L \vdash i \Rightarrow_{\mathsf{w}} i$$

$$\frac{\textit{for any } w}{L \vdash ?\tau \Rightarrow_{\mathsf{w}} w} \qquad \frac{L \vdash v \Rightarrow_{\mathsf{w}} w}{L \vdash \mathsf{fold}\, v\, \mathsf{as}\, \tau \Rightarrow_{\mathsf{w}} w}$$

INSTRUCTIONS

$$\begin{array}{ll}
L \vdash \mathsf{add}\ r_d, r_s, r_t & \Rightarrow_{\mathsf{i}} \mathtt{add}\ r_d, r_s, r_t \\
L \vdash \mathsf{addi}\ r_d, r_s, i & \Rightarrow_{\mathsf{i}} \mathtt{addi}\ r_d, r_s, i \\
L \vdash \mathsf{alloc}\ r_d[\overrightarrow{\tau}] & \Rightarrow_{\mathsf{i}} \mathtt{addi}\ r_d, \mathsf{r15}, 0 \\
L \vdash \mathsf{bump}\ i & \Rightarrow_{\mathsf{i}} \mathtt{addi}\ \mathsf{r15}, \mathsf{r15}, i \\
L \vdash \mathsf{fold}\ r_d[r_s], \tau & \Rightarrow_{\mathsf{i}} \mathtt{addi}\ r_d, r_s, 0 \\
L \vdash \mathsf{unfold}\ r_d, r_s & \Rightarrow_{\mathsf{i}} \mathtt{addi}\ r_d, r_s, 0 \\
L \vdash \mathsf{ld}\ r_d, r_s(i) & \Rightarrow_{\mathsf{i}} \mathtt{ld}\ r_d, r_s(i) \\
L \vdash \mathsf{st}\ r_d(i), r_s & \Rightarrow_{\mathsf{i}} \mathtt{st}\ r_d(i), r_s \\
L \vdash \mathsf{mov}\ r_d, r_s & \Rightarrow_{\mathsf{i}} \mathtt{addi}\ r_d, r_s, 0 \\
L \vdash \mathsf{movi}\ r_d, i & \Rightarrow_{\mathsf{i}} \mathtt{movi}\ r_d, i \\
L \vdash \mathsf{movl}\ r_d, l' & \Rightarrow_{\mathsf{i}} \mathtt{movi}\ r_d, L(l') \\
L \vdash \mathsf{bgt}\ r_s, r_t, l & \Rightarrow_{\mathsf{i}} \mathtt{bgt}\ r_s, r_t, L(l)
\end{array}$$

INSTRUCTION SEQUENCES

$$\frac{L \vdash c \Rightarrow_{\mathsf{i}} \mathsf{Dc}(\mathbb{M}(i)) \quad L \vdash I \Rightarrow_{\mathsf{s}} \mathbb{M}_{(i+1)..j}}{L \vdash c; I \Rightarrow_{\mathsf{s}} \mathbb{M}_{i..j}}$$

$$\frac{\mathsf{Dc}(\mathbb{M}(i)) = \mathtt{jd}\ (L(l'))}{L \vdash \mathsf{jd}\ l' \Rightarrow_{\mathsf{s}} \mathbb{M}_{i..i}} \qquad \frac{\mathsf{Dc}(\mathbb{M}(i)) = \mathtt{jmp}\ r}{L \vdash \mathsf{jmp}\ r \Rightarrow_{\mathsf{s}} \mathbb{M}_{i..i}}$$

HEAP VALUES

$$\frac{L \vdash v_i \Rightarrow_{\mathsf{w}} \mathbb{M}(j+i) \quad \textit{for } 0 \le i \le n}{L \vdash \langle v_o, \ldots, v_n \rangle \Rightarrow_{\mathsf{h}} \mathbb{M}_{j..(j+n)}}$$

$$\frac{L \vdash I \Rightarrow_{\mathsf{s}} \mathbb{M}_{i..j}}{L \vdash \mathsf{code}\ [\Gamma].I \Rightarrow_{\mathsf{h}} \mathbb{M}_{i..j}}$$

HEAP, REGISTER FILE, PROGRAM

$$\frac{L \vdash H(l) \Rightarrow_{\mathsf{h}} \mathbb{M}_{L(l)..L(l+1)-1} \quad \mathbf{0} \le l < |H|}{L \vdash H \Rightarrow \mathbb{M}}$$

$$\frac{L \vdash R(\hat{r}_i) \Rightarrow_{\mathsf{w}} \mathbb{R}(r_i) \quad 0 \le i \le 15}{L \vdash R \Rightarrow \mathbb{R}}$$

$$\frac{\begin{array}{ll}
\textit{Layout}(H) \vdash H \Rightarrow \mathbb{M} & \textit{Layout}(H) \vdash I \Rightarrow_{\mathsf{s}} \mathbb{M}_{pc..pc+|I|-1}, \\
\textit{Layout}(H) \vdash R \Rightarrow \mathbb{R} & \textit{where} \quad \exists l \in Dom(H).(H(l) = \mathsf{code}\ [\Gamma].I', I \subseteq_{\textit{tail}} I', \textit{ and} \\
 & \qquad\qquad pc = \textit{Layout}(H)(l) + |I'| - |I|)
\end{array}}{(H, R, I) \Rightarrow (\mathbb{M}, \mathbb{R}, pc)}$$

Figure 3.9: Relating FTAL programs to machine states.

machine program, and provide proofs of:

FPCC Initial Condition: $\mathsf{Inv}\,(\mathbb{S}_0)$

FPCC Preservation: $\Pi\mathbb{S} : \mathit{State}.\ \mathsf{Inv}\,(\mathbb{S}) \rightarrow \mathsf{Inv}\,(\mathsf{Step}\,(\mathbb{S}))$

FPCC Progress: $\Pi\mathbb{S} : \mathit{State}.\ \mathsf{Inv}\,(\mathbb{S}) \rightarrow \mathsf{SP}\,(\mathbb{S})$

The invariant simply has to ensure that the machine state at each step corresponds to a well-typed FTAL program, which will allow us to use the formalized versions of the proofs of the progress and preservation lemmas for FTAL to generate formal proofs of the corresponding properties of the invariant. Since the definition of Inv requires us to state that an FTAL program is well-typed, it must be expressed not just in terms of FTAL programs, but of their typing derivations:

$$\mathsf{Inv}(\mathbb{S}) = \exists \mathcal{P} : \mathit{program}.\ ((\vdash \mathcal{P})\ \wedge\ (\mathcal{P} \Rightarrow \mathbb{S}))$$

Hence, the invariant holds on a state if there exists an FTAL program that is well-typed and translates to the state.

The proof of the initial condition can now be obtained directly in the process of translating an initial well-formed FTAL program to machine state as described in Section 3.2.1. It remains, therefore, to prove the two lemmas.

3.2.3 The Preservation and Progress Properties

Progress in our case is easy to prove: since the invariant states that there exists a well-typed FTAL program that translates to the current state, it is obvious by examination of the translation rules that such an FTAL program will never translate to a state in which the program counter points to an illegal instruction.

The remaining proof term, for Preservation, is thus the most involved of the generated FPCC proofs. It is obtained in the following way:

Given a program $\mathcal{P}$ and a typing derivation for $\vdash \mathcal{P}$, we know by FTAL progress that there exists a program $\mathcal{P}'$ such that $\mathcal{P} \longmapsto \mathcal{P}'$. Furthermore, by FTAL preservation,

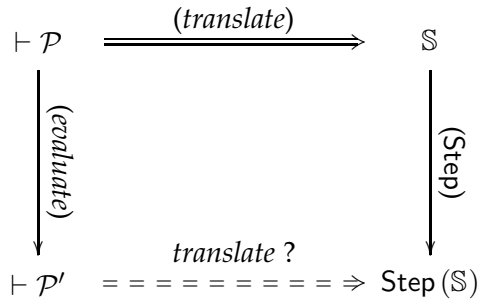


Figure 3.10: Relationship between FTAL evaluation and machine semantics.

we know that $\vdash \mathcal{P}'$. Now, the premise of our FPCC Preservation theorem provides us with a machine state $\mathbb{S}$ such that $\mathcal{P} \Rightarrow \mathbb{S}$, and we need to show that there exists another well-typed program that translates to $\mathsf{Step}(\mathbb{S})$. The semantics of FTAL has been set up so that this well-typed program is exactly $\mathcal{P}'$. It remains now for us to prove that indeed $\mathcal{P}' \Rightarrow \mathsf{Step}(\mathbb{S})$, as diagrammed in Figure 3.10.

Essentially, we need to show that the FTAL evaluation relation corresponds to the machine's step function. This is proved by induction on the typing derivation of $\vdash \mathcal{P}$. For each possible case, we use inversion[1] on the structure of $\mathcal{P}$, the FTAL evaluation relation, the translation relation, and the machine Step function to gain the necessary information about the structure of $\mathcal{P}'$, $\mathbb{S}$, and $\mathsf{Step}(\mathbb{S})$. Many of the cases of this proof are fairly straightforward.

Let us briefly consider one of the interesting cases of the Preservation proof, which is when the current instruction is alloc. Corresponding to the diagram in Figure 3.10, we

[1] "Inversion" is simply a process of backwards reasoning– given a premise, one infers what judgment rule must have been used to prove it and then adds the assumptions of that rule to one's list of available facts. For example, if we know an FTAL program is well-typed then, by inversion, the only possible rule that would allow one to prove such a judgment is (PROG) on page 35; hence, we can infer, for instance, that there must also exist a proof that the heap component of the program is well-typed.

have the following setup:

$$\mathcal{P} = (H, R, \mathsf{alloc}\ r_d[\tau_1, \ldots, \tau_n]; I)$$
$$\mathcal{P}' = (H', R', I)$$
$$\mathbb{S} = (\mathbb{M}, \mathbb{R}, pc)$$
$$\mathsf{Step}\,(\mathbb{S}) = (\mathbb{M}, \mathbb{R}', (pc + 1))$$

where H', R', and $\mathbb{R}'$ can be determined by the operational semantics of FTAL and the definition of the Step function (Figure 2.2).

We now need to prove that $\mathcal{P}'$ is related to $\mathsf{Step}\,(\mathbb{S})$ by the translation. First, we know by the properties of the layout function that applying it to an extended heap maintains the mapping of all the existing labels in the old heap. Now, the FTAL heap is updated after evaluation but the memory stays the same after the step. However, since the update to the heap is only with uninitialized values, which can be translated to any word, the translation will still hold on the unchanged memory. Thus, we can show that the updated heap translates to the unaltered memory. Then, relating the two updated register files is not difficult, nor is showing that the residual instruction sequence corresponds to the next program counter value. Well-formedness of $\mathcal{P}$ (*i.e.* $\vdash \mathcal{P}$) is used in various steps of this proof, for instance, to reason that any labels in the registers are within the domain of the heap, hence the layout function on the updated heap, H', preserves the mappings of existing labels.

This completes the translation, or compilation, of a well-typed FTAL program to an FPCC code package. The FTAL program can be shown to correspond to an initial machine state and that state can be shown safe using the proofs of Preservation and Progress developed here.

Coq code: `translate_ftal.v` defines the translation of Section 3.2.1 as well as contains the complete FPCC proofs described in this section.

3.3 Implementation in Coq

An implementation of the syntactic approach presented in this thesis consists of an FTAL compiler which generates FPCC packages. An FPCC package consists of two parts: the initial machine state and the proof of safety. The proof of safety can be further divided into two pieces: one is the proof of the Preservation and Progress theorems and the other is the proof that the initial machine state satisfies the Initial Condition property. Note that the proofs of Preservation and Progress (which are constructed semi-automatically) do not change for any machine state which has been generated by compiling an FTAL program. Thus, these properties need only be proven once and can then be reused for all FPCC packages produced by this compiler.

In the following sections, I first describe our Coq representation of the machine and the encoding of FTAL syntax and semantics and soundness theorems. Next I discuss implementation of the formal proofs of FPCC Preservation and Progress, which were done interactively using the Coq proof assistant. Finally, I describe a compiler which parses an FTAL program, performs type-checking, and automatically produces the Coq term representing the typing derivation. This typing derivation is then used to construct the proof of the Initial Condition property.

Coq is a proof assistant tool for the calculus of inductive constructions (Section 2.2). It provides an interactive interface for constructing formal proofs in the logic. The Coq syntax[2] for λ-abstraction, $\lambda X : A.\, B$, is `[X:A]B`. The syntax for dependent products, $\Pi X : A.\, B$, is `(X:A)B` and Coq allows for the normal arrow abbreviation of this when the bound variable does not occur in the body, *e.g.* `A->B`. Coq syntax for inductive definitions is that described in Section 2.2. Coq uses the sort `Prop` for logical propositions and the sort `Set` for the type of data type specifications (booleans, natural numbers, lists, programs, *etc.*).

[2]This syntax is for Coq version 7, using which these proofs were built.

3.3.1 Encoding Machine Semantics

The Coq encoding of the machine to which FTAL programs are translated is very similar to the presentation in Section 2.1. For example, having defined the registers as an inductive set with 16 constructors, I then define the memory and register file as being functions and the state as a triple of memory, register file, and program counter:

```
Definition Word := nat.
Inductive _Reg : Set := _r0 : _Reg | _r1 : _Reg | ...
Definition Mem  := Word -> Word.
Definition _RegFile := _Reg -> Word.
Definition State := (Mem * (_RegFile * Word)).
```

The instruction set is then defined as an inductive definition with appropriate constructors:

```
Inductive _Instr : Set
 := _add  : _Reg -> _Reg -> _Reg -> _Instr
 |  _addi : _Reg -> _Reg -> Word -> _Instr
 |  _movi : _Reg -> Word -> _Instr
 |  _bgt  : _Reg -> _Reg -> Word -> _Instr
 |  _jd   : Word -> _Instr
 |  _jmp  : _Reg -> _Instr
 |  _ld   : _Reg -> _Reg -> Word -> _Instr
 |  _st   : _Reg -> Word -> _Reg -> _Instr
 |  _ill  : _Instr.
```

I next decide on how to encode the instructions above as natural numbers and write a Coq function which uses the appropriate arithmetic operations to decode a natural number into an `_Instr`:

1ex

```
Definition Dc : Word -> _Instr := ...
```

We are now ready to encode the semantics of the machine as given in Section 2.1. For updating the register file and memory, I define auxiliary functions, as in the code below (`beq_reg` compares equality of two register names and returns a boolean):

```
Definition updateregfile
 : _RegFile -> _Reg -> Word -> _RegFile
```

```
 := [R:_RegFile; rd:_Reg; v:Word]
    ([r:_Reg] if (beq_reg r rd) then v else (R r)).

Definition Step : State -> State
 := [St:State] Cases St of (M, (R, pc)) =>
    Cases (Dc (M pc)) of
        (_add rd rs rs')
           => (M, ((updateregfile R rd
                                  (plus (R rs) (R rs'))),
                (S pc)))
      | (_jd l)
           => (M, (R, l))
      | ...
      | _ill => St
    end
  end.
```

Finally, we can state the safety policy we wish to enforce and define what a safe machine state is. The `MultiStep` function simply applies the `Step` function to the given state `n` times:

```
Definition SP [S:State]
 := (let (M,T')=S in (let (R,PC)=T' in  ~(Dc (M PC))=_ill)).

Definition Safe [S:State]
 := (n:nat)(SP (MultiStep n S)).
```

3.3.2 Encoding FTAL Syntax

Encoding the FTAL language is a more involved process. I start by defining each syntactic category as an inductive type. For example, the FTAL types are encoded as follows:

```
Definition initflag := bool.

Inductive Omega : Set
 := intty : Omega
 | codety : (Map Reg Omega) -> APTy -> Omega
 | tupty  : (list Omega) -> (list initflag) -> Omega
 | recty  : (OmegaL (S O)) -> Omega.
```

The `list` in the tuple type constructor is the usual definition of a list, found in the Coq library. Hence, the tuple type constructor takes as arguments a list of types and a list of initialization flags, encoded as booleans. (Alternatively, I could have used a list of pairs

but in practice this encoding seemed easier to work with.) `Map` is defined as a list of pairs where the first element of each pair is used as a key for lookup and update operations. The type of a register file (used by `codety`) is a map from registers (definition presented below) to types. I also define a "well-formed `Map`", used later, as being a list of pairs in which the first element of every pair in the list is distinct from all others.

A well-formed type in the FTAL language will never have free type variables, but variables may appear in a recursive type. Hence, I represent the type under the recursive type constructor by a "lifted" version of `Omega` which uses deBruijn indices to represent variables. The parameter of the `OmegaL` type below tracks the number of free type variables in the term to ensure the correctness of our substitution and unfolding functions for recursive types:[3]

```
Inductive OmegaL : nat -> Set
 := inttyL  : (OmegaL 0)
 |  codetyL : (i:nat) (Map Reg (OmegaL i)) ->
                      APTy -> (OmegaL i)
 |  tuptyL  : (i:nat) (list (OmegaL i)) ->
                      (list initflag) -> (OmegaL i)
 |  rectyL  : (i:nat) (OmegaL (S i)) -> (OmegaL i)
 |  varL    : (i:nat) (OmegaL (S i))
 |  liftL   : (i:nat) (OmegaL i) -> (OmegaL (S i)).
```

FTAL registers are defined as in the machine above. Unlike the presentation in previous sections, we carry the special allocation pointer separately from the rest of of the register file, hence there are only 15 registers defined for FTAL. The `r15` register, or `AP` below, is simply a label (which is defined to be a natural number). The special allocation pointer types are encoded as an inductive definition and the types of register files and heaps are maps from registers or labels, respectively, to `Omega` (the heap type also requires that the map be well-formed, as defined above):

[3]This encoding of variables is discussed in more detail in Section 6.2.

```
Inductive Reg : Set := r0 : Reg | r1 : Reg | ...
Definition label := nat.
Definition AP := label. (* alloc. ptr. (r15) *)
Inductive APTy : Set
 := fresh : APTy
 |  used  : nat -> APTy.
Definition RegFileTy := (Map Reg Omega).
Definition HeapTy := (WFMap label Omega).
```

The remainder of the definitions for FTAL syntax are fairly intuitive and match closely the presentation in Figure 3.1, except that r15 and its type are carried separately as AP and APTy:

```
Inductive Instr : Set
 := add    : Reg -> Reg -> Reg -> Instr
 |  addi   : Reg -> Reg -> int -> Instr
 |  alloc  : Reg -> (list Omega) -> Instr
 |  bgt    : Reg -> Reg -> label -> Instr
 |  bump   : int -> Instr
 |  fold   : Reg -> Omega -> Reg -> Instr
 |  ld     : Reg -> Reg -> int -> Instr
 |  mov    : Reg -> Reg -> Instr
 |  movi   : Reg -> int -> Instr
 |  movl   : Reg -> label -> Instr
 |  st     : Reg -> int -> Reg -> Instr
 |  unfold : Reg -> Reg -> Instr.

Inductive InstrSeq : Set
 := iseq : Instr -> InstrSeq -> InstrSeq
 |  jd   : label -> InstrSeq
 |  jmp  : Reg -> InstrSeq.

Inductive WordVal : Set
 := wl      : label -> WordVal
 |  wi      : int -> WordVal
 |  wuninit : Omega -> WordVal
 |  wfold   : WordVal -> Omega -> WordVal.

Inductive HeapVal : Set
 := tuple : (list WordVal) -> HeapVal
 |  code  : RegFileTy -> APTy -> InstrSeq -> HeapVal.

Definition Heap    := (WFMap label HeapVal).
Definition RegFile := (Map Reg WordVal).

Definition Program := (Heap * (RegFile * (AP * InstrSeq))).
```

3.3.3 Encoding FTAL Semantics and Soundness

Each judgment form of the dynamic and static semantics can be viewed as a relation and is also encoded as an inductive definition. For every evaluation or typing rule, there is an associated constructor of the appropriate inductive definition. (This allows us to use Coq's inductive elimination constructs to perform inversion and induction on typing derivations.) I show the encoding of several evaluation rules in Figure 3.11.

The `reglookup` and `regupdext` are to be read as propositions stating that looking up the value of a given register in a register file (which is defined to be a `Map`) yields the given word value and that updating or extending the mapping of a register in a register file results in a new register file, respectively. For the heap (and similarly heap type, which are both defined as well-formed `Maps`) the `hextend` proposition requires that the label being added to the domain of the heap is not already being mapped in the heap. The `hupdate` proposition only holds true when the label is in fact present in the heap mapping. These propositions are defined inductively as relations on `Maps`.

The encodings of the main static judgments are given in Figures 3.12 and 3.13.

In order to formally prove the soundness of FTAL as encoded above, we proceed by first proving the same lemmas that are listed in Section 3.1.5. The statements of these lemmas in Coq, while slightly verbose, are essentially the same as those listed in that section. I generate the proofs of these lemmas interactively using Coq proof "tactics." The tactics of the proof assistant correspond to the steps that would be used in a hand proof, *e.g.* induction, inversion, rewriting, application of rules (constructors), *etc.* I present the statements of a few of these lemmas in Coq below (Register File Update, the second case of Canonical Word Forms, and several cases of the Heap Extension lemma):

```
Lemma regfile_update
 : (HT:HeapTy; R,R':RegFile; G,G':(Map Reg Omega))
   (rd:Reg; v:WordVal; t:Omega)
   (WFRegFile HT R G) ->
   (WFWordVal HT v t) ->
   (regupdext R rd v R') ->
   (regupdext G rd t G') ->
   (WFRegFile HT R' G').
```

```
Lemma can_word_forms_code
 : (H:Heap; HT:HeapTy; v:WordVal; G:RegFileTy; T:APTy)
   (WFHeap H HT) ->
   (WFWordVal HT v (codety G T)) ->
   (EX l | v=(wl l) /\ (EX I | (hlookup H l (code G T I)))).

Lemma heap_ext_2
 : (H,H':Heap; HT,HT':HeapTy; t:Omega; l:label)
   (v:WordVal; t':Omega)
   (WFHeap H HT) ->
   (hsize H l) ->
   (htextend HT l t HT') ->
   (WFWordVal HT  v t') ->
   (WFWordVal HT' v t').

Lemma heap_ext_4
 : (I:InstrSeq)
   (H,H':Heap; HT,HT':HeapTy; t:Omega; l:label)
   (R:RegFileTy; A:APTy)
   (WFHeap H HT) ->
   (hsize H l) ->
   (htextend HT l t HT') ->
   (WFInstrSeq HT  R A I) ->
   (WFInstrSeq HT' R A I).

Lemma heap_ext_7
 : (H,H':Heap; HT,HT':HeapTy; t:Omega; l:label)
   (h:HeapVal)
   (WFHeap H HT) ->
   (hsize H l) ->
   (htextend HT l t HT') ->
   (hextend H l h H') ->
   (WFHeapVal HT' h t) ->
   (WFHeap H' HT').
```

The main theorems for the soundness of FTAL, preservation and progress, follow from the various lemmas:

```
   Theorem ftal_preserv
 : (P,P':Program) (WFProgram P) -> (Eval P P') -> (WFProgram P').

Theorem ftal_progress
    : (P:Program) (WFProgram P) -> (EX P' | (Eval P P')).
```

We have now completely formalized the syntactic soundness proof of FTAL. In the next section, I discuss the encoding of the translation relations between FTAL and the machine, and how FTAL soundness is used to produce the proofs of the FPCC Preservation and Progress theorems.

```
Inductive Eval : Program -> Program -> Prop
:= ev_add
   : (H:Heap; R,R':RegFile; r15:AP; I':InstrSeq)
     (rd,rs,rs':Reg; rsval,rsval':int)
     (reglookup R rs (wi rsval)) ->
     (reglookup R rs' (wi rsval')) ->
     (regupdext R rd (wi (plus rsval rsval')) R') ->
     (Eval (H,(R,(r15,(iseq (add rd rs rs') I'))))
           (H,(R',(r15,I'))))
|  ev_alloc
   : (H,H':Heap; R,R':RegFile; r15:AP; I':InstrSeq)
     (rd:Reg; V:(list Omega))
     (regupdext R rd (wl r15) R') ->
     (hextend H r15 (tuple (makeUninitTup V)) H') ->
     (Eval (H, (R, (r15, (iseq (alloc rd V) I'))))
           (H', (R', (r15, I'))))
|  ev_bump
   : (H:Heap; R:RegFile; r15:AP; I':InstrSeq)
     (i:int; l:nat)
     (hsize H l) ->
     (Eval (H, (R, (r15, (iseq (bump i) I'))))
           (H, (R, (l, I'))))
|  ev_jd
   : (H:Heap; R:RegFile; r15:AP)
     (l:label; G:RegFileTy; T:APTy; I':InstrSeq)
     (hlookup H l (code G T I')) ->
     (Eval (H, (R, (r15, (jd l))))
           (H, (R, (r15, I'))))
|  ev_movl
   : (H:Heap; R,R':RegFile; r15:AP; I':InstrSeq)
     (rd:Reg; l:label)
     (regupdext R rd (wl l) R') ->
     (Eval (H, (R, (r15, (iseq (movl rd l) I'))))
           (H, (R',(r15, I'))))
|  ev_store
   : (H,H':Heap; R:RegFile; r15:AP; I':InstrSeq)
     (rd,rs:Reg; i:int; l:label;
      V,V':(list WordVal); w:WordVal)
     (reglookup R rd (wl l)) ->
     (reglookup R rs w) ->
     (hlookup H l (tuple V)) ->
     (updatetuple V i w V') ->
     (hupdate H l (tuple V') H') ->
     (Eval (H, (R, (r15, (iseq (st rd i rs) I'))))
           (H',(R, (r15, I'))))
|  ...
```

Figure 3.11: Coq encoding of FTAL dynamic semantics

```
Inductive RegFileSubtype             (* register file subtyping: G <= G' *)
  : RegFileTy -> RegFileTy -> Prop
  := weaken
     : (G,G':RegFileTy)
       ((r:Reg; t:Omega) (reglookup G' r t) -> (reglookup G r t)) ->
       (RegFileSubtype G G').

Inductive WFWordVal         (* well-formed word values: HT |- w : t wval *)
  : HeapTy -> WordVal -> Omega -> Prop
  := int_wval : (HT:HeapTy; i:int)(WFWordVal HT (wi i) intty)
  |  label_wval
     : (HT:HeapTy; l:label; t,t':Omega)
       (htlookup HT l t') ->
       (Subtype t' t) ->
       (WFWordVal HT (wl l) t)
  |  fold_word_wval
     : (HT:HeapTy; w:WordVal;  t:OmegaR; t':Omega)
       (RUnlift (RUnfold t))=t' ->
       (WFWordVal HT w t') ->
       (WFWordVal HT (wfold w (recty t)) (recty t)).

Inductive WFInstrSeq    (* well-formed instruction sequences: HT; G |- I *)
 : HeapTy -> RegFileTy -> APTy -> InstrSeq -> Prop
 := s_add
    : (HT:HeapTy; G,G':RegFileTy; T:APTy; I:InstrSeq)
      (rd,rs,rs':Reg)
      (reglookup G rs intty) ->
      (reglookup G rs' intty) ->
      (regupdext G rd intty G') ->
      (WFInstrSeq HT G' T I) ->
      (WFInstrSeq HT G T (iseq (add rd rs rs') I))
 |  s_alloc
    : (HT:HeapTy; G,G':RegFileTy; I:InstrSeq)
      (rd:Reg; n:nat; V:(list Omega))
      n=(length V) ->
      (regupdext G rd (tupty V (makeUninitTupty V)) G')->
      (WFInstrSeq HT G' (used n) I) ->
      (WFInstrSeq HT G fresh (iseq (alloc rd V) I))
  |  s_jd
     : (HT:HeapTy; G,G':RegFileTy; T:APTy)
       (l:label)
       (htlookup HT l (codety G' T)) ->
       (RegFileSubtype G G') ->
       (WFInstrSeq HT G T (jd l))
 |  s_st
    : (HT:HeapTy; G,G':RegFileTy; T:APTy; I:InstrSeq)
      (rd,rs:Reg; i:int;
      V,V':(list initflag); Ts:(list Omega); t:Omega)
      (reglookup G rd (tupty Ts V)) ->
      (reglookup G rs t) ->
      (ListNth ? Ts i t) ->
      (updatetupty V i V') ->
      (regupdate G rd (tupty Ts V') G') ->
      (WFInstrSeq HT G' T I) ->
      (WFInstrSeq HT G T (iseq (st rd i rs) I))
 |  ...
```

Figure 3.12: Coq encoding of FTAL static semantics: main definitions (1 of 2)

```
Inductive WFHeapVal         (* well-formed heap values: HT |- h : t hval *)
 : HeapTy -> HeapVal -> Omega -> Prop
 := tuple_wf
    : (HT:HeapTy; wl:(list WordVal); tl:(list Omega); il:(list initflag))
      (WFWordValinitList HT wl tl il) ->
      (WFHeapVal HT (tuple wl) (tupty tl il))
 |  code_wf
    : (HT:HeapTy; G:RegFileTy; I:InstrSeq; T:APTy)
      (WFInstrSeq HT G T I) ->
      (WFHeapVal HT (code G T I) (codety G T)).

Inductive WFHeap                                    (* well-formed heap *)
 : Heap -> HeapTy -> Prop
 := heap_wf
    : (H:Heap; HT:HeapTy)
      (EX s | (hsize H s) /\
              (htsize HT s) /\
              ((n:label; h:HeapVal) (hlookup H n h) -> (lt n s)) /\
              ((n:label; t:Omega) (htlookup HT n t) -> (lt n s)) /\
              ((n:label) (lt n s) -> (EX h | (hlookup H n h))) /\
              ((n:label) (lt n s) -> (EX t | (htlookup HT n t))) /\
              ((n:label; h:HeapVal; t:Omega)
                (hlookup H n h)->(htlookup HT n t)->(WFHeapVal HT h t)) /\
              (OrdHeap H)
      ) ->
      (WFHeap H HT).

Inductive WFRegFile                        (* well-formed register file *)
  : HeapTy -> RegFile -> RegFileTy -> Prop
  := regfile_wf
     : (HT:HeapTy; R:RegFile; G:RegFileTy)
       ((r:Reg; t:Omega)
          (reglookup G r t) ->
             (EX w | (reglookup R r w) /\ (WFWordVal HT w t))) ->
       ((r:Reg; v:WordVal; l:label; n:nat)
          (reglookup R r v) ->
          (stripWV v)=(wl l) ->
          (htsize HT n) ->
          (lt l n)) ->
       (WFRegFile HT R G).

Inductive WFProgram                              (* well-formed program *)
 : Program -> Prop
 := program_wf
    : (H:Heap; HT:HeapTy; R:RegFile; G:RegFileTy;
       l:AP; t:APTy; I:InstrSeq)
      (WFHeap H HT) ->
      (WFRegFile HT R G) ->
      (WFap HT l t) ->
      (WFInstrSeq HT G t I) ->
      (EX l | (EX G' | (EX T' | (EX I' | (EX n |
         (hlookup H l (code G' T' I')) /\
         (ISubDepth I I' n)))))) ->
      (WFProgram (H, (R, (l, I)))).
```

Figure 3.13: Coq encoding of FTAL static semantics: main definitions (2 of 2)

3.3.4 Encoding FPCC Preservation and Progress

The translation relations (not shown here) are represented as a set of inductive definitions which follow precisely the presentation in Figure 3.9, for example,

```
Inductive TrProgram
 : Program -> State -> Prop := ...
```

The global invariant for FPCC can be defined in terms of the translation between a well-formed FTAL program and the machine state:

```
Definition Inv [S:State]
 := (EX P:Program | (WFProgram P) /\ (TrProgram P S)).
```

Now we proceed to prove the FPCC Progress theorem:

```
Theorem Progress : (S:State) (Inv S) -> (SP S).
```

As mentioned in Section 3.2.3, the Progress theorem is straightforward. Using several Coq "Inversion" tactics, we determine that there exists a well-formed instruction sequence which translates to the program counter of the state. Then we perform case analysis on the well-formed instruction sequence judgment and show that in every possible case, the program counter of the state must be pointing to a non-`illegal` instruction.

Next is the FPCC Preservation theorem, which is more involved to prove but which follows the discussion in Section 3.2.3:

```
Theorem Preservation : (S:State) (Inv S) -> (Inv (Step S)).
```

With these two theorems, we can now prove that a machine state will be safe if the FPCC Initial Condition property is satisfied:

```
Theorem Safety : (S:State) (Inv S) -> (Safe S).
```

3.3.5 Generating the Initial Condition

In order to generate the Initial Condition, we would use a compiler that takes an FTAL program and compiles it to a machine state, producing the necessary proofs in the process.[4] The structure of this compiler is fairly straightforward: After parsing an FTAL

[4] A complete compiler has not actually been developed because FTAL is a very simplistic language and we have been working with an "ideal" machine anyway.

source file, type-checking is performed. The algorithm for type-checking follows closely the structure of the inductively defined static semantics in Coq. (Similarly, the compiler structures for FTAL abstract syntax mirror the Coq encoding.) Thus, the type-checker, as it analyzes the FTAL program, simultaneously builds a Coq term representing the proof of well-formedness of the program. In particular, if `P:Program`, then the type-checking phase produces a term, `D:(WFProgram P)`.

Once type-checking is successfully completed, the compiler then translates the FTAL program into a machine state. Again, this is done in such a manner that a Coq term representing the machine state and the proof of the relation between the FTAL program and the machine state can be generated. That is, for some `S:State`, a term, `T:(TrProgram P S)`, is constructed. Along with the typing derivation term of `P` produced above, we can now construct a proof that the global invariant holds on `S`. This can then be composed with the `Safety` theorem of the previous section to produce a complete proof of the safety of the machine state `S`, as specified by our safety policy.

3.3.6 The Complete System

We now have a complete system that starts with a typed assembly language program and compiles it into an FPCC package, consisting of an initial machine state and a proof of safety. Although my current implementation is not as realistic as [15, 4], the advantages of the syntactic FPCC approach are still clear.

With respect to PCC implementations in general, the two most practical considerations are the extent of the trusted computing base (TCB) and the size of the proofs that are shipped with code. As for the former, the TCB of my syntactic FPCC implementation would consist of the following: (1) a parser, which converts the state of the raw machine into the encoding in the logic; (2) the encoding of the machine step function in the logic, which must accurately capture the semantics of the real machine (that is, it must be adequate); and (3) the proof-checker of the logic. The first two will necessarily exist in any

PCC system. For syntactic FPCC (and FPCC in general), the proof-checker is smaller and more reliable than that of existing PCC systems because the logic used is much simpler.[5] In addition, the VCgen is completely eliminated from the system.

Regarding the proofs that are shipped with syntactic FPCC packages, note that a large portion of the safety proof is static—the Progress and Preservation theorems hold regardless of the particular FTAL program from which the machine state was compiled. Hence, this part of the proof does not need to be re-supplied (or even re-checked) with every individual FPCC package. Furthermore, the remaining portion of the proof simply consists of the initial FTAL program and its typing derivation. The typing derivation can be easily and quickly generated by either the code producer or consumer. Thus, if proof size is especially critical, the only additional information that needs to be supplied with the initial machine state is the FTAL program itself with minimal type annotations.

3.4 Summary

This chapter has illustrated one of the main contributions of my dissertation, which is the development of a new "syntactic" approach to producing foundational proof-carrying code. Unlike earlier FPCC methods, I have been able to present a relatively simple and straightforward compilation from TAL programs to certified machine code. Albeit the technical product may not be as tangible (I have used a very restricted form of TAL and an idealized machine), I have dealt with several theoretical aspects that required much more effort in existing semantic FPCC frameworks. Among these features are, primarily, mutable memory, first-class code pointers, and recursive types.

[5] That is, taking into consideration the type-related axioms that need to be added to the base logic of those systems. Furthermore, although our prototype uses the Coq proof assistant which integrates the proving and checking processes it would not be at all difficult to separate the proof-checker out into a very small, simple program, as has been done in previous FPCC approaches.

Chapter 4

Interfacing Type Systems and Certified Machine Code

As discussed in the introduction, the initial proof-carrying code (PCC) systems specified a safety policy using a logic extended with many (source) language-specific rules. While allowing implementation of a scalable system, this approach to PCC suffers from too large of a trusted computing base (TCB). It is still difficult to trust that the components of this system – the verification-condition generator, the proof-checker, and even the logical axioms and typing rules – are free from error.

The development of another family of PCC implementations, known as Foundational Proof-Carrying Code (FPCC), was intended to reduce the TCB to a minimum by expressing and proving safety using only a foundational mathematical logic without additional language-specific axioms or typing rules. The trusted components in such a system are mostly reduced to a much simpler logic and the proof-checker for it.

Both these approaches to PCC have so far one feature in common, which is that they begin with a single source language and compile type-correct programs from that language into machine code with a safety proof. However, the runtime systems of these frameworks still include components that are not addressed in the safety proof [3, 19]: low-level memory management libraries, garbage collection, debuggers, marshallers, *etc.*

Furthermore, the issue of producing a safety proof for code that is compiled and linked together from two different source languages was not addressed. (Some recent efforts on this aspect in the context of the original PCC systems is cited in the Related Work.)

In this chapter, I introduce an FPCC framework for constructing certified machine code packages from typed assembly language that will interface with a similarly certified runtime system. The framework permits the typed assembly language to have a "foreign function" interface in which stubs, initially provided when the program is being written, are eventually compiled and linked to code that may have been written in a language with a different type system, or even certified directly in the FPCC logic using a proof assistant.

Experience has shown that foundational proofs are much harder to construct than those in a logic extended with type-specific axioms. The earliest FPCC systems built proofs by constructing sophisticated semantic models of types in order to reason about safety at the machine level. That is, the final safety proof incorporated no concept of source level types – each type in the source language would be interpreted as a predicate on the machine state and the typing rules of the language would turn into lemmas which must prove properties about the interaction of these predicates. While it seems that this method of FPCC would already be amenable to achieving the goals outlined in the previous paragraph, the situation is complicated by the complexity of the semantic models [25, 5, 2] that were required to support a realistic type system. Thus, it is not clear yet how one would integrate the semantic models of different source type systems and safety proofs of the runtime components to produce a complete package.

For my work, I adopt the "syntactic" approach to FPCC, introduced in the previous chapter. In this framework, the machine level proofs do indeed incorporate and use the syntactic encoding of elements of the source type system to derive safety. The presentation of the syntactic approach in Chapter 3 involves a monolithic translation from type-correct source programs to a package of certified machine code. In this chapter, I refine the approach by inserting a generic layer of reasoning above the machine code which can (1) be a target for the compilation of typed assembly languages, (2) certify low-level runtime sys-

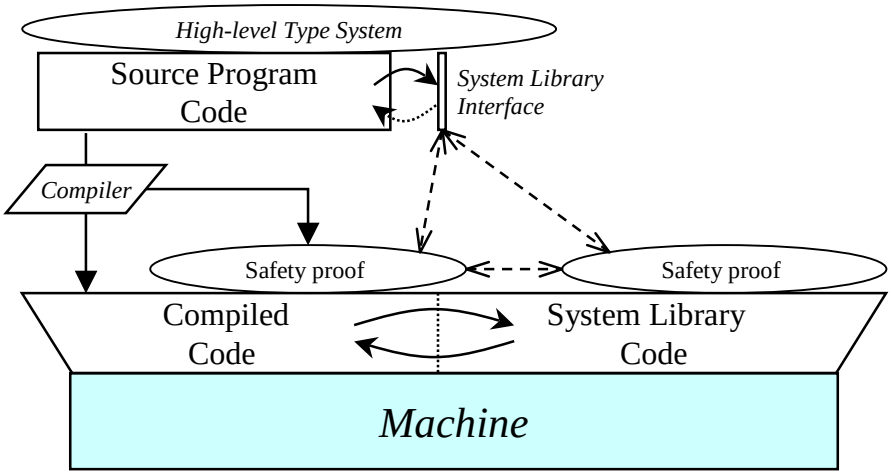


Figure 4.1: FPCC Certified Runtime Framework.

tem components using assertions as in Hoare logic, and (3) "glue" together these pieces by reasoning about the compatibility of the interfaces specified by the various types of source code.

A diagram of my framework is given in Figure 4.1. Source programs are written in a typed high-level language and then passed through a certifying compiler to produce machine code along with a proof of safety. The source level type system may provide a set of functionality that is accessed through a library interface. At the machine level, there is an actual library code implementation that should satisfy that interface. The non-trivial problem is how to design the framework such that not only will the two pieces of machine code link together to run, but that the safety proofs originating from two different sources are also able to "link" together, consistent with the high-level interface specification, to produce a unified safety proof for the entire set of code.

Notice that the interaction between program and library is two-way: either piece of code may make direct or indirect function calls and returns to the other. Ideally, I want to be able to certify the library code with no knowledge of the source language and type system that will be interacting with it. At the same time I would like to support first-class code pointers at all levels of the code. Methods for handling code pointers properly have been one of the main challenges of FPCC and are one of the differentiating factors between

semantic and syntactic FPCC approaches. For the framework in this paper, I have factored out most of the code pointer reasoning that is needed when certifying library code so that the proofs thereof can be relatively straightforward.

In the following section I present the layer of reasoning which will serve as the common interface for code compiled from different sources. Then I present a typical typed assembly language, extended with library interfaces and external call facilities. I finally show how to compile this language to the target machine, expanding external function stubs, and linking in the runtime library, at the same time producing the proof of safety of the complete package.

Coq code: The Coq (version 8.0) prototype code of the development in this chapter may be downloaded from:

`http://flint.cs.yale.edu/flint/publications/rtpcc.html`

4.1 A Language for Certified Machine Code (CAP)

Recalling the discussion in Section 2.2.4, we know what type of proof we are looking for; the hard part is to generate that proof of safety. Previous approaches for FPCC [4, 5, 33] have achieved this by constructing an induction hypothesis, also known as the global invariant, which can be proven (*e.g.* by induction) to hold for all states reachable from the initial state and is strong enough to imply the safety condition. The nature of the invariant has ranged from a semantic model of types at the machine level (Appel *et al.* [4, 5, 77]) to a purely syntactic well-formedness property [33, 34] based on a type-correct source program in a typed assembly language.

What I have developed in this chapter refines these previous approaches. I will still be presenting a typed assembly language in Section 4.3, in which most source programs are written. However, I introduce another layer between the source type system and the "raw" encoding of the target machine in the FPCC logic. This is a "type system" or "specification system" that is defined upon the machine encoding, allowing us to reason

about its state using assertions that essentially capture Hoare logic-style reasoning. Such a layer allows more generality for reasoning than a fixed type system, yet at the same time is more structured than reasoning directly in the logic about the machine encoding.

The language is called CAP and it uses the same machine syntax as presented in Figure 2.1. The syntax of the additional assertion layer is given below:

$$P, Q, R \in \mathit{Pred} \quad = \mathit{State} \to \mathit{Prop}$$

$$\Phi \in \mathit{CdSpec} = \mathit{Word} \rightharpoonup (\mathit{Word} \times \mathit{Pred})$$

$$\mathit{CmdList} \ni \mathbb{C} ::= \emptyset \mid c; \mathbb{C}$$

$$\mathit{WordList} \ni \mathbb{W} ::= \emptyset \mid w; \mathbb{W}$$

The name CAP is derived from its being a "Certified Assembly Programming" language. An initial version was introduced in joint work with Yu [97] and used to certify a dynamic storage allocation library. The version I use for this thesis introduces some improvements such as a unified data and code memory, assertions on the whole machine state, and support for user-specifiable safety policies. Yu [99] has independently extended the CAP system in a different direction to support certification of temporal properties for concurrent assembly code.

Assertions (P,Q,R) are predicates on the machine state; the code specification (Φ) is a partial function mapping memory addresses to a pair of an integer and a predicate. The integer gives the length of the command sequence at that address and the predicate is the precondition for the block of code. (The function of the length element is to allow us to specify the addresses of valid code areas of memory based on Φ.)

The operational semantics of the machine has already been presented in Section 2.1. I now introduce CAP inference rules followed by some important safety theorems.

$$\frac{\begin{array}{l} c \in \{\texttt{add}, \texttt{addi}, \texttt{mov}, \texttt{movi}, \texttt{ld}\} \\ \forall \mathbb{S}.(P(\mathbb{S}) \ \wedge\ \mathsf{curcmd}(\mathbb{S}) = c) \ \rightarrow\ (Q(\mathsf{Step}(\mathbb{S})) \ \wedge\ \mathsf{SP}(\mathbb{S})) \\ \Phi \vdash_{\mathsf{SP}} \{Q\}\, \mathbb{C} \end{array}}{\Phi \vdash_{\mathsf{SP}} \{P\}\, c; \mathbb{C}} \text{ (CAP-PURE)}$$

$$\frac{\begin{array}{l} \forall \mathbb{S}.(P(\mathbb{S}) \ \wedge \mathsf{curcmd}(\mathbb{S}) = \texttt{st}\ r_d(i), r_s) \\ \qquad \rightarrow\ (Q(\mathsf{Step}(\mathbb{S})) \ \wedge\ \mathsf{SP}(\mathbb{S}) \ \wedge\ \neg \mathsf{InCodeArea}(\Phi, \mathbb{S}.\mathbb{R}(r_d) + i)) \\ \Phi \vdash_{\mathsf{SP}} \{Q\}\, \mathbb{C} \end{array}}{\Phi \vdash_{\mathsf{SP}} \{P\}\, \texttt{st}\ r_d(i), r_s; \mathbb{C}} \text{ (CAP-ST)}$$

$$\frac{\begin{array}{l} \forall \mathbb{S}.\ (P(\mathbb{S}) \ \wedge\ \mathsf{curcmd}(\mathbb{S}) = \texttt{bgt}\ r_s, r_t, w) \\ \quad \rightarrow\ ((\mathbb{S}.\mathbb{R}(r_s) \leq \mathbb{S}.\mathbb{R}(r_t) \rightarrow Q(\mathsf{Step}(\mathbb{S}))) \\ \qquad \wedge\ (\mathbb{S}.\mathbb{R}(r_s) > \mathbb{S}.\mathbb{R}(r_t) \rightarrow Q'(\mathsf{Step}(\mathbb{S}))) \\ \qquad \wedge\ \mathsf{SP}(\mathbb{S})) \\ \Phi \vdash_{\mathsf{SP}} \{Q\}\, \mathbb{C} \end{array} \quad \textit{where } \Phi(w) = (n, Q')}{\Phi \vdash_{\mathsf{SP}} \{P\}\, \texttt{bgt}\ r_s, r_t, w; \mathbb{C}} \text{ (CAP-BGT)}$$

$$\frac{\forall \mathbb{S}.(P(\mathbb{S}) \ \wedge\ \mathsf{curcmd}(\mathbb{S}) = \texttt{jd}\ w) \rightarrow (Q'(\mathsf{Step}(\mathbb{S})) \ \wedge\ \mathsf{SP}(\mathbb{S})) \quad \textit{where } \Phi(w) = (n, Q')}{\Phi \vdash_{\mathsf{SP}} \{P\}\, \texttt{jd}\ w; \emptyset} \text{ (CAP-JD)}$$

$$\frac{\forall \mathbb{S}.(P(\mathbb{S}) \ \wedge\ \mathsf{curcmd}(\mathbb{S}) = \texttt{jmp}\ r) \rightarrow (Q'(\mathsf{Step}(\mathbb{S})) \ \wedge\ \mathsf{SP}(\mathbb{S})) \quad \textit{where } \Phi(\mathbb{S}.\mathbb{R}(r)) = (n, Q')}{\Phi \vdash_{\mathsf{SP}} \{P\}\, \texttt{jmp}\ r; \emptyset} \text{ (CAP-JMP)}$$

$$\frac{\mathsf{Flatten}(\mathbb{W}, \mathbb{M}, f) \qquad \Phi \vdash_{\mathsf{SP}} \{P\}\, (\mathsf{Map}(\mathsf{Dc}, \mathbb{W})) \quad \textit{for all } f \textit{ where } \Phi(f) = (\mathsf{length}(\mathbb{W}), P)}{\vdash_{\mathsf{SP}} \mathbb{M} : \Phi} \text{ (CAP-CDSPEC)}$$

$$\frac{\vdash_{\mathsf{SP}} \mathbb{M} : \Phi \qquad \Phi \vdash_{\mathsf{SP}} \{P\}\, (\mathsf{Map}(\mathsf{Dc}, \mathbb{W})) \qquad \mathsf{Flatten}(\mathbb{W}, \mathbb{M}, pc) \qquad \mathsf{InCodeArea}(\Phi, pc) \qquad P(\mathbb{M}, \mathbb{R}, pc)}{\vdash_{\mathsf{SP}} (\mathbb{M}, \mathbb{R}, pc)} \text{ (CAP-STATE)}$$

Figure 4.2: CAP inference rules.

4.1.1 Inference Rules

CAP adds a layer of inference rules ("typing rules") allowing us to prove specification judgments of the forms:

$$\Phi \vdash_{\mathsf{SP}} \{P\}\,\mathbb{C} \quad \textit{well-formed command sequence}$$

$$\vdash_{\mathsf{SP}} \mathbb{M} : \Phi \quad \textit{well-formed code specification}$$

$$\vdash_{\mathsf{SP}} (\mathbb{M}, \mathbb{R}, pc) \quad \textit{well-formed machine state}$$

The inference rules for these judgments are shown in Figure 4.2. The rules for well-formed command sequences essentially require that if the given precondition P is satisfied in the current state, then (1) the global predicate SP holds on that state and (2) there must be some postcondition Q, which is the precondition of the remaining sequence of commands, that holds on the state after executing one step.[1] The rules directly refer to the Step function of the machine; control flow instructions additionally use the code specification environment Φ in order to allow for the certification of mutually dependent code blocks. The global predicate SP is provided as a parameter to the whole process of type-checking a program; it is threaded through all the rules but does not change.

I group as "pure" commands all those which do *not* involve control flow and do not change the memory (*i.e.* everything other than branches, jumps, and `st`). The `st` command requires an additional proof that the address being stored to is not in the code area (*i.e.* I do not permit self-modifying code). $\mathsf{curcmd}(\mathbb{S})$ is defined as:

$$\mathsf{curcmd}(\mathbb{M}, \mathbb{R}, pc) = \mathsf{Dc}(\mathbb{M}(pc))$$

The InCodeArea predicate in the rules uses the code addresses and sequence lengths in Φ to determine whether a given address lies within the code area. The (CAP-CDSPEC)

[1] In the abstract syntax of CAP given here, only the precondition of an entire code block is specified explicitly; the postconditions of intermediate commands in the code block are embedded in the term representing the well-formedness of the code. Of course, in practice we could provide a concrete syntax for CAP that allows the user to annotate instructions explicitly with pre- and postconditions.

rule ensures that the addresses and sequence lengths specified in Φ are consistent with the code actually in memory.

The Flatten predicate is defined as:

$$\mathsf{Flatten}(\emptyset, \mathbb{M}, f) = \mathsf{True}$$
$$\mathsf{Flatten}(w; \mathbb{W}, \mathbb{M}, f) = \ \mathbb{M}(f) = w \ \wedge \ \mathsf{Flatten}(\mathbb{W}, \mathbb{M}, f+1)$$

Note also, I use dot notation to refer to the components of a triple like the state: $\mathbb{S}.\mathbb{M}$, $\mathbb{S}.\mathbb{R}$, *etc.*

4.1.2 Safety Properties

The machine of Section 2.1 will execute continuously, even if an `illegal` instruction is encountered. Given a well-formed CAP state, however, we can prove that it satisfies:[2]

Theorem 4.1 (CAP Soundness)
For any safety policy SP and state $\mathbb{S}$, if $\vdash_{\mathsf{SP}} \mathbb{S}$ then (1) $\mathsf{SP}(\mathbb{S})$ and (2) $\vdash_{\mathsf{SP}} \mathsf{Step}^n(\mathbb{S})$ for all integers $n \geq 0$.

Proof sketch The proof is very straightforward: (1) the CAP inference rules ensure that SP holds on every well-formed state, and (2) by induction on the well-formed command sequence judgment. □

For the purposes of FPCC, we are interested in obtaining safety proofs in the context of our policy as described in Section 2.2.3. For this chapter, I again define the basic safety policy as requiring that the machine always be at a valid instruction:

$$\mathsf{BasicSP}\,(\mathbb{M}, \mathbb{R}, pc) = \ \ (\mathsf{Dc}\,(\mathbb{M}(pc)) \neq \texttt{illegal})$$

[2] An even stronger statement of CAP soundness can be made, as given in [97] but I do not need it for my development here.

It is fairly straightforward to show that any well-formed CAP state will be safe according to this policy. That is, no matter what the global SP parameter is instantiated to (*e.g.* it could simply be a trivial True predicate), the CAP inference rules at a minimum will enforce the basic safety policy:

Lemma 4.2 For any SP, if $\vdash_{\mathsf{SP}} \mathbb{S}$ then $\mathsf{BasicSP}(\mathbb{S})$.

Proof Follows directly from examination of the well-formedness rules. Since there is no rule for checking an `illegal` command, we know that the current command is not that, since it is well-formed by the top-level rule (CAP-STATE). □

Now, from this lemma and the CAP Soundness theorem, we can derive the following result for general safety policies:

Theorem 4.3 (CAP Safety)

For any SP and $\mathbb{S}$, if $\vdash_{\mathsf{SP}} \mathbb{S}$ then $\mathsf{Safe}(\mathbb{S}, \lambda\mathbb{S}' : \mathit{State}.\, \mathsf{SP}(\mathbb{S}') \wedge \mathsf{BasicSP}(\mathbb{S}'))$ according to the definition of Safe on page 25.

Proof Directly from Lemma 4.2 and CAP Soundness. □

Threading an arbitrary SP through the typing rules is a novel feature not found in the initial version of CAP [97]. In that case, there was no way to specify that an arbitrary safety policy beyond BasicSP (which essentially provides type safety) must hold at every step of execution. For this chapter, I will actually not make use of this feature but in the next chapter, I will show how it is used with a more realistic and complex FPCC safety policy.

From now on, I will write CAP judgment forms with a bulleted placeholder ($\vdash_{\bullet} \mathbb{S}$) to indicate the use of a trivial global safety predicate ($\lambda\mathbb{S}.\mathsf{True}$) for SP.

Thus, to produce an FPCC package we just need to prove that the initial machine state is well-formed in CAP with respect to the desired safety policy. This provides a structured method for constructing FPCC packages in our logic. However, programming and reasoning in CAP is still much too low-level for the practical programmer. We need to

provide a method for compiling programs from a higher-level language and type system to CAP. The main purpose of programming directly in CAP will then be to "glue" code together from different source languages and to semi-automatically certify particularly low-level libraries such as memory management, threads, *etc.*. In the next few sections, I present a "conventional" typed assembly language and show how to compile it to CAP.

Coq code: The definition of CAP and the proved theorems of this section are in the file `captis.v`.

4.2 The Code Pointer Problem

Before going on to a typed assembly language, let us try to understand one of the difficulties that arise when trying to use a Hoare logic-based framework to certify low-level machine code. Consider the following annotated CAP code blocks that may be part of a larger set of code making up a program or a system library:

```
23: { R(r0) > 0 }
    movi r2, 9;
    { R(r0) > 0  /\  R(r2) = 9 }
    ⋮
    { R(r2) > 9 }
    jd 52

52: { R(r2) > 5 }
    ⋮
    { R(r0) = R(r2)  /\  R(r2) > 10 }
    jd 78

78: { R(r2) > 8 }
    ⋮
    { R(r0) = 7 }
    jd 23
```

The relevant part of the CAP code specification is composed from the preconditions of these code blocks:

```
Φ = { 23:{ R(r0) > 0 }, 52:{ R(r2) > 5 }, 78:{ R(r2) > 8 }, ... }
```

It is easy enough to verify these code blocks because the control flow only involves direct jumps. Now, suppose instead of `jd 52`, the last command of the first block was `jmp r5`; *i.e.* register `r5` contains a pointer to a continuation. It may be that `r5` holds either the address 52 or 78. Now, it is a bit more tricky to see how one would verify correctness of this jump. One immediate option is to have the precondition of the jump to `r5` specify all the possible targets of the jump:

```
23: { R(r0) > 0 }
    ⋮
    { R(r2) > 9  /\  (R(r6) = 52 \/ R(r6) = 78) }
    jmp r6
```

Then we would verify that the precondition (or postcondition) of the jump is stronger than the preconditions of all the possible targets. One obvious problem with this approach is that it requires analysis of the whole program in order to determine all possible targets of such indirect jumps. While this might be easily enough done, it is not appropriate if this code is supposed to be used as a runtime library. We do not want to have to recompute the return targets of library code, in order to form new preconditions, each time it is linked with a newly compiled source language program. Basically, when verifying library code, we would like a natural way to do so without worrying about the user code with which it will interact.

In order to achieve this modularity, one possibility could be to introduce an abstract predicate, `codeptr`, to be used in the Hoare logic assertions. Thus, we could have a precondition like:

```
23: { R(r0) > 0 }
    ⋮
    { R(r2) > 9  /\  codeptr( R(r6), { R(r2) > 5 } ) }
    jmp r6
```

stating that the value in register `r6` is a code pointer whose precondition requires the value in `r2` be greater than 5. The difficulty then is how to keep the reasoning about the `codeptr` predicates consistent. Since these predicates will appear within the definition of

Φ, we cannot define `codeptr(w,P)` as meaning $\Phi(w) = P$ because that would introduce a circularity in the definition. One might imagine trying to add additional inference rules to CAP for the `jmp` rule and the top-level rules that tie together the use of `codeptr` as an abstract predicate, but I have not been able to achieve that. Furthermore, there is the complication of mutually recursive code. Consider the case where the code pointer in `r6` above may itself perform an indirect jump through a register back to location `23`. Not only will the code pointer predicates be nested, they require a recursive type because each depends on the other.

The solution I have adopted for my purposes, based on that introduced in [97, 98], is to take advantage of the fact that our CAP language is in fact embedded in a higher-order logic. Thus, when verifying a CAP code block such as `23` above, I quantify over the code specification and precondition with an assumption that the postcondition will meet the requirements of the indirect jump. For example, the block:

```
23: { Pre }                 // R(r0) > 0
    ℂ;              // command list
    { Post }                // R(r2) > 9
    jmp r6
```

would actually be certified in CAP using the judgment form (assuming some safety policy, SP):

$$\Phi \vdash_{\mathsf{SP}} \{Pre\}\, \mathbb{C}; \mathtt{jmp\ r6}$$

Now, instead of directly certifying the code like this, where the simple precondition will not be sufficient to handle the indirect jump, I prove this judgment in CAP as the following lemma:

$$\forall \Phi, P.\ P \Rightarrow Pre\ \wedge\ (\exists Q.\ \Phi(\mathbb{R}(\mathsf{r6})) = Q\ \wedge\ Post \Rightarrow Q) \qquad (1)$$
$$\rightarrow \Phi \vdash_{\mathsf{SP}} \{P\}\, \mathbb{C}; \mathtt{jmp\ r6}$$

The quantified predicate P here implies the necessary precondition of the code block

itself, as well as the fact that there is a code pointer in the return register whose precondition is satisfied by $Post$.[3]

Now, this is only half the story. It allows us to verify the indirect jump of a single code block, but when linking a whole program together, we have to provide a concrete predicate for P. Again, P can be instantiated using a disjunction of all possible targets of the jump, just like our earlier concrete precondition:

```
{ R(r2) > 9  /\  (R(r6) = 52 \/ R(r6) = 78) }
```

We are left with the premise of the lemma (1) above which is to show that this instantiation of P satisfies the necessary conditions on the return code pointer. This can again easily be done by analysis over the various cases of the disjunction.

So far, we have somewhat improved modularity of the certification process since the code block itself is verified independent of its actual targets. While this approach may be sufficient for verifying the actual code blocks of a runtime library (as shown in [97, 98]), when compiling from a higher-level type system, we still need a way to link together the code pointers of the high-level type system (and their associated type interfaces) with the low-level CAP preconditions of the library code. For this purpose, I leverage my syntactic approach to certified code in order to provide an automatic method of linking compiled source programs from a type system to library code certified safe using the Hoare logic based inference system of CAP. Details of how this is done are contained in the remainder of this chapter.

As a brief overview, the idea is to instantiate the predicate P with an invariant based on Inv of the previous chapter (Section 3.2.2). For the example above, we could use something like: $\lambda\mathbb{S}.\ \exists\mathcal{P} : \mathit{program}.\ (\vdash \mathcal{P} : \Gamma)\ \wedge\ (\mathcal{P}_\Gamma \Rightarrow \mathbb{S})$. The program $\mathcal{P}$ is not completely abstract here but is constrained by the TAL type Γ for which this program must be well-formed in order to show the necessary premise of lemma (1) above. Γ in this case would be a TAL code type such as $\{\mathsf{r0} : \mathsf{int}_{>0}, \mathsf{r6} : \forall[\{\mathsf{r2} : \mathsf{int}_{>9}\}]\}$ (assuming a fancier TAL type system which tracks integer values). By reasoning (syntactically) over the properties of the type

[3] $P \Rightarrow Q$ is defined as $\forall\mathbb{S}.P(\mathbb{S}) \rightarrow Q(\mathbb{S})$.

system, we would be able to show that this invariant implies the necessary precondition Pre of the code block, and it also enforces the correctness of the indirect jump. To show the latter, we actually need to know more about the formation of the entire CAP code specification– *e.g.* , that its preconditions are consistent with the TAL code types. Details and an example of how the complete CAP code specification is generated when linking compiled TAL code to certified CAP code are given in Sections 4.4.3 and 4.4.4.

Using the method outlined in the last few paragraphs, I am able to verify runtime library code in CAP independent of the source language type system or other code that it interacts with. Furthermore, use of syntactic type system-based invariants allows the safety proof of compiled source code to automatically "link" with the proof of these certified CAP code blocks, once the necessary lemmas such as (1) above have been established.

4.3 Extensible Typed Assembly Language with Runtime System

In this section, I introduce an extensible typed assembly language (XTAL), again based on that of Morrisett *et al.* [52]. After presenting the full syntax of XTAL, I give here only a brief overview of its static and dynamic semantics,[4] since it is very similar to FTAL in the previous chapter, except for the addition of primitive arrays, a newpair "macro" instruction, and stub values in the code heap. The data and code portions of the heap have also been separated in XTAL and it uses pairs instead of the more general tuples of FTAL.

4.3.1 Syntax

To simplify the presentation, XTAL's types (see Figure 4.3) involve only integers, pairs, and integer arrays. (I have extended the Coq prototype implementation to include existential, recursive, and polymorphic code types.) The code type $\forall[\Gamma]$ describes a code pointer that expects a register file satisfying Γ. The register file type assigns types to the

[4]Chapter 5 gives the gory details of a more realistic variant of XTAL.

word values in each register and the heap type keeps track of the heap values in the data heap. I have separated the code and data heaps at this level of abstraction because the code heap will remain unchanged throughout the execution of a program.

Unlike many conventional TALs, my language supports "stub values" in its code heap. These are placeholders for code that will be linked in later from another source (outside the XTAL system). Primitive "macro" instructions that might be built into other TALs, such as array creation and access operations, can be provided as an external library with interface specified as XTAL types. I have also included a typical macro instruction for allocating pairs (newpair) in the language. When polymorphic types are added to the language, this macro instruction could potentially be provided through the external code interface; however, in general, providing built-in primitives can allow for a richer specification of the interface where the type system is too limited (see the typing rule for newpair below).

The abstract state of an XTAL program is composed of code and data heaps, a register file, and current instruction sequence. Labels are simply integers and the domains of the code and data heaps are to be disjoint. Besides the newpair operation, the arithmetic, memory access, and control flow instructions of XTAL correspond directly to those of the machine defined in 2.1. The movl instruction is constrained to refer only to code heap labels. Note that programs are written in continuation passing style; thus every code block ends with some form of jump to another location in the code heap.

4.3.2 Static and Dynamic Semantics

The dynamic (operational) semantics of the XTAL abstract machine is defined by a set of rules of the form $\mathcal{P} \mapsto \mathcal{P}'$. This evaluation relation is entirely standard (as in Figure 3.2) except that the case when jumping to a stub value in the code heap is not handled. The complete rules are omitted here.

For the static semantics, I define a set of judgments as illustrated in Figure 4.4. Only

$$
\begin{array}{lll}
\textit{(type)} & \tau & ::= \mathsf{int} \mid \mathsf{array} \mid \tau_0 \times \tau_1 \mid \forall[\Gamma] \\
\textit{(reg file type)} & \Gamma & ::= \{r_0 : \tau_0, \ldots, r_n : \tau_n\} \\
\textit{(heap type)} & \Psi & ::= \{l_0 : \tau_0, \ldots, l_n : \tau_n\} \\
\textit{(label)} & l & ::= 0 \mid 1 \mid \ldots \\
\textit{(register)} & r & ::= \mathsf{r0} \mid \mathsf{r1} \mid \ldots \mid \mathsf{r7} \\
\textit{(word val)} & v & ::= l \mid i \\
\textit{(code heap val)} & \overline{h} & ::= \mathsf{code}\ [\Gamma].I \mid \mathsf{stub}\ [\Gamma].\emptyset \\
\textit{(heap val)} & h & ::= [i_0, \ldots, i_n] \mid \langle v_0, v_1 \rangle \\
\textit{(instr)} & \iota & ::= \mathsf{add}\ r_d, r_s, r_t \mid \mathsf{movi}\ r_d, i \mid \mathsf{movl}\ r_d, l \mid \mathsf{ld}\ r_d, r_s(i) \\
 & & \quad \mid \mathsf{st}\ r_d(i), r_s \mid \mathsf{newpair}\ r_d[\tau_0, \tau_1] \\
\textit{(instr seq)} & I & ::= \iota; I \mid \mathsf{jd}\ l \mid \mathsf{jmp}\ r \\
\textit{(code heap)} & \mathcal{C} & ::= \{l_0 \mapsto \overline{h}_0, \ldots, l_n \mapsto \overline{h}_n\} \\
\textit{(data heap)} & H & ::= \{l_0 \mapsto h_0, \ldots, l_n \mapsto h_n\} \\
\textit{(reg file)} & R & ::= \{r_0 \mapsto v_0, \ldots, r_n \mapsto v_n\} \\
\textit{(program)} & \mathcal{P} & ::= (\mathcal{C}, H, R, I)
\end{array}
$$

Figure 4.3: XTAL syntax.

a few of the critical XTAL typing rules are presented here. The top-level typing rule for XTAL programs requires well-formedness of the code and data heaps, register file, and current instruction sequence, and that I is somewhere in the code heap:

$$
\frac{\vdash \mathcal{C} \qquad \vdash H : \Psi \qquad \mathcal{C}; \Psi \vdash R : \Gamma \qquad \mathcal{C}; \Gamma \vdash I \qquad \exists l \in Dom(\mathcal{C}).\ \mathcal{C}(l) = \mathsf{code}\ [\Gamma'].I' \ \textit{and}\ I \subseteq_{\textit{tail}} I'}{\vdash (\mathcal{C}, H, R, I)}\ (\text{PROG})
$$

Heap and register file typing depends on the well-formedness of the elements in each. Stub values are simply assumed to have the specified code type. From the instruction typing rules, we show below the rules for newpair, jd, and jmp. The newpair instruction expects initialization values for the newly allocated space in registers r0 and r1 and a pointer to the new pair is put in r_d.

Judgment	Meaning
$\vdash \Gamma_0 \subseteq \Gamma_1$	Γ_0 is a register file subtype of Γ_1
$\vdash (\mathcal{C}, H, R, I)$	$(\mathcal{C}, H, R, I)$ is a well-formed program
$\vdash \mathcal{C}$	$\mathcal{C}$ is a well-formed code heap
$\vdash H : \Psi$	H is a well-formed data heap of type Ψ
$\mathcal{C}; \Psi \vdash R : \Gamma$	R is a well-formed reg. file of type Γ
$\mathcal{C} \vdash \overline{h}$ cdval	$\overline{h}$ is a well-formed code heap value
$\Psi \vdash h : \tau$ hval	h is a well-formed data heap value of type τ
$\Psi \vdash v : \tau$	v is a well-formed word value of type τ
$\mathcal{C}; \Gamma \vdash I$	I is a well-formed instruction sequence

Figure 4.4: Static judgments.

$$\frac{\mathcal{C}; \Gamma \vdash I}{\mathcal{C} \vdash \mathsf{code}\ [\Gamma].I\ \mathsf{cdval}}\ (\textsc{code}) \qquad \frac{}{\mathcal{C} \vdash \mathsf{stub}\ [\Gamma].\emptyset\ \mathsf{cdval}}\ (\textsc{stub})$$

$$\frac{\Gamma(\mathsf{r0}) = \tau_0 \qquad \Gamma(\mathsf{r1}) = \tau_1 \qquad \mathcal{C}; \Gamma\{r_d : \tau_0 \times \tau_1\} \vdash I}{\mathcal{C}; \Gamma \vdash \mathsf{newpair}\ r_d[\tau_0, \tau_1]; I}\ (\textsc{is-newpair})$$

$$\frac{\mathsf{typeof}(\mathcal{C}(l)) = \forall[\Gamma'] \quad \vdash \Gamma \subseteq \Gamma'}{\mathcal{C}; \Gamma \vdash \mathsf{jd}\ l}\ (\textsc{is-jd}) \qquad \frac{\Gamma(r) = \forall[\Gamma'] \quad \vdash \Gamma \subseteq \Gamma'}{\mathcal{C}; \Gamma \vdash \mathsf{jmp}\ r}\ (\textsc{is-jmp})$$

Although the details of the type system are certainly important, the key thing to be understood here is just that we are able to encode the syntactic judgment forms of XTAL well-formedness in the FPCC logic and prove soundness in Wright-Felleisen style [93]. We will then refer to these judgments in CAP assertions during the process of proving machine code safety.

4.3.3 External Code Stub Interfaces

XTAL can pass around pointers to arrays in its data heap but has no built-in operations for allocating, accessing, or modifying arrays. These are provided through code stubs:

$$\mathsf{newarray} \mapsto \mathsf{stub}\ [\{\ \mathsf{r0}:\mathsf{int},\ \mathsf{r1}:\mathsf{int},\ \mathsf{r7}:(\forall[\{\mathsf{r0}:\mathsf{array}\}])\ \}].\emptyset$$

$$\mathsf{arrayget} \mapsto \mathsf{stub}\ [\{\ \mathsf{r0}:\mathsf{array},\ \mathsf{r1}:\mathsf{int},\ \mathsf{r7}:(\forall[\{\mathsf{r0}:\mathsf{int}\}])\ \}].\emptyset$$

$$\mathsf{arrayset} \mapsto \mathsf{stub}\ [\{\ \mathsf{r0}:\mathsf{array},\ \mathsf{r1}:\mathsf{int},\ \mathsf{r2}:\mathsf{int},\ \mathsf{r7}:(\forall[\{\mathsf{r0}:\mathsf{array}\}])\ \}].\emptyset$$

newarray expects a length and initial value as arguments, allocates and initializes a new array accordingly, and then jumps to the code pointer in r7. The accessor operations similarly expect an array and index arguments and will return to the continuation pointer in r7 when they have performed the operation. As is usually the case when dealing with external libraries, the interfaces (code types) defined above do not provide a complete specification of the operations (such as bounds-checking issues). Section 4.4.3 discusses how we deal with this in the context of the safety of XTAL programs and the final executable machine code.

4.3.4 Soundness

As usual, we need to show that our XTAL type system is sound with respect to the operational semantics of the abstract machine. This can be done using the standard progress and preservation lemmas. However, in the presence of code stubs, the complete semantics of a program is undefined, so at this level of abstraction we can only assume that those typing rules are sound. In the next section, when compiling XTAL programs to the real machine and linking in code for these libraries and stubs, we will need to prove at that point that the linked code is sound with respect to the XTAL typing rules. Let us define the state when the current XTAL program is jumping to external code:

Definition 4.4 (External call state) *We define the current instruction of a program,* $(\mathcal{C}, H, R, I)$*, to be an* external call *if* $I \in \{\mathsf{jd}\ l,\ \mathsf{jmp}\ r,\ \mathsf{bgt}\ldots,\ \mathsf{bgti}\ldots\}$ *and* $\mathcal{C}(l) = \mathsf{stub}\ [\Gamma].\emptyset$ *or* $\mathcal{C}(R(r)) = \mathsf{stub}\ [\Gamma].\emptyset$*, as appropriate.*

Now we can state qualified versions of the standard soundness lemmas:

Theorem 4.5 (XTAL Progress) If $\vdash \mathcal{P}$ and the current instruction of $\mathcal{P}$ is not an external call then there exists $\mathcal{P}'$ such that $\mathcal{P} \mapsto \mathcal{P}'$.

Theorem 4.6 (XTAL Preservation) If $\vdash \mathcal{P}$ and $\mathcal{P} \mapsto \mathcal{P}'$ then $\vdash \mathcal{P}'$.

These theorems are proved by induction on the well-formed instruction premise ($\mathcal{C}; \Gamma \vdash I$) of the top level typing rule ($\vdash \mathcal{P}$). Of course the proof of these must be done entirely in the FPCC logic in which the XTAL language is encoded.

In the previous chapter, I demonstrated how to get from these proofs of soundness directly to the FPCC safety proof. However, now we have an extra level to go through (the CAP system) in which we will also be linking external code to XTAL programs, and we must ensure safety of the complete package at the end.

Coq code: `axtalp.v` contains the definitions and proofs for an alternate version of XTAL than that presented here. The version in the file does not actually include the newpair instruction, but it does include primitive arrays as described in this section and also supports polymorphic code types, some of the details of which are more involved and will be covered in Chapters 5 and 6.

4.4 Compilation and Linking

In this section I first define how abstract XTAL programs will be translated to, and laid out in, the real machine state (the runtime memory layout). I also define the necessary library routines as CAP code (the runtime system). Then, after compiling and linking an XTAL program to CAP, I must show how to maintain the well-formedness of that CAP state so that we can apply Theorem 4.3 to obtain the final FPCC proof of safety.

4.4.1 The Runtime System

In my simple runtime system, memory is divided into three sections–a static data area (used for global constants and library data structures), a read-only code area (which might

be further divided into subareas for external ($\mathcal{E}$) and program ($\mathcal{C}$) code), and the dynamic heap area, which can grow indefinitely in our idealized machine. I use a data allocation framework where a heap limit, stored in a fixed allocation pointer register,[5] designates a finite portion of the dynamic heap area as having been allocated for use. (The safety policy could use this to specify the definition of "readable" and "writeable" memory.)

4.4.2 Translating XTAL Programs to CAP

I now outline how to construct (compile) an initial CAP machine state from an XTAL program. Given an initial XTAL program, we need the following (partial) functions or mappings to produce the CAP state:

- $\mathcal{A}_C : \mathit{label} \rightharpoonup \mathit{Word}$ – a layout mapping from XTAL code heap labels to CAP machine addresses.

- $\mathcal{A}_D : \mathit{label} \rightharpoonup \mathit{Word}$ – a layout mapping from XTAL data heap labels to CAP machine addresses. Both the domain and range of the two layout functions should be disjoint. I use $\mathcal{A}$ without any subscript to indicate the union of the two: $\mathcal{A} = \mathcal{A}_C \cup \mathcal{A}_D$.

- $\mathcal{E} : \mathit{Word} \rightharpoonup \mathit{CmdList} \times \mathit{Pred}$ – the external (from XTAL's point of view) code blocks and their CAP preconditions for well-formedness. Proving that these blocks are well-formed according to the preconditions will be a proof obligation when verifying the safety of the complete CAP state. The range of $\mathcal{A}_C$ may overlap with the domain of $\mathcal{E}$ – these addresses are the implementation of XTAL code stubs.

With these elements, the translation from XTAL programs to CAP is quite straightforward. As in Chapter 3, I describe the translation by a set of relations and associated inference rules (Figures 4.5 and 4.6). Register files and word values translate fairly directly between XTAL and the machine. XTAL labels are translated to machine addresses using the $\mathcal{A}$ functions. Every heap value in the code and data heaps must correspond to

[5] XTAL source programs use fewer registers than the actual machine provides.

WORD VALUES $\boxed{\mathcal{A} \vdash v \Rightarrow w}$

$$\frac{}{\mathcal{A} \vdash l \Rightarrow \mathcal{A}(l)}\ \text{(TR-LAB)} \qquad \frac{}{\mathcal{A} \vdash i \Rightarrow i}\ \text{(TR-INT)}$$

INSTRUCTION SEQUENCES *(selected rules)* $\boxed{\mathcal{A}_C \vdash I \Rightarrow \mathbb{C}}$

$$\frac{\mathcal{A}_C \vdash I \Rightarrow \mathbb{C}}{\mathcal{A}_C \vdash \mathsf{add}\ r_d, r_s, r_t; I \Rightarrow \mathtt{add}\ r_d, r_s, r_t; \mathbb{C}}\ \text{(TR-ADD)}$$

$$\frac{\mathcal{A}_C \vdash I \Rightarrow \mathbb{C}}{\mathcal{A}_C \vdash \mathsf{movi}\ r_d, i; I \Rightarrow \mathtt{movi}\ r_d, i; \mathbb{C}}\ \text{(TR-MOVI)}$$

$$\frac{\mathcal{A}_C(l) = w \qquad \mathcal{A}_C \vdash I \Rightarrow \mathbb{C}}{\mathcal{A}_C \vdash \mathsf{movl}\ r_d, l; I \Rightarrow \mathtt{movi}\ r_d, w; \mathbb{C}}\ \text{(TR-MOVL)} \qquad \frac{\mathcal{A}_C(l) = w}{\mathcal{A}_C \vdash \mathsf{jd}\ l \Rightarrow \mathtt{jd}\ w}\ \text{(TR-JD)}$$

$$\frac{\mathcal{A}_C \vdash I \Rightarrow \mathbb{C}}{\mathcal{A}_C \vdash \mathsf{newpair}\ r_d[\tau_0, \tau_1]; I \Rightarrow \mathbb{C}_{\mathsf{newp}}; \mathbb{C}}\ \text{(TR-NEWPAIR)}$$

HEAP VALUES $\boxed{\mathcal{A}_C \vdash \overline{h} \Rightarrow \mathbb{W} \qquad \mathcal{A} \vdash h \Rightarrow \mathbb{W}}$

$$\frac{\mathcal{A}_C \vdash I \Rightarrow \mathbb{C} \qquad \mathbb{C} = \mathsf{Map}(\mathsf{Dc}, \mathbb{W})}{\mathcal{A}_C \vdash \mathsf{code}\ [\Gamma].I \Rightarrow \mathbb{W}}\ \text{(TR-CODE)}$$

$$\frac{}{\mathcal{A} \vdash [i_0, \ldots, i_n] \Rightarrow n; i_0; \ldots; i_n}\ \text{(TR-ARRAY)}$$

$$\frac{\mathcal{A} \vdash v_i \Rightarrow w_i \qquad i \in \{0, 1\}}{\mathcal{A} \vdash \langle v_0, v_1 \rangle \Rightarrow w_0; w_1}\ \text{(TR-PAIR)}$$

where

$$\mathbb{C}_{\mathsf{newp}} = \mathtt{mov}\ r_d, r_{ap}; \mathtt{addi}\ r_{ap}, r_{ap}, 2; \mathtt{st}\ r_d(0), \mathsf{r0}; \mathtt{st}\ r_d(1), \mathsf{r1}$$

Figure 4.5: XTAL to CAP translation (1 of 2).

REGISTER FILE, HEAPS, AND PROGRAM

$$\frac{\mathcal{A} \vdash R(r) \Rightarrow \mathbb{R}(r) \qquad \textit{for all } r \in Dom(R)}{\mathcal{A} \vdash R \Rightarrow \mathbb{R}} \text{ (TR-RF)}$$

$$\frac{\mathcal{A}_C \vdash \mathcal{C}(l) \Rightarrow \mathbb{W} \qquad \mathsf{Flatten}(\mathbb{W}, \mathbb{M}, \mathcal{A}_C(l)) \qquad \textit{for all } l \in Dom(\mathcal{C})}{\mathcal{A}_C \vdash \mathcal{C} \Rightarrow \mathbb{M}} \text{ (TR-CHEAP)}$$

$$\frac{\begin{array}{l}\mathcal{A} \vdash H(l) \Rightarrow \mathbb{W} \qquad \mathsf{Flatten}(\mathbb{W}, \mathbb{M}, \mathcal{A}(l)) \qquad \textit{for all } l \in Dom(H) \\ \mathcal{A}_D(l) + \mathsf{length}(\mathbb{W}) < \mathbb{R}(r_{ap}) \qquad\qquad \mathsf{DLytConsist}(\mathcal{A}_D, H)\end{array}}{\mathcal{A}; \mathbb{R} \vdash H \Rightarrow \mathbb{M}} \text{ (TR-HEAP)}$$

$$\frac{\mathcal{A}_C \vdash \mathcal{C} \Rightarrow \mathbb{M} \qquad \mathcal{A} \vdash H \Rightarrow_{\mathbb{R}} \mathbb{M}}{\mathcal{A}; \mathbb{R} \vdash (\mathcal{C}, H) \Rightarrow \mathbb{M}} \text{ (TR-HEAPS)}$$

$$\frac{\begin{array}{l}\mathcal{A}; \mathbb{R} \vdash (\mathcal{C}, H) \Rightarrow \mathbb{M} \quad \mathcal{A} \vdash R \Rightarrow \mathbb{R} \quad \mathcal{A}_C \vdash I \Rightarrow \mathbb{W} \quad \mathsf{Flatten}(\mathbb{W}, \mathbb{M}, pc) \\ \exists l.\, \mathcal{C}(l) = \mathsf{code}\ [\Gamma].I' \ \wedge \ I \subseteq_{\mathit{tail}} I' \ \wedge \ pc = \mathcal{A}_C(l) + |I'| - |I| \\ \forall w \in Dom(\mathcal{E}).\, \mathsf{Flatten}(\mathsf{Fst}(\mathcal{E}(w)), \mathbb{M}, w) \\ \mathsf{CLytConsist}(\mathcal{A}_C, \mathcal{A}_D, \mathcal{C}, \mathcal{E})\end{array}}{\mathcal{E}; \mathcal{A} \vdash (\mathcal{C}, H, R, I) \Rightarrow (\mathbb{M}, \mathbb{R}, pc)} \text{ (TR-PROG)}$$

where

$$\begin{array}{rl}\mathsf{CLytConsist}(\mathcal{A}_C, \mathcal{A}_D, \mathcal{C}, \mathcal{E}) = & \mathsf{onetoone}(\mathcal{A}_C) \ \wedge \ Dom(\mathcal{A}_C) = Dom(\mathcal{C}) \\ & \wedge\, Dom(\mathcal{A}_C) \cap Dom(\mathcal{A}_D) = \emptyset \\ & \wedge\, \mathcal{C}(l) = \mathsf{code}\ [\Gamma].I \ \rightarrow \ \mathcal{A}_C(l) \notin Dom(\mathcal{E}) \\ & \wedge\, \mathcal{C}(l) = \mathsf{stub}\ [\Gamma].\emptyset \ \rightarrow \ \mathcal{A}_C(l) \in Dom(\mathcal{E})\end{array}$$

$$\begin{array}{rl}\mathsf{DLytConsist}(\mathcal{A}_D, H) = \forall l, l' \in Dom(H).\, (l \neq l' \ \wedge \ \mathcal{A} \vdash H(l) \Rightarrow \mathbb{W}) & \\ \rightarrow \ (\mathcal{A}_D(l) + \mathsf{length}(\mathbb{W}) < \mathcal{A}_D(l'))\end{array}$$

Figure 4.6: XTAL to CAP translation (2 of 2).

an appropriately translated sequence of words in memory. All XTAL instructions translate directly to a single machine command except newpair which translates to a series of commands that adjust the allocation pointer to make space for a new pair and then copy the initial values from r0 and r1 into the new space. The stubs in the XTAL code heap translation are handled in the top-level translation rule (when $\mathcal{E}$ is Flatten'ed).

The top-level rules (Figure 4.6) also enforce the set of necessary consistency constraints on the layout functions and external code blocks. CLytConsist ensures (1) that the code layout mapping is one-to-one,[6] (2) that the domains of the code heap and the code heap mapping are exactly the same, (3) the code heap and data heap labels are distinct, and (4) that the external code library does not provide code for those addresses that are already associated with XTAL blocks. DLytConsist simply ensures that all data heap entities will be mapped to non-overlapping sections of memory.

4.4.3 Generating the CAP Proofs

In this section I proceed in a top-down manner by first stating the main theorem we need to establish. The theorem says that for a given runtime system, any well-typed XTAL program that compiles and links to the runtime will result in an initial machine state that is well-formed according to the CAP typing rules. Applying Theorem 4.3, we would then be able to produce an FPCC package certifying the safety of the initial machine state.

Theorem 4.7 (XTAL-CAP Safety Theorem) For some specified external code environment $\mathcal{E}$, and for all $\mathcal{P}$ and $\mathcal{A}$, if $\vdash \mathcal{P}$ (in XTAL) and $\mathcal{E}; \mathcal{A} \vdash \mathcal{P} \Rightarrow \mathbb{S}$, then $\vdash_{\bullet} \mathbb{S}$ (in CAP, recalling that $\bullet$ indicates a trivial global safety predicate).

To prove that the CAP state is well-formed (using the (CAP-STATE) rule, Figure 4.2), we need a code heap specification, Φ, and a top-level precondition, P, for the command sequence at the current program counter. The code specification is generated as follows:

[6] Such a strong constraint might not be strictly necessary but it makes the proofs easier, because one does not have to worry about the case of two distinct XTAL code heap values, with potentially different types, mapping to the same CAP code.

$\Phi = \mathsf{CpGen}(\mathcal{E}, \mathcal{A}_C, \mathcal{C})$, where

$$\mathsf{CpGen}(\mathcal{E}, \mathcal{A}_C, \mathcal{C})(w) = \begin{cases} \mathsf{CpInv}(\mathcal{A}_C, \mathcal{C}, \Gamma) & \textit{if } w \notin \mathit{Dom}(\mathcal{E}) \textit{ and } \exists l. \mathcal{A}_C(l) = w \ \wedge \ \mathcal{C}(l) = (\mathsf{code}\ [\Gamma].I) \\ \mathsf{Snd}(\mathcal{E}(w)) & \textit{if } w \in \mathit{Dom}(\mathcal{E}) \end{cases}$$

That is, for external code blocks, the precondition comes directly from $\mathcal{E}$, while for code blocks that have been compiled from XTAL, the CAP preconditions are constructed by the following predicate generator:

$$\mathsf{CpInv}(\mathcal{A}_C, \mathcal{C}, \Gamma) = \lambda\mathbb{S}.\, \exists \mathcal{A}_D, \Psi, H, R.\ (\vdash \mathcal{C}) \ \wedge \ (\vdash H : \Psi) \ \wedge \ (\mathcal{C}; \Psi \vdash R : \Gamma) \\ \wedge\, (\mathcal{A}; \mathbb{S}.\mathbb{R} \vdash (\mathcal{C}, H) \Rightarrow \mathbb{S}.\mathbb{M}) \ \wedge \ (\mathcal{A} \vdash R \Rightarrow \mathbb{S}.\mathbb{R})$$

For any given program, the code heap and layout ($\mathcal{C}$ and $\mathcal{A}_C$) must be unchanged, therefore they are global parameters of these predicate generators. CpInv captures the fact that at a particular machine state there is a well-typed XTAL memory and register file that syntactically corresponds to it. We only need to specify the register file type as an argument to CpInv because the XTAL typing rules for the well-formed register file and heap will imply all the necessary restrictions on the data heap structure. One of the main insights of this work is the definition of CpInv, which allows us to both establish a syntactic invariant on CAP machine states as well as define the interface between XTAL and library code at the CAP level. CpInv is based on a similar idea as the global invariant defined in the previous chapter but instead of a generic, monolithic safety proof using the syntactic encoding of the type system, CpInv makes clear what the program-specific preconditions are for each command (instruction) and allows for relatively easy manipulation and reasoning thereupon, as well as interaction with other type system-based invariants.

Returning to the proof of Theorem 4.7, if we define the top-level precondition of the (CAP-STATE) rule to be $\mathsf{CpInv}(\mathcal{A}_C, \mathcal{C}, \Gamma)$, then it is trivially satisfied on the initial state $\mathbb{S}$ by the premises of the theorem. We now have to show well-formedness of the code at the current program counter, $\Phi \vdash_{\bullet} \{P\}\, \mathbb{C}$, and, in fact, proofs of the same judgment form must

be provided for each of the code blocks in the heap, according to the (CAP-CDSPEC) rule. The correctness of the CAP code memory is shown by the theorem:

Theorem 4.8 (XTAL-CAP Code Heap Safety) For a specified $\mathcal{E}$, and for any XTAL program state $(\mathcal{C}, H, R, I)$, layout functions $\mathcal{A}$, and machine state $(\mathbb{M}, \mathbb{R}, pc)$, such that $\vdash (\mathcal{C}, H, R, I)$ and $\mathcal{E}; \mathcal{A} \vdash (\mathcal{C}, H, R, I) \Rightarrow (\mathbb{M}, \mathbb{R}, pc)$, if $\Phi = \mathsf{CpGen}(\mathcal{E}, \mathcal{A}_C, \mathcal{C})$, then $\vdash_\bullet \mathbb{M} : \Phi$.

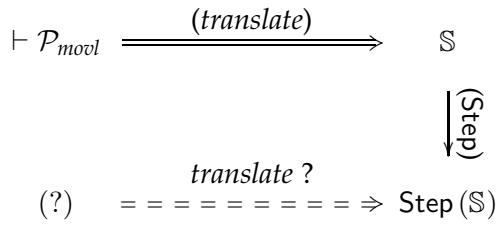
This depends in turn on the proof that each well-typed XTAL instruction sequence translated to machine commands will be well-formed in CAP under CpInv:

Theorem 4.9 (XTAL-CAP Instruction Safety) For a specified $\mathcal{E}$, and for all $\mathcal{A}_C$, $\mathcal{C}$, I, Γ, and $\mathbb{C}$ (where $\Phi = \mathsf{CpGen}(\mathcal{E}, \mathcal{A}_C, \mathcal{C})$), if *(a)* $\mathcal{C}; \Gamma \vdash I$ and *(b)* $\mathcal{A}_C \vdash I \Rightarrow \mathbb{C}$, then *(c)* $\Phi \vdash_\bullet \{\mathsf{CpInv}(\mathcal{A}_C, \mathcal{C}, \Gamma)\}\, \mathbb{C}$.

Proof sketch The proof proceeds by induction on I. I outline here the proof of three cases. The complete proof, especially the mechanized one, is of course somewhat involved so I try here to just cover the high-level, intuitive ideas.

Case movl $r_d, l; I'$: From (a) and inversion on the typing rule for movl we can derive that $\mathsf{typeof}(\mathcal{C}(l)) = \forall[\Gamma']$ and $\Psi; \Gamma\{r_d : \forall[\Gamma']\} \vdash I'$. Then from (b) and rule (TR-MOVL), we get $\mathcal{A}_C(l) = w$, $\mathcal{A}_C \vdash I \Rightarrow \mathbb{C}'$, and $\mathbb{C} = \texttt{movi}\ r_d, w; \mathbb{C}'$.

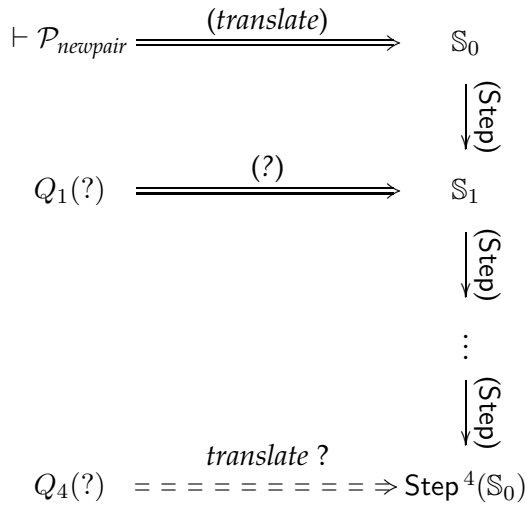
We can now apply (CAP-PURE) with $Q = \mathsf{CpInv}(\mathcal{A}_C, \mathcal{C}, \Gamma\{r_d : \forall[\Gamma']\})$. The premises of (CAP-PURE) require that we show three premises hold - that Q holds on $\mathsf{Step}(\mathbb{S})$, that $\mathsf{SP}(\mathbb{S})$, and that $\mathbb{C}'$ is well-formed under Q. The latter follows by the induction hypothesis. Our global safety predicate here is trivial ($\mathsf{SP} = \lambda\mathbb{S}.\mathsf{True}$). Thus, the remaining obligation is to show that given $\mathsf{CpInv}(\mathcal{A}_C, \mathcal{C}, \Gamma)(\mathbb{S})$ and the fact that the current CAP command is a movi command, then $Q(\mathsf{Step}(\mathbb{S}))$. The state of our proof here can be illustrated as follows:



That is, CpInv essentially captures the proposition that a well-formed XTAL program translates to a CAP state.[7] By the premise of this theorem, we have the top arrow in the diagram above, and by our instantiation of Q, we must provide an XTAL program (or at least the heap and register file) to fill in the question mark and show that it corresponds to the new CAP state. Just as in the previous chapter, the idea is to use the formalized XTAL soundness lemmas to obtain a well-typed XTAL program to fill in the question mark. Then, the diagram above becomes very similar to that of Figure 3.10. In this case, however, we don't explicitly care about the relationship between the two XTAL programs. This part of the proof now has the same structure as the Preservation proof described in Section 3.2.3.

Case newpair $r_d[\tau_0, \tau_1]; I'$: I will outline this case diagrammatically. Translating a single XTAL new pair instruction, we get a sequence of four machine commands. Thus, consider the following figure:

[7] Actually CpInv does not use the complete judgment $\vdash \mathcal{P}$, only the well-formedness of the heaps and register file, but I abuse the notation in these diagrams for the sake of conciseness.



As in the previous case, we are able to determine by the semantics of the XTAL new pair operation and the CAP commands implementing it that, in fact, the last predicate of the sequence can be

$$Q_4 = \mathsf{CpInv}(\mathcal{A}_C, \mathcal{C}, \Gamma\{r_d : \Gamma(\mathsf{r0}) \times \Gamma(\mathsf{r1})\})$$

because we have ended up in a state where the destination register is now pointing to a pair in memory. Notice that unlike the previous case, when we need to instantiate the new heap and heap type of the Q_4 predicate (expanding the definition of CpInv), they will now be instantiated with modified versions to reflect that the XTAL data memory now contains a new pair in it.

Now the question remains as to the intermediate pre/post-conditions. A simple way to deal with these is to thread along the relationship between $\mathcal{P}_{newpair}$ and the initial CAP state, keeping track of updates in each step. So, each of the predicates, for $i \in \{1, 2, 3\}$, will be of the form:

$$Q_i = \lambda\mathbb{S}_i.\ \exists\mathbb{S}.\ \mathsf{CpInv}(\mathcal{A}_C, \mathcal{C}, \Gamma)(\mathbb{S})\ \wedge\ \mathbb{S}_i = \mathsf{upd}_i(\mathbb{S})$$

At each step, I keep track of the changes made so far from the original CAP state that corresponded to a XTAL program with a current newpair instruction. The upd_i function,

describing the updates to the new state, can be simply generated from the strongest postcondition given the current CAP command. By the fourth command, the update function will have described the appropriate changes to memory and register file to allow the CAP machine state to once again correspond to the next well-formed XTAL program state and instruction sequence.

Case jd l: By examination of the translation rules, we know here that $\mathbb{C} = \texttt{jd}\ w$ where $\mathcal{A}_C(l) = w$. Also, by inversion on the XTAL typing rule (IS-JD), we get that $\mathsf{typeof}(\mathcal{C}(l)) = \forall[\Gamma']$. It follows therefore that either $\mathcal{C}(l) = \mathsf{code}\ [\Gamma'].I'$ for some I', or else $\mathcal{C}(l) = \mathsf{stub}\ [\Gamma'].\emptyset$. In the first case, we are jumping to other XTAL code; in the second, we are jumping to code external to the actual XTAL program. We have to apply the (CAP-JD) rule to prove the well-formedness of $\mathbb{C}$. This mainly requires showing that there exists some Q' to which $\Phi(w)$ is mapped[8] such that $Q'(\mathsf{Step}(\mathbb{S}))$.
In the first subcase, let $Q' = \mathsf{CpInv}(\mathcal{A}_C, \mathcal{C}, \Gamma')$. We know that
$\Phi(w) = \mathsf{CpInv}(\mathcal{A}_C, \mathcal{C}, \Gamma') = Q'$ by the definition of CpGen (page 88) and the fact that $\mathcal{A}_C(l) = w$ and $\mathcal{C}(l) = \mathsf{code}\ [\Gamma'].I'$. Since the Q' here has the form of a CpInv predicate, the reasoning for the remainder of this subcase follows the two previous instruction cases, where we are just filling in pieces of the diagram.
In the second subcase, we can determine from the translation (see the CLytConsist predicate in Figure 4.6) that $\mathcal{A}_C(l) \in Dom(\mathcal{E})$. Thus, let $Q' = \mathsf{Snd}(\mathcal{E}(w))$. Now, to show that Q' is satisfied by the jump command, we use the result of Proof Obligation 4.11 below. Proof Obligation 4.11 enforces that the XTAL type specification of an external code block is consistent with its actual precondition in $\mathcal{E}$. Thus, we are once again returned to showing that the CpInv predicate holds after the jump. Notice that CpInv does not require anything of the program counter or current instruction sequence so in this case of a jump command, $\mathsf{CpInv}(\mathcal{A}_C, \mathcal{C}, \Gamma')(\mathbb{S}) \rightarrow \mathsf{CpInv}(\mathcal{A}_C, \mathcal{C}, \Gamma')(\mathsf{Step}(\mathbb{S}))$.
Hence, one of the main components for each case of this theorem's proof is to show that

[8] Actually $\Phi(w)$ maps to a pair (n, Q') but I ignore the n here.

a correspondence similar to that of Figure 3.10 holds. □

Finally, establishing the theorems above depends on satisfying some proof obligations with respect to the external library code and its interfaces as specified at the XTAL level. First, we must show that the external library code is well-formed according to its supplied preconditions:

Proof Obligation 4.10 (External Code Safety) *For a given $\mathcal{E}$, if $\Phi = \mathsf{CpGen}(\mathcal{E}, \mathcal{A}_C, \mathcal{C})$ for any $\mathcal{A}_C$ and $\mathcal{C}$, then $\Phi \vdash_{\bullet} \{\mathsf{Snd}(\mathcal{E}(w))\}\,\mathsf{Fst}(\mathcal{E}(w))$, for all $w \in Dom(\mathcal{E})$.*

For now, we assume that the proofs of this lemma are constructed semi-automatically using the rules for well-formedness of CAP commands. This lemma is utilized in the proof of Theorem 4.8.

Secondly, when linking the external code with a particular XTAL program, where certain labels of the XTAL code heap are mapped to external code addresses, we have to show that the typing environment that would hold at any XTAL program that is jumping to that label implies the actual precondition of that external code:

Proof Obligation 4.11 (Interface Correctness) *For a given $\mathcal{E}$, $\mathcal{A}_C$, and $\mathcal{C}$, and for all l such that $\mathcal{C}(l) = \mathsf{stub}\ [\Gamma].\emptyset$ and $\mathcal{A}_C(l) = w$, if $\mathsf{CpInv}(\mathcal{A}_C, \mathcal{C}, \Gamma)(\mathbb{S})$ then $\mathsf{Snd}(\mathcal{E}(w))(\mathbb{S})$.*

These properties must be proved for each instantiation of the runtime system $\mathcal{E}$. With them, the proofs of Theorems 4.9, 4.8, and, finally, 4.7 can be completed.

4.4.4 arrayget Example

As a concrete example of the process discussed in the foregoing subsection, let us consider arrayget. The XTAL type interface is defined in Section 4.3.3. A CAP machine implementation of this function could be:

$$\mathbb{C}_{aget} = [\texttt{ld}\ \mathsf{r8}, \mathsf{r0}(0);\ \texttt{addi}\ \mathsf{r1}, \mathsf{r1}, 1;\ \texttt{bgt}\ \mathsf{r1}, \mathsf{r8}, \mathsf{bnderr};\ \texttt{add}\ \mathsf{r0}, \mathsf{r0}, \mathsf{r1};\ \texttt{ld}\ \mathsf{r0}, \mathsf{r0}(0);\ \texttt{jmp}\ \mathsf{r7}]$$

The runtime representation of an array in memory is a length field followed by the actual array of data. We assume that there is some exception handling routine for out-of-bounds accesses with a trivial precondition defined by $\mathcal{E}(\mathsf{bnderr}) = (\mathbb{C}_{bnderr}, Q_{bnderr})$.

Before describing the CAP assertions for the safety of $\mathbb{C}_{aget}$, notice that the code returns indirectly to an XTAL function pointer. Similarly, the arrayget address can be passed around in XTAL programs as a first-class code pointer. While the syntactic type system handles these code pointers quite easily using the relevant XTAL types, dealing with code pointers in a Hoare logic-based setup like CAP requires a little bit of machinery.

We can thus proceed to directly define the precondition of $\mathbb{C}_{aget}$ as,

$$Q_{aget} = \mathsf{CpInv}(\mathcal{A}_C, \mathcal{C}, \{\ \mathsf{r0}:\mathsf{array},\ \mathsf{r1}:\mathsf{int},\ \mathsf{r7}:(\forall[\{\mathsf{r0}:\mathsf{int}\}])\ \})$$

for some $\mathcal{A}_C$ and $\mathcal{C}$. Then we certify the library code in CAP by providing a derivation of $(\Phi \vdash_{\bullet} \{Q_{aget}\}\,\mathbb{C}_{aget})$. We do this by applying the appropriate rules from Figure 4.2 to track the changes that are made to the state with each command. When we reach the final jump to r7, we can then show that $\mathsf{CpInv}(\mathcal{A}_C, \mathcal{C}, \{\mathsf{r0}:\mathsf{int}\})$ holds, which must be the precondition specified for the return code pointer by $\Phi(\mathbb{S}.\mathbb{R}(\mathsf{r7}))$ (see the definition of Φ in the beginning of Section 4.4.3). The problem with this method of certifying arrayget, however, is that we have explicitly included details about the source language type system in its preconditions. In order to make the proof more generic, while at the same time be able to leverage the syntactic type system for certifying code pointers, I follow a similar approach as in [97]: First, I define generic predicates for the pre- and postconditions, abstracting over an arbitrary external predicate, P_{aget}. The actual requirements of the arrayget code are minimal (for example, that the memory area of the array is readable according to the safety policy). The post-condition predicate relates the state of the machine upon exiting the code block to the initial entry state:

$$\mathsf{Pre} = \lambda P_{aget}.\lambda \mathbb{S}.\ P_{aget}(\mathbb{S}) \ \wedge\ \mathsf{SafeToRead}(\mathbb{S}.\mathbb{M},\ \mathbb{S}.\mathbb{R}(\mathsf{r0}),\ \mathbb{S}.\mathbb{R}(\mathsf{r1})+1)$$

$$\mathsf{Post} = \lambda(\mathbb{M},\mathbb{R},pc).\ \lambda(\mathbb{M}',\mathbb{R}',pc').\quad \mathbb{M}' = \mathbb{M} \ \wedge\ pc' = \mathbb{S}.\mathbb{R}(\mathsf{r7})$$
$$\wedge\ \mathbb{R}'(\mathsf{r0}) = \mathbb{M}(\mathbb{R}(\mathsf{r0})+\mathbb{R}(\mathsf{r1})+1) \ \wedge\ \ldots$$

Now we certify the arrayget code block, quantifying over all P_{aget} and complete code specifications Φ, but imposing some appropriate restrictions on them:

$$\forall \Phi, P_{aget}.\ \Phi(\mathsf{bnderr}) = Q_{bnderr} \ \wedge\ (\forall \mathbb{S},\mathbb{S}'.\mathsf{Pre}(P_{aget})(\mathbb{S}) \ \wedge\ \mathsf{Post}(\mathbb{S})(\mathbb{S}') \ \rightarrow\ \Phi(\mathbb{S}.\mathbb{R}(\mathsf{r7}))(\mathbb{S}'))$$
$$\rightarrow \Phi \vdash_{\mathsf{SP}} \{\mathsf{Pre}(P_{aget})\}\, \mathbb{C}_{aget}$$

Thus, under the assumption that the Pre predicate holds, we can again apply the inference rules for CAP commands to show the well-formedness of the $\mathbb{C}_{aget}$ code. When we reach the final jump, we show that the Post predicate holds and then use that fact with the premise of the formula above to show that it is safe to jump to the return code pointer.

The arrayget code can thus be certified independent of any type system, by introducing the quantified P_{aget} predicate. Now, when we want to use this as an external function for XTAL programs, we instantiate P_{aget} with Q_{aget} above. We have to prove the premise of the formula above, $(\forall \mathbb{S},\mathbb{S}'.\mathsf{Pre}(Q_{aget})(\mathbb{S}) \ \wedge\ \mathsf{Post}(\mathbb{S})(\mathbb{S}') \ \rightarrow\ \Phi(\mathbb{S}.\mathbb{R}(\mathsf{r7}))(\mathbb{S}'))$. Proving this is not difficult, because we use properties of the XTAL type system to show that from a state satisfying the precondition– *i.e.* there is a well-formed XTAL program whose register file satisfies the arrayget type interface– the changes described by the Post predicate will result in a state to which there does correspond another well-formed XTAL program, one where the register r0 is updated with the appropriate element of the array.

Hence, we can let $\mathcal{E}(\mathsf{arrayget}) = (\mathbb{C}_{aget}, \mathsf{Pre}(Q_{aget}))$ and we have satisfied Proof Obligation 4.10. Proof Obligation 4.11 follows almost directly given the definition of Q_{aget}.

4.5 Summary

I have shown in this chapter the design of a typed assembly language supporting "foreign function" calls to external code. Programs written in this language will be safe if the external code operates according to the type specification. I have then shown how to certify the external (or runtime library) code independent of the source language. In order to handle code pointers, I simply assume their safety as a premise; then, when using the library with a particular source language type system, instantiate with a syntactic well-formedness predicate in the form of CpInv and use the facilities of the type system for checking code pointers to prove the safety of indirect jumps.

By associating typed assembly language programs with the compiled machine code state, in a purely syntactic manner, a large part of the safety proof has the same structure as the FPCC Preservation and Progress lemmas discussed in the previous chapter. However, by inserting the CAP specification language between the type system and the compiled machine code, I have been able to break the global invariant of the previous chapter into more local invariants, allowing greater control over integration with external code. Although I have not demonstrated the interoperation of code from two different type systems in the current work, the same approach can be used to achieve certified interoperability between code compiled from different high-level type systems.

In the next chapter, I describe an application of this approach to a more realistic type system – one that uses regions and capabilities to manage memory allocation and deallocation.

Chapter 5

A Certified Memory-Management Framework

Most, if not all, software systems need to provide facilities for memory reuse. In the previous chapters, we have seen languages which only support *allocation* of memory. Since the ideal machine I am working with has an infinite memory, this does not cause any problems. However, in real machines, the amount of memory is finite and there must be mechanisms to allocate areas of memory for use in a computation, and then *free* them for reuse after the computation is finished.

One way to categorize memory management schemes is into implicit and explicit frameworks. The latter would fit well with the languages I have presented in the earlier chapters and usually consists of having a runtime "garbage collector." In such a framework, the programmer only worries about allocating new memory. Whenever a request is made for a new object in memory, the runtime system checks to see if there is enough memory to satisfy the request. If there is not enough memory left, then the garbage collector looks through all the previously allocated objects to determine which of them are "garbage" – that is, unreachable from a set of root pointers (usually the registers). The collector then automatically frees the space being consumed by the unreachable objects and tries again to satisfy the request for a new object. This collection does not need to

take place only when an allocation request is made, and there are a number of strategies of when to run the collector [91] and how to manage used and free memory areas [92].

Adding a garbage collector to the XTAL language in the previous chapter, for instance, would require that an XTAL newpair instruction translate to a sequence of commands that check if there is enough memory for the new object, and if not then a call to a garbage collector would be made. Adding this to the XTAL runtime library would then require certifing the safety of the entire collector. While this is doable (as results by [10] indicate), it is quite an undertaking and I have chosen to take a simpler first step for this thesis.

An alternative to using implicit memory management schemes is to allow the programmer explicit control, through language constructs and the runtime system, over the allocation and deallocation of objects in memory. The C language offers this facility through "malloc" and "free" system calls. However, the C runtime system is generally not certified correct and certainly the C language type system does not ensure that these functions are used properly by programmers. One approach that has been developed to provide type safe memory management is that of region-based type systems [80, 81, 85]. In this chapter, I adopt a region-based memory management framework for the typed assembly languages presented in previous chapters. The actual primitives for managing regions will be provided as an external runtime library, while the type system of the language, RgnTAL, provides facilities for tracking the use of regions.

The type system I use for RgnTAL is essentially the capability calculus of Walker, *et al.* [85]. In the following sections, I will present the syntax and semantics, along with details of the Coq encoding. I will then discuss the translation from well-typed RgnTAL programs to certified CAP machine code, which follows the same structure as the discussion in Section 4.4. I will also give some details about the implementation of one of the region management primitives, which has been completely certified in Coq. At the time of writing this thesis, the other main primitive is still in the process of being certified correct using the proof assistant.

This chapter does not introduce any new features to the theoretical framework devel-

oped in the previous two chapters. What it does is demonstrate the technical details of a complete development using a more realistic example of a typed assembly language and runtime system.

5.1 Typed Assembly Language with Regions

5.1.1 RgnTAL Syntax

The syntax of my region-based typed assembly language is given in Figure 5.1. Note, this region-based TAL is not a new contribution, since [75, 85] have already designed such a system, although its details have not been published. However, as with FTAL, my contribution is the encoding of this system in a formal logic, along with the proofs of soundness, and the provision of a *certified* runtime library implementing the region-based memory management primitives. In previous works, the region primitives have still been part of the trusted computing base.

Terms. Beginning from the bottom of Figure 5.1, the abstract machine state of RgnTAL is made up of a static code memory and a dynamic program state. As in XTAL, the code memory maps code labels, f, to either code blocks (sequences of instructions) or stubs. The code types are now a bit more complex, to account for polymorphism and capability constraints, discussed in more detail below. The code memory does not change for the entire execution of a program.

The dynamic program state is made up of a data memory, register file, and current instruction sequence. The data memory is a finite mapping of region names, ν, to heap regions, where a region is a block of memory in which a collection of individual objects may be stored. Regions will be created at runtime using a `newrgn` library function and data will be allocated within regions using an `alloc` library function. Furthermore, a `freergn` library function will allow regions to be deleted from the data memory (effectively freeing up space in the data memory for reuse). In this version of the language, I have restricted the actual data that can be stored in regions (the data heap values) simply to pairs. Data

$$
\begin{array}{lll}
(\textit{kinds}) & \kappa & ::= \mathsf{Type} \mid \mathsf{Rgn} \mid \mathsf{Cap} \\
(\textit{constructors}) & c & ::= \tau \mid g \mid A \\
\quad(\textit{types}) & \tau & ::= \alpha \mid \mathsf{int} \mid g\ \mathsf{handle} \mid \langle \tau_1 \times \tau_2 \rangle\ \mathsf{at}\ g \mid \forall[\Delta](A,\Gamma) \mid \mu\alpha.\tau \\
\quad(\textit{regions}) & g & ::= \rho \mid \nu \\
\quad(\textit{capabilities}) & A & ::= \epsilon \mid \emptyset \mid \{g^1\} \mid \{g^+\} \mid A_1 \oplus A_2 \mid \overline{A} \\
(\textit{con. contexts}) & \Delta & ::= \cdot \mid \Delta, \alpha:\kappa \mid \Delta, \epsilon \leq A \\
(\textit{register file types}) & \Gamma & ::= \{\mathsf{r0}:\tau_0, \ldots, \mathsf{r7}:\tau_7\} \\
(\textit{region types}) & \Upsilon & ::= \{l_0:\tau_0, \ldots, l_n:\tau_n\} \\
(\textit{memory types}) & \Psi & ::= \{\nu_0:\Upsilon_0, \ldots, \nu_n:\Upsilon_n\} \\
(\textit{labels}) & l, f & ::= \mathbf{0} \mid \mathbf{1} \mid \ldots \\
(\textit{user registers}) & r & ::= \mathsf{r0} \mid \mathsf{r1} \mid \ldots \mid \mathsf{r7} \\
(\textit{word values}) & v & ::= i \mid \nu.l \mid f \mid \mathsf{handle}\ (\nu) \mid v[c] \mid \mathsf{fold}\, v\, \mathsf{as}\, \tau \\
(\textit{register file}) & R & ::= \{\mathsf{r0} \mapsto v_0, \ldots, \mathsf{r7} \mapsto v_7\} \\
(\textit{data heap values}) & h & ::= (v_1, v_2) \\
(\textit{heap region}) & H & ::= \{l_0 \mapsto h_0, \ldots, l_n \mapsto h_n\} \\
(\textit{data memory}) & \mathcal{D} & ::= \{\nu_0 \mapsto H_0, \ldots, \nu_n \mapsto H_n\} \\
(\textit{instructions}) & \iota & ::= \mathsf{add}\ r_d, r_s, r_t \mid \mathsf{addi}\ r_d, r_s, i \mid \mathsf{sub}\ r_d, r_s, r_t \mid \mathsf{subi}\ r_d, r_s, i \\
 & & \quad \mid \mathsf{mov}\ r_d, r_s \mid \mathsf{movi}\ r_d, i \mid \mathsf{movf}\ r_d, f \mid \mathsf{ld}\ r_d, r_s(i) \\
 & & \quad \mid \mathsf{st}\ r_d(i), r_s \mid \mathsf{bgt}\ r_s, r_t, f \mid \mathsf{bgti}\ r_s, i, f \mid \mathsf{tapp}\ r[c] \\
 & & \quad \mid \mathsf{fold}\ r[\tau] \mid \mathsf{unfold}\ r \\
(\textit{instr. sequences}) & I & ::= \iota; I \mid \mathsf{jd}\ f \mid \mathsf{jmp}\ r \\
(\textit{code heap values}) & \overline{h} & ::= \mathsf{code}\ [\Delta](A,\Gamma).I \mid \mathsf{stub}\ [\Delta](A,\Gamma).\emptyset \\
(\textit{code memory}) & \mathcal{C} & ::= \{f_0 \mapsto \overline{h}_0, \ldots, f_n \mapsto \overline{h}_n\} \\
(\textit{program}) & \mathcal{P} & ::= (\mathcal{D}, R, I)
\end{array}
$$

Figure 5.1: RgnTAL syntax.

is read from, or stored to, regions using the `ld` and `st` instructions.

The word values of the language are integers, data heap addresses given by a pair of a region name and label ($\nu.l$), code labels, region handles, type applications, and fold annotations for recursive data structures. Region names and handles are distinguished in order to maintain a phase distinction between compile-time and run-time entities. The names are important at compile-time for they are used to ensure statically that a given memory address is safe to load or store from. However, they have no run-time significance. On the other hand, region handles are the run-time data structures necessary to manipulate regions. The handles are used during allocation, to increment the region's allocation pointer, and when freeing a region, to return the region memory onto a free list. (Section 5.3.3 describes the underlying representation of region handles and the library functions that operate on them.)

Types. At the type level we now have, besides types of kind Type, two other kinds of constructors – regions and capabilities. The types include type constructor variables, the basic integer type, the type of region handles, pairs, code types, and recursive types. Notice that the type of a pair not only gives the type of its two elements but also the region in which the pair is stored. The code type, which can now be made polymorphic over types, regions, or capabilities, also specifies a set of region capabilities, A, which must be satisfied before jumping to the code block. I use the metavariables ρ and ϵ for variables of kind Rgn and Cap, respectively, and use the metavariable α for type variables and constructor variables in general. For convenience, the types, regions, and capabilities are merged into a single syntactic category of "constructors," distinguished by kinds.

Type contexts, Δ, keep track of quantified constructor variables as well as bounded quantification over capability sets. Register file types and heap region types map registers and labels, respectively, to types; and the data memory type keeps track of all the regions in memory. The data memory type maps region names to their types.

Capabilities. The real novelty in the region type system is the presence of capabilities. Capabilities, A, indicate the set of regions that are currently valid to access and also keep

track of aliasing information, or more accurately, *non*-aliasing or uniqueness. For details of how capabilities are used, I refer the reader to Walker's thesis [84] or [85].

In short, a set of capabilities contains a collection of valid region names, each one tagged with one of two multiplicities: $\{g^+\}$ simply indicates the capability to access region g, while $\{g^1\}$ adds the additional information that g is unique – that is, g is different from any other regions appearing in a set of capabilities formed using $\{g^1\}$. The uniqueness of a region is important for the purpose of freeing (deleting) a region from memory, because it ensures that there are no other region handles in use that are aliases of the one being freed. Providing the ability to safely free regions is the main purpose of having capabilities in the RgnTAL type system. Capabilities can also include variables ϵ which indicate access to an unspecified set of regions.

In order to get a feel for the use of capabilities, I summarize here a series of examples from [85], modified for the context of my particular version of the region type system. Suppose we have a function that simply needs to access data values stored in some regions. The type of such a code block could be something like:

$$\forall[\rho_1, \rho_2, \epsilon](\{\rho_1^+\} \oplus \{\rho_2^+\} \oplus \epsilon, \{\mathsf{r1} : \langle \mathsf{int} \times \mathsf{int} \rangle \ \mathsf{at}\ \rho_1,\ \mathsf{r2} : \langle \mathsf{int} \times \mathsf{int} \rangle \ \mathsf{at}\ \rho_2,\ \ldots\})$$

This function is parameterized over two regions (which may or may not be aliased) and an unspecified set of capabilities. When jumping to this code block, the region variables ρ_1 and ρ_2 may be instantiated with the same region name, or with different region names.

Now, if instead the code block needs to free one of the regions, it will require knowledge that the region variable is unique among the entire set of capabilities. If the region variable is aliased with another in the set of capabilities, the type system would not be able to prevent an illegal access to data in that region once the aliased region is freed. So, for example, suppose the function will want to free the region ρ_1 after processing its data.

The type must then specify uniqueness:

$$\forall[\rho_1, \rho_2, \epsilon](\{\rho_1^1\} \oplus \{\rho_2^+\} \oplus \epsilon, \{\ldots\})$$

In this case, it will not be possible to instantiate ρ_1 and ρ_2 with the same region name upon jumping to this code.

Suppose, instead, that the code block *creates* a new region and allocates data in it before jumping to its continuation. In that case, the type of the continuation will indicate this:

$$\forall[\epsilon](\epsilon, \{\ldots, \ \mathsf{r7} : \forall[\rho](\{\rho^1\} \oplus \epsilon, \{\mathsf{r0} : \rho \ \mathsf{handle}, \mathsf{r1} : \langle \mathsf{int} \times \mathsf{int} \rangle \ \mathsf{at} \ \rho, \ldots\})\})$$

Here, the continuation in r7 expects the creation of a new, unique region ρ, with the handle to that region and a pointer to some data allocated in the region in the registers r0 and r1.

Finally, suppose we hold a unique capability $\{\nu^1\}$ and a code block f has type:

$$\forall[\rho_1, \rho_2](\{\rho_1^+\} \oplus \{\rho_2^+\}, \{\ldots, \ \mathsf{r7} : \forall[](\{\rho_1^+\} \oplus \{\rho_2^+\}, \ldots)\})$$

Since f again does not care about aliasing between ρ_1 and ρ_2, we can instantiate both variables with ν. But what if we want to be able to recapture the uniqueness of the region variable in the continuation of f so that it can be freed later on? It would be too restrictive to change the strengthen the capability of the continuation in r7 to just $\{\rho_1^1\}$ because then f would not be able to call it.

We could try using a capability variable to recover the uniqueness information so that f is given type:

$$\forall[\rho_1, \rho_2, \epsilon](\epsilon, \{\ldots, \ \mathsf{r7} : \forall[](\epsilon, \ldots)\})$$

Now, we could instantiate ϵ with the unique capability we hold, $\{\nu^1\}$, and then f would be able to call its continuation and the continuation would have the unique capability necessary to free the region. Unfortunately, now f is unable to actually access any data

in the regions ρ_1 and ρ_2 because its own capability ϵ is completely abstract – there is no relationship between the region variables and ϵ. The solution to this problem is to use bounded quantification to relate ρ_1, ρ_2, and ϵ. Using this feature, we can give f the type:

$$\forall[\rho_1,\ \rho_2,\ \epsilon \leq \{\rho_1^+\} \oplus \{\rho_2^+\}](\epsilon, \{\ldots,\ \mathsf{r7} : \forall[](\epsilon, \ldots)\})$$

In this type, we can instantiate ρ_1 and ρ_2 with ν and ϵ with $\{\nu^1\}$. This instantiation works because $\{\nu^1\} \leq \{\rho_1^+\} \oplus \{\rho_2^+\}$ according to the definition of the subcapability relation. Furthermore, the continuation will possess the unique capability $\{\nu^1\}$ allowing it to free the region, and the body of f will be able to access data in the region through either of the handles, ρ_1 or ρ_2. In general, the bounded quantification allows one to hide some privileges when jumping to a code block, and then regain those privileges in the continuation. I have sketched here the intuitive use of regions, capabilities, and bounded quantification. The precise typing rules for the system will be given later in this chapter, but again, for complete details the reader is referred to the earlier works on this type system.

I have included a fuller example of a RgnTAL program to compute pairs of Fibonacci numbers in Appendix C.

Coq encoding

Figure 5.2 details the definition in the Coq logic of RgnTAL regions and capabilities. Regions are identified with natural numbers (defined inductively), and thus a decidable equality predicate can be defined for them. I then represent a set of region capabilities as a function from regions to multiplicities. To model a partial function, I define three multiplicities, one to indicate the lack of a capability (`noC`) and the other two as discussed in the previous section. I then encode the definition of capabilities in Figure 5.1 as appropriate function terms. There is, however, one complication. On paper, it is easy to implicitly require through definitions of equality and other type system rules that unique capabilities

```
(* regions and decidable equality *)
Definition rgn : Set := nat.
Definition beq_rgn : rgn -> rgn -> bool := ...

(* capability multiplicities and decidable equality *)
Inductive accap : Set := noC : accap | uniC : accap | mulC : accap.
Definition beq_accap : accap -> accap -> bool := ...

(* set of capabilities *)
Definition capset := rgn -> accap.

Definition nullcap : capset := fun r => noC.

Definition uniqcap : rgn -> capset
  := fun rk r => if (beq_rgn r rk) then uniC else noC.

Definition multcap : rgn -> capset
  := fun rk r => if (beq_rgn r rk) then mulC else noC.

Definition disjcap : capset -> capset -> Prop :=
  fun A1 A2 => forall p, (A1 p = noC) \/ (A2 p = noC).

Definition pluscap : forall (A1 A2:capset), (disjcap A1 A2) -> capset :=
  fun A1 A2 D r => if (beq_accap (A1 r) noC) then (A2 r) else
                     if (beq_accap (A2 r) noC) then (A1 r) else noC.

Definition barcap : capset -> capset
  := fun A r => match (A r) with uniC => mulC | _ => (A r) end.
```

Figure 5.2: Coq encoding: RgnTAL regions and capabilities

are not duplicated. We generally need to avoid the situation of having a union of two capabilities takes place where each capability is for the same, unique region – $\{g^1\} \oplus \{g^1\}$.[1] I have found for my purposes that the most convenient way to work this in the Coq encoding, in order to keep the proofs simple, is to syntactically enforce that a union of two capability sets may not mention duplicate regions. Hence the definition of `disjcap`, which is used by `pluscap` to enforce this syntactic restriction. Every union of two capability sets, $A_1 \oplus A_2$, must be provided with a proof that the domains of the two sets A_1 and A_2 are disjoint.

Notice that in the Coq encoding I have defined capabilities such that the equalities which must be axiomatized in [85] are directly derivable in the logic. We shall see this in the section on RgnTAL static semantics.

The encoding of labels, registers, and RgnTAL types is shown is Figure 5.3. Here, again, labels are identified with natural numbers, and registers are defined as an appropriate inductive definition. In the inductive definition of type constructors (`omega`), integer, region handle, and pair types are defined directly. The encoding of code types, however, deviates somewhat from the presentation in Figure 5.1.[2] To begin with, I separate the type constructors for polymorphism and the code type. The code type constructor (`tcode`) simply specifies a capability constraint and the expected type of the register file. I then have a number of constructors which provide polymorphism over the various kinds of the type system. Region and capability polymorphism is handled using a form of higher-order abstract syntax (HOAS).[3] However, due to constraints on the shape of inductive definitions in the Coq logic, one cannot define a constructor of the type `(omega -> omega) -> omega`, which would be the HOAS encoding of type abstraction. Hence, I have had to use a first-order encoding (deBruijn indices) of polymorphism over constructors of kind `Type`. The deBruijn index encoding of types utilizes a mirror image definition of `omega`, called

[1] Again, to avoid lengthening this thesis beyond its scope, I refer the reader to [84, 85] for a discussion of why this must be avoided.

[2] The existence of these discrepancies and the lack of adequacy in the encoding is discussed later in Section 6.3.

[3] See Section 6.2.

```
Definition label := nat.

Inductive regt : Set := r0 : regt | r1 : regt | ... | r7 : regt.

Inductive omega : Set :=
  | tint    : omega                                  (* int *)
  | thandle : rgn -> omega                           (* p handle *)
  | tpair   : omega -> omega -> rgn -> omega         (* t1 x t2 at p *)
  | tcode   : capset -> (regt -> omega) -> omega     (* code A,G *)

  | tabsr   : (rgn -> omega) -> omega                (* \/ p:Rgn. t *)
  | tabsc   : (capset -> omega) -> omega             (* \/ c:Cap. t *)
  | tabscb  : (capset -> omega) -> capset -> omega (* bounded poly over cap *)
  | tabscd  : forall (c1 c2:capset), ((disjcap c1 c2) -> omega) -> omega
  | tabst   : (omegaV 1) -> omega                    (* \/ t:Type. t' *)

  | trec    : (omegaV 1) -> omega                    (* \mu t:Type. t' *).

Inductive constr : Set :=
  | c_rgn : rgn -> constr
  | c_cap : capset -> constr
  | c_disj : forall c1 c2, disjcap c1 c2 -> constr
  | c_type : omega -> constr.

Definition rftype : Set := regt -> omega.
Definition rgntype : Set := fmap label omega.
Definition memtype : Set := fmap rgn rgntype.
```

Figure 5.3: Coq encoding: RgnTAL types

omegaV, which may have free type variables in the body of type constructors. The encoding keeps track of the maximum number of free type variables using a dependent type. Hence at the top level of a type abstraction (tabst), only one free type variable may be introduced, as captured by the definition in Figure 5.3. The details of the deBruijn encoding will be described in Section 6.2. Recursive types (trec) also use the deBruijn index notation for type variables.

The motivation for having two versions of omega is that during all the proof developments related to the soundness of RgnTAL and its translation to certified CAP code, we are always working at the top-level of typing derivations so there are never any free type variables in the context. Hence, the proofs are much simpler to manage using the top-level definition of omega defined in Figure 5.3, as opposed to the definition of omegaV with variables and lift terms described in Section 6.2, where we would continuously have to eliminate the applicability of those cases.

At the bottom of Figure 5.3 are defined the constructors of the type system (region names, capabilities, disjunction of capability sets, and types), as well as the register file, region, and data memory types. fmap is defined as

```
Definition fmap : Set -> Set -> Set = fun A B => A -> option B
```

and used to encode partial functions. I have developed a complete library for reasoning about partial functions using this definition.

Encoding the remainder of RgnTAL syntax is fairly straightforward and is shown in Figure 5.4. One major difference from the presentation in Figure 5.1 is that there are separate instructions and word values for handling the different kinds of type application. Furthermore, having defined the basic form of instruction sequences, iseq, the ciseq definition allows them to be parameterized over a set of constructor variables, corresponding to the code type in the syntax.

```
Inductive wordval : Set :=
  | wi     : nat -> wordval               (* i *)
  | wl     : rgn -> label -> wordval      (* p.l *)
  | wf     : label -> wordval             (* codeptr(f) *)
  | wh     : rgn -> wordval               (* handle(p) *)
  | wappr  : wordval -> rgn -> wordval    (* v[p] *)
  | wappc  : wordval -> capset -> wordval (* v[A] *)
  | wappcd : forall (c1 c2:capset),
             wordval -> (disjcap c1 c2) -> wordval
  | wappt  : wordval -> omega -> wordval  (* v[t] *)
  | wfold  : wordval -> omega -> wordval  (* fold v as t *).

Inductive instr : Set :=
  | iadd  : regt -> regt -> regt -> instr
  | iaddi : regt -> regt -> nat -> instr
  | imov  : regt -> regt -> instr
  | imovi : regt -> nat -> instr
  | imovf : regt -> label -> instr
  | ild   : regt -> regt -> nat -> instr
  | ist   : regt -> nat -> regt -> instr
  | ibgt  : regt -> regt -> label -> instr
  | ibgti : regt -> nat -> label -> instr
  | iappr  : regt -> rgn -> instr
  | iappc  : regt -> capset -> instr
  | iappcd : forall c1 c2, regt -> (disjcap c1 c2) -> instr
  | iappt  : regt -> omega -> instr
  | ifold   : regt -> omega -> instr
  | iunfold : regt -> instr.

Inductive iseq : Set :=
  | icons    : instr -> iseq -> iseq
  | ijd      : label -> iseq
  | ijmp     : regt -> iseq.

Inductive ciseq : Set :=
  | cibase : iseq -> ciseq
  | ciabsr : (rgn -> ciseq) -> ciseq
  | ciabsc : (capset -> ciseq) -> ciseq
  | ciabsd : forall c1 c2, ((c1 |-| c2) -> ciseq) -> ciseq
  | ciabst : (omega -> ciseq) -> ciseq.

Inductive codeval : Set :=
  | cvcode : omega -> ciseq -> codeval
  | cvstub : omega -> codeval.

Inductive heapval : Set :=
  | hvpair : wordval -> wordval -> heapval.

Definition heap := fmap label heapval.    (* heap regions *)
Definition datamem := fmap rgn heap.
Definition regfile := regt -> wordval.
Definition codemem := fmap label codeval.
Definition progstate := datamem * regfile * iseq.
```

Figure 5.4: Coq encoding: RgnTAL abstract machine

$\mathcal{C} \vdash (\mathcal{D}, R, I) \longmapsto \mathcal{P}$ where	
if $I =$	then $\mathcal{P} =$
add $r_d, r_s, r_t; I'$	$(\mathcal{D}, R\{r_d \mapsto R(r_t) + R(r_s)\}, I')$
addi $r_d, r_s, i; I'$	$(\mathcal{D}, R\{r_d \mapsto R(r_s) + i\}, I')$
sub $r_d, r_s, r_t; I'$	$(\mathcal{D}, R\{r_d \mapsto R(r_t) - R(r_s)\}, I')$
subi $r_d, r_s, i; I'$	$(\mathcal{D}, R\{r_d \mapsto R(r_s) - i\}, I')$
mov $r_d, r_s; I'$	$(\mathcal{D}, R\{r_d \mapsto R(r_s)\}, I')$
movi $r_d, i; I'$	$(\mathcal{D}, R\{r_d \mapsto i\}, I')$
movf $r_d, f; I'$	$(\mathcal{D}, R\{r_d \mapsto f\}, I')$
ld $r_d, r_s(i); I'$	$(\mathcal{D}, R\{r_d \mapsto v_i\}, I')$ where $R(r_s) = \nu.l$, $i \in \{0, 1\}$, and $\mathcal{D}(R(r_s)) = (v_0, v_1)$
st $r_d(i), r_s; I'$	$(\mathcal{D}\{R(r_s) + i \mapsto h'\}, R, I')$ where $R(r_d) = \nu.l$, $\mathcal{D}(R(r_d)) = (v_0, v_1)$, and $h' = (R(r_s), v_1)$ if $i = 0$ or $h' = (v_0, R(r_s))$ if $i = 1$
bgt $r_s, r_t, f; I'$	$(\mathcal{D}, R, I')$ when $R(r_s) \leq R(r_t)$; and $(\mathcal{D}, R, \mathcal{C}(f))$ when $R(r_s) > R(r_t)$
bgti $r_s, i, f; I'$	$(\mathcal{D}, R, I')$ when $R(r_s) \leq i$; and $(\mathcal{D}, R, \mathcal{C}(f))$ when $R(r_s) > i$
tapp $r_d[c]; I'$	$(\mathcal{D}, R\{r_d \mapsto R(r_d)[c]\}, I')$
fold $r_d[\tau]; I'$	$(\mathcal{D}, R\{r_d \mapsto \mathsf{fold}\, R(r_d)\, \mathsf{as}\, \tau\}, I')$
unfold $r_d; I'$	$(\mathcal{D}, R\{r_d \mapsto v\}, I')$ where $R(r_d) = \mathsf{fold}\, v\, \mathsf{as}\, \tau$
jd f	$(\mathcal{D}, R, I')$ where $\mathcal{C}(f) = \mathsf{code}\, [\Delta](A, \Gamma).I'$
jmp r	$(\mathcal{D}, R, I')$ where $\mathcal{C}(R(r)) = \mathsf{code}\, [\Delta](A, \Gamma).I'$

Figure 5.5: Operational semantics of RgnTAL.

5.1.2 Dynamic Semantics

The operational semantics of RgnTAL is defined in Figure 5.5. There is very little difference from FTAL or XTAL semantics of previous chapters. Note that I use the notation $\mathcal{D}(\nu.l)$ as an abbreviation for address lookup $\mathcal{D}(\nu)(l)$. Because the region management primitives for allocating and freeing regions will be provided as external library functions, the base instruction set and dynamics semantics of RgnTAL is quite simple.

Coq encoding

The encoding of RgnTAL operational semantics is completely straightforward. For each row of Figure 5.5, there is a constructor of the inductive definition:

Judgment	Meaning
$\Delta \vdash c_1 = c_2 : \kappa$	constructor equality
$\Delta \vdash A_1 \leq A_2$	subcapability relation
$\mathcal{C}; \Psi \vdash v : \tau$	well-formed word value
$\mathcal{C}; \Psi \vdash h$ at $\nu : \tau$	well-formed data heap value in region
$\mathcal{C}; \Psi \vdash H$ at $\nu : \Upsilon$	well-formed heap region
$\mathcal{C} \vdash \mathcal{D} : \Psi$	well-formed data memory
$\mathcal{C}; \Psi \vdash R : \Gamma$	well-formed register file
$\mathcal{C}; \Delta; A; \Gamma \vdash I$	well-formed instruction sequence
$\mathcal{C} \vdash \overline{h}$	well-formed code heap value
$\vdash \mathcal{C}$	well-formed code memory
$\Psi \vdash A$ sat	memory type–capability satisfiability
$\mathcal{C}; \Psi; A; \Gamma \vdash (\mathcal{D}, R)$	well-formed program state
$\mathcal{C} \vdash (\mathcal{D}, R, I)$	well-formed program

Figure 5.6: Static judgments of RgnTAL.

```
Inductive rt_eval : codemem -> progstate -> progstate -> Prop :=
  | ev_iadd
    : forall CM DM R IS rd rs rt s t,
      let i:=(iadd rd rs rt) in
        let R':=(rf_upd R rd (wi (plus s t))) in
          (R rs)=(wi s) ->
          (R rt)=(wi t) ->
          (rt_eval CM (DM, R, (icons i IS)) (DM, R', IS))
  | ...
```

5.1.3 Static Semantics

As in previous chapters, the static semantics of RgnTAL is defined by a series of typing judgments, summarized in Figure 5.6. Since I am encoding the definition of RgnTAL within the Coq logic, I delegate the judgment of constructor equality to the built-in equality of the logic. However, since I use a HOAS-inspired encoding for some of the type constructors and partial functions for the capability sets (see definitions of `capset` in Figure 5.2 and `omega` in Figure 5.3), I must also provide for an extensional definition of equality on those constructors. For instance,

$$\frac{A_1(g) = A_2(g), \textit{ for all } g}{\Delta \vdash A_1 = A_2 : \mathsf{Cap}}$$

```
Definition eqcap (A1 A2:capset) : Prop := forall g, (A1 g)=(A2 g).
```

$$\frac{\tau_1 = \tau_1}{\Delta \vdash \tau_1 = \tau_2 : \mathsf{Type}} \qquad \frac{\Delta \vdash A_1 = A_2 : \mathsf{Cap} \qquad \Delta \vdash \Gamma_1(r) = \Gamma_2(r) : \mathsf{Type}, \textit{ for all } r}{\Delta \vdash \forall[\cdot](A_1, \Gamma_1) = \forall[\cdot](A_2, \Gamma_2) : \mathsf{Type}} \qquad \cdots$$

```
Inductive eqtype : omega -> omega -> Prop :=
  | eqrefl : forall t1, eqtype t1 t1
  | eqcode : forall A1 A2 G1 G2,
             eqcap A1 A2 ->
             (forall r, eqtype (G1 r) (G2 r)) ->
             eqtype (tcode A1 G1) (tcode A2 G2)
  ⋮
```

The definition of `eqtype` contains more constructors which handle the various cases of abstraction over constructors. With the above definitions of equality and my encoding of capability sets, the various equality rules that are axiomatized in Walker's presentation [85] are derivable in my encoding. These rules are listed in Figure 5.7. Briefly, the equality rules allow one to rearrange the structure of a capability set and to duplicate capabilities of multiplicity $+$. We use these rearrangements within typing rules for jumping to a code block in order to match the current capability with the specification of the code block. Another necessary feature is the ability to lift unique capabilities ($\{g^1\}$) to duplicatable ones ($\{g^+\}$). The former should provide all the privileges of the latter, although they are not the same. Thus, the region type system specifies a subtyping relation which allows uniqueness information to be "forgotten." In my Coq encoding, I define subtyping on multiplicities and use that to define the subcapability relation:

$$\frac{}{\vdash 1 \leq 1} \qquad \frac{}{\vdash + \leq +} \qquad \frac{}{\vdash 1 \leq +}$$

```
Inductive subaccap : accap -> accap -> Prop :=
  | subaccap_refl : forall c, subaccap c c
  | subaccap_mult : subaccap uniC mulC.
```

$$\frac{\vdash A_1(g) \leq A_2(g), \textit{ for all } g}{\Delta \vdash A_1 \leq A_2}$$

$$\frac{\Delta \vdash c : \kappa}{\Delta \vdash c = c : \kappa}\ (\text{EQ-REFLEX}) \qquad \frac{\Delta \vdash c_2 = c_1 : \kappa}{\Delta \vdash c_1 = c_2 : \kappa}\ (\text{EQ-SYMM})$$

$$\frac{\Delta \vdash c_1 = c_2 : \kappa \qquad \Delta \vdash c_2 = c_3 : \kappa}{\Delta \vdash c_1 = c_3 : \kappa}\ (\text{EQ-TRANS})$$

$$\frac{\Delta \vdash A_1 = A_1' : \mathsf{Cap} \qquad \Delta \vdash A_2 = A_2' : \mathsf{Cap}}{\Delta \vdash A_1 \oplus A_2 = A_1' \oplus A_2' : \mathsf{Cap}}\ (\text{EQ-CONGR-PLUS})$$

$$\frac{\Delta \vdash A = A' : \mathsf{Cap}}{\Delta \vdash \overline{A} = \overline{A'} : \mathsf{Cap}}\ (\text{EQ-CONGR-BAR}) \qquad \frac{\Delta \vdash A : \mathsf{Cap}}{\Delta \vdash \emptyset \oplus A = A : \mathsf{Cap}}\ (\text{EQ-}\emptyset)$$

$$\frac{\Delta \vdash A_1 : \mathsf{Cap} \quad \Delta \vdash A_2 : \mathsf{Cap}}{\Delta \vdash A_1 \oplus A_2 = A_2 \oplus A_1 : \mathsf{Cap}}\ (\text{EQ-COMM})$$

$$\frac{\Delta \vdash A_i : \mathsf{Cap}, \textit{ for } 1 \leq i \leq 3}{\Delta \vdash (A_1 \oplus A_2) \oplus A_3 = A_1 \oplus (A_2 \oplus A_3) : \mathsf{Cap}}\ (\text{EQ-ASSOC})$$

$$\frac{\Delta \vdash A : \mathsf{Cap}}{\Delta \vdash \overline{A} = \overline{A} \oplus \overline{A} : \mathsf{Cap}}\ (\text{EQ-DUP}) \qquad \frac{}{\Delta \vdash \overline{\emptyset} = \emptyset : \mathsf{Cap}}\ (\text{EQ-BAR-}\emptyset)$$

$$\frac{\Delta \vdash g : \mathsf{Rgn}}{\Delta \vdash \overline{\{g^1\}} = \{g^+\} : \mathsf{Cap}}\ (\text{EQ-FLAG}) \qquad \frac{\Delta \vdash A : \mathsf{Cap}}{\Delta \vdash \overline{\overline{A}} = \overline{A} : \mathsf{Cap}}\ (\text{EQ-BAR-IDEM})$$

$$\frac{\Delta \vdash A_1 : \mathsf{Cap} \quad \Delta \vdash A_2 : \mathsf{Cap}}{\Delta \vdash \overline{A_1 \oplus A_2} = \overline{A_1} \oplus \overline{A_2} : \mathsf{Cap}}\ (\text{EQ-DISTRIB})$$

Figure 5.7: RgnTAL static semantics: equality (derivable rules).

```
Definition subcap : capset -> capset -> Prop
  := fun A1 A2 => forall p, subaccap (A1 p) (A2 p).
```

Once again, the set of subcapability rules that are axiomatized in [85, Fig. 5] can be easily derived from my encoding.

The next five judgment forms in Figure 5.6 cover the rules for typechecking data values and memory. The rules for these are given in Figure 5.8. The rule for integers is trivial, and data pointers are given the type assigned to the address in memory. (V-PAIR) allows pointers to be given a type even after their region has been deallocated. Although such dangling pointers can exist in a program, the type system will not allow them to be dereferenced. The rule for region handles ties together the dynamic (run-time) and static (compile-time) entities. Code pointers are given their type as assigned in the code memory specification. The rules for constructor application and folding a recursive type are standard, with (V-SUB) taking into account the bounded quantification over capabilities. The data heap only contains pairs and thus we only have one rule for heap values. Regions, the data memory, and the register file all make sure that their respective elements are well-typed.

The remaining RgnTAL typing rules are given in Figure 5.9. Many of the instruction typing rules are the same as our previous flavors of TAL, such as the (I-MOV) rule. One difference is that the store and load instructions must make sure that there exists a capability to read or write from a tuple in memory. Also the rules for jumps will make sure that the capability specification matches the current capability set, as well as checking that the register file specifications are compatible. Finally, RgnTAL has rules for constructor application. The (I-APPCB) handles the case of bounded quantification over capabilities. The typing rules for instructions are used to type code values in the code heap. As before, we simply assume at this stage that all stubs are well-typed.

The top-level rules bring together the well-formedness of the code memory, data memory, and register file, along with a judgment that a capability A satisfies the data memory type Ψ. The (SAT) rule ensures that the capabilities being used to type check memory

$\boxed{\mathcal{C};\Psi \vdash v : \tau}$

$$\frac{}{\mathcal{C};\Psi \vdash i : \mathsf{int}}\ (\text{V-INT}) \qquad \frac{\Psi(\nu.l) = \tau}{\mathcal{C};\Psi \vdash \nu.l : \tau}\ (\text{V-ADDR})$$

$$\frac{\nu \notin Dom(\Psi)}{\mathcal{C};\Psi \vdash \nu.l : \langle \tau_1 \times \tau_2 \rangle\ \mathsf{at}\ \nu}\ (\text{V-PAIR})$$

$$\frac{}{\mathcal{C};\Psi \vdash \mathsf{handle}\ (\nu) : \nu\ \mathsf{handle}}\ (\text{V-HNDL}) \qquad \frac{\mathsf{typeof}(\mathcal{C}(f)) = \tau}{\mathcal{C};\Psi \vdash f : \tau}\ (\text{V-CODE})$$

$$\frac{\mathcal{C};\Psi \vdash v : \forall[\Delta, \alpha:\kappa](A,\Gamma) \qquad \vdash c : \kappa}{\mathcal{C};\Psi \vdash v[c] : \forall[\Delta](A,\Gamma)[c/\alpha]}\ (\text{V-TYPE})$$

$$\frac{\mathcal{C};\Psi \vdash v : \forall[\Delta, \epsilon \leq A''](A',\Gamma) \qquad \Delta \vdash A \leq A''}{\mathcal{C};\Psi \vdash v[A] : \forall[\Delta](A',\Gamma)[A/\alpha]}\ (\text{V-SUB})$$

$$\frac{\mathcal{C};\Psi \vdash v : \tau[\mu\alpha.\tau/\alpha]}{\mathcal{C};\Psi \vdash \mathsf{fold}\ v\ \mathsf{as}\ \mu\alpha.\tau : \mu\alpha.\tau}\ (\text{V-FOLD})$$

$\boxed{\mathcal{C};\Psi \vdash h\ \mathsf{at}\ \nu : \tau}$

$$\frac{\mathcal{C};\Psi \vdash v_1 : \tau_1 \qquad \mathcal{C};\Psi \vdash v_2 : \tau_2}{\mathcal{C};\Psi \vdash (v_1, v_2)\ \mathsf{at}\ \nu : \langle \tau_1 \times \tau_2 \rangle\ \mathsf{at}\ \nu}\ (\text{H-PAIR})$$

$\boxed{\mathcal{C};\Psi \vdash H\ \mathsf{at}\ \nu : \Upsilon}$

$$\frac{Dom(H) = Dom(\Upsilon) \quad \mathcal{C};\Psi \vdash H(l_i)\ \mathsf{at}\ \nu : \Upsilon(l_i) \quad \textit{for all } l_i \in Dom(\Upsilon)}{\mathcal{C};\Psi \vdash H\ \mathsf{at}\ \nu : \Upsilon}\ (\text{REGION})$$

$\boxed{\mathcal{C} \vdash \mathcal{D} : \Psi}$

$$\frac{Dom(\mathcal{D}) = Dom(\Psi) \qquad \vdash \Psi \quad \mathcal{C};\Psi \vdash \mathcal{D}(\nu_i)\ \mathsf{at}\ \nu_i : \Psi(\nu_i) \quad \textit{for all } \nu_i \in Dom(\Psi)}{\mathcal{C} \vdash \mathcal{D} : \Psi}\ (\text{DMEM})$$

$\boxed{\mathcal{C};\Psi \vdash R : \Gamma}$

$$\frac{\mathcal{C};\Psi \vdash R(r) : \Gamma(r) \quad \textit{for all } r}{\mathcal{C};\Psi \vdash R : \Gamma}\ (\text{REGFILE})$$

Figure 5.8: RgnTAL static semantics: Data values, memory, and register file.

$\boxed{\mathcal{C}; \Delta; A; \Gamma \vdash I}$

$$\frac{\mathcal{C}; \Delta; A; \Gamma\{r_d \mapsto \Gamma(r_s)\} \vdash I}{\mathcal{C}; \Delta; A; \Gamma \vdash \mathsf{mov}\ r_d, r_s; I}\ (\text{I-MOV})$$

$$\frac{\begin{array}{ll}\Gamma(r_d) = \langle \tau_1 \times \tau_2 \rangle\ \mathsf{at}\ g & \Gamma(r_s) = \tau_i \\ \Delta \vdash A \leq A' \oplus \{g^+\} & \mathcal{C}; \Delta; A; \Gamma \vdash I \\ i \in \{0,1\} & \end{array}}{\mathcal{C}; \Delta; A; \Gamma \vdash \mathsf{st}\ r_d(i), r_s; I}\ (\text{I-ST})$$

$$\frac{\Gamma(r) = \forall[\Delta, \epsilon \leq A''](A', \Gamma') \quad \Delta \vdash A_0 \leq A'' \quad \mathcal{C}; \Delta; A; \Gamma\{r : \forall[\Delta](A', \Gamma')[A_0/\epsilon]\} \vdash I}{\mathcal{C}; \Delta; A; \Gamma \vdash \mathsf{tapp}\ r[A_0]; I}\ (\text{I-APPCB})$$

$$\frac{\Gamma(r) = \forall[](A', \Gamma') \qquad \Delta \vdash \Gamma = \Gamma' \qquad \Delta \vdash A \leq A'}{\mathcal{C}; \Delta; A; \Gamma \vdash \mathsf{jmp}\ r}\ (\text{I-JMP})$$

$\boxed{\mathcal{C} \vdash \overline{h}}$ $\qquad \dfrac{\mathcal{C}; \Delta; A; \Gamma \vdash I}{\mathcal{C} \vdash \mathsf{code}\ [\Delta](A, \Gamma).I}\ (\text{C-CODE})$ $\qquad \dfrac{}{\mathcal{C} \vdash \mathsf{stub}\ [\Delta](A, \Gamma).\emptyset}\ (\text{C-STUB})$

$\boxed{\vdash \mathcal{C}}$ $\qquad \dfrac{\mathcal{C} \vdash \mathcal{C}(f) \qquad \textit{for all } f \in Dom(\mathcal{C})}{\vdash \mathcal{C}}\ (\text{CMEM})$

$\boxed{\Psi \vdash A\ \mathsf{sat} \qquad \mathcal{C}; \Psi; A; \Gamma \vdash (\mathcal{D}, R) \qquad \mathcal{C} \vdash (\mathcal{D}, R, I)}$

$$\frac{\Delta \vdash A = \{\nu_0{}^{\varphi_0}\} \oplus \ldots \oplus \{\nu_n{}^{\varphi_n}\} : \mathsf{Cap} \qquad \nu_0, \ldots, \nu_1\ \textit{distinct}}{\{\nu_0 : \Upsilon_0, \ldots, \nu_n : \Upsilon_n\} \vdash A\ \mathsf{sat}}\ (\text{SAT})$$

$$\frac{\vdash \mathcal{C} \quad \mathcal{C} \vdash \mathcal{D} : \Psi \quad \mathcal{C}; \Psi \vdash R : \Gamma \quad \Psi \vdash A\ \mathsf{sat}}{\mathcal{C}; \Psi; A; \Gamma \vdash (\mathcal{D}, R)}\ (\text{STATE})$$

$$\frac{\mathcal{C}; \Psi; A; \Gamma \vdash (\mathcal{D}, R) \qquad \mathcal{C}; \cdot; A; \Gamma \vdash I}{\mathcal{C} \vdash (\mathcal{D}, R, I)}\ (\text{PROG})$$

Figure 5.9: RgnTAL static semantics: Instructions (selected), code, and top-level rules.

accesses accurately reflect the regions that exist in memory.

For each judgment form, the rules in Figures 5.8 and 5.9 are encoded in the Coq logic using inductively defined predicates. Thus we have, corresponding to the judgments in Figure 5.6:

```
wf_wordval : codemem -> memtype -> wordval -> omega -> Prop
wf_heapval : codemem -> memtype -> heapval -> rgn -> omega -> Prop
wf_heap    : codemem -> memtype -> heap -> rgn -> rgntype -> Prop
wf_datamem : codemem -> datamem -> memtype -> Prop
wf_regfile : codemem -> memtype -> regfile -> rftype -> Prop

wf_iseq    : codemem -> capset -> rftype -> iseq -> Prop
wf_codeval : codemem -> codeval -> Prop
wf_codemem : codemem -> Prop

sat_cap_memtype : memtype -> capset -> Prop
wf_state : codemem -> memtype -> capset -> rftype -> datamem -> regfile -> Prop
wf_program : codemem -> progstate -> Prop
```

Notice that constructor contexts are implicit (in the typing judgment for instruction sequences) because I use a HOAS-style encoding for constructor variables other than types; thus RgnTAL variables are represented using metavariables of the logic. For types, as mentioned earlier and described in Section 6.2, I use a lifted version of the type encoding that utilizes deBruijn indices to track type variables.

5.2 Soundness

As for XTAL, I cannot prove a complete soundness theorem for the RgnTAL type system with the presence of stub values in the code heap. However, for each individual instruction, I prove the standard progress and preservation lemmas. That is, given a well-typed RgnTAL program, it is possible to execute the current instruction, resulting in another well-typed program state. For example, the Coq statements of these lemmas in the case of a simple move instruction are:
If $\mathcal{C}; \Psi; A; \Gamma \vdash (\mathcal{D}, R)$ and $\mathcal{C}; \Delta; A; \Gamma \vdash \mathsf{mov}\ r_d, r_s; I$, then there exists a program $\mathcal{P}$ such that $\mathcal{C} \vdash (\mathcal{D}, R, I) \longmapsto \mathcal{P}$.

```
Lemma progress_imov
  : forall CM M R MT A G rd rs Is,
    let curIs := (icons (imov rd rs) Is) in
      wf_state CM MT A G M R ->
      wf_iseq CM A G curIs ->
      exists P, rt_eval CM (M, R, curIs) P.
```

If $\mathcal{C}; \Psi; A; \Gamma \vdash (\mathcal{D}, R)$, $\mathcal{C}; \Delta; A; \Gamma \vdash$ mov $r_d, r_s; I$, and $\mathcal{C} \vdash (\mathcal{D}, R, I) \longmapsto (\mathcal{D}', R', I')$, then there exists some data memory type, Ψ', such that $\mathcal{C}; \Psi'; A; \Gamma\{r_d : \Gamma(r_s)\} \vdash (\mathcal{D}', R')$.

```
Lemma preserv_imov
  : forall CM M R MT A G rd rs Is M' R' Is',
    let curIs := (icons (imov rd rs) Is) in
    let P := (M', R', Is') in
      wf_state CM MT A G M R ->
      wf_iseq CM A G curIs ->
      rt_eval CM (M, R, curIs) P ->
      exists MT',
        wf_state CM MT' A (rft_upd G rd (G rs)) M' R'.
```

I have generalized the statements of these lemmas so that they have about the same form for every instruction. In the case of mov above, we could actually prove the preservation lemma using $\Psi' = \Psi$. The instructions that deal with memory are more complicated to prove, as are the instructions involving type application, but overall the formalized proofs in Coq are essentially the same as the soundness proofs presented in [85], after taking into account differences in my Coq encoding.

5.3 Compilation and Runtime Library

5.3.1 Specifying the Translation of Programs to Machine States

The compilation of well-typed RgnTAL programs to CAP with safety proofs is somewhat more complicated that the compilation of XTAL in the previous chapter, mainly because of the reasoning about memory that is needed. In order to keep the memory reasoning tractable, I have encoded the primitives of separation logic [67] in Coq, along with a library of derived lemmas. Some of the basic operators that I use are listed in the following table. (In this table, "memory" refers to a partial function mapping integer addresses to integer words. In the Coq encoding, separation logic assertions have the type `mempred`.)

Sep. log. primitive	My Coq encoding	Describes
emp	`emptymp`	an empty memory area
$e \mapsto e'$	`e |-> e'`	memory containing one cell at address e which contains e'
$e \mapsto \_$	`e |-> ?`	memory containing one cell at address e which contains some unspecified value
$p * q$	`p && q`	memory that can be split into two disjoint parts in which p and q hold respectively
$\exists a : b.p$	`'ex fun a:b => p`	existential quantification

To compile RgnTAL programs into a CAP machine state, I divide the machine memory into three parts: (1) an area for external code, beginning at address `ec_min` and of size `ec_size`, where the implementation of the external code (runtime library functions) will be, (2) an area for the compiled program code, at address `cm_min` and of size `cm_size`, and (3) an area for program data, at address `dm_min` and of size `dm_size`, which may expand dynamically. In the next few subsections, I describe the contents of each of these areas.

Data Memory Area

The data area stores the translated RgnTAL data memory containing regions. It will keep track of region allocation using a simple memory management scheme involving a free list of regions. Within each region, allocation of data (RgnTAL pairs) will be handled by a simple linear allocation scheme that keeps track of the current allocation pointer within the region. Each RgnTAL region corresponds to a block of memory with a three word header. In C-like syntax, a region block is defined as:

```
Definition mempred := memarea -> Prop.

Fixpoint ptrtoanylist (a sz:word) struct sz : mempred :=
  match sz with | 0 => emptymp
                | (S n) => ((a |-> ?) && (ptrtoanylist (a+1) n))
  end.

Definition freeblock (a sz nxt:word) : mempred :=
  `(a <> nilptr) &
  ((a |-> sz) && (((a+1) |-> ?) && (((a+2) |-> nxt) && (ptrtoanylist (a+3) sz)))).

Definition newblock (a sz:word) : mempred :=
  `(a <> nilptr) &
  ((a |-> sz) && (((a+1) |-> 0) && (((a+2) |-> ?) && (ptrtoanylist (a+3) sz)))).

Fixpoint fblist (n a:word) struct n : mempred :=
  match n with | 0 => `(a=nilptr) & emptymp
               | (S n') => `(a<>nilptr)
                                & (`ex fun sz => `ex fun a' =>
                                    ((freeblock a sz a') && (fblist n' a')))
  end.

Definition freelist (a:word) : mempred := `ex fun n => (fblist n a).
```

Figure 5.10: Coq definitions of region blocks and free list.

```
struct rgnblock {
  int size;       // size of the block, excluding header
  int ap;         // allocation pointer within the region
  rgnblock* next; // next region in the free list
  int* data;      // array of [size] integers containing data values (the
                  //   translated RgnTAL heap values)
}
```

Empty space in the data memory and deallocated regions are kept track of in a free list structure using the `next` pointer of the region block. The Coq encoding of the `rgnblock` data structure and free list is given in Figure 5.10.

The `ptrtoanylist` predicate describes a block of memory pointed to by address `a` and of size `sz` which contains arbitrary data values. This definition is used by `freeblock`, which adds on the header fields for such a free region. The `newblock` predicate is what is returned upon allocation of a new region; notice that its next field is unspecified and its allocation pointer field is set to 0. `fblist` describes a linked list of `n` free blocks, and by existential quantification over the number of free blocks we obtain the definition of the

freelist.

The free list will describe the unallocated portions of data memory. In order to describe the allocated portion of data memory, I first define, in Figure 5.11, the translation from RgnTAL word values to machine words (integers). Then tr_heapval will translate a RgnTAL heap value (pair) and its location in the RgnTAL data memory into a separation logic assertion. The tr_dataptr function maps a RgnTAL pointer into the corresponding machine address and will be constructed appropriately during the compilation process by the compiler. tr_dataptr is the equivalent of the $\mathcal{A}_D$ function for XTAL compilation in Section 4.4.2. The next function, tr_hvs, translates a RgnTAL region (H:heap) containing a number of heap values (pairs) into a separation logic assertion describing how the region will be laid out in the machine memory. Since I model the RgnTAL heap regions as functions from integer labels to heap values, I make sure the functions are a finite mapping by insuring that they have a greatest label in their domain (Lim). Then I recurse from the value of Lim down to zero to define a terminating function which will process all the heap values in the range of H. Finally, the definition of tr_heap at the bottom of the figure details the complete translation from a RgnTAL region to the actual rgnblock data structure defined earlier. It uses another layout function, RLyt, which maps RgnTAL region names to addresses in memory. The three-word header information is laid out, as well as checks to ensure that the allocated and unallocated amount of data add up to the total region size. The allocation pointer is set to be exactly the number of allocated words in the region.

The compilation of RgnTAL instructions to CAP machine commands is fairly simple. The first two definitions in Figure 5.12 illustrate this. A RgnTAL instruction sequence corresponds to a list of CAP commands as defined by tr_iseq. This definition is then used by tr_codeval to translate RgnTAL code values to a sequence of words (encoded instructions), which are appropriately laid out in memory by tr_codemem. The coversfnat predicate ensures that the memory area described by the tr_codemem assertion starts at address cm_min and extends for cm_size words.

```
Inductive tr_wordval : wordval -> word -> Prop :=
  | tr_wi    : forall n, tr_wordval (wi n) n                    (* Integer *)
  | tr_wl    : forall p l w,
               tr_dataptr p l = Some w ->
               tr_wordval (wl p l) w        (* RgnTAL pointer to address w *)
  | tr_wlnop : forall p l w,
               notindomf DMLyt p ->
               tr_wordval (wl p l) w                  (* Dangling pointer *)
  | tr_wf    : forall f w,
               fmaplook CMLyt f w ->
               tr_wordval (wf f) w                        (* Code pointer *)
  | tr_wappt : forall v t w,
               tr_wordval v w ->
               tr_wordval (wappt v t) w        (* Type application (type info
                                                          is erased) *)
  | ...  (* other cases of constructor application similar *)

Definition tr_heapval (p:rgn) (l:nat) (a:word) (hv:heapval) : mempred :=
  match hv with [v0,v1] => `ex fun w0 => `ex fun w1 =>
          `(tr_dataptr p l = (Some a)) & `(tr_wordval v0 w0) &
          `(tr_wordval v1 w1) & ( (a |-> w0) && ((a+1) |-> w1))
  end.

Fixpoint tr_hvs (p:rgn) (a:word) (Lim:word) (H:heap) struct Lim : mempred :=
  match Lim with | 0 => (`(nulldomf H) & (@emptyfp _ _))
                 | (S m) => match (H Lim) with
                                | None => tr_hvs p a m H
                                | Some hv => ((tr_heapval p Lim a hv) &&
                                        (tr_hvs p (2+a) m (fmapdelN _ H Lim)))
                            end
  end.

Definition tr_heap : rgn -> heap -> mempred :=
  fun p H => `ex fun a => `ex fun hplim => `ex fun rsize => `ex fun diff =>
      `(a <> nilptr) &                    (* region address cannot be null *)
      `(fmaplook RLyt p a) &                 (* map region name to address *)
      `(@limitf heapval H hplim) &             (* region has finite domain *)
      `((hvlist_size hplim H)+diff = rsize) &  (* allocated and unallocated
                                                          space in the region
                                                          add up to the size *)
      (((a |-> rsize) &&                         (* first header word: size *)
        (((a+1) |-> (hvlist_size hplim H)) &&    (* header word: alloc ptr *)
        ((a+2) |-> ?))) &&                        (* header word: next ptr *)
        ((tr_hvs p (a+3) hplim H) &&                 (* actual region data *)
        (ptrtoanylist (a+3+(hvlist_size hplim H)) diff))). (* unused space *)
```

Figure 5.11: Coq encoding of RgnTAL heap values and region translation.

```
Inductive tr_instr : cmdlist -> instr -> cmdlist -> Prop :=
  | tr_iadd  : forall Cs rd rs rt,
               tr_instr Cs (iadd rd rs rt) (add rd rs rt :: Cs)
  | tr_iaddi : forall Cs rd rs t,
               tr_instr Cs (iaddi rd rs t) (addi rd rs t :: Cs)
  | ...

Inductive tr_iseq : iseq -> cmdlist -> Prop :=
  | tr_icons : forall i Is cs Cs,
               tr_iseq Is Cs -> tr_instr Cs i cs -> tr_iseq (icons i Is) cs
  | tr_ijd   : forall f w, fmaplook CMLyt f w -> tr_iseq (ijd f) (jd w :: nil)
  | tr_ijmp  : forall r, tr_iseq (ijmp r) (jmp r :: nil).

Inductive tr_codeval : codeval -> wordlist -> Prop :=
  | tr_cvcode : forall G Is Cs Ws,
                tr_ciseq Is Cs -> Cs = map Dc Ws -> tr_codeval (cvcode G Is) Ws
  | tr_cvstub : forall G, tr_codeval (cvstub G) nil.

Definition tr_codemem  (CM:codemem) : mempred :=
  fun M => (coversfnat _ M cm_min (cm_min+cm_size)) /\
    (forall f cv, fmaplook CM f cv ->
        exists Ws, exists a, fmaplook CMLyt f a /\ tr_codeval cv Ws
                          /\ ((wordsinmem Ws a) && truemp) M).
```

Figure 5.12: Coq encoding of RgnTAL code values and code memory translation.

Finally, the top-level translation relations are specified by the definitions in Figure 5.13. `tr_datamem` brings together the translation of all the regions in the RgnTAL data memory (using an auxiliary definition, `tr_datamem_aux`, not shown) with the `freelist` specification to form a separation logic assertion describing all of the data area of machine memory (which starts at address `dm_min` and extends to size `dm_size`). Also, the first address in the data area points to the size of the entire data area, and the second word is a pointer to the `freelist` structure. The next two definitions handle the layout of external code in the appropriate area of the machine memory. `tr_memstate_aux` and `tr_memstate` describe the combined layout of the RgnTAL code and data memories. Notice also that they specify that address 0 in the machine memory is not used, and that addresses 1 and 2 are pointers to the beginning of code and data memory areas. With the register file and program counter translation, I obtain a complete translation predicate (`tr_program`) relating a RgnTAL program (`codemem` and `progstate`) to a Coq machine state (`state`).

```
Definition tr_datamem (DM:datamem) dm_min dm_size : mempred :=
    `ex fun dmlim =>
    `ex fun fp =>                                  (* free list pointer *)
      (fun M => coversfnat _ M dm_min (dm_min+dm_size)) &
      `(limitf _ DM dmlim) &
      ((dm_min |-> dm_size) &&
        (((dm_min+1) |-> fp) && ((tr_datamem_aux dmlim DM) && (freelist fp)))).

Definition extcode_in_ec : Prop :=
  (forall a Cs Pcs, ExtCode a = someT (Cs,*Pcs) ->
                        (ec_min <= a /\ a+(length Cs) < ec_min+ec_size)).

Definition tr_extcode : codemem -> mem -> Prop :=
  fun CM MM => exists M,
      (extcode_in_ec) /\
      (coversfnat _ M ec_min (ec_min+ec_size)) /\
      (forall f w G Is, fmaplook CMLyt f w -> fmaplook CM f (cvcode G Is)
                          -> ExtCode w = noneT _) /\
      (forall a Cs Pcs, ExtCode a = someT (Cs,*Pcs) -> cmdsinmem Cs a M) /\
      (appliesto M MM).

Definition tr_memstate_aux (CM:codemem) (DM:datamem) : mempred :=
  `ex fun dm_min => `ex fun dm_size =>
      `(eqdomf _ _ _ CMLyt CM) & `(eqdomf _ _ _ DMLyt DM)
    & (((0 |-> ?) && ((1 |-> cm_min) && ((2 |-> dm_min))))
    && ((tr_codemem CM) && (tr_datamem DM dm_min dm_size))).

Definition tr_memstate (CM:codemem) (DM:datamem) (MM:mem) : Prop :=
  exists M, tr_memstate_aux CM DM M /\ (appliesto M MM).

Definition tr_regfile (Rf:regfile) (R:rfile) : Prop :=
  forall r v w, Rf r = v -> R r = w -> tr_wordval v w.

Definition tr_pc (CM:codemem) (Is:iseq) (pc:word) : Prop :=
  exists f, exists G', exists Is', exists w, exists n,
    fmaplook CM f (cvcode G' Is') /\
    subseqis Is Is' n /\
    fmaplook CMLyt f w /\
    pc = n + w.

Inductive tr_program : codemem -> progstate -> state -> Prop :=
  | tr_prog :
      forall CM DM R Is MM RR pc,
        tr_extcode CM MM ->
        tr_memstate CM DM MM ->
        tr_regfile R RR ->
        tr_pc CM Is pc ->
        tr_program CM (DM,R,Is) (MM,RR,pc).
```

Figure 5.13: Coq encoding of RgnTAL memory, register file, and program translation.

5.3.2 Safety Policy, Invariants, and Proofs

For the system described in this chapter, I use a slightly more realistic safety policy when certifying the compiled code. It states that for any memory reads or writes, the address being accessed must lie within the data area of memory, or else the program counter must be in the external code area of memory. The reasoning is that RgnTAL code should only access memory within its data area, while the external code (runtime library functions) may access memory anywhere but it will have the burden then of proving that any writes to memory will maintain basic type safety. In Coq syntax,

```
Definition MySP (St:state) :=
  match St with (M,R,pc) =>
    let dm_min := (M 2) in
    let dm_size := (M dm_min) in
      match (curcmd St) with
        | ld rd rs n  => let a:=(R rs)+n in  dm_min < a < (dm_min+dm_size)
                         \/ (ec_min <= pc < (ec_min+ec_size))
        | st rd n rs  => let a:=(R rd)+n in  dm_min < a < (dm_min+dm_size)
                         \/ (ec_min <= pc < (ec_min+ec_size))
        | _ => True
      end
  end.
```

I also specify in Coq the generation of CAP code preconditions from RgnTAL code types and code memory specification. Thus, corresponding to XTAL's Cplnv (Section 4.4.3), I define the following for the RgnTAL system:

```
Definition cpinv (CM:codemem) (T:omega) : pred :=
  fun St =>
    match St with (MM,RR,pc) =>
      exists DLyt, exists RLyt,        (* data ptr and region ptr mappings *)
      exists DM, exists RF, exists MT, exists Ts, exists A, exists G,
        instcodetype T Ts (tcode A G) /\
        wf_state CM MT A G DM RF /\
        tr_memstate DLyt RLyt CMLyt cm_min cm_size CM DM MM /\
        tr_regfile DLyt RLyt CMLyt RF RR
    end.
```

As the invariant Cplnv is important to understand the connection between the RgnTAL type system and CAP predicates, it might be more convenient to view it in typeset form:

$$
\begin{aligned}
&\mathsf{CpInv}(\mathcal{A}_C, \mathcal{C}, \forall[\Delta](A', \Gamma')) \\
&\quad = \lambda\mathbb{S}.\, \exists \mathcal{A}_D, \mathcal{A}_R, \mathcal{D}, R, \Psi, \vec{c}, A, \Gamma.\ \forall[](A', \Gamma')[\vec{c}/\Delta] = \forall[](A, \Gamma) \\
&\qquad\qquad \wedge\ \mathcal{C}; \Psi; A; \Gamma \vdash (\mathcal{D}, R) \\
&\qquad\qquad \wedge\ \mathcal{A} \vdash (\mathcal{C}, \mathcal{D}) \Rightarrow \mathbb{S}.\mathbb{M} \\
&\qquad\qquad \wedge\ \mathcal{A} \vdash R \Rightarrow \mathbb{S}.\mathbb{R}
\end{aligned}
$$

This version of CpInv is very similar to the one for XTAL. One difference here is an additional layout function mapping region names to addresses, $\mathcal{A}_R$. Another difference is that RgnTAL code types support polymorphism. Thus, CpInv states that there must exist some list of constructors, $\vec{c}$ (or `Ts` in the Coq version above), with which the code type can be instantiated. The conjuncts on the lower three lines are the real essence of CpInv. Again, they state that CpInv holds on a machine state $\mathbb{S}$ if there exists some well-typed RgnTAL program such that the memory and register file components of that program correspond to the memory and register file of the machine state according to the translation relations defined in the previous section.

Given the definition of CpInv, I define the generation of the complete CAP code specification (CpGen) given an external code library, RgnTAL code memory, and code layout function, just as in Section 4.4.3. Thus, in Coq I define a predicate that describes the formation of an appropriate CAP code specification:

```
Definition iscpgen : extcodety -> codelyt -> codemem -> cdspec -> Prop
   := ...
```

Now, I prove the same sequence of safety theorems as for XTAL (Theorems 4.9, 4.8, 4.7). Among the lemmas that are used to develop these proofs are a set of lemmas, one for each RgnTAL instruction, that prove the preservation of CpInv over one step of execution. As a few representative examples,

```
Lemma cpinv_preserv_add :
  forall CM A G rd rs rt Is St
    (D0 : cpinv CM (tcode A G) St)
    (D1 : wf_iseq CM A G (icons (iadd rd rs rt) Is))
    (D2 : curcmd St = (add rd rs rt)),
    cpinv CM (tcode A (rft_upd G rd tint)) (Step St).

Lemma cpinv_preserv_ld :
  forall CM A G rd rs n Is St
    (D0 : cpinv CM (tcode A G) St)
    (D1 : wf_iseq CM A G (icons (ild rd rs n) Is))
    (D2 : curcmd St = (ld rd rs n)),
    exists t, exists t1, exists t2, exists g,
      (n = 0 /\ t = t1 \/ n = 1 /\ t = t2) /\
      G rs = tpair t1 t2 g /\
      cpinv CM (tcode A (rft_upd G rd t)) (Step St).

Lemma cpinv_preserv_appr :
  forall CM A G r p Is St
    (D0 : cpinv CM (tcode A G) St)
    (D1 : wf_iseq CM A G (icons (iappr r p) Is))
    (D2 : curcmd St = (mov r r)),
    exists Fr,
      G r = tabsr Fr /\
      cpinv CM (tcode A (rft_upd G r (Fr p))) (Step St).
```

These are directly used to prove the well-formedness of translation from RgnTAL instructions to CAP commands:

```
Lemma rgntal2wfcapcmds :
 forall ExtCode CMLyt CT CM Is T Ws Cs
   (D0 : iscpgen ExtCode CMLyt CT CM)                (* CT = CpGen(..., CM) *)
   (D1 : extcode_in_ec ExtCode ec_min ec_size)          (* ExtCode is valid *)
   (D2 : tr_codeval CMLyt (cvcode T Is) Ws)               (* Ac |- Is => Ws *)
   (D3 : Cs = map Dc Ws)
   (D4 : forall Ts A G, instcodetype T Ts (tcode A G)     (* CM; A; G |- Is *)
                                -> wf_iseq CM A G Is),
   WFCapCmds MySP CT (cpinv CM T) Cs.
```

That is, if a well-typed RgnTAL instruction sequence `Is` (with code type `T`) translates to a list of CAP commands `Cs`, then the list of commands is well-formed under the CAP code specification `CT` (generated from the RgnTAL code memory type `CM`) and precondition `cpinv CM T`.

Well-formedness of the individual compiled instruction sequences leads to well-formedness of the entire code memory (the `extcode_wf` premise corresponds to Proof Obligation 4.10 in the XTAL context),

```
Lemma rgntal2wfcapcdspec :
  forall ExtCode CMLyt DLyt RLyt CT CM DM R Is MM RR pc
    (D0:iscpgen ExtCode CMLyt CT CM)
    (D1:wf_program (CM, (DM,R,Is)))
    (D2:tr_program ExtCode DLyt RLyt CMLyt CM (DM,R,Is) (MM,RR,pc))
    (extcode_wf: forall f Cs P,
         ExtCode f = someT (Cs ,* P) ->
         forall CT', iscpgen ExtCode CMLyt CT' CM -> WFCapCmds MySP CT' P Cs),
    WFCapcdspec MySP MM CT.
```

which in turns leads to the final safety theorem,

```
Theorem rgntal2cap :
  forall ExtCode CMLyt DLyt RLyt CT CM DM R Is St
    (D0:iscpgen ExtCode CMLyt CT CM)
    (D1:wf_program (CM, (DM,R,Is)))
    (D2:tr_program ExtCode DLyt RLyt CMLyt CM (DM,R,Is) St)
    (extcode_wf: forall f Cs P,
         ExtCode f = someT (Cs ,* P) ->
         forall CT', iscpgen ExtCode CMLyt CT' CM -> WFCapCmds MySP CT' P Cs),
    WFCapstate MySP St.
```

This states that given a well-typed RgnTAL program that translates to a CAP machine state `St`, such that the necessary proof obligations are satisfied on external code, and the code specification is generated properly, we can prove that the CAP machine state is well-formed for the memory safety policy `MySP`.

5.3.3 Region-based Memory Management Library

The development in the previous section establishes an assembly language with a region-based type system but does not provide the primitives for creating and deleting regions, or for allocating new data values in a region. For that purpose, I design a runtime library that will link in with RgnTAL code providing the necessary external functions to support region management.

Figures 5.14 and 5.15 illustrate my implementation of region library in C-like syntax. The library provides three main functions and has three utility functions that are used by those. The primary functions of the library are: `freergn`, which takes a pointer to a region (`rgnblock*`) and returns it to the `freelist`; `newrgn`, which returns a new region ready to

```
// region data structure
struct rgnblock {
   int size;       // size of the block, excluding header
   int ap;         // allocation pointer within the region
   rgnblock* next; // next region in the free list
   int* data;      // array of [size] integers containing data values (the
                   //   translated RgnTAL heap values)
}

// data structure allocated within regions
struct pair {
   int fst, snd;
}

const int BASESIZE = 10;
const rgnblock* NULLPTR = 0;

rgnblock* freelist;     // assume it is appropriately initialized somehow
int dm_min, dm_size;    // same for these (defined in the previous section)

void freergn(rgnblock* p) {
   p.next = freelist;
   freelist = p;
}

rgnblock* newrgn() {
   if (freelist != NULLPTR) {
      rgnblock* p = freelist;
      freelist = freelist.next;
      return p;
   } else {
      rgnblock* p = morememem(BASESIZE);
      return p;
   }
}

pair* alloc(int v1, int v2, rgnblock* p) {
   if (p.ap > p.size) {       // check if there is space in the region
      rgnblock* p' = growrgn(p, p.size + p.size);
      return alloc(v1, v2, p');
   } else {
      pair* s = p + p.ap;
      p.ap = p.ap + 2;
      s.fst = v1;
      s.snd = v2;
      return s;
   }
}
```

Figure 5.14: Region library functions (C-like implementation)

```
rgnblock* morememem(int nsize) {
   rgnblock* p = dm_min + dm_size;         // end of the data memory area
   p.size = nsize;
   p.ap = 0;
   p.next = NULLPTR;
   dm_size += nsize + 3;               // increase the limit of data area
   return p;
}

rgnblock* growrgn(rgnblock* p, int nsize) {
   rgnblock* p' = morememem(nsize);
   p'.ap = p.ap;
   copyrgn(p, p', 0);
   freergn(p);
   return p';
}

void copyrgn(rgnblock* p, rgnblock* p', int elem) {
   if (elem >= p.ap) return;
   p'.data[elem] = p.data[elem];
   copyrgn(p, p', elem+1);
}
```

Figure 5.15: Region library utility functions (C-like implementation)

be used, taking it from the `freelist` if it is non-empty and otherwise allocating space for the region by adjusting the size of the data area of memory; and `alloc`, which creates a new pair of data in a region, again resizing the region as necessary if there is not enough space.

Notice that these functions are (albeit simplistic) realistic, efficient C implementations of a library. They do not introduce any extra checks for safety. The RgnTAL type system will guarantee proper use of the API; for example, that `free`'d region pointers are not passed to the `alloc` function. At the time of this writing, I am in the process of mechanizing the proofs of safety for this library. I will discuss in this section, therefore, only one of the functions, `freergn`, for which the formal Coq proofs have been almost entirely completed. The process of certifying the other functions in the library is similar to the description for `freergn` below although, of course, the details of the specifications and interface types will be different.

Certifying `freergn` in CAP

To begin with, I manually translate the `freergn` code in Figure 5.14 into a CAP code block. The result is the following command list:

```
Definition freergn_cmds : cmdlist :=
  (movi r8 2) ::       (* address 2 points to dm_min *)
  (ld r8 r8 0) ::      (* r8 := dm_min *)
  (ld r9 r8 1) ::      (* r9 := freelist *)
  (st r0 2 r9) ::      (* p.next := freelist *)
  (st r8 1 r0) ::      (* freelist := p *)
  (jmp r7).
```

In standard Hoare-triple notation, we would specify pre- and post-conditions for this set of commands as follows:

`{ Pre } freergn_cmds { Post };`

However, because CAP programs are written in continuation-passing style, and also because it is difficult in the simple Hoare logic system to handle the jump to a code pointer (the last command in the sequence), the actual process of certifying this code in CAP is a bit more convoluted. Specifically, corresponding to assertions of the Hoare-triple above, I specify a predicate on what constraints the CAP precondition of the `freergn_cmds` block should satisfy:

$$\texttt{freergn\_req}(\Phi, P_{free}) = \lambda\mathbb{S}.P_{free}(\mathbb{S}) \rightarrow \; Pre(\mathbb{S}) \wedge \\ \exists Q.(\Phi(\mathbb{S}.\mathbb{R}(\mathsf{r7})) = Q \; \wedge \; (\forall\mathbb{S}'.Post(\mathbb{S}') \rightarrow Q(\mathbb{S}')))$$

I will prove the safety of `freergn_cmds` using a quantified precondition, P_{free}, which must imply a concrete precondition, *Pre*, and which must also imply that there exists a return code pointer in r7 the precondition of which (Q) is implied by *Post*. The predicates *Pre* and *Post* are used to reason about the changes in memory and register file that are effected by the series of commands up till the jump. Such reasoning is pure first order reasoning, as we will see, and I use separation logic to define these two predicates and reason the intermediate steps. The higher-order quantification over P_{free} and the definition

of freergn_req then allows reasoning about the safety of the final jump while maintaining modularity of the library's safety proof (because it is not instantiated with a predicate particular to one type system).

Let us now examine the definitions of *Pre* and *Post*. *Pre* is defined in Coq as:

```
fun St => match St with ((MM,RR),pc) =>
   ec_min <= pc /\ (pc + (length freergn_cmds)) < ec_min+ec_size       (* 1 *)
   /\
   exists M, exists PmemA, exists PmemB,
   exists dmmin, exists dmsize, exists fp, exists rsize, exists nxt,
      (appliesto M MM) /\                                              (* 2 *)
      (PmemA &&                                                        (* 3 *)
       (2 |-> dmmin)) &&                                               (* 4 *)
       ((fun Md => (forall a, indomf Md a -> ~(iscodearea CT a 1))) &   (* 5 *)
        (fun Md => (coversfnat _ Md dmmin (dmmin+dmsize))) &           (* 6 *)
        (PmemB && (freeblock (RR r0) rsize nxt)                        (* 7 *)
               && (dmmin |-> dmsize)                                   (* 8 *)
               && ((dmmin+1) |-> fp)                                   (* 9 *)
               && (freelist fp))) M.                                  (* 10 *)
```

This specifies on line (1) that the entire code block of freergn_cmds should lie within the external code area. The remaining lines specify the required layout of memory when freergn is called. MM is the representation of the entire CAP machine memory (a function with infinite domain) while M is a finite mapping from addresses to words. The appliesto predicate (2) ensures that the contents of M and MM are consistent and then I use a separation logic assertion to specify the contents of M. M is broken into three disjoint portions. On one portion (3), an arbitrary predicate, PmemA, will hold and this part of memory will not be modified (PmemA will hold in the postcondition as well and the area of memory that it holds on is actually the external code and RgnTAL code areas). Next, address 2 in memory points to the beginning of the data area of memory. Lines (5) and (6) state that there is no code in this data area of memory and that it has size dmsize. The last four lines describe the parts of memory that freergn will actually manipulate. Within the data memory, there is a freelist (10) and freergn also expects a pointer to a rgnblock data structure in register r0 (7). The code may also need to access the size of the data area (8) and the pointer to the freelist (9). Finally, the remaining portions of the data memory will remain unchanged as described by another abstract predicate PmemB (7).

The postcondition of `freergn`, namely the state of memory and register file just before the final jump, is described by the following predicate, which takes the `PmemA`, `PmemB` predicates of the precondition as parameters and also the state of the register file at the beginning of the function. The postcondition specifies that the registers used by RgnTAL programs (the first 8 registers) will be unchanged by the end of this function. Also, memory will be mostly the same except that the region that was to be freed has been added to the `freelist` and therefore the `freeblock` assertion that appeared on line (7) above is gone:

```
fun PmemA PmemB RR =>
fun St' => match St' with ((MM',RR'),pc') =>
   (eqonregs talregs RR RR')
   /\
   exists M', exists dmmin, exists dmsize, exists fp,
      (appliesto M' MM') /\
      (PmemA && (2 |-> dmmin)) &&
      ((fun Md => (forall a, indomf Md a -> ~(iscodearea CT a 1))) &
       (fun Md => (coversfnat _ Md dmmin (dmmin+dmsize))) &
       (PmemB && (dmmin |-> dmsize)
              && ((dmmin+1) |-> fp)
              && (freelist fp))) M'.
```

Putting all the pieces together, then, we have the complete definition of `freergn_req` in Figure 5.16. Now, the `freergn` code block can be certified safe as a Coq lemma stating that for any precondition `Pfree` satisfying the constraints of `freergn_req`, the command sequence will be well-formed:

Lemma 5.1 (Well-formedness of `freergn_cmds`)

```
Lemma freergn_wfcap : forall CT Pfree,
    freergn_req CT Pfree -> WFCapCmds MySP CT Pfree freergn_cmds.
```

Proving this lemma just involves application of the appropriate CAP inference rules in Figure 4.2. For most commands the postcondition (Q in the inference rules) is simply generated by computing the strongest postcondition based on the operational semantics of the CAP machine. For example, the postcondition of the first command, `movi r8 2`, is:

```
fun (St2:state) => match St2 with ((MM2,RR2),pc2) =>
   MM2 = MM /\ RR2=(updatereg RR r8 2) /\ pc2=(S pc) end.
```

```
(* postcondition requirements for the final jump *)
Definition freergn_jmp_req (CT:cdspec) (PmemA PmemB:mempred) RR dmmin dmsize
  : Prop :=
  exists n, exists Q, CT(RR r7) = someT (n,*Q) /\
    forall (M:fmap word word) (MM:mem) RR' fp,
        (eqonregs talregs RR RR') /\
        (appliesto M MM) /\
        (PmemA && (2 |-> dmmin)) &&
        ((fun Md => (forall a, indomf Md a -> ~(iscodearea CT a 1))) &
         (fun Md => (coversfnat _ Md dmmin (dmmin+dmsize))) &
         (PmemB && (dmmin |-> dmsize)
                && ((dmmin+1) |-> fp)
                && (freelist fp))) M
        -> Q(MM,RR',RR r7).

(* constraints on the precondition of freergn_cmds *)
Definition freergn_req : cdspec -> pred -> Prop :=
  fun CT Pfree =>
    forall St, Pfree(St) ->
      match St with ((MM,RR),pc) =>
        ec_min <= pc /\ (pc + (length freergn_cmds)) < ec_min+ec_size
        /\
        exists M, exists PmemA, exists PmemB,
          exists dmmin, exists dmsize, exists fp, exists rsize, exists nxt,
            (appliesto M MM) /\
            (PmemA && (2 |-> dmmin)) &&
            ((fun Md => (forall a, indomf Md a -> ~(iscodearea CT a 1))) &
             (fun Md => (coversfnat _ Md dmmin (dmmin+dmsize))) &
             (PmemB && (freeblock (RR r0) rsize nxt)
                    && (dmmin |-> dmsize)
                    && ((dmmin+1) |-> fp)
                    && (freelist fp))) M /\
            (freergn_jmp_req CT PmemA PmemB RR dmmin dmsize)
      end.
```

Figure 5.16: Complete pre- and post-condition specification for `freergn`

By the time we have reached the second store command, just before the jump, the state of the memory has been modified in such a way that it is possible to show that the postcondition *Post* described above is satisfied. Then, to show that the jump is safe, we use the fact that *Post* holds, along with the definition of `freergn_req`, to satisfy the premise of the (CAP-JMP) rule in Figure 4.2.

The RgnTAL Interface to `freergn`

Now that the `freergn` function has been certified in CAP we must define the interface through which RgnTAL programs can access it. This involves giving the function a RgnTAL type, which is fairly easy to do:

$$\texttt{freergn\_type} = \forall[\rho, \epsilon, \alpha_1, \ldots, \alpha_6](\{\rho^1\} \oplus \epsilon,\ \{\mathsf{r0}:\rho\ \mathsf{handle}, \mathsf{r1}:\alpha_1, \ldots, \mathsf{r6}:\alpha_6, \mathsf{r7}:\forall[\alpha_0, \alpha_7](\epsilon, \{\mathsf{r0}:\alpha_0, \mathsf{r1}:\alpha_1, \ldots \mathsf{r7}:\alpha_7\})$$

`freergn` in RgnTAL expects register r0 to contain a pointer to a region for which a unique capability is held ($\{\rho^1\}$) and register r7 should contain a return continuation which expects region ρ to have been removed from the capability set. Having defining the RgnTAL code type, I now instantiate the `Pfree` predicate of the previous section using the code invariant generator, CpInv (see its definition on page 125):

$$\texttt{Pfree} = \mathsf{CpInv}(\mathcal{A}_C, \mathcal{C}, \texttt{freergn\_type})$$

When proving well-formedness of a complete RgnTAL program that links to `freergn`, we will need to apply Lemma 5.1, which in turn requires showing that this definition of `Pfree` satisfies the constraints of `freergn_req`. A final Coq lemma therefore is:

```
Definition Pfree (CMLyt:heaplyt) (CM:codemem) : pred :=
   fun St => (cpinv CMLyt CM freergn_type).

Lemma freergn_type_satisfies_req : forall ExtCode CMLyt CT CM,
   iscpgen ExtCode CMLyt CT CM -> freergn_req CT (Pfree CMLyt CM).
```

5.4 Summary

A language with certified region management operations is the first step towards a runtime system with garbage collection. One of the most common sources of program error and security holes today is code that accesses and manages memory improperly. In this chapter, I have shown how to integrate the memory-safety properties provided by a high-level type system with low-level proofs of runtime system correctness. By continuing this line of research it may eventually be possible to eliminate most or all of the security breaches that occur due to faulty memory management. The details in this chapter may seem overwhelming but in fact the approach is fairly straightforward. As I am applying tools (*e.g.* the CiC calculus and Coq proof assistant) in a domain where they have not been used much before, the notation and development is still primitive. Yet the progress so far is encouraging and opens up future research areas in streamlining the production and presentation of such systems for integrating verified code.

Chapter 6

Tools and Techniques

During the process of mechanizing the framework described in the preceding chapters, I (and others working on related research) have encountered a variety of issues– some interesting, some disappointing– with which we have had to deal. In this chapter, I touch on a few of the more major hurdles that we have addressed using an assortment of tools and techniques. I also point out some of the insights gained in reasoning about the safety of our machine code.

6.1 Proof Development and Automation

One of the major challenges of this research is producing the necessary formalized proofs of safety. For instance, as mentioned earlier, the region-based runtime library of Chapter 5 has not been completely formalized in Coq yet. There is no apparent difficulty in the theoretical aspects but the main issue is the technical detail, time, and tedium required for building the proofs.

I have adopted the Coq proof assistant, which implements CiC– the logic on which my framework is based. One of the primary motivating factors for the use of CiC as opposed to other frameworks (such as those mentioned in Related Work, Chapter 7) has been its strong support for inductive definitions, making it very convenient to encode

language syntax and develop syntactic proofs of soundness. Nonetheless, the Coq tool has been developed and used primarily for purposes other than the proof-carrying code framework described in this thesis. Thus, its level of automation for the sort of reasoning I have been using it is somewhat primitive.

Coq provides, in addition to basic concrete syntax for CiC terms and types (*i.e.* the logical language), a tactic language, Ltac [79], which allows interactive development of proof terms. Coq comes equipped with some built-in tactics and also allows user-programmable tactics. Let us examine a proof script for a very small lemma, one that states that if a data heap value is well-typed in RgnTAL, then its type will not be a code type (because only pairs are stored in the data heap). In Coq, we state the lemma and then start the proof script by performing case analysis on the heap value `hv`:

```
Lemma wf_heapval_not_tcode
  : forall CM MT hv p t A G,
    (wf_heapval CM MT hv p t) -> t <> (tcode A G).
Proof.
induction hv; intros.
```

Coq returns to us the following state at this point (the terms above the double line are our premises or derived propositions and our goal is to prove that which is below the line):

```
1 subgoal

  CM : codemem
  MT : memtype
  w : wordval
  w0 : wordval
  p : rgn
  t : omega
  A : capset
  G : regt -> omega
  H : wf_heapval CM MT [w, w0] p t
  ============================
   t <> tcode A G
```

Here we can perform backwards reasoning on the `H` hypothesis and the typing rule for pairs to determine that `t` must be a pair type:

```
inversion H; auto.
```

The proof state becomes:

```
...
H6 : tpair t1 t2 p = t
============================
 tpair t1 t2 p <> tcode A G
```

Based on injectivity of constructors of an inductive definition, we can prove this inequality using a built-in tactic:

```
discriminate. Qed.
```

Notice that for terms which one does not specify a name, Coq introduces them into the proof context with automatically generated names, like `H`, `H0`, `H1`, ... The tactics then refer to these hypotheses by the introduced name. When developing large proofs that have many of these automatically generated names, the proof script may contain references to a number of hypotheses `H`i. During the development, one might discover the need for an additional hypothesis, or delete an unnecessary one. This causes the order of introduced names to change and the tactic commands in the proof script are all referring to the wrong names, requiring one to go through and update the entire proof script manually. This is a major problem and I have addressed it by trying to redefine the tactics so that one would refer to the judgment `wf_heapval` in the proof script instead of the arbitrary name `H`. Hence, instead of `inversion H` above, I use `mcinv wf_heapval` (`mc` for match-in-the-context). Not only does this relieve one from editing the entire proof script if the order of hypotheses change, it makes the proof script slightly more intelligible.

Writing such a tactic as `mcinv` is not hard in Coq but it seems to be awkward given the tactic language design. Coq provides a general matching construct that allows pattern matching on terms in the goal or (hypotheses) context. However, it does not support matching on curried application so the definition of `mcinv` ends up looking like the following:

```
Ltac mcinv t :=
  match goal with
  | H:t |- _ => inversion H; auto
  | H:(t _) |- _ => inversion H; auto
  | H:(t _ _) |- _ => inversion H; auto
  | H:(t _ _ _) |- _ => inversion H; auto
  | H:(t _ _ _ _) |- _ => inversion H; auto
  | H:(t _ _ _ _ _) |- _ => inversion H; auto
  | H:(t _ _ _ _ _ _) |- _ => inversion H; auto
  end.
```

Thus, the tactic only matches constructors with up to 6 arguments. There is no way to pattern match on the head of the constructor in general, regardless of its number of arguments. For `mcinv` this is not so bad but for other tactics it leads to verbose matching constructs. At any rate, I have redefined a number of the primitive tactics to match the heads of terms instead of hypothesis names, including rewrite, elimination, injection, clear (removing unnecessary hypotheses) and unfold (expand a definition) tactics.[1]

Returning to the lemma example above, it would be nice once we have proved this simple lemma for Coq to automatically try and apply it whenever we have a goal that matches its conclusion. Coq does allow one to add lemmas to the database for its `auto` and `eauto` tactics. However these `auto` tactics often fail to apply lemmas like `wf_heapval_not_tcode` because there are variables in the premises which do not appear in the goal and the tactics are not always able to infer them automatically from a context. There are many other situations when it seems that it should be possible to automatically infer proofs terms or perform simple proof search to establish a goal but the standard Coq tactics do not provide that facility. There is the possibility of implementing such advanced tactics in the underlying OCAML implementation of Coq, and this has been done for tactics related to arithmetic reasoning, for example, but I have not yet had the opportunity to dig deeply into the data structures and proof representation of the implementation.

One construct that has been used heavily in my implementation is the partial function, which I define as:

```
Definition fmap (A,B:Set) := A -> option B.
```

[1]Most of such tactics will be found in the Coq file `mystuff.v` of my developments.

In addition to standard access and update operations on partial functions, I define (extensional) equality and disjointness, which are used especially in the encoding of separation logic primitives. I have then developed a large library, including lemmas and tactics, for reasoning about these various operations, which is available in the `fmap.v` source file of the Coq developments of Chapters 4 and 5.

6.2 Inductive Types, Impredicativity, and Encoding Polymorphism

Typed assembly languages do not have term-level variables because one works only with a fixed set of registers. Therefore, encoding the term language of TAL in Coq does not present a problem because we can simply use a first-order representation. However, the type system for realistic TALs, as for RgnTAL in Chapter 5, will include features such as polymorphism, existentials, and recursive types. This requires us to deal with TAL type variables and related issues like substitution in the logical encoding. An elegant way to handle binding constructs and variables is higher-order abstract syntax (HOAS) [63]. The basic idea of HOAS is to represent object level (*e.g.* TAL) binding constructs and variables using the functional types and metavariables of the logical system (*e.g.* Coq). The benefit of doing so is that we avoid having to reason properties about the binding constructs of the object language, such as variable substitution, renaming, and scope– all the necessary properties are inherited from the logic itself.

Let us look again, then, at my encoding of RgnTAL types, from Figure 5.3:

```
Inductive omega : Set :=
  | tint    : omega                                 (* int *)
  | thandle : rgn -> omega                          (* p handle *)
  | tpair   : omega -> omega -> rgn -> omega        (* t1 x t2 at p *)
  | tcode   : capset -> (regt -> omega) -> omega    (* code A,G *)

  | tabsr   : (rgn -> omega) -> omega               (* \/ p:Rgn. t *)
  | tabsc   : (capset -> omega) -> omega            (* \/ c:Cap. t *)
  | tabscb  : (capset -> omega) -> capset -> omega (* bounded poly over cap *)
  | tabscd  : forall (c1 c2:capset), ((disjcap c1 c2) -> omega) -> omega
  | tabst   : (omegaV 1) -> omega                   (* \/ t:Type. t' *)

  | trec    : (omegaV 1) -> omega                   (* \mu t:Type. t' *).
```

Notice in the `tabsr` and `tabsc` constructors that abstraction over RgnTAL region and capability variables is represented by a Coq function from the appropriate set to `omega`. Then, constructor application in RgnTAL corresponds directly to application in the logic– we do not need a separate definition of substitution, variable renaming, *etc.* However, HOAS has not been used in the `tabst` case – for abstraction over RgnTAL types themselves. It would have been nice to be able to define the `tabst` constructor as well to have type `(omega -> omega) -> omega`, but the leftmost `omega` in this expression represents a negative occurrence, which is not allowed in CiC for the inductive type that is being defined. This restriction on inductive type definitions is necessary to enforce consistency of the logic [79].

One approach to try and bypass this restriction is to use the impredicativity of the CiC universe `Set`. That is, instead of defining separate constructors for abstraction over regions, capabilities, and types, we define a single constructor which captures abstraction over any `Set`. Since `omega` is itself in `Set`, we could then instantiate this constructor with `omega`:

```
Inductive omega : Set :=
  ...
  | tabs : forall (j:Set), (j -> omega) -> omega
```

Thus, for example, the RgnTAL type $\forall[\alpha:\mathsf{Type}](A,\Gamma)$ would be represented as

```
(tabs omega (fun (t:omega) => (tcode A G)))
```

for appropriate `A` and `G`.

This approach had been used previously in [74]. However, it turns out that this trick does not work for my purposes. By defining `omega` in this way, we have built a so-called *large inductive definition* on which additional restrictions apply for elimination, again in order to maintain consistency of the logic [78]. In particular, we cannot write projection functions that return the first or second arguments of `tabs`. This limitation is quite unfortunate because we would need such projections to prove injectivity of the constructors when developing the syntactic soundness proof for the type system.

[74] was not interested in proving soundness of the encoded type system in Coq itself

and therefore did not run into this problem. Nonetheless, the latest version of Coq has removed the impredicativity of the sort `Set` in order to have a more consistent logical system in the eyes of the intuitionistic mathematician community. Therefore, the default configuration of the Coq proof checker will not even allow large inductive definitions such as the one above any more.[2]

In any event, it seems to be difficult to represent RgnTAL type abstraction using HOAS and so I have utilized a deBruijn index representation for type variables. In order to keep the effects of the deBruijn representation localized from the main development and proofs, I define a "lifted" version of `omega` as follows:

```
Inductive omegaV : nat -> Set :=
  | tvvar    : forall i, omegaV (S i)
  | tvlift   : forall i, omegaV i -> omegaV (S i)

  | tvint    : omegaV 0
  | tvhandle : rgn -> omegaV 0
  | tvpair   : forall i, omegaV i -> omegaV i -> rgn -> omegaV i
  | tvcode   : forall i, capset -> (regt -> omegaV i) -> omegaV i

  | tvabsr   : forall i, (rgn -> omegaV i) -> omegaV i
  | tvabsc   : forall i, (capset -> omegaV i) -> omegaV i
  | tvabscb  : forall i, (capset -> omegaV i) -> capset -> omegaV i
  | tvabscd  : forall i, forall (c1 c2:capset),
                         ((disjcap c1 c2) -> omegaV i) -> omegaV i
  | tvabst   : forall i, omegaV (S i) -> omegaV i
  | tvrec    : forall i, omegaV (S i) -> omegaV i.
```

This definintion keeps track of the number of free type variables in the term. Thus `(omegaV 1)` is the kind of an encoded RgnTAL type which may have up to one free type variable in it. The first two constructors of `omegaV` are for the deBruijn representation– the first is for the actual type variables and the second is to lift the degree of a type. That is, a type with up to `i` free variables in it can also be treated as a type with up to `i+1` free variables. The remaining constructors of `omegaV` are simply mirror images of the `omega` definition that propagate the number of free type variables. Notice that the `tvabst` constructor (and the `tvrec`) binds a type variable– it takes a type with `i+1` free variables and binds the topmost one to produce a type with only `i` free variables.

[2] Although the feature may be restored in the current version of Coq using a command-line argument.

```
Fixpoint subst_aux (i:nat) (t:omegaV i) struct t
  : forall j, i=(S j) -> omegaV j -> omegaV j
  := match
       t as X in (omegaV i) return (forall j (p:i=S j) (e':omegaV j), omegaV j)
     with
       | tvvar n     => fun j _ e' => e'
       | tvlift n t' => fun j (p:S n=S j) _ =>
                                eq_rec n _ t' j (myeqaddS n j p)
       | tvint       => fun j (p:0=S j) _ => O_S_set _ j p
       | tvhandle _  => fun j (p:0=S j) _ => O_S_set _ j p
       | tvpair n t1 t2 p'
                     => fun j (p:n=S j) e' =>
                                tvpair j (subst_aux n t1 j p e')
                                         (subst_aux n t2 j p e') p'
       | tvcode n A G => fun j (p:n=S j) e' =>
                                tvcode j A
                                  (fun r => (subst_aux n (G r) j p e'))
       | tvabsr n Fr => fun j (p:n=S j) e' =>
                                tvabsr j
                                  (fun p' => (subst_aux n (Fr p') j p e'))
       | tvabsc n Fc => fun j (p:n=S j) e' =>
                                tvabsc j
                                  (fun c => (subst_aux n (Fc c) j p e'))
       | tvabscb n Fc A => fun j (p:n=S j) e' =>
                                tvabscb j
                                  (fun c => (subst_aux n (Fc c) j p e')) A
       | tvabscd n c1 c2 Fcd  => fun j (p:n=S j) e' =>
                                tvabscd j c1 c2
                                  (fun D =>
                                    (subst_aux n (Fcd D) j p e'))
       | tvabst n t' => fun j (p:n=S j) e' =>
                                tvabst j
                                  (subst_aux (S n) t' (S j)
                                          (myfequal _ _ S _ _ p)
                                          (tvlift j e'))
       | tvrec n t'  => fun j (p:n=S j) e' =>
                                tvrec j
                                  (subst_aux (S n) t' (S j)
                                          (myfequal _ _ S _ _ p)
                                          (tvlift j e'))
     end.
```

Figure 6.1: Substitution for the encoding of RgnTAL types.

The deBruijn representation means that I do not need to worry about alpha-equivalence in the encoding of types. However, I do need to explicitly define the substition operation. The definition of `omegaV` allows the use of dependent types to enforce the correctness of substitution. Hence, I can define a function,

```
Definition substV : omegaV 1 -> omegaV 0 -> omegaV 0
  := fun T t => (subst_aux _ T _ (refl_equal 1) t).
```

which takes a type with one free variable `T` and a type with no free variables `t` and substitutes `t` for the variable in `T` to return a type with no free variables. The function actually makes use of a more generalized version shown in Figure 6.1. The Coq syntax needed to code this function is somewhat opaque,[3] but it basically takes a type `t`, with `j+1` (= `i`) free variables, and a type `e'`, with one less free variable, and substitutes `e'` for the topmost free variable in `t`.

In the `tvvar` case, we have found the free variable which is to be substituted so we replace it with `e'`. In `tvlift` we know that the lifted term does not contain the variable we are substituting for so we simply return `t'`. However, it is necessary to cast the type of `t'` from `(omegaV n)` to `(omegaV j)` where `i = j + 1 = n + 1`, using the elimination operator on equality `eq_rec`. The simple constructors such as `tvint` should not have to be considered because they will not contain any free type variables (they have kind `omegaV 0`); thus the lemma `O_S_set` eliminates these cases based on the absurd proof `p` that `O = j + 1`. In the remaining cases, the substitution is simply propagated through the constructors as necessary.

After substitution, I define functions using similar techniques which convert between the two versions of `omega` and also a function to unfold a recursive type (by substituting itself for the free variable in its body):

```
Definition unliftV0 : omegaV 0 -> omega := ...
Definition lifttoV : omega -> omegaV 0 := ...

Definition unfoldV : omegaV 1 -> omega
   := fun t:(omegaV 1) => unliftV0 (substV t (tvrec _ t)).
```

[3] Much credit goes to Valery Trifonov for working out the way to write this function in Coq.

As mentioned earlier in this section, the use of two versions of `omega` keeps the reasoning about deBruijn encodings out of most of the proofs. The reason should be clear from the standard statements of the preservation and progress lemmas for showing soundness of a type system in the syntactic method (see Section 5.2 for example): in these proofs we need to reason about well-typed programs at the top-level, where the type variable context is empty. The only time that it is necessary to deal with type variables is when the current instruction is a jump or branch to another code block whose specification may be a polymorphic type. At this point, we need to instantiate the variables of the code block type context with types of the correct kind from our top-level context, but once that is done, we have immediately again a type with no free variables.

The other place in the type system where type variables are introduced into the context is the typing rule for code heap values. For example, in RgnTAL we had:

$$\frac{\mathcal{C}; \Delta; A; \Gamma \vdash I}{\mathcal{C} \vdash \mathsf{code}\ [\Delta](A, \Gamma).I}\ (\text{C-CODE})$$

I encode this rule using a hybrid scheme of HOAS and deBruijn substitution:

```
Inductive wf_codeval : codemem -> codeval -> Prop :=
  | wf_cvcode : forall CM (t:omega) Is,
                (forall (Ts:list constr) A G,
                        instcodetype t Ts (tcode A G) ->
                        wf_iseq CM A G Is) ->
                wf_codeval CM (cvcode t Is)
```

The `instcodetype` relation corresponds to substitution of a list of constructors for the variables in a RgnTAL type context: $\tau[\vec{c}/\Delta] = \forall[\cdot](A, \Gamma)$. Thus, in `wf_cvcode` I use the idea of HOAS to introduce a set of Coq variables `Ts` and then substitute them into the deBruijn encoding to obtain a code type where the variables of the object language are, in fact, represented as variables of the meta-language.

One final detail in this method of encoding RgnTAL variables is that I end up needing to specify an extensional definition of equality on types because of the use of functions (for HOAS) in the constructors representing abstraction over regions and capabilities cou-

pled with the representation of type abstraction using first-order abstract syntax (deBruijn indices).

The need for deBruijn indices certainly has an effect on the elegance and convenience of encoding the RgnTAL type system in Coq and reasoning about it. Furthermore, I cannot derive in my type systems the immediate benefits of the impredicative alternative described above for the purposes detailed in [74, 82]. Nonetheless, I do manage to have an encoding of the TAL type systems for which the necessary proofs can be developed to satisfy the needs of the FPCC framework. As future work, it will be beneficial to explore the use of alternate logical frameworks, such as the Twelf framework, instead of Coq, which would allow the proper use of HOAS, thus simplifying much of the effort involved in formalizing the binding constructs of the type system. On the other hand, the benefit of using Coq is that we reuse the Coq logic for the assertions of our actual object language (CAP). If we use a logical framework like Twelf, then we will also need to encode the assertion logic and perhaps prove its properties that are gotten for free by reusing Coq. This will not necessarily be difficult, but will require a new definition of CAP and some reorganization of the overall framework.

6.3 Coq Encodings and Adequacy

The handling of type abstraction and variables described in the previous section will perhaps be somewhat disturbing for the reader familiar with logical frameworks and the use of HOAS. The issue that may be raised is the one of *adequacy*. That is, does the Coq encoding accurately capture the syntax and semantics of the language as defined, for example, in Figures 5.1, 5.5, and 5.6. In fact, as it should be clear from the presentation of Coq details, the encoding is not adequate with respect to the typeset description of the RgnTAL language.

I have, for the most part, implemented a *shallow embedding* of RgnTAL in Coq. That is, I use the underlying binding mechanism of the logic to represent the binding constructs of

RgnTAL (*e.g.* providing polymorphism over regions and capabilities). Several problems can arise from the use of a shallow embedding. One is that the usual induction principles over the structure of terms is lost. This has not been a problem with Coq since the generated induction principles for `omega` are sufficient for my purposes. Another problem that can arise is that reasoning on the syntax becomes more difficult. For example, it would be hard or impossible to define a function that counts the number of free region variables in a `omega` term because there are no bound variables at the object level– they have been lifted to Coq variables and the logic cannot be used to reason about itself in that way. Nonetheless, I have not had a need to perform such reasoning on the syntax of RgnTAL and so this also is not a problem.

Finally, the major issue with shallow embeddings is the difficulty in preserving the adequacy of the encoding. This exhibits itself in the ability to define *exotic terms*– terms that do not encode any valid term of the language according to the BNF description. For example, in Coq I can define the following very strange `omega` term, which does not correspond to any valid RgnTAL type according to Figure 5.1:

```
Definition exotic : omega
  := tabsr (fun r => match r with
                     | 0 => tint
                     | _ => (thandle r) end).
```

The occurrence of such exotic terms in the RgnTAL encoding is compounded by the fact that I have represented type abstraction using a first-order encoding (deBruijn indices), described in the preceding section of this chapter. Thus, one could define a term that performs strange manipulations on the deBruijn index representing a type variable.

When using HOAS to encode an object language, one usually either uses a logical framework that does not exhibit such problems, or else one would define a `valid` predicate on `omega` terms to somehow prevent the formation of such exotic terms. However, in my case, I do not need to do this. The reason is that even if the encoded language does not adequately represent the original, I have still proven formally the type soundness of the system. Therefore, even if exotic terms are somehow introduced into a well-typed

program, they will not affect the soundness of the language and the translation into CAP will still satisfy the safety policy.

The normal use of logical frameworks is to encode an object language and prove properties about the encoding that will be applicable to the object language. Usually, in such a situation, one implements a type-checker or compiler for the object language separate from the logical encoding and it is therefore necessary to be sure that the proofs developed for the logical encoding will also hold for the implementation. In my case, the logical encoding is also the implementation and therefore, even if the encoding actually captures a more complex language than what is initially "on paper", it is irrelevant as long as the necessary safety proofs can be generated in the end.

Note, though, that there is one aspect of the encoding that must be adequate in my framework. That is the encoding of the machine semantics in Section 2.1. Ultimately, I am proving the safety property for the machine encoding. If the encoding in Coq does not match the actual behavior of the machine, then the Coq proofs are useless. Adequacy in this situation is fairly easy to establish, however, because the machine encoding is simply first-order and can be directly checked against the specifications of the hardware's operation.

In summary of the discussion in this section and the previous two, then, I can list the major advantages and disadvantages of using the Coq tool to develop my prototypes. Most of these are actually issues with the Coq logic– CiC– as opposed to the actual software implementation. The advantages are:

- The inductive types, along with base logic of CC, provide a very expressive system in which to encode systems and reason about them. (Unfortunately, this great expressiveness plays a part in some of the disadvantages listed below.)

- By using the logic at different layers, we are able to directly embed the entire reasoning power of CiC into the assertions of our object language, CAP. In defining the syntax of CAP, the assertions are arbitrary CiC predicates on state (another instance

of the use of shallow embedding). That means that CAP assertions can easily refer to the RgnTAL encoding and use its properties to certify safety of its own code.

On the other hand, the main disadvantages that I have encountered are:

- Proof automation is very primitive. Because of the expressiveness of the logic, and especially the presence of inductive types, the search space seems to be too large to handle much automated searching for proof terms. Thus, one must use primitive tactics when developing proofs and there is not a reasonably good way to compose more effective tactics for a given proof development without resorting to programming in the underlying OCaml implementation.

- The fragility of Coq proof scripts, which are sequences of tactic commands, means that making a change to some part of the object encoding may potentially affect a major part of the scripts, requiring manual editing, to adjust the order of operations, or to rename hypotheses, for example. This is not even taking into account the changes in syntax or the behavior of tactics that occur with new releases of the Coq tool. Explicitly building the proof terms, instead of using the tactics, is not much more use because again when a change is made to the system being encoded, the proof terms have to be edited to reflect that.

- The limited ability for using higher-order abstract syntax in encodings means that one must resort to reasoning about terms with deBruijn indices or some similar first-order representation. Not an enjoyable task for a human being, even if the amount of such reasoning needed is limited, as I have tried to do. Eventually, programs that are even as complex as the Fibonacci example in Appendix C result in one having to manage quite large proof terms in order to build a type checking derivation. While doable for a prototype implementation, this quickly becomes too tedious for a human to handle, and should be handled by an automated tool built for that purpose.

- In addition to the deBruijn indices, another area of reasoning that becomes cumber-

some in Coq is that of equality. Because of the dependent types that are used in the encodings, it sometimes becomes complex to reason about equality between terms. Figure 6.1 demonstrates a simple example of this.

These issues certainly indicate that there is much future work to be engaged in, including the evaluation of alternate frameworks for reasoning than Coq or CiC.

6.4 Function Pointers, Mutable Memory, and Reflection

The discussion in this section may perhaps be best viewed as future work but I include it here since it might throw some light on how the syntactic approach to FPCC that I have developed handles tough language features like function pointers and mutable memory. In a recent paper, Reynolds makes the following observation about Hoare logic:

> Even as a low-level language, the simple imperative language axiomatized by Hoare is deficient in making no provision for the occurrence in data structures of addresses that refer to machine code. Such code pointers appear in the compiled translation of programs in higher-order languages such as Scheme or SML, or object-oriented languages such as Java or C#. Moreover, they also appear in low-level programs that use the techniques of higher-order or object-oriented programming.[67]

The difficulty of handling first-class code pointers in Hoare logic arises immediately in proof-carrying code frameworks because they are usually based on proving safety using Hoare logic-style reasoning with pre- and post-conditions for each machine command. Besides having to deal with first-class code pointers, programs compiled from the TALs presented in this thesis will also have blocks of code that make recursive calls to each other indirectly. Suppose we have a CAP program with the following command block in memory:

$$l_2 \mapsto \mathbb{C}_2; \texttt{jmp r2}$$

To prove this code safe, we would need to specify somehow that the address in r2 is a pointer to code that requires, for example, $\mathsf{r1} = 1$ before jumping to it.[4] In CAP, therefore, the code specification for code address l_2 would be given by something like:

$$\Phi(l_2) = \{\mathsf{codeptr}(\mathsf{r2}, \{\mathsf{r1} = 1\})\}$$

where codeptr is a predicate that somehow specifies the precondition of r2 to be $\{\mathsf{r1} = 1\}$. Defining codeptr for this can already be tricky but it gets much more complicated with recursive functions. Consider another code block in memory that may be mutually recursive with l_2:

$$l_2 \mapsto \mathbb{C}_2; \texttt{jmp r2}$$
$$l_3 \mapsto \mathbb{C}_3; \texttt{jmp r3}$$

At runtime the register r3 may contain l_2 and r2 may contain l_3. One way to handle this would be to analyze the whole program to find out all possible targets of these jumps and then build a specification which is a disjunction of these concrete target labels. However, this does not allow for very modular specification and checking of code blocks. What we want to be able to specify is something like:

$$\Phi(l_2) = \{\Phi(\mathsf{r2}) \ \wedge \ \mathsf{codeptr}(\mathsf{r2}, \Phi(\mathsf{r2}))\}$$
$$\Phi(l_3) = \{\Phi(\mathsf{r3}) \ \wedge \ \mathsf{codeptr}(\mathsf{r3}, \Phi(\mathsf{r3}))\}$$

Namely, for l_2, there is some precondition of r2 specified in Φ which holds at the point that the jump from l_2 takes place. Of course, this cannot be defined in Coq because we are trying to use Φ in its own definition.

It is interesting that Reynolds, following the statement quoted at the beginning of this section, goes on to suggest the introduction of a reflection operator to handle such code pointers. Reflection, in general, refers to an entity's (the entity usually being an executable

[4] More accurately, I should write $\mathbb{R}(\mathsf{r1}) = 1$, where the specification is a predicate on machine state, but for simplicity I leave the state components implicit in these examples.

program) ability to represent and operate on itself in the same way it deals with its other constituents [21]. Notice that in the specifications of l_1 and l_2 above, Φ is trying to reflect on its own contents.

It seems that the manner in which the syntactic approach to FPCC handles first-class code pointers is similar to the use of reflection and especially the related process of reification. In terms of executable programs, reification is the provision of a mechanism for encoding execution state as data. Once the execution state has been represented as data somehow, the data can be inspected or modified and reintroduced into the execution state. In a similar way, we may view the TAL code types as reified data which are reflected into the definition of the CAP code specification. The CpInv function in Section 4.4.3 defines the relationship between syntactic data structures (the encoding of the TAL language and type system) and the predicates on state which make up the code specification of CAP through CpGen. During the process of proving safety, we know that the specifications for code compiled from TAL are based on the syntactic terms that encode the syntax of TAL. In other words, we know that the CAP preconditions, which can potentially be predicates of arbitrary form, have been generated from the TAL encoding and therefore we can reason about their properties and interaction based on the structure of TAL syntax. The syntactic encoding, in turn, has appropriate typing rules for handling recursive code and the encoded soundness proof of the language shows by induction one step at a time that the typing rules are meaningful. Therefore circularity in the reasoning about specifications is eliminated.

The way mutable memory is dealt with in the syntactic framework is also similar to code pointers. Recall from the Introduction, Section 1.3, that the semantic approach tries to verify the store instruction by introducing an *allocset*. The *allocset* mapping keeps track of the types at each location in memory in order to maintain consistency in the presence of aliasing. In the semantic framework, however, types are predicates in the logic, which in turn depend on the *allocset*. This introduced a troublesome circularity in the definitions.

In the syntactic approach, instead of using an *allocset* that maps to predicates, I am essentially mapping addresses to syntactic terms (again, the inductive definition of TAL types) that describe the layout of memory. Since these terms are simply first-order objects in the logic, it is easy to reason about them, in particular, to ensure that the description of memory is consistent in the presence of aliasing. Then I define how to interpret these syntactic terms (*i.e.* reflect them) as predicates, which again takes place in the definition of CpInv.

The use of CpInv thus enforces a number of constraints on the CAP predicates which are useful for proving the safety theorems. However, many of the issues are still meshed together and I hope in future work to separate the handling of mutable memory from code pointers, for example. Even as we went from a monolithic invariant in Chapter 3 to the local construction of CpInv for individual instructions in Chapter 4, I hope to further distill the components of CpInv. This could lead to a better understanding of how to produce foundational PCC and allow for more general support of interaction between different type systems.

Chapter 7

Related Work

The development of this thesis has taken place in the context of a great number of related works. Not only have I found useful ideas in older researches, there are a number of contemporary developments in various areas which complement my framework. In this chapter, I give an overview of related work in several areas that my research has touched and benefitted from.

7.1 Proof-Carrying Code

I have already given some overview of the development of proof-carrying code in the Introduction. Here I summarize the lines of research as they have developed historically and continue to do so. The idea of PCC was introduced by Necula and Lee and applied in several case studies [56, 54, 58, 57]. The original framework was also used to build a certifying compiler for Java, described in Colby, *et al.* [15].

More recently, there have been ongoing efforts to reduce the trusted code base of these traditional PCC systems, which have mostly focused on removing parts of the VCGen from the TCB while maintaining the scalability and industrial-strength engineering quality that marked the first implementations. To this end, Necula and Schneck have presented a series of incremental improvements to their PCC framework [59, 70, 60]. In their

latest work, the PCC code producer is responsible not only for supplying the machine code but also a verifier which proves the code safe. The verifier is an actual executable instead of being a static proof. This untrusted verifier interacts with a smaller, trusted, core VCGen to prove safety of the code. The system is intended to be more secure and flexible by removing a large part of the original VCGen from the trusted base, but in doing so they have begun to encounter the technical issues that arise in foundational PCC, such as the handling of code pointers. In his thesis, Schneck [69] also begins to address the issue of interoperation between different languages in the PCC system. The Open Verifier framework, as it is called, is still under development but appears to be approaching a solution similar to my use of the CAP specification system. Currently, the integration seems to be somewhat limited in nature, with programs being able to interoperate only if their memory areas are completely disjoint. The Open Verifier does not use a language like CAP as a common intermediate level, but it does involve specifying invariants (preconditions) of each machine instruction which are based on the syntactic encoding of the type system, as I have developed. It may be therefore, that these lines of research will merge at some point in the future, or at least produce results that are mutually beneficial for each approach.

In another line of research, Bernard and Lee [9] investigated the use of temporal logic to remove the VC generator from the TCB. In particular, the idea is to use temporal logic as the basis for a formal security-policy language, instead of having the security policy and enforcement mechanism built-in to a trusted VCGen. This work also may be viewed as complementary to other PCC research. Furthermore, another group of researchers [90, 89] have been using the Isabelle/HOL framework to verify the VC generator component of a traditional proof-carrying code system. They currently have developed a prototype framework with a generic VCGen that can be instantiated with a particular programming language, safety policy, and safety logic.

In the meantime, Appel, *et al.* have introduced the notion of foundational proof-carrying code [4, 48, 5, 3]. As discussed also in the Introduction, their FPCC project aims to provide a more flexible and secure PCC framework by developing proofs using only the

foundations of mathematical logic. In particular, this system uses Church's higher-order logic with axioms for arithmetic. The complexities of the semantic approach followed by this FPCC group meant several years of developing models for various type system features – the troublesome mutable references, recursive types, and code pointers. Ahmed produced a stratified semantic model for handling mutable memory references [2]. A low-level framework for typed machine language is being developed to encapsulate the complex portions of the semantic model by serving as a general compilation target from high-level typed assembly languages [76, 14]. Appel, *et al.* have also investigated the development of a minimal TCB, producing one that includes less than 2,700 lines of code [6], as well as work on minimizing the size of transmitted FPCC proofs [94].

Taking inspiration from the initial developments of FPCC, I have introduced a framework based on syntactic soundness proofs. While I have concentrated on the use of the CiC logic, Crary and others have adopted the syntactic FPCC approach to the Twelf [65] metalogical framework [18, 19]. In the metalogical approach, the operational semantics of the architecture and encodings of object languages are specified in one logic, while safety proofs are supplied as metatheorems in the metalogic. The advantage of this system is ease of development, as illustrated in the design of an expressive typed assembly language [18] targeted at a real machine architecture (the Intel IA-32, or x86). One tradeoff, however, is that the logical system is more complicated and thus may result in a larger TCB. This research has also produced a low-level language with a type system that is capable of expressing the interface of a realistic garbage collector [83]. This is similar in nature to my provision of a region library through code stubs in the TAL heap, except that it is not as general since the interface is built into the type system. Also, while the interface has been specified, the actual implementation of the garbage collector has not been certified and therefore does not link together to provide a proof of safety of the complete application, as the region library does in my case.

Finally, in work predating the development of proof-carrying code, Burstall and McKinna introduced the notion of *deliverables*– a program paired with a proof of correctness.

Despite the correspondence to the idea of PCC, this research was focused on a categorical approach to program development, did not support programs with effects (*i.e.* imperative programs), and does not seem to have been carried on beyond the simple framework given in McKinna's thesis [47].

7.2 Typed Assembly Languages and Region Type Systems

One of the complementary and, in fact, enabling technologies of PCC has been the development of typed assembly languages. Morrisett, *et al.* [52, 50] introduced the design of a low-level, statically typed assembly language aimed at being more suitable than other targets (such as Java bytecode) for supporting a wide variety of source languages and optimizations. There have been a great number of variants of the original TAL (three in this dissertation!), each incorporating some interesting feature into the type system. In most cases, it would be possible to utilize the syntactic approach to FPCC to compile programs from these TAL variants into certified machine code. A related project that grew out from TAL is the development of a safe dialect of C, called Cyclone [43]. Cyclone enforces safety through advanced type system features, while at the same time providing compatibility with C and retaining its low-level control and performance. Cyclone may potentially be a good language to use for writing certified system libraries, such as the region management library, and then compiling it down to a version of TAL. Cyclone provides support for a number of different memory management mechanisms [31, 36] including regions (a memory management discipline introduced by Tofte and Talpin [80, 81]) and garbage collection.

The type system I use for RgnTAL is based on the capability calculus of Walker, *et al.* [85, 20] for typed memory management. This work in turn was based on earlier research [75, 86] using a linear type system to track pointer aliasing and destructive operations on memory objects. I have left out many of the details and background of the development of a region-based type system in this dissertation because I have, for the

most part, directly adopted the work of [85, 84] in this regard.

7.3 Encoding Object Languages with Variable Binding

As discussed in Sections 6.2 and 6.3, one important issue that must be faced with when encoding object languages is how to encode binding constructs and variables of the language. One option is to take a first-order approach resulting in a *deep embedding*: one encodes the variables and binders in the framework by explicit defining the type of variables (*e.g.* inductive, as natural numbers), and the necessary concepts of alpha-equivalence, beta-reduction, *etc.* . This approach allows us full reasoning power over the object language but at the expense of a great deal of machinery to implement the necessary concepts. Some representative examples of this approach in the Coq framework are those of Huet, who formalized the lambda-calculus [41] using a deBruijn representation, and Barras, who also used a deBruijn representation to prove the correctness of the Coq kernel in Coq itself [7].

A more elegant and convenient approach is to use a higher-order abstract syntax (HOAS) [63] and a *shallow embedding*, where the binding constructs of the object system are represented using the binding constructs of the logic itself. However, as discussed in the earlier chapter, this approach has difficulties in a formal system of inductive definitions like Coq. There is some ongoing research on how to adapt Coq to the HOAS approach, among them [23, 22], which introduce a specific (non-inductively defined) type for variable identifiers that delegates alpha-conversion to the metalanguage, while eliminating the possibility of so-called exotic terms. However, the drawback of this approach is that the definition of substitution must still be encoded in the logic. In a separate line, [24, 72] develop a new logic by extending the simply typed lambda-calculus with primitive recursive constructs that allow the programming of adequate HOAS encodings. The goal is to synthesis the methodology of HOAS used in a logical framework like LF [35] with systems based on induction principles such as Coq.

Another approach to reasoning with HOAS in logics such as Coq is to axiomatize the necessary theory needed for reasoning about such encodings. This axiomatic approach has been adopted and developed as the Theory of Contexts by [68, 38, 39, 49]. The issue here of course is that one begins adding axioms to the base logic, which we would like to avoid for FPCC, even though much effort has been put into proving their consistency. [1, 30] also develop an axiomatic approach to reasoning about lambda calculi identified up to alpha-conversion. A similar work to these others has been presented in [26] which involves a two-level axiomatic methodology for reasoning about object languages. In this approach, a specification logic is encoded in Coq as an inductive definition and then the object logic is encoded in the specification logic.

7.4 Low-level Reasoning and Separation Logic

As in the other areas of related work, there are a great number of developments in reasoning about safety at a low level in the language hierarchy. I only try to list here those that I have found most relevant to my work. Some common features of most, if not all, of these works are that they involve (1) semi-automatic (*i.e.* human assisted) generation of proofs or other form of certification for the low-level code, because of the very intricate nature of reasoning required, and (2) they only concentrate on certifying or reasoning about code at one level. My work also exhibits the former feature but in respect of the latter, I have developed a framework to allow code certified using different methods, at different levels of abstraction, to interoperate safely.

Historically, advancements in proving program properties have been spurred by the works of McCarthy [46], Floyd [29], and Hoare [37]. Much of the research projects in the last couple of decades have focused on formal, mechanical verification of high-level language programs, as opposed to low-level machine code verification. Of the few exceptions, Yu and Boyer [100, 11] mechanically check the correctness of object code programs for the Motorola MC68020 microprocessor. A substantial piece of research, they veri-

fied implementations of a number of well-knowing searching, sorting, and string library operations. In the process, they had to deal with some of the same issues that I have mentioned in this thesis. For "functional parameters" (*i.e.* code pointers), they essentially end up hardcoding the addresses of code blocks into their correctness specifications. This is similar to what we would do if we were programming and certifying code in CAP without reference to predicates related to a syntactic type system. Much of Yu's work also involved reasoning about memory operations, which was quite complicated for them because they did not benefit from the development of separation logic primitives (discussed below) for that purpose. Finally, their approach was to compile high-level code and then specify and reason about the object code correctness– unlike my approach where I can start with high-level code that has been certified correct (according to a type system) and then compile to machine code, possibly linking into other machine code that has been certified correct directly at the object code level.

There has been some use of the Coq proof assistant itself for low-level program verification. Much work has been done in the VerifiCard project at INRIA (France) in the context of certifying programs for smartcards running the JavaCard virtual machine, for example [8]. This project in general is very specific to the Java VM language, deals with high-level specifications of safety and/or correctness, and also focuses on developing proofs using a proof-assistant (as opposed to fully automatically during a compilation process). The WHY system developed by Filliâtre [28, 27] is a software certification tool that produces proof obligations for annotated imperative programs. The imperative language used is a high-level language and the proof obligation generator resembles the VC generator of PCC. It is not clear, however, if the tool can really be extended to support the necessary low-level reasoning for PCC.

For the purposes of reasoning about low-level manipulation of memory, I have encoded the separation logic primitives of Reynolds [67, 66] in Coq. The semantics of the primitives are defined in terms of the machine memory model and the axioms and inference rules defined by Reynolds in his works are all derivable within the Coq encoding.

More details of the use of separation logic for our FPCC reasoning may be found in [97, 98] or Yu's thesis [96]. Reynolds' work extends earlier work by Burstall [12] and parallels the independent development of bunched logic by Ishtiaq, O'Hearn, and others [13, 42, 61].

Chapter 8

Future Work and Conclusion

My path to the research culminating in this thesis began with an investigation of Java reflection. Reflection, mentioned in Section 6.4, is an advanced language feature that allows programs to examine, adapt to, and modify representations of their own code and execution state at runtime. As part of the research related to the FLINT project [73] for developing a common typed intermediate language, League *et al.* [45] showed a formal translation of a large subset of Java language features into a variant of the polymorphic λ-calculus, F_ω.

I began to investigate the possibility of extending the FLINT framework to handle the reflection library of Java, which in standard Java compilers is implemented in C. It soon became apparent that proving that runtime library type-safe required reasoning about memory in ways that the type system of the intermediate language did not easily support. I have now developed the syntactic approach to FPCC which, as we have seen in this thesis, provides a reasonably simple and straightforward approach to generate safe machine code from a typical typed assembly language (such as Java or FLINT might eventually compile to). Additionally, I have worked out a method in which runtime libraries (such as the Java reflection package) can be certified safe using low-level Hoare logic-style reasoning and then linked together with the high-level compiled code. Of course, I have presented this work with very ideal languages, not Java or FLINT, so there is much poten-

tial for future work. I outline below some of the directions in which this research could logically proceed.

8.1 Limitations and Future Work

Clearly, a first practical step will be to apply the framework to a real machine and to a realistic typed assembly language. As mentioned in the related works, Crary [19, 18] has already begun applying my syntactic framework, using a different logic framework, to the Intel family of processors. With others in the FLINT project, I have also been working on a version of CAP for the Intel x86 architecture. The results of these efforts will allow for a better comparison of the viability of the syntactic approach with respect to previous developments of proof-carrying code. Although the idealized machine I have used for the prototype implementation of this dissertation is quite simplistic, the results should be scalable to different machine architectures. This will not necessarily be a simple task, but only because of technical details in different architectures, not because of any fundamental limitations. As TAL technology is scalable to different architectures so would be the corresponding syntactic FPCC framework.

In order to produce the deliverable of a realistic certified system, we will have to extend the research along both the engineering and the theoretical aspects. In the engineering aspect, there is much potential for proof development tools and checkers that are tuned to the FPCC framework. Coq has been an excellent resource for the prototyping, but it was obviously developed with different purposes in mind and would therefore be better supplemented by tools specific for our purposes. For example, as described in Section 3.3.5, we will need a compiler that deals directly with the logical encodings, as well as the source and target languages, so as to produce the necessary proofs of well-typedness and the translation relations. When using a realistic language, we will also need to deal with real operating system libraries, which will potentially be much more complex to certify, and definitely much larger, than the simple runtime library examples my prototype

uses.

Along more theoretical paths of research, there is the possibility of investigating the use of alternate logics for the framework. It is certainly not bound to the CiC logic, although the higher-order reasoning and inductive definitions are very handy for our purposes. Whether CiC is used or another logic, the encoding of type variables will remain an issue to be dealt with– for polymorphic, existential, recursive, and other such complex types. It is probably also safe to say that the fundamental differences between semantic and syntactic approaches to FPCC are still not well-understood. Future research may unify the frameworks. Ideas such as those I mention in Section 6.4 could be worked out to produce a more well-grounded understanding of the underlying relationship between type systems and Hoare logic.

In addition to the logic used, the syntactic approach to FPCC clearly depends much on the type system that is being encoded. Not only does the type system have to be encodable in the logic, it must also enforce the constraints specified by the safety policy. The FPCC proof will require a complete representation of the typing derivation for a source program, so the type checking algorithm must be decidable as well. That is, if a fancy type system such as DTAL [95] involves solving a constraint satisfaction problem – NP-complete in general, but can often be efficiently solved in practice – one probably cannot expect to include the constraint solver in the logical encoding. Of course, it might be possible to use an external type checker (written in a general-purpose programming language) to solve the necessary constraints and have that represented in the logical encoding so that the constraint solution only needs to be verified by the proof checker, not solved. It does not appear that this would be a serious limitation of the syntactic approach, but nonetheless will require further investigation and research.

Since most type systems will disallow the writing of some perfectly safe programs, this also means that such programs may not be immediately certifiable for PCC using the syntactic approach. This may be alleviated somewhat by being able to directly certify the programs using the low-level CAP layer of reasoning. However, even with CAP, certain

types of code may not be certifiable. An immediate example would be self-modifying or dynamically generated code. CAP currently will not support such programs because the CAP code specification must be fixed statically at the point of proof-checking. Thus, there are some aspects of my framework that will bear further research in this respect.

Finally, two more issues that will need to be addressed in a practical development of such a framework will be the size of proofs that need to be transmitted and the development time for proofs. As mentioned earlier in Section 3.3.6, the proofs needed for a syntactic FPCC framework can be viewed as two portions– a static portion and a dynamic one. The former, which is also the larger, portion will be composed of the proofs of soundness for the type system, proofs about the translation of well-typed source programs, such as the theorems in Section 4.4.3, or proofs certifying the safety of the runtime library. These proofs are produced semi-automatically by a human being interacting with a proof assistant (at least in the current framework), but they only need to be developed once for a given system. On the other hand, the dynamic portion of the FPCC proof will be composed of the typing derivation for source programs that have been compiled to be run by the code consumer. The size of these proofs can probably be greatly minimized and they are produced automatically during the compilation process.

Thus, the issues of proof size and development time (or effort) will need to be addressed mainly for the "static" proofs described in the previous paragraph. Since the mechanized proofs for the syntactic FPCC framework follow very closely the standard proofs of soundness that are done "on paper," the mechanization can be achieved with reasonable effort, although it may not be entirely trivial. For the developments described in Chapters 2 and 3, the Coq implementation was complete within half a year by a single graduate student with no previous experience in Coq or CiC. The proofs of soundness and correctness of compilation for the region-based system of Chapter 5 were completed in only a couple of months, although the certification of the runtime library has taken longer to complete, mainly because of tedious issues such as reasoning with deBruijn indices. It therefore seems that the syntactic approach, in terms of implementation effort,

is not any harder at the worst than other methods of producing PCC. The issue of proof sizes is somewhat harder to evaluate at this point because I only have a prototype framework. The Coq proof scripts for Chapter 3 are about 10,000 lines of specifications and proof tactics.

8.2 Conclusion

It seems at times that I have introduced more new problems than I have solved. Yet the progress is promising. The work described in this thesis goes a good way towards the realistic goal of laying a foundation for certifying safety of "the whole code." More than ever, codes that are written and compiled from varying levels of the language hierarchy need to be verified for safe interoperation. It is becoming less and less desirable to allow infrastructure code, such as operating systems and runtime libraries, to remain in the trusted computing base of software systems. In summary then, my main contributions have been:

- Development of the syntactic approach to foundational proof-carrying code, in which syntactic soundness proofs of a type system are mechanized in a formal logic and used to produce a safety proof for the compiled machine code.
- A framework for FPCC supporting interoperation between code compiled from high-level type systems and code for low-level runtime libraries certified semi-automatically at the machine or assembly level.
- Initial development of a certified region-based memory management library that links together with a typed assembly language incorporating region capabilities in its type system.

Appendix A

Coq Files for the Syntactic Approach to Foundational Proof-Carrying Code

This chapter lists the Coq code for the system described in Chapter 3. Proofs of all the lemmas and theorems listed here have been completed. (I have omitted the actual Coq proof scripts, composed of long sequences of tactic commands, from these listings as well as some of the utility libraries and lemmas.)

A.1 An Idealized Machine for FPCC

Machine state

```
(* Register definitions and utilities *)
Load tisreg.

Definition word := nat.
Definition mem := word → word.
Definition rfile := reg → word.
Definition state := (mem × rfile × word)%type.

(* Commands (machine instruction set) *)
Inductive cmd : Set :=
  | add : reg → reg → reg → cmd
  | addi : reg → reg → word → cmd
  | sub : reg → reg → reg → cmd
  | subi : reg → reg → word → cmd
```

```
  | mov : reg → reg → cmd
  | movi : reg → word → cmd
  | bgt : reg → reg → word → cmd
  | bgti : reg → word → word → cmd
  | jd : word → cmd
  | jmp : reg → cmd
  | ld : reg → reg → word → cmd
  | st : reg → word → reg → cmd
  | ill : cmd.

(* Decode utilities *)
Definition icoded (w : word) : nat := mod_ w 8.
Definition r1argd (w : word) : nat := mod_ (div8 w) 32.
Definition r2argd (w : word) : nat := mod_ (div32 (div8 w)) 32.
Definition r3argd (w : word) : nat := mod_ (div32 (div32 (div8 w))) 32.
Definition w1argd (w : word) : nat := div8 w.
Definition w2argd (w : word) : nat := div32 (div8 w).
Definition w3argd (w : word) : nat := div32 (div32 (div8 w)).

(* Decode function *)
Definition Dc (w : word) : cmd :=
  match icoded w with
  | (1) ⇒ add (nat2reg (r1argd w)) (nat2reg (r2argd w)) (nat2reg (r3argd w))
  | (2) ⇒ addi (nat2reg (r1argd w)) (nat2reg (r2argd w)) (w3argd w)
  | (3) ⇒ sub (nat2reg (r1argd w)) (nat2reg (r2argd w)) (nat2reg (r3argd w))
  | (4) ⇒ subi (nat2reg (r1argd w)) (nat2reg (r2argd w)) (w3argd w)
  | (5) ⇒ mov (nat2reg (r1argd w)) (nat2reg (r2argd w))
  | (6) ⇒ movi (nat2reg (r1argd w)) (w2argd w)
  | (7) ⇒ bgt (nat2reg (r1argd w)) (nat2reg (r2argd w)) (w3argd w)
  | (8) ⇒ bgti (nat2reg (r1argd w)) (w2argd w) (w3argd w)
  | (9) ⇒ jd (w1argd w)
  | (10) ⇒ jmp (nat2reg (r1argd w))
  | (11) ⇒ ld (nat2reg (r1argd w)) (nat2reg (r2argd w)) (w3argd w)
  | (12) ⇒ st (nat2reg (r1argd w)) (w3argd w) (nat2reg (r2argd w))
  | _ ⇒ ill
  end.
```

Machine semantics

```
Definition updatereg (R : rfile) (rd : reg) (v : word) : rfile :=
  fun r : reg ⇒ if beq_reg r rd then v else R r.

Definition updatemem (M : mem) (a v : word) : mem :=
```

```
  fun w : word ⇒ if beq_nat w a then v else M w.

Definition Step (St : state) : state :=
  match St with
  | (M, R, pc) ⇒
      match Dc (M pc) with
      | add rd rs rs' ⇒ (M, updatereg R rd (R rs + R rs'), S pc)
      | addi rd rs w ⇒ (M, updatereg R rd (R rs + w), S pc)
      | sub rd rs rs' ⇒ (M, updatereg R rd (R rs - R rs'), S pc)
      | subi rd rs w ⇒ (M, updatereg R rd (R rs - w), S pc)
      | mov rd rs ⇒ (M, updatereg R rd (R rs), S pc)
      | movi rd w ⇒ (M, updatereg R rd w, S pc)
      | bgt rs rt w ⇒
          if blt_nat (R rt) (R rs) then (M, R, w) else (M, R, S pc)
      | bgti rs i w ⇒
          if blt_nat i (R rs) then (M, R, w) else (M, R, S pc)
      | jd w ⇒ (M, R, w)
      | jmp r ⇒ (M, R, R r)
      | ld rd rs w ⇒ (M, updatereg R rd (M (R rs + w)), S pc)
      | st rd w rs ⇒ (updatemem M (R rd + w) (R rs), R, S pc)
      | ill ⇒ St
      end
  end.

Fixpoint MultiStep (n : nat) : state → state :=
  match n with
  | O ⇒ fun S : state ⇒ S   | S m ⇒ fun S : state ⇒ Step (MultiStep m S)
  end.
```

A.2 Featherweight Typed Assembly Language

Syntax

```
Inductive Reg : Set := r0 : Reg | r1 : Reg | r2 : Reg | r3 : Reg | r4 : Reg
                     | r5 : Reg | r6 : Reg | r7 : Reg | r8 : Reg | r9 : Reg
                     | r10: Reg | r11 : Reg | r12 : Reg | r13 : Reg | r14 : Reg
                     | r15: Reg | r16 : Reg | r17 : Reg | r18 : Reg | r19 : Reg
                     | r20: Reg | r21 : Reg | r22 : Reg | r23 : Reg | r24 : Reg
                     | r25: Reg | r26 : Reg | r27 : Reg | r28 : Reg | r29 : Reg
                     | r30: Reg.

Syntactic Definition label := nat.
Syntactic Definition int := nat.
```

```
Syntactic Definition initflag := bool.

(* The allocation pointer register is not merged into the normal register file - it just contains a label
with a special type *)
Definition AP := label.
Inductive APTy : Set := fresh : APTy
                      | used : nat → APTy.

(* Lifted types *)
Inductive OmegaL : nat → Set
  := varL : (i:nat) (OmegaL (S i))
  | liftL : (i:nat) (OmegaL i) → (OmegaL (S i))
  | inttyL : (OmegaL O)
  | codetyL : (i:nat) (OmegaL_Map i) → APTy → (OmegaL i)
  | tuptyL : (i:nat) (OmegaL_List i) → (list initflag) → (OmegaL i)
  | rectyL : (i:nat) (OmegaL (S i)) → (OmegaL i)
with OmegaL_Map : nat → Set
  := memptyL : (i:nat) (OmegaL_Map i)
  | melemL : (i:nat) Reg → (OmegaL i) → (OmegaL_Map i) → (OmegaL_Map i)
with OmegaL_List : nat → Set
  := nilL : (i:nat) (OmegaL_List i)
  | consL : (i:nat) (OmegaL i) → (OmegaL_List i) → (OmegaL_List i).

Syntactic Definition OmegaR := (OmegaL (S O)).

(* The actual types *)
Inductive Omega : Set := intty : Omega
                       | codety : (Map Reg Omega) → APTy → Omega
                       | tupty : (list Omega) → (list initflag) → Omega
                       | recty : (OmegaL (S O)) → Omega.

(* deBruijn substitution functions *)
Load rftal.

Syntactic Definition RegFileTy := (Map Reg Omega).

Inductive WordVal : Set
  := wl : label → WordVal
  | wi : int → WordVal
  | wuninit : Omega → WordVal
  | wfold : WordVal → Omega → WordVal.

Inductive Instr : Set
  := add : Reg → Reg → Reg → Instr
  | addi : Reg → Reg → int → Instr
  | alloc : Reg → (list Omega) → Instr
```

```
  | bgt : Reg → Reg → label → Instr
  | bump : int → Instr
  | fold : Reg → Omega → Reg → Instr
  | ld : Reg → Reg → int → Instr
  | mov : Reg → Reg → Instr
  | movi : Reg → int → Instr
  | movl : Reg → label → Instr
  | st : Reg → int → Reg → Instr
  | unfold : Reg → Reg → Instr.

Inductive InstrSeq : Set
  := iseq : Instr → InstrSeq → InstrSeq
  | jd : label → InstrSeq
  | jmp : Reg → InstrSeq.

Inductive HeapVal : Set
  := tuple : (list WordVal) → HeapVal
  | code : RegFileTy → APTy → InstrSeq → HeapVal.

Definition RegFile := (Map Reg WordVal).

Definition Heap := (WFMap label HeapVal).

Syntactic Definition hunwrap := (unwrapMap label HeapVal).
Syntactic Definition hsize := (msize label HeapVal).
Syntactic Definition hindom := (mindom label HeapVal).
Syntactic Definition hnindom := (mnotindom label HeapVal).
Syntactic Definition hlookup := (mlookup label HeapVal).
Syntactic Definition hupdate := (mupdate label HeapVal).
Syntactic Definition hextend := (mextend label HeapVal).
Syntactic Definition hupdext := (mupdext label HeapVal).

Definition HeapTy := (WFMap label Omega).

Syntactic Definition htunwrap := (unwrapMap label Omega).
Syntactic Definition htsize := (msize label Omega).
Syntactic Definition htindom := (mindom label Omega).
Syntactic Definition htnindom := (mnotindom label Omega).
Syntactic Definition htlookup := (mlookup label Omega).
Syntactic Definition htupdate := (mupdate label Omega).
Syntactic Definition htextend := (mextend label Omega).
Syntactic Definition htupdext := (mupdext label Omega).

Syntactic Definition reglookup := (mlookup ? ?).
Syntactic Definition regupdext := (mupdext ? ?).

Definition Program := (Heap × (RegFile × (AP × InstrSeq))).
```

```
(* Stuff for the heap *)
Syntactic Definition HeapMap := (Map label HeapVal).

Inductive OrdHeap : HeapMap → Prop
  := ordheap0 : (OrdHeap (mempty ? ?))
  | ordheap1 : (l:label; hv:HeapVal; h:(Map label HeapVal))
                  (msize ? ? h l) → (OrdHeap h) → (OrdHeap (melem ? ? l hv h)).

Inductive PropSubHeap : HeapMap → HeapMap → Prop
  := propsubh0 : (h:HeapMap; l:label; hv:HeapVal) (PropSubHeap h (melem ? ? l hv h))
  | propsubh1 : (h,h':HeapMap; l:label; hv:HeapVal)
                   (PropSubHeap h h') → (PropSubHeap h (melem ? ? l hv h')).

Inductive SubHeap : HeapMap → HeapMap → Prop
  := subheap0 : (h:HeapMap) (SubHeap h h)
  | subheap1 : (h,h':HeapMap) (PropSubHeap h h') → (SubHeap h h').

(* Length and positions of an instruction sequence and other utilities *)
Inductive ISubDepth : InstrSeq → InstrSeq → nat → Prop
  := isubd0 : (I:InstrSeq) (ISubDepth I I O)
  | isubdS : (I,I':InstrSeq; i:Instr; n:nat)
                (ISubDepth I' I n) → (ISubDepth I' (iseq i I) (S n)).

Fixpoint lenInstrSeq [I:InstrSeq] : nat
  := Cases I of (iseq _ I') ⇒ (S (lenInstrSeq I'))
              | _ ⇒ (S O) end.

Fixpoint getInstrPos [n:nat] : InstrSeq → InstrSeq
  := [I] Cases n of O ⇒ I
                  | (S n') ⇒ Cases I of (iseq _ I') ⇒ (getInstrPos n' I')
                                      | _ ⇒ I end end.

Inductive ListNth [A:Set] : (list A) → nat → A → Prop
  := listnth0 : (V:(list A); a:A) (ListNth A (cons a V) O a)
  | listnth1 : (V:(list A); a,a':A; n:nat)
                  (ListNth A V n a') → (ListNth A (cons a V) (S n) a').

Fixpoint makeUninitTup [V:(list Omega)] : (list WordVal)
  := Cases V of nil ⇒ (nil WordVal) | (cons t V') ⇒ (cons (wuninit t) (makeUninitTup V')) end.

Fixpoint makeUninitTupty [V:(list Omega)] : (list initflag)
  := Cases V of nil ⇒ (nil initflag)
              | (cons t V') ⇒ (cons false (makeUninitTupty V'))
    end.

Inductive updatetuple
  : (list WordVal) → int → WordVal → (list WordVal) → Prop
```

```
  := updtup0 : (V:(list WordVal); w,w':(WordVal))
                  (updatetuple (cons w V) O w' (cons w' V))
  | updtup1 : (V,V':(list WordVal); w,w':(WordVal); i:int)
                  (updatetuple V i w' V') →
                  (updatetuple (cons w V) (S i) w' (cons w V')).

Inductive updatetupty    : (list initflag) → int → (list initflag) → Prop
  := updtpt0 : (V:(list initflag); b:initflag)
                  (updatetupty (cons b V) O (cons true V))
  | updtpt1 : (V,V':(list initflag); b:initflag; i:int)
                  (updatetupty V i V') →
                  (updatetupty (cons b V) (S i) (cons b V')).
```

Operational Semantics

```
Inductive Eval : Program → Program → Prop
  := ev_add : (H:Heap; R,R':RegFile; r31:AP; I':InstrSeq)
                  (rd,rs,rs':Reg; rsval,rsval':int)
                  (reglookup R rs (wi rsval)) →
                  (reglookup R rs' (wi rsval')) →
                  (regupdext R rd (wi (plus rsval rsval')) R') →
                  (Eval (H, (R, (r31, (iseq (add rd rs rs') I'))))
                        (H, (R', (r31, I'))))
  | ev_addi : (H:Heap; R,R':RegFile; r31:AP; I':InstrSeq)
                  (rd,rs:Reg; i:int; rsval:int)
                  (reglookup R rs (wi rsval)) →
                  (regupdext R rd (wi (plus rsval i)) R') →
                  (Eval (H, (R, (r31, (iseq (addi rd rs i) I'))))
                        (H, (R', (r31, I'))))

  | ev_alloc : (H,H':Heap; R,R':RegFile; r31:AP; I':InstrSeq)
                  (h,h':(Map label HeapVal); wfh:(mWF ? ? h); wfh':(mWF ? ? h'))
                  (rd:Reg; V:(list Omega))
                  H=(wfmap ? ? h wfh) →
                  H'=(wfmap ? ? h' wfh') →
                  (regupdext R rd (wl r31) R') →
                  (hextend h r31 (tuple (makeUninitTup V)) h') →
                  (Eval (H, (R, (r31, (iseq (alloc rd V) I'))))
                        (H', (R', (r31, I'))))
  | ev_bgt0 : (H:Heap; R:RegFile; r31:AP; I':InstrSeq)
                  (rs,rt:Reg; l:label; rsval,rtval:int)
                  (reglookup R rs (wi rsval)) →
```

(*reglookup R rt* (*wi rtval*)) →
(*le rsval rtval*) →
(*Eval* (*H*, (*R*, (*r31*, (*iseq* (*bgt rs rt l*) *I′*))))
(*H*, (*R*, (*r31*, *I′*))))

| *ev_bgt1* : (*H*:*Heap*; *R*:*RegFile*; *r31*:*AP*; *I′*:*InstrSeq*)
(*G*:*RegFileTy*; *T*:*APTy*; *I″*:*InstrSeq*)
(*rs*,*rt*:*Reg*; *l*:*label*; *rsval*,*rtval*:*int*)
(*reglookup R rs* (*wi rsval*)) →
(*reglookup R rt* (*wi rtval*)) →
(*gt rsval rtval*) →
(*hlookup* (*hunwrap H*) *l* (*code G T I″*)) →
(*Eval* (*H*, (*R*, (*r31*, (*iseq* (*bgt rs rt l*) *I′*))))
(*H*, (*R*, (*r31*, *I″*))))

| *ev_bump* : (*H*:*Heap*; *R*:*RegFile*; *r31*:*AP*; *I′*:*InstrSeq*)
(*i*:*int*; *l*:*nat*)
(*hsize* (*hunwrap H*) *l*) →
(*Eval* (*H*, (*R*, (*r31*, (*iseq* (*bump i*) *I′*))))
(*H*, (*R*, (*l*, *I′*))))
| *ev_fold* : (*H*:*Heap*; *R*,*R′*:*RegFile*; *r31*:*AP*; *I′*:*InstrSeq*)
(*rd*,*rs*:*Reg*; *t*:*Omega*; *rsval*:*WordVal*)
(*reglookup R rs rsval*) →
(*regupdext R rd* (*wfold rsval t*) *R′*) →
(*Eval* (*H*, (*R*, (*r31*, (*iseq* (*fold rd t rs*) *I′*))))
(*H*, (*R′*, (*r31*, *I′*))))

| *ev_jd* : (*H*:*Heap*; *R*:*RegFile*; *r31*:*AP*)
(*l*:*label*; *G*:*RegFileTy*; *T*:*APTy*; *I′*:*InstrSeq*)
(*hlookup* (*hunwrap H*) *l* (*code G T I′*)) →
(*Eval* (*H*, (*R*, (*r31*, (*jd l*))))
(*H*, (*R*, (*r31*, *I′*))))
| *ev_jmp* : (*H*:*Heap*; *R*:*RegFile*; *r31*:*AP*)
(*r*:*Reg*; *l*:*label*; *G*:*RegFileTy*; *T*:*APTy*; *I′*:*InstrSeq*)
(*reglookup R r* (*wl l*)) →
(*hlookup* (*hunwrap H*) *l* (*code G T I′*)) →
(*Eval* (*H*, (*R*, (*r31*, (*jmp r*))))
(*H*, (*R*, (*r31*, *I′*))))

| *ev_ld* : (*H*:*Heap*; *R*,*R′*:*RegFile*; *r31*:*AP*; *I′*:*InstrSeq*)
(*rd*,*rs*:*Reg*; *i*:*int*)
(*l*:*label*; *V*:(*list WordVal*); *v*:*WordVal*)
(*reglookup R rs* (*wl l*)) →

(*hlookup* (*hunwrap H*) *l* (*tuple V*)) →
(*ListNth* ? *V i v*) →
(*regupdext R rd v R′*) →
(*Eval* (*H*, (*R*, (*r31*, (*iseq* (*ld rd rs i*) *I′*))))
(*H*, (*R′*,(*r31*, *I′*))))

| *ev_mov* : (*H*:*Heap*; *R*,*R′*:*RegFile*; *r31*:*AP*; *I′*:*InstrSeq*)
(*rd*,*rs*:*Reg*; *v*:*WordVal*)
(*reglookup R rs v*) →
(*regupdext R rd v R′*) →
(*Eval* (*H*, (*R*, (*r31*, (*iseq* (*mov rd rs*) *I′*))))
(*H*, (*R′*,(*r31*, *I′*))))
| *ev_movi* : (*H*:*Heap*; *R*,*R′*:*RegFile*; *r31*:*AP*; *I′*:*InstrSeq*)
(*rd*:*Reg*; *i*:*int*)
(*regupdext R rd* (*wi i*) *R′*) →
(*Eval* (*H*, (*R*, (*r31*, (*iseq* (*movi rd i*) *I′*))))
(*H*, (*R′*,(*r31*, *I′*))))
| *ev_movl* : (*H*:*Heap*; *R*,*R′*:*RegFile*; *r31*:*AP*; *I′*:*InstrSeq*)
(*rd*:*Reg*; *l*:*label*)
(*regupdext R rd* (*wl l*) *R′*) →
(*Eval* (*H*, (*R*, (*r31*, (*iseq* (*movl rd l*) *I′*))))
(*H*, (*R′*,(*r31*, *I′*))))

| *ev_store* : (*H*,*H′*:*Heap*; *R*:*RegFile*; *r31*:*AP*; *I′*:*InstrSeq*)
(*h*,*h′*:(*Map label HeapVal*); *wfh*:(*mWF* ? ? *h*); *wfh′*:(*mWF* ? ? *h′*))
(*rd*,*rs*:*Reg*; *i*:*int*; *l*:*label*; *V*,*V′*:(*list WordVal*); *w*:*WordVal*)
H=(*wfmap* ? ? *h wfh*) →
H′=(*wfmap* ? ? *h′ wfh′*) →
(*reglookup R rd* (*wl l*)) →
(*reglookup R rs w*) →
(*hlookup h l* (*tuple V*)) →
(*updatetuple V i w V′*) →
(*hupdate h l* (*tuple V′*) *h′*) →
(*Eval* (*H*, (*R*, (*r31*, (*iseq* (*st rd i rs*) *I′*))))
(*H′*,(*R*, (*r31*, *I′*))))

| *ev_unfold* : (*H*:*Heap*; *R*,*R′*:*RegFile*; *r31*:*AP*; *I′*:*InstrSeq*)
(*rd*,*rs*:*Reg*; *v*:*WordVal*; *t*:*Omega*)
(*reglookup R rs* (*wfold v t*)) →
(*regupdext R rd v R′*) →
(*Eval* (*H*, (*R*, (*r31*, (*iseq* (*unfold rd rs*) *I′*))))
(*H*, (*R′*, (*r31*, *I′*)))).

Static Semantics

```
(* Register file subtyping *)
Inductive RegFileSubtype : RegFileTy → RegFileTy → Prop
  := weaken : (G,G':RegFileTy)
                ((r:Reg; t:Omega) (mlookup ? ? G' r t) → (mlookup ? ? G r t)) →
                (RegFileSubtype G G').

(* Subtyping *)
Inductive SubtypeB : initflag → initflag → Prop
  := subb_refl : (i:initflag) (SubtypeB i i)
  | subb_true : (SubtypeB true false).

Inductive SubtypeBList : (list initflag) → (list initflag) → Prop
  := subb_nil : (SubtypeBList (nil ?) (nil ?))
  | subb_cons : (L1,L2:(list initflag); i1,i2:initflag)
                  (SubtypeB i1 i2) → (SubtypeBList L1 L2) → (SubtypeBList (cons i1 L1)
(cons i2 L2)).

Inductive Subtype : Omega → Omega → Prop
  := reflex : (t:Omega) (Subtype t t)
  | tuple_sub : (ol:(list Omega); il1,il2:(list initflag))
                  (SubtypeBList il1 il2) → (Subtype (tupty ol il1) (tupty ol il2)).

(* Well-formed word values *)
Inductive WFWordVal : HeapTy → WordVal → Omega → Prop
  := int_wval : (HT:HeapTy; i:int) (WFWordVal HT (wi i) intty)
  | label_wval : (HT:HeapTy; l:label; t,t':Omega)
                   (htlookup (htunwrap HT) l t') →
                   (Subtype t' t) →
                   (WFWordVal HT (wl l) t)
  | fold_word_wval
                 : (HT:HeapTy)
                   (w:WordVal; t:OmegaR; t':Omega)
                   (RUnlift (RUnfold t))=t' →
                   (WFWordVal HT w t') →
                   (WFWordVal HT (wfold w (recty t)) (recty t)).

Inductive WFWordValinit : HeapTy → WordVal → Omega → initflag → Prop
  := init : (HT:HeapTy)
                   (w:WordVal; t:Omega; f:initflag)
                   (WFWordVal HT w t) →
                   (WFWordValinit HT w t f)
  | uninit : (HT:HeapTy)
                   (t:Omega)
```

```
                    (WFWordValinit HT (wuninit t) t false).

(* Well-formed instruction sequences *)
Inductive WFInstrSeq : HeapTy → RegFileTy → APTy → InstrSeq → Prop
  := s_add : (HT:HeapTy; G,G':RegFileTy; T:APTy; I:InstrSeq)
                    (rd,rs,rs':Reg)
                    (mlookup ? ? G rs intty) →
                    (mlookup ? ? G rs' intty) →
                    (mupdext ? ? G rd intty G') →
                    (WFInstrSeq HT G' T I) →
                    (WFInstrSeq HT G T (iseq (add rd rs rs') I))
  | s_addi : (HT:HeapTy; G,G':RegFileTy; T:APTy; I:InstrSeq)
                    (rd,rs:Reg; i:int)
                    (mlookup ? ? G rs intty) →
                    (mupdext ? ? G rd intty G') →
                    (WFInstrSeq HT G' T I) →
                    (WFInstrSeq HT G T (iseq (addi rd rs i) I))
  | s_alloc : (HT:HeapTy; G,G':RegFileTy; I:InstrSeq)
                    (rd:Reg; n:nat; V:(list Omega))
                    n=(length V) →
                    (mupdext ? ? G rd (tupty V (makeUninitTupty V)) G') →
                    (WFInstrSeq HT G' (used n) I) →
                    (WFInstrSeq HT G fresh (iseq (alloc rd V) I))
  | s_bgt : (HT:HeapTy; G,G':RegFileTy; T:APTy; I:InstrSeq)
                    (rs,rt:Reg; l:label)
                    (mlookup ? ? G rs intty) →
                    (mlookup ? ? G rt intty) →
                    (htlookup (htunwrap HT) l (codety G' T)) →
                    (RegFileSubtype G G') →
                    (WFInstrSeq HT G T I) →
                    (WFInstrSeq HT G T (iseq (bgt rs rt l) I))
  | s_bump : (HT:HeapTy; G:RegFileTy; I:InstrSeq; n:nat)
                    (WFInstrSeq HT G fresh I) →
                    (WFInstrSeq HT G (used n) (iseq (bump n) I))
  | s_fold : (HT:HeapTy; G,G':RegFileTy; T:APTy; I:InstrSeq)
                    (rd,rs:Reg; f:OmegaR; rst:Omega)
                    (mlookup ? ? G rs rst) →
                    (RUnlift (RUnfold f))=rst →
                    (mupdext ? ? G rd (recty f) G') →
                    (WFInstrSeq HT G' T I) →
                    (WFInstrSeq HT G T (iseq (fold rd (recty f) rs) I))
```

| *s_jd* : (*HT*:*HeapTy*; *G*,*G′*:*RegFileTy*; *T*:*APTy*)
(*l*:*label*)
(*htlookup* (*htunwrap HT*) *l* (*codety G′ T*)) →
(*RegFileSubtype G G′*) →
(*WFInstrSeq HT G T* (*jd l*))
| *s_jmp* : (*HT*:*HeapTy*; *G*,*G′*:*RegFileTy*; *T*:*APTy*)
(*r*:*Reg*)
(*mlookup* ? ? *G r* (*codety G′ T*)) →
(*RegFileSubtype G G′*) →
(*WFInstrSeq HT G T* (*jmp r*))
| *s_ld* : (*HT*:*HeapTy*; *G*,*G′*:*RegFileTy*; *T*:*APTy*; *I*:*InstrSeq*)
(*rd*,*rs*:*Reg*; *i*:*int*)
(*ol*:(*list Omega*); *il*:(*list initflag*); *t*:*Omega*)
(*mlookup* ? ? *G rs* (*tupty ol il*)) →
(*ListNth* ? *ol i t*) →
(*ListNth* ? *il i true*) →
(*mupdext* ? ? *G rd t G′*) →
(*WFInstrSeq HT G′ T I*) →
(*WFInstrSeq HT G T* (*iseq* (*ld rd rs i*) *I*))
| *s_mov* : (*HT*:*HeapTy*; *G*,*G′*:*RegFileTy*; *T*:*APTy*; *I*:*InstrSeq*)
(*rd*,*rs*:*Reg*; *t*:*Omega*)
(*mlookup* ? ? *G rs t*) →
(*mupdext* ? ? *G rd t G′*) →
(*WFInstrSeq HT G′ T I*) →
(*WFInstrSeq HT G T* (*iseq* (*mov rd rs*) *I*))
| *s_movi* : (*HT*:*HeapTy*; *G*,*G′*:*RegFileTy*; *T*:*APTy*; *I*:*InstrSeq*)
(*rd*:*Reg*; *i*:*int*)
(*mupdext* ? ? *G rd intty G′*) →
(*WFInstrSeq HT G′ T I*) →
(*WFInstrSeq HT G T* (*iseq* (*movi rd i*) *I*))
| *s_movl* : (*HT*:*HeapTy*; *G*,*G′*:*RegFileTy*; *T*:*APTy*; *I*:*InstrSeq*)
(*rd*:*Reg*; *l*:*label*; *t*,*t′*:*Omega*)
(*htlookup* (*htunwrap HT*) *l t*) →
(*Subtype t t′*) →
(*mupdext* ? ? *G rd t′ G′*) →
(*WFInstrSeq HT G′ T I*) →
(*WFInstrSeq HT G T* (*iseq* (*movl rd l*) *I*))
| *s_st* : (*HT*:*HeapTy*; *G*,*G′*:*RegFileTy*; *T*:*APTy*; *I*:*InstrSeq*)
(*rd*,*rs*:*Reg*; *i*:*int*;
V,*V′*:(*list initflag*); *Ts*:(*list Omega*); *t*:*Omega*)

```
                    (mlookup ? ? G rd (tupty Ts V)) →
                    (mlookup ? ? G rs t) →
                    (ListNth ? Ts i t) →
                    (updatetupty V i V') →
                    (mupdate ? ? G rd (tupty Ts V') G') →
                    (WFInstrSeq HT G' T I) →
                    (WFInstrSeq HT G T (iseq (st rd i rs) I))

  | s_unfold : (HT:HeapTy; G,G':RegFileTy; T:APTy; I:InstrSeq)
                    (rd,rs:Reg; f:OmegaR; unf,rst:Omega)
                    (mlookup ? ? G rs rst) →
                    rst=(recty f) →
                    (RUnlift (RUnfold f))=unf →
                    (mupdext ? ? G rd unf G') →
                    (WFInstrSeq HT G' T I) →
                    (WFInstrSeq HT G T (iseq (unfold rd rs) I)).

Inductive WFWordValinitList
  : HeapTy → (list WordVal) → (list Omega) → (list initflag) → Prop
  := wfwvilist0 : (HT:HeapTy)
                    (WFWordValinitList HT (nil ?) (nil ?) (nil ?))
  | wfwvilist1 : (HT:HeapTy;
                     wl:(list WordVal); tl:(list Omega); il:(list initflag))
                    (w:WordVal; t:Omega; i:initflag)
                    (WFWordValinit HT w t i) →
                    (WFWordValinitList HT wl tl il) →
                    (WFWordValinitList HT (cons w wl) (cons t tl) (cons i il)).

Inductive WFHeapVal : HeapTy → HeapVal → Omega → Prop
  := tuple_wf : (HT:HeapTy; wl:(list WordVal);
                     tl:(list Omega); il:(list initflag))
                    (WFWordValinitList HT wl tl il) →
                    (WFHeapVal HT (tuple wl) (tupty tl il))
  | code_wf : (HT:HeapTy; G:RegFileTy; I:InstrSeq; T:APTy)
                    (WFInstrSeq HT G T I) →
                    (WFHeapVal HT (code G T I) (codety G T)).

Inductive WFHeap : Heap → HeapTy → Prop
  := heap_wf : (H:Heap; HT:HeapTy)
                     (EX s | (hsize (hunwrap H) s) ∧
                              (htsize (htunwrap HT) s) ∧
                              ((n:label; h:HeapVal)
                                (hlookup (hunwrap H) n h) → (lt n s)) ∧
```

```
                    ((n:label; t:Omega)
                     (htlookup (htunwrap HT) n t) → (lt n s)) ∧
                    ((n:label) (lt n s) →
                                (EX h | (hlookup (hunwrap H) n h))) ∧
                    ((n:label) (lt n s) →
                                (EX t | (htlookup (htunwrap HT) n t))) ∧
                    ((n:label; h:HeapVal; t:Omega)
                                 (hlookup (hunwrap H) n h) →
                                 (htlookup (htunwrap HT) n t) →
                                 (WFHeapVal HT h t)) ∧
                    (OrdHeap (hunwrap H))
                ) →
                (WFHeap H HT).

Inductive WFap : HeapTy → AP → APTy → Prop
  := fresh_wf : (HT:HeapTy; l:AP)
                (htsize (htunwrap HT) l) →
                (WFap HT l fresh)
  | used_wf : (HT:HeapTy; l:AP; n:nat; tl:(list Omega); ol:(list initflag))
                n=(length tl) →
                (htsize (htunwrap HT) (S l)) →
                (WFWordVal HT (wl l) (tupty tl ol)) →
                (WFap HT l (used n)).

Fixpoint stripWV [w:WordVal] : WordVal
  := Cases w of (wfold w' t) ⇒ (stripWV w')
              | _ ⇒ w
     end.

Inductive WFRegFile : HeapTy → RegFile → RegFileTy → Prop
  := regfile_wf : (HT:HeapTy; R:RegFile; G:RegFileTy)
                  ((r:Reg; t:Omega)
                   (mlookup ? ? G r t) →
                      (EX w | (mlookup ? ? R r w) ∧ (WFWordVal HT w t))) →
                  ((r:Reg; v:WordVal; l:label; n:nat)
                   (mlookup ? ? R r v) →
                   (stripWV v)=(wl l) →
                   (htsize (htunwrap HT) n) →
                   (lt l n)) →
                  (WFRegFile HT R G).

(* Well-formed program *)
```

Inductive *WFProgram* : *Program* → *Prop*
:= *program_wf* : (*H*:*Heap*; *HT*:*HeapTy*; *R*:*RegFile*; *G*:*RegFileTy*;
l:*AP*; *t*:*APTy*; *I*:*InstrSeq*)
(*WFHeap H HT*) →
(*WFRegFile HT R G*) →
(*WFap HT l t*) →
(*WFInstrSeq HT G t I*) →
(*EX l* | (*EX G′* | (*EX T′* | (*EX I′* | (*EX n* —
(*hlookup* (*hunwrap H*) *l* (*code G′ T′ I′*)) ∧
(*ISubDepth I I′ n*)))))) →
(*WFProgram* (*H*, (*R*, (*l*, *I*)))).

Soundness Proofs

Utility lemmas

Lemma *regfile_ext_eq_instr_wf*
: (*I*:*InstrSeq*; *HT*:*HeapTy*; *G1*,*G2*:*RegFileTy*; *T*:*APTy*)
((*r*:*Reg*; *t*:*Omega*) (*mlookup* ? ? *G1 r t*) → (*mlookup* ? ? *G2 r t*)) →
(*WFInstrSeq HT G1 T I*) →
(*WFInstrSeq HT G2 T I*).

Lemma *wfwordval_label_lt_heap_size*
: (*H*:*Heap*; *HT*:*HeapTy*; *v*:*WordVal*; *l*:*label*; *t*:*Omega*; *n*:*nat*)
(*WFHeap H HT*) → (*WFWordVal HT v t*) → (*stripWV v*)=(*wl l*) → (*htsize* (*htunwrap HT*) *n*) → (*lt l n*).

Lemma *regfile_upd_wf*
: (*H*:*Heap*; *HT*:*HeapTy*; *R*,*R′*:*RegFile*; *G*,*G′*:(*Map Reg Omega*))
(*rd*:*Reg*; *w*:*WordVal*; *t*:*Omega*)
(*WFHeap H HT*) →
(*WFRegFile HT R G*) →
(*WFWordVal HT w t*) →
(*mupdext* ? ? *R rd w R′*) →
(*mupdext* ? ? *G rd t G′*) →
(*WFRegFile HT R′ G′*).

Lemma *heap_lookup_not_recty*
: (*H*:*Heap*; *HT*:*HeapTy*; *l*:*label*; *t*:*Omega*; *f*:*OmegaR*)
(*WFHeap H HT*) →
(*htlookup* (*htunwrap HT*) *l t*) →
¬*t*=(*recty f*).

Lemma *heap_lookup_not_intty*

```
  : (H:Heap; HT:HeapTy; l:label; t:Omega)
    (WFHeap H HT) →
    (htlookup (htunwrap HT) l t) →
    ¬t=intty.

(* Canonical forms lemmas *)
Lemma can_word_forms_rec
  : (H:Heap; HT:HeapTy; w:WordVal; f:OmegaR)
    (WFHeap H HT) →
    (WFWordVal HT w (recty f)) →
    (EX v | (EX t | w=(wfold v t))).

Lemma can_word_forms_int
  : (H:Heap; HT:HeapTy; w:WordVal; t:Omega)
    (WFHeap H HT) →
    (WFWordVal HT w t) →
    (t=intty) →
    (EX i | w=(wi i)).

Lemma can_reg_forms_any
  : (HT:HeapTy; R:RegFile; G:(Map Reg Omega);
     r:Reg; t:Omega)
    (WFRegFile HT R G) →
    (mlookup Reg Omega G r t) →
    (EX w | (mlookup Reg WordVal R r w)).

Lemma can_reg_forms_tuple
  : (HT:HeapTy; R:RegFile; G:RegFileTy;
     r:Reg; tl:(list Omega); il:(list initflag))
    (WFRegFile HT R G) →
    (mlookup Reg Omega G r (tupty tl il)) →
    (EX l | (mlookup ? ? R r (wl l))).

Lemma tupty_lists_length_eq
  : (H:Heap; HT:HeapTy; R:RegFile; G:RegFileTy;
       r:Reg; l:label; tl:(list Omega); il:(list initflag); V:(list WordVal))
    (WFHeap H HT) →
    (WFRegFile HT R G) →
    (mlookup Reg Omega G r (tupty tl il)) →
    (mlookup ? ? R r (wl l)) →
    (hlookup (hunwrap H) l (tuple V)) →
    (length V)=(length tl).

Lemma can_heap_forms_reg_tuple
  : (H:Heap; HT:HeapTy; R:RegFile; G:RegFileTy;
```

r:*Reg*; *l*:*label*; *tl*:(*list Omega*); *il*:(*list initflag*))
(*WFHeap H HT*) →
(*WFRegFile HT R G*) →
(*mlookup* ? ? *G r* (*tupty tl il*)) →
(*mlookup* ? ? *R r* (*wl l*)) →
(*EX V* | (*hlookup* (*hunwrap H*) *l* (*tuple V*))).

Lemma *can_heap_forms_code*
: (*HT*:*HeapTy*; *H*:*Heap*; *l*:*label*; *G*:(*Map Reg Omega*); *T*:*APTy*)
(*WFHeap H HT*) →
(*mlookup nat Omega* (*unwrapMap nat Omega HT*) *l* (*codety G T*)) →
(*EX I* | (*mlookup* ? ? (*unwrapMap* ? ? *H*) *l* (*code G T I*))).

Lemma *can_heap_forms_code_I*
: (*HT*:*HeapTy*; *h*:*HeapVal*; *t*:*Omega*; *G*:*RegFileTy*; *I*:*InstrSeq*; *T*:*APTy*)
(*WFHeapVal HT h t*) →
t=(*codety G T*) →
h=(*code G T I*) →
(*WFInstrSeq HT G T I*).

Lemma *can_word_forms_code*
: (*H*:*Heap*; *HT*:*HeapTy*; *w*:*WordVal*; *G*:*RegFileTy*; *T*:*APTy*)
(*WFHeap H HT*) →
(*WFWordVal HT w* (*codety G T*)) →
(*EX l* | *w*=(*wl l*) ∧
(*EX I* | (*mlookup* ? ? (*unwrapMap* ? ? *H*) *l* (*code G T I*)))).

Lemma *wfwordval_in_reg*
: (*HT*:*HeapTy*; *R*:*RegFile*; *G*:*RegFileTy*)
(*r*:*Reg*; *w*:*WordVal*; *t*:*Omega*)
(*WFRegFile HT R G*) →
(*mlookup* ? ? *G r t*) →
(*mlookup* ? ? *R r w*) →
(*WFWordVal HT w t*).

Lemma *wfheapval_in_heap*
: (*H*:*Heap*; *HT*:*HeapTy*; *l*:*label*; *h*:*HeapVal*; *t*:*Omega*)
(*WFHeap H HT*) →
(*hlookup* (*hunwrap H*) *l h*) →
(*htlookup* (*htunwrap HT*) *l t*) →
(*WFHeapVal HT h t*).

(Reasoning about the heap *)*

Lemma *size_notin_wfheap*

: (*H*:*Heap*; *HT*:*HeapTy*; *h*:(*Map label HeapVal*); *wfh*:(*mWF ? ? h*); *l*:*label*)
H=(*wfmap label HeapVal h wfh*) → (*WFHeap H HT*) → (*msize ? ? h l*) → (*mnotindom ? ? h l*).

Lemma *size_notin_wfheapty*
: (*H*:*Heap*; *HT*:*HeapTy*; *ht*:(*Map label Omega*); *wfht*:(*mWF ? ? ht*); *l*:*label*)
HT=(*wfmap label Omega ht wfht*) → (*WFHeap H HT*) → (*msize ? ? ht l*) → (*mnotindom ? ? ht l*).

Lemma *wfh_ht_size_eq*
: (*h*:(*Map label HeapVal*); *wfh*:(*mWF ? ? h*); *ht*:(*Map label Omega*); *wfht*:(*mWF ? ? ht*); *l*:*nat*)
(*WFHeap* (*wfmap ? ? h wfh*) (*wfmap ? ? ht wfht*)) → (*hsize h l*) → (*htsize ht l*).

Lemma *wfh_h_size_eq*
: (*h*:(*Map label HeapVal*); *wfh*:(*mWF ? ? h*); *ht*:(*Map label Omega*); *wfht*:(*mWF ? ? ht*); *l*:*nat*)
(*WFHeap* (*wfmap ? ? h wfh*) (*wfmap ? ? ht wfht*)) → (*htsize ht l*) → (*hsize h l*).

Lemma *makeUninitTupty_len_eq*
: (*n*:*nat*; *V*:(*list Omega*)) *n*=(*length V*) → *n*=(*length* (*makeUninitTupty V*)).

Lemma *makeUninitWF*
: (*HT*:*HeapTy*; *V*:(*list Omega*)) (*WFHeapVal HT* (*tuple* (*makeUninitTup V*)) (*tupty V* (*make-UninitTupty V*))).

Lemma *reg_tuple_ex_heaptype*
: (*ht*:(*Map label Omega*); *wfht*:(*mWF ? ? ht*); *R*:*RegFile*; *G*:*RegFileTy*; *r*:*Reg*; *tl*:(*list Omega*); *il*:(*list initflag*); *l*:*label*)
(*WFRegFile* (*wfmap ? ? ht wfht*) *R G*) →
(*mlookup ? ? G r* (*tupty tl il*)) →
(*mlookup ? ? R r* (*wl l*)) →
(*EX s*:*Omega* | (*mlookup nat Omega ht l s*) ∧ (*Subtype s* (*tupty tl il*))).

Heap extension lemmas

Lemma *heap_ext_6*
: (*h*,*h'*:(*Map label HeapVal*); *wfh*:(*mWF ? ? h*); *wfh'*:(*mWF ? ? h'*);
ht,*ht'*:(*Map label Omega*); *wfht*:(*mWF ? ? ht*); *wfht'*:(*mWF ? ? ht'*))
(*t*:*Omega*; *l*:*label*)
(*wv*:*WordVal*; *t'*:*Omega*)
(*WFHeap* (*wfmap ? ? h wfh*) (*wfmap ? ? ht wfht*)) →
(*hsize h l*) →
(*htextend ht l t ht'*) →
(*WFWordVal* (*wfmap ? ? ht wfht*) *wv t'*) →
(*WFWordVal* (*wfmap ? ? ht' wfht'*) *wv t'*).

Lemma *heap_ext_7*
: (*h*,*h'*:(*Map label HeapVal*); *wfh*:(*mWF ? ? h*); *wfh'*:(*mWF ? ? h'*);

ht,ht':(*Map label Omega*); *wfht*:(*mWF ? ? ht*); *wfht'*:(*mWF ? ? ht'*))
(*t*:*Omega*; *l*:*label*)
(*wv*:*WordVal*; *t'*:*Omega*; *b*:*initflag*)
(*WFHeap* (*wfmap ? ? h wfh*) (*wfmap ? ? ht wfht*)) →
(*hsize h l*) →
(*htextend ht l t ht'*) →
(*WFWordValinit* (*wfmap ? ? ht wfht*) *wv t' b*) →
(*WFWordValinit* (*wfmap ? ? ht' wfht'*) *wv t' b*).

Lemma *heap_ext_4*
: (*I*:*InstrSeq*)
(*h,h'*:(*Map label HeapVal*); *wfh*:(*mWF ? ? h*); *wfh'*:(*mWF ? ? h'*);
ht,ht':(*Map label Omega*); *wfht*:(*mWF ? ? ht*); *wfht'*:(*mWF ? ? ht'*))
(*t*:*Omega*; *l*:*label*)
(*R*:*RegFileTy*; *A*:*APTy*)
(*WFHeap* (*wfmap ? ? h wfh*) (*wfmap ? ? ht wfht*)) →
(*hsize h l*) →
(*htextend ht l t ht'*) →
(*WFInstrSeq* (*wfmap ? ? ht wfht*) *R A I*) →
(*WFInstrSeq* (*wfmap ? ? ht' wfht'*) *R A I*).

Lemma *heap_ext_5_aux*
: (*h,h'*:(*Map label HeapVal*); *wfh*:(*mWF ? ? h*); *wfh'*:(*mWF ? ? h'*);
ht,ht':(*Map label Omega*); *wfht*:(*mWF ? ? ht*); *wfht'*:(*mWF ? ? ht'*))
(*wl*:(*list WordVal*); *tl*:(*list Omega*); *il*:(*list initflag*))
(*t*:*Omega*; *l*:*label*)
(*WFHeap* (*wfmap ? ? h wfh*) (*wfmap ? ? ht wfht*)) →
(*hsize h l*) →
(*htextend ht l t ht'*) →
(*WFWordValinitList* (*wfmap ? ? ht wfht*) *wl tl il*) →
(*WFWordValinitList* (*wfmap ? ? ht' wfht'*) *wl tl il*).

Lemma *heap_ext_3*
: (*h,h'*:(*Map label HeapVal*); *wfh*:(*mWF ? ? h*); *wfh'*:(*mWF ? ? h'*);
ht,ht':(*Map label Omega*); *wfht*:(*mWF ? ? ht*); *wfht'*:(*mWF ? ? ht'*))
(*R,R'*:*RegFile*; *G,G'*:(*Map Reg Omega*))
(*rd*:*Reg*; *l*:*nat*; *tl*:(*list Omega*); *il*:(*list initflag*))
(*WFHeap* (*wfmap ? ? h wfh*) (*wfmap ? ? ht wfht*)) →
(*hsize h l*) →
(*WFRegFile* (*wfmap ? ? ht wfht*) *R G*) →
(*mupdext Reg WordVal R rd* (*wl l*) *R'*) →
(*mupdext Reg Omega G rd* (*tupty tl il*) *G'*) →
(*mextend nat Omega ht l* (*tupty tl il*) *ht'*) →

(*WFRegFile* (*wfmap* ? ? *ht′ wfht′*) *R′ G′*).

Lemma *heap_ext_3ap*
: (*HT*,*HT′*:*HeapTy*; *a*:*AP*; *tl*:(*list Omega*); *il*:(*list initflag*); *n*:*nat*)
(*WFap HT a fresh*) →
n=(*length tl*) →
(*mextend nat Omega* (*unwrapMap nat Omega HT*) *a* (*tupty tl il*)
(*unwrapMap nat Omega HT′*)) →
(*WFap HT′ a* (*used n*)).

Lemma *heap_ext_5*
: (*h*,*h′*:(*Map label HeapVal*); *wfh*:(*mWF* ? ? *h*); *wfh′*:(*mWF* ? ? *h′*);
ht,*ht′*:(*Map label Omega*); *wfht*:(*mWF* ? ? *ht*); *wfht′*:(*mWF* ? ? *ht′*))
(*hv*:*HeapVal*; *t*,*t′*:*Omega*; *l*:*label*)
(*WFHeap* (*wfmap* ? ? *h wfh*) (*wfmap* ? ? *ht wfht*)) →
(*hsize h l*) →
(*htextend ht l t ht′*) →
(*WFHeapVal* (*wfmap* ? ? *ht wfht*) *hv t′*) →
(*WFHeapVal* (*wfmap* ? ? *ht′ wfht′*) *hv t′*).

Lemma *heap_ext_2*
: (*h*,*h′*:(*Map label HeapVal*); *wfh*:(*mWF* ? ? *h*); *wfh′*:(*mWF* ? ? *h′*);
ht,*ht′*:(*Map label Omega*); *wfht*:(*mWF* ? ? *ht*); *wfht′*:(*mWF* ? ? *ht′*))
(*hv*:*HeapVal*; *t*:*Omega*; *l*:*label*)
(*WFHeap* (*wfmap* ? ? *h wfh*) (*wfmap* ? ? *ht wfht*)) →
(*hsize h l*) →
(*htextend ht l t ht′*) →
(*hextend h l hv h′*) →
(*WFHeapVal* (*wfmap* ? ? *ht′ wfht′*) *hv t*) →
(*WFHeap* (*wfmap* ? ? *h′ wfh′*) (*wfmap* ? ? *ht′ wfht′*)).

Lemma *heap_ext_1*
: (*H*:*Heap*; *HT*:*HeapTy*; *ht*:(*Map label Omega*); *wfht*:(*mWF* ? ? *ht*);
l:*label*; *t*:*Omega*)
HT=(*wfmap label Omega ht wfht*) →
(*WFHeap H HT*) →
(*msize* ? ? *ht l*) →
(*EX ht′* | (*EXT wfht′* | (*EX HT′*:*HeapTy* |
HT′=(*wfmap label Omega ht′ wfht′*) ∧
(*mextend* ? ? *ht l t ht′*)))).

Heap update lemmas

Lemma *subtype_update*
: (*i*:*int*; *il1*,*il1′*,*il0*,*il0′*:(*list initflag*))

(*SubtypeBList il1 il0*) → (*updatetupty il1 i il1′*) → (*updatetupty il0 i il0′*) →
(*SubtypeBList il1′ il0′*).

Lemma *subtypeb_trans*
: (*b*,*a*,*c*:*initflag*)(*SubtypeB a b*) → (*SubtypeB b c*) → (*SubtypeB a c*).

Lemma *subtypeblist_trans*
: (*il2*,*il1*,*il3*:(*list initflag*))
(*SubtypeBList il1 il2*) → (*SubtypeBList il2 il3*) → (*SubtypeBList il1 il3*).

Lemma *subtype_trans*
: (*t1*,*t2*,*t3*:*Omega*) (*Subtype t1 t2*) → (*Subtype t2 t3*) → (*Subtype t1 t3*).

Lemma *heap_upd_6*
: (*h*,*h′*:(*Map label HeapVal*); *wfh*:(*mWF ? ? h*); *wfh′*:(*mWF ? ? h′*);
ht,*ht′*:(*Map label Omega*); *wfht*:(*mWF ? ? ht*); *wfht′*:(*mWF ? ? ht′*))
(*t*,*s*:*Omega*; *l*:*label*)
(*wv*:*WordVal*; *t′*:*Omega*)
(*WFHeap* (*wfmap ? ? h wfh*) (*wfmap ? ? ht wfht*)) →
(*htlookup ht l s*) →
(*Subtype t s*) →
(*htupdate ht l t ht′*) →
(*WFWordVal* (*wfmap ? ? ht wfht*) *wv t′*) →
(*WFWordVal* (*wfmap ? ? ht′ wfht′*) *wv t′*).

Lemma *heap_upd_7*
: (*h*,*h′*:(*Map label HeapVal*); *wfh*:(*mWF ? ? h*); *wfh′*:(*mWF ? ? h′*);
ht,*ht′*:(*Map label Omega*); *wfht*:(*mWF ? ? ht*); *wfht′*:(*mWF ? ? ht′*))
(*t*,*s*:*Omega*; *l*:*label*)
(*wv*:*WordVal*; *t′*:*Omega*; *b*:*initflag*)
(*WFHeap* (*wfmap ? ? h wfh*) (*wfmap ? ? ht wfht*)) →
(*htlookup ht l s*) →
(*Subtype t s*) →
(*htupdate ht l t ht′*) →
(*WFWordValinit* (*wfmap ? ? ht wfht*) *wv t′ b*) →
(*WFWordValinit* (*wfmap ? ? ht′ wfht′*) *wv t′ b*).

Lemma *heap_upd_4*
: (*I*:*InstrSeq*)
(*h*,*h′*:(*Map label HeapVal*); *wfh*:(*mWF ? ? h*); *wfh′*:(*mWF ? ? h′*);
ht,*ht′*:(*Map label Omega*); *wfht*:(*mWF ? ? ht*); *wfht′*:(*mWF ? ? ht′*))
(*t*,*s*:*Omega*; *l*:*label*)
(*R*:*RegFileTy*; *A*:*APTy*)
(*WFHeap* (*wfmap ? ? h wfh*) (*wfmap ? ? ht wfht*)) →
(*htlookup ht l s*) →

(*Subtype t s*) →
(*htupdate ht l t ht'*) →
(*WFInstrSeq* (*wfmap ? ? ht wfht*) *R A I*) →
(*WFInstrSeq* (*wfmap ? ? ht' wfht'*) *R A I*).

Lemma *heap_upd_5_aux*
: (*h,h'*:(*Map label HeapVal*); *wfh*:(*mWF ? ? h*); *wfh'*:(*mWF ? ? h'*);
ht,ht':(*Map label Omega*); *wfht*:(*mWF ? ? ht*); *wfht'*:(*mWF ? ? ht'*))
(*t,s*:*Omega*; *l*:*label*)
(*wl*:(*list WordVal*); *tl*:(*list Omega*); *il*:(*list initflag*))
(*WFHeap* (*wfmap ? ? h wfh*) (*wfmap ? ? ht wfht*)) →
(*htlookup ht l s*) →
(*Subtype t s*) →
(*htupdate ht l t ht'*) →
(*WFWordValinitList* (*wfmap ? ? ht wfht*) *wl tl il*) →
(*WFWordValinitList* (*wfmap ? ? ht' wfht'*) *wl tl il*).

Lemma *heap_upd_3*
: (*h,h'*:(*Map label HeapVal*); *wfh*:(*mWF ? ? h*); *wfh'*:(*mWF ? ? h'*);
ht,ht':(*Map label Omega*); *wfht*:(*mWF ? ? ht*); *wfht'*:(*mWF ? ? ht'*))
(*t,s*:*Omega*; *l*:*label*)
(*R*:*RegFile*; *G*:(*Map Reg Omega*))
(*WFHeap* (*wfmap ? ? h wfh*) (*wfmap ? ? ht wfht*)) →
(*htlookup ht l s*) →
(*Subtype t s*) →
(*htupdate ht l t ht'*) →
(*WFRegFile* (*wfmap ? ? ht wfht*) *R G*) →
(*WFRegFile* (*wfmap ? ? ht' wfht'*) *R G*).

Lemma *heap_upd_3ap*
: (*h,h'*:(*Map label HeapVal*); *wfh*:(*mWF ? ? h*); *wfh'*:(*mWF ? ? h'*);
ht,ht':(*Map label Omega*); *wfht*:(*mWF ? ? ht*); *wfht'*:(*mWF ? ? ht'*))
(*t,s*:*Omega*; *l*:*label*)
(*a*:*AP*; *at*:*APTy*)
(*WFHeap* (*wfmap ? ? h wfh*) (*wfmap ? ? ht wfht*)) →
(*htlookup ht l s*) →
(*Subtype t s*) →
(*htupdate ht l t ht'*) →
(*WFap* (*wfmap ? ? ht wfht*) *a at*) →
(*WFap* (*wfmap ? ? ht' wfht'*) *a at*).

Lemma *heap_upd_5*
: (*h,h'*:(*Map label HeapVal*); *wfh*:(*mWF ? ? h*); *wfh'*:(*mWF ? ? h'*);

ht,ht':(*Map label Omega*); *wfht*:(*mWF ? ? ht*); *wfht'*:(*mWF ? ? ht'*))
(*t,s*:*Omega*; *l*:*label*)
(*hv*:*HeapVal*; *t'*:*Omega*)
(*WFHeap* (*wfmap ? ? h wfh*) (*wfmap ? ? ht wfht*)) →
(*htlookup ht l s*) →
(*Subtype t s*) →
(*htupdate ht l t ht'*) →
(*WFHeapVal* (*wfmap ? ? ht wfht*) *hv t'*) →
(*WFHeapVal* (*wfmap ? ? ht' wfht'*) *hv t'*).

Lemma *ord_updateheap_ord*
: (*h,h'*:*HeapMap*; *l*:*label*; *hv*:*HeapVal*)
(*mupdate ? ? h l hv h'*) →
(*OrdHeap h*) →
(*OrdHeap h'*).

Lemma *heap_upd_2*
: (*h,h'*:(*Map label HeapVal*); *wfh*:(*mWF ? ? h*); *wfh'*:(*mWF ? ? h'*);
ht,ht':(*Map label Omega*); *wfht*:(*mWF ? ? ht*); *wfht'*:(*mWF ? ? ht'*))
(*t,s*:*Omega*; *l*:*label*)
(*hv*:*HeapVal*)
(*WFHeap* (*wfmap ? ? h wfh*) (*wfmap ? ? ht wfht*)) →
(*htlookup ht l s*) →
(*Subtype t s*) →
(*htupdate ht l t ht'*) →
(*hupdate h l hv h'*) →
(*WFHeapVal* (*wfmap ? ? ht' wfht'*) *hv t*) →
(*WFHeap* (*wfmap ? ? h' wfh'*) (*wfmap ? ? ht' wfht'*)).

Cases of the FTAL progress theorem

Lemma *progress_add* : (*H*:*Heap*; *R*:*RegFile*; *I*:*InstrSeq*; *a*:*AP*;
rd,rs,rs':*Reg*)
(*WFProgram* (*H*, (*R*, (*a*, (*iseq* (*add rd rs rs'*) *I*))))) →
(*EX P'* |
(*Eval* (*H*, (*R*, (*a*, (*iseq* (*add rd rs rs'*) *I*)))) *P'*)).

Lemma *progress_addi* : (*H*:*Heap*; *R*:*RegFile*; *I*:*InstrSeq*; *a*:*AP*;
rd,rs:*Reg*; *i*:*int*)
(*WFProgram* (*H*, (*R*, (*a*, (*iseq* (*addi rd rs i*) *I*))))) →
(*EX P'* |
(*Eval* (*H*, (*R*, (*a*, (*iseq* (*addi rd rs i*) *I*)))) *P'*)).

Lemma *progress_jmp* : (*H*:*Heap*; *R*:*RegFile*; *a*:*AP*; *r*:*Reg*)
(*WFProgram* (*H*, (*R*, (*a*, (*jmp r*))))) →

(*EX P′* |
(*Eval* (*H*, (*R*, (*a*, (*jmp r*)))) *P′*)).

Lemma *progress_jd* : (*H*:*Heap*; *R*:*RegFile*; *a*:*AP*; *l*:*label*)
(*WFProgram* (*H*, (*R*, (*a*, (*jd l*))))) →
(*EX P′* |
(*Eval* (*H*, (*R*, (*a*, (*jd l*)))) *P′*)).

Lemma *progress_ld* : (*H*:*Heap*; *R*:*RegFile*; *a*:*AP*; *I*:*InstrSeq*)
(*rd*,*rs*:*Reg*; *i*:*int*)
(*WFProgram* (*H*, (*R*, (*a*, (*iseq* (*ld rd rs i*) *I*))))) →
(*EX P′* |
(*Eval* (*H*, (*R*, (*a*, (*iseq* (*ld rd rs i*) *I*)))) *P′*)).

Lemma *progress_mov* : (*H*:*Heap*; *R*:*RegFile*; *a*:*AP*; *I*:*InstrSeq*)
(*rd*,*rs*:*Reg*)
(*WFProgram* (*H*,(*R*,(*a*,(*iseq* (*mov rd rs*) *I*))))) →
(*EX P′* —
(*Eval* (*H*,(*R*,(*a*,(*iseq* (*mov rd rs*) *I*)))) *P′*)).

Lemma *progress_movi* : (*H*:*Heap*; *R*:*RegFile*; *a*:*AP*; *I*:*InstrSeq*)
(*rd*:*Reg*; *i*:*int*)
(*WFProgram* (*H*,(*R*,(*a*,(*iseq* (*movi rd i*) *I*))))) →
(*EX P′* —
(*Eval* (*H*,(*R*,(*a*,(*iseq* (*movi rd i*) *I*)))) *P′*)).

Lemma *progress_movl* : (*H*:*Heap*; *R*:*RegFile*; *a*:*AP*; *I*:*InstrSeq*)
(*rd*:*Reg*; *l*:*label*)
(*WFProgram* (*H*,(*R*,(*a*,(*iseq* (*movl rd l*) *I*))))) →
(*EX P′* —
(*Eval* (*H*,(*R*,(*a*,(*iseq* (*movl rd l*) *I*)))) *P′*)).

Lemma *progress_unfold* : (*H*:*Heap*; *R*:*RegFile*; *a*:*AP*; *I*:*InstrSeq*)
(*rd*,*rs*:*Reg*)
(*WFProgram* (*H*, (*R*, (*a*, (*iseq* (*unfold rd rs*) *I*))))) →
(*EX P′* |
(*Eval* (*H*, (*R*, (*a*, (*iseq* (*unfold rd rs*) *I*)))) *P′*)).

Lemma *progress_fold* : (*H*:*Heap*; *R*:*RegFile*; *a*:*AP*; *I*:*InstrSeq*)
(*rd*,*rs*:*Reg*; *t*:*Omega*)
(*WFProgram* (*H*, (*R*, (*a*, (*iseq* (*fold rd t rs*) *I*))))) →
(*EX P′* |
(*Eval* (*H*, (*R*, (*a*, (*iseq* (*fold rd t rs*) *I*)))) *P′*)).

Lemma *progress_bgt* : (*H*:*Heap*; *R*:*RegFile*; *a*:*AP*; *I*:*InstrSeq*)
(*rs*,*rt*:*Reg*; *l*:*label*)

(*WFProgram* (*H*, (*R*, (*a*, (*iseq* (*bgt rs rt l*) *I*))))) →
(*EX P′* |
(*Eval* (*H*, (*R*, (*a*, (*iseq* (*bgt rs rt l*) *I*)))) *P′*)).

Lemma *progress_bump* : (*H*:*Heap*; *R*:*RegFile*; *a*:*AP*; *I*:*InstrSeq*)
(*n*:*nat*)
(*WFProgram* (*H*, (*R*, (*a*, (*iseq* (*bump n*) *I*))))) →
(*EX P′* |
(*Eval* (*H*, (*R*, (*a*, (*iseq* (*bump n*) *I*)))) *P′*)).

Lemma *progress_alloc* : (*H*:*Heap*; *R*:*RegFile*; *a*:*AP*; *I*:*InstrSeq*)
(*rd*:*Reg*; *V*:(*list Omega*))
(*WFProgram* (*H*, (*R*, (*a*, (*iseq* (*alloc rd V*) *I*))))) →
(*EX P′* |
(*Eval* (*H*, (*R*, (*a*, (*iseq* (*alloc rd V*) *I*)))) *P′*)).

Lemma *progress_st* : (*H*:*Heap*; *R*:*RegFile*; *a*:*AP*; *I*:*InstrSeq*)
(*rd*,*rs*:*Reg*; *i*:*int*)
(*WFProgram* (*H*, (*R*, (*a*, (*iseq* (*st rd i rs*) *I*))))) →
(*EX P′* |
(*Eval* (*H*, (*R*, (*a*, (*iseq* (*st rd i rs*) *I*)))) *P′*)).

Cases of the FTAL preservation theorem

Lemma *preserv_add* : (*H*:*Heap*; *R*:*RegFile*; *I*:*InstrSeq*; *a*:*AP*;
rd,*rs*,*rs′*:*Reg*; *P′*:*Program*)
(*WFProgram* (*H*, (*R*, (*a*, (*iseq* (*add rd rs rs′*) *I*))))) →
(*Eval* (*H*, (*R*, (*a*, (*iseq* (*add rd rs rs′*) *I*)))) *P′*) →
(*WFProgram P′*).

Lemma *preserv_addi* : (*H*:*Heap*; *R*:*RegFile*; *I*:*InstrSeq*; *a*:*AP*;
rd,*rs*:*Reg*; *i*:*int*; *P′*:*Program*)
(*WFProgram* (*H*, (*R*, (*a*, (*iseq* (*addi rd rs i*) *I*))))) →
(*Eval* (*H*, (*R*, (*a*, (*iseq* (*addi rd rs i*) *I*)))) *P′*) →
(*WFProgram P′*).

Lemma *preserv_alloc* : (*H*:*Heap*; *R*:*RegFile*; *I*:*InstrSeq*; *a*:*AP*;
rd:*Reg*; *V*:(*list Omega*); *P′*:*Program*)
(*WFProgram* (*H*, (*R*, (*a*, (*iseq* (*alloc rd V*) *I*))))) →
(*Eval* (*H*, (*R*, (*a*, (*iseq* (*alloc rd V*) *I*)))) *P′*) →
(*WFProgram P′*).

Lemma *preserv_bgt* : (*H*:*Heap*; *R*:*RegFile*; *I*:*InstrSeq*; *a*:*AP*; *P′*:*Program*)
(*rs*,*rt*:*Reg*; *l*:*label*)
(*WFProgram* (*H*, (*R*, (*a*, (*iseq* (*bgt rs rt l*) *I*))))) →
(*Eval* (*H*, (*R*, (*a*, (*iseq* (*bgt rs rt l*) *I*)))) *P′*) →

(*WFProgram P'*).

Lemma *preserv_bump* : (*H*:*Heap*; *R*:*RegFile*; *I*:*InstrSeq*; *a*:*AP*; *n*:*nat*; *P'*:*Program*)
(*WFProgram* (*H*, (*R*, (*a*, (*iseq* (*bump n*) *I*))))) →
(*Eval* (*H*, (*R*, (*a*, (*iseq* (*bump n*) *I*)))) *P'*) →
(*WFProgram P'*).

Lemma *preserv_fold* : (*H*:*Heap*; *R*:*RegFile*; *I*:*InstrSeq*; *a*:*AP*; *P'*:*Program*)
(*rd*,*rs*:*Reg*; *t*:*Omega*)
(*WFProgram* (*H*, (*R*, (*a*, (*iseq* (*fold rd t rs*) *I*))))) →
(*Eval* (*H*, (*R*, (*a*, (*iseq* (*fold rd t rs*) *I*)))) *P'*) →
(*WFProgram P'*).

Lemma *preserv_jmp* : (*H*:*Heap*; *R*:*RegFile*; *a*:*AP*; *r*:*Reg*; *P'*:*Program*)
(*WFProgram* (*H*, (*R*, (*a*, (*jmp r*))))) →
(*Eval* (*H*, (*R*, (*a*, (*jmp r*)))) *P'*) →
(*WFProgram P'*).

Lemma *preserv_jd* : (*H*:*Heap*; *R*:*RegFile*; *a*:*AP*; *l*:*label*; *P'*:*Program*)
(*WFProgram* (*H*, (*R*, (*a*, (*jd l*))))) →
(*Eval* (*H*, (*R*, (*a*, (*jd l*)))) *P'*) →
(*WFProgram P'*).

Lemma *preserv_ld* : (*H*:*Heap*; *R*:*RegFile*; *I*:*InstrSeq*; *a*:*AP*; *P'*:*Program*)
(*rd*,*rs*:*Reg*; *i*:*int*)
(*WFProgram* (*H*, (*R*, (*a*, (*iseq* (*ld rd rs i*) *I*))))) →
(*Eval* (*H*, (*R*, (*a*, (*iseq* (*ld rd rs i*) *I*)))) *P'*) →
(*WFProgram P'*).

Lemma *preserv_mov* : (*H*:*Heap*; *R*:*RegFile*; *a*:*AP*; *I*:*InstrSeq*; *P'*:*Program*)
(*rd*,*rs*:*Reg*)
(*WFProgram* (*H*,(*R*,(*a*,(*iseq* (*mov rd rs*) *I*))))) →
(*Eval* (*H*,(*R*,(*a*,(*iseq* (*mov rd rs*) *I*)))) *P'*) →
(*WFProgram P'*).

Lemma *preserv_movi* : (*H*:*Heap*; *R*:*RegFile*; *a*:*AP*; *I*:*InstrSeq*; *P'*:*Program*)
(*rd*:*Reg*; *i*:*int*)
(*WFProgram* (*H*,(*R*,(*a*,(*iseq* (*movi rd i*) *I*))))) →
(*Eval* (*H*,(*R*,(*a*,(*iseq* (*movi rd i*) *I*)))) *P'*) →
(*WFProgram P'*).

Lemma *preserv_movl* : (*H*:*Heap*; *R*:*RegFile*; *a*:*AP*; *I*:*InstrSeq*; *P'*:*Program*)
(*rd*:*Reg*; *l*:*label*)
(*WFProgram* (*H*,(*R*,(*a*,(*iseq* (*movl rd l*) *I*))))) →
(*Eval* (*H*,(*R*,(*a*,(*iseq* (*movl rd l*) *I*)))) *P'*) →
(*WFProgram P'*).

Lemma *preserv_st_aux2*

: (*h*,*h'*:(*Map label HeapVal*); *wfh*:(*mWF ? ? h*); *wfh'*:(*mWF ? ? h'*))
(*ht*:(*Map label Omega*); *wfht*:(*mWF ? ? ht*))
(*R*:*RegFile*; *G*:*RegFileTy*)
(*rd*,*rs*:*Reg*; *i*:*int*; *l*:*label*; *w*:*WordVal*; *t*:*Omega*)
(*V*,*V'*:(*list WordVal*); *Ts*:(*list Omega*);
il,*V0*,*V'0*:(*list initflag*))
(*WFHeap* (*wfmap ? ? h wfh*) (*wfmap ? ? ht wfht*)) →
(*WFRegFile* (*wfmap ? ? ht wfht*) *R G*) →
(*mlookup ? ? R rd* (*wl l*)) →
(*mlookup ? ? R rs w*) →
(*mlookup ? ? h l* (*tuple V*)) →
(*updatetuple V i w V'*) →
(*mupdate ? ? h l* (*tuple V'*) *h'*) →
(*mlookup ? ? G rd* (*tupty Ts V0*)) →
(*mlookup ? ? G rs t*) →
(*ListNth ? Ts i t*) →
(*updatetupty V0 i V'0*) →
(*mlookup ? ? ht l* (*tupty Ts il*)) →
(*Subtype* (*tupty Ts il*) (*tupty Ts V0*)) →
(*EX ht'*:(*Map nat Omega*) —
(*EXT wfht'*:(*mWF nat Omega ht'*) —
(*EX il'* —
(*mupdate nat Omega ht l* (*tupty Ts il'*) *ht'*) ∧
(*updatetupty il i il'*) ∧
(*Subtype* (*tupty Ts il'*) (*tupty Ts V'0*)) ∧
(*Subtype* (*tupty Ts il'*) (*tupty Ts il*)) ∧
(*WFHeapVal* (*wfmap ? ? ht' wfht'*) (*tuple V'*)
(*tupty Ts il'*))))).

Lemma *preserv_st* : (*H*:*Heap*; *R*:*RegFile*; *I*:*InstrSeq*; *a*:*AP*;
rd,*rs*:*Reg*; *i*:*int*; *P'*:*Program*)
(*WFProgram* (*H*, (*R*, (*a*, (*iseq* (*st rd i rs*) *I*))))) →
(*Eval* (*H*, (*R*, (*a*, (*iseq* (*st rd i rs*) *I*)))) *P'*) →
(*WFProgram P'*).

Lemma *preserv_unfold* : (*H*:*Heap*; *R*:*RegFile*; *I*:*InstrSeq*; *a*:*AP*; *P'*:*Program*)
(*rd*,*rs*:*Reg*)
(*WFProgram* (*H*, (*R*, (*a*, (*iseq* (*unfold rd rs*) *I*))))) →
(*Eval* (*H*, (*R*, (*a*, (*iseq* (*unfold rd rs*) *I*)))) *P'*) →
(*WFProgram P'*).

A.3 Translation to Machine State

The Safety Policy

```
Definition SP := [S:State] (let (M,T')=S in (let (R,PC)=T' in ~(Dc (M PC))=_ill )).

Definition Safe := [S:State] (n:nat)(SP (MultiStep n S)).
```

Translation Relations

```
(* The Layout function *)
Definition HV_size : HeapVal → nat
  := [hv] Cases hv of (tuple V) ⇒ (length V)
                    | (code G T I') ⇒ (lenInstrSeq I')
     end.

Fixpoint Layout_aux [H:(Map label HeapVal)] : label → Word
  := Cases H of mempty ⇒ ([l':label] O)
                | (melem l h H') ⇒ [l':label] (if (blt_nat l l')
                                                then (plus (Layout_aux H' l') (HV_size h))
                                                else (Layout_aux H' l'))
     end.

Definition Layout : Heap → label → Word := [H] (Layout_aux (hunwrap H)).

Syntactic Definition LayoutF := (label → Word).

(* Register translations *)
Require translate_ftal_aux.

Inductive TrWordVal : LayoutF → WordVal → Word → Prop
  := trwv_label : (L:LayoutF; l:label) (TrWordVal L (wl l) (L l))
  | trwv_int : (L:LayoutF; i:int) (TrWordVal L (wi i) i)
  | trwv_uninit : (L:LayoutF; t:Omega; l:label) (TrWordVal L (wuninit t) l)
  | trwv_fold : (L:LayoutF; v:WordVal; t:Omega; w:Word)
                (TrWordVal L v w) → (TrWordVal L (wfold v t) w).

Inductive TrWordValList : LayoutF → (list WordVal) → Mem → Word → Prop
  := trwvl_nil : (L:LayoutF; M:Mem; l:Word)
                 (TrWordValList L (nil ?) M l)
  | trwvl_cons : (L:LayoutF; v:WordVal; V:(list WordVal); M:Mem; w,l:Word)
                 (TrWordVal L v w) →
                 (M l)=w →
                 (TrWordValList L V M (S l)) →
                 (TrWordValList L (cons v V) M l).

Inductive TrInstr : LayoutF → Instr → _Instr → Prop
```

```
:= tri_add : (L:LayoutF; rd,rs,rt:Reg; _rd,_rs,_rt:_Reg)
                    (TrRegF rd)=_rd →
                    (TrRegF rs)=_rs →
                    (TrRegF rt)=_rt →
                    (TrInstr L (add rd rs rt) (_add _rd _rs _rt))
| tri_addi : (L:LayoutF; rd,rs:Reg; _rd,_rs:_Reg; i:int)
                    (TrRegF rd)=_rd →
                    (TrRegF rs)=_rs →
                    (TrInstr L (addi rd rs i) (_addi _rd _rs i))
| tri_alloc : (L:LayoutF; rd:Reg; _rd:_Reg; Ts:(list Omega))
                    (TrRegF rd)=_rd →
                    (TrInstr L (alloc rd Ts) (_addi _rd _r31 O))
| tri_bgt : (L:LayoutF; rs,rt:Reg; _rs,_rt:_Reg; l:label)
                    (TrRegF rs)=_rs →
                    (TrRegF rt)=_rt →
                    (TrInstr L (bgt rs rt l) (_bgt _rs _rt (L l)))
| tri_bump : (L:LayoutF; i:int)
                    (TrInstr L (bump i) (_addi _r31 _r31 i))
| tri_fold : (L:LayoutF; rd,rs:Reg; t:Omega; _rd,_rs:_Reg)
                    (TrRegF rd)=_rd →
                    (TrRegF rs)=_rs →
                    (TrInstr L (fold rd t rs) (_addi _rd _rs O))
| tri_ld : (L:LayoutF; rd,rs:Reg; i:int; _rd,_rs:_Reg)
                    (TrRegF rd)=_rd →
                    (TrRegF rs)=_rs →
                    (TrInstr L (ld rd rs i) (_ld _rd _rs i))
| tri_mov : (L:LayoutF; rd,rs:Reg; _rd,_rs:_Reg)
                    (TrRegF rd)=_rd →
                    (TrRegF rs)=_rs →
                    (TrInstr L (mov rd rs) (_addi _rd _rs O))
| tri_movi : (L:LayoutF; r:Reg; _r:_Reg; i:int)
                    (TrRegF r)=_r →
                    (TrInstr L (movi r i) (_movi _r i))
| tri_movl : (L:LayoutF; r:Reg; _r:_Reg; l:label)
                    (TrRegF r)=_r →
                    (TrInstr L (movl r l) (_movi _r (L l)))
| tri_st : (L:LayoutF; rd,rs:Reg; i:int; _rd,_rs:_Reg)
                    (TrRegF rd)=_rd →
                    (TrRegF rs)=_rs →
                    (TrInstr L (st rd i rs) (_st _rd i _rs))
```

```
  | tri_unfold : (L:LayoutF; rd,rs:Reg; _rd,_rs:_Reg)
                 (TrRegF rd)=_rd →
                 (TrRegF rs)=_rs →
                 (TrInstr L (unfold rd rs) (_addi _rd _rs O)).

Inductive TrInstrSeq : LayoutF → InstrSeq → Mem → Word → Prop
  := tris_iseq : (L:LayoutF; i:Instr; _i:_Instr; I:InstrSeq; M:Mem; l:Word)
                 (TrInstr L i _i) →
                 (Dc (M l)) = _i →
                 (TrInstrSeq L I M (S l)) →
                 (TrInstrSeq L (iseq i I) M l)
  | tris_jd : (L:LayoutF; l:label; M:Mem; w:Word)
                 (Dc (M w)) = (_jd (L l)) →
                 (TrInstrSeq L (jd l) M w)
  | tris_jmp : (L:LayoutF; r:Reg; _r:_Reg; M:Mem; w:Word)
                 (TrRegF r)=_r →
                 (Dc (M w)) = (_jmp _r) →
                 (TrInstrSeq L (jmp r) M w).

Inductive TrHeapVal : LayoutF → HeapVal → Mem → Word → Prop
  := trhv_tuple : (L:LayoutF; V:(list WordVal); M:Mem; l:Word)
                 (TrWordValList L V M l) →
                 (TrHeapVal L (tuple V) M l)
  | trhv_code : (L:LayoutF; G:RegFileTy; T:APTy; I:InstrSeq; M:Mem; l:Word)
                 (TrInstrSeq L I M l) →
                 (TrHeapVal L (code G T I) M l).

Inductive TrHeap : LayoutF → Heap → Mem → Prop
  := trheap : (L:LayoutF; H:Heap; s:nat; M:Mem)
                 (hsize (hunwrap H) s) →
                 ( (n:nat; h:HeapVal)
                    (hlookup (hunwrap H) n h) →
                    (TrHeapVal L h M (L n)) ) →
                 (TrHeap L H M).

Inductive TrRegFile : LayoutF → RegFile → _RegFile → Prop
  := trregfile : (L:LayoutF; R:RegFile; _R:_RegFile)
                 ( (r:Reg; _r:_Reg; v:WordVal)
                    (mlookup ? ? R r v) →
                    (TrRegF r)=_r →
                    (TrWordVal L v (_R _r))) →
                 (TrRegFile L R _R).

Inductive TrAP : LayoutF → AP → _RegFile → Prop
```

```
  := trap : (L:LayoutF; A:AP; _R:_RegFile)
                    (_R _r31)=(L A) → (TrAP L A _R)
.

Inductive TrProgram : Program → State → Prop
  := trprogram : (H:Heap; R:RegFile; A:AP; I:InstrSeq;
                   M:Mem; _R:_RegFile; pc:Word; L:LayoutF)
                  L=(Layout H) →
                  (TrHeap L H M) →
                  (TrRegFile L R _R) →
                  (TrAP L A _R) →
                  (EX l | (EX G | (EX T | (EX I' | (EX n |
                      (hlookup (hunwrap H) l (code G T I')) ∧
                      (ISubDepth I I' n) ∧
                      (TrInstrSeq L I' M (L l)) ∧
                      (plus (L l) n)=pc))))) →
                  (TrProgram (H,(R,(A,I))) (M,(_R,pc))).
```

FPCC Proofs

The global invariant

```
Definition Inv := [S:State] (EX P:Program | (WFProgram P) ∧ (TrProgram P S)).
```

FPCC Progress theorem

```
Theorem Progress : (S:State) (Inv S) → (SP S).
```

Cases of the FPCC Preservation theorem

```
Lemma Preservation_add
                  : (S:State; H:Heap; R:RegFile; a:AP;
                     rd,rs,rs':Reg; I:InstrSeq;
                     P,P':Program)
                    P = (H,(R,(a,(iseq (add rd rs rs') I)))) →
                    (WFProgram P) →
                    (TrProgram P S) →
                    (Eval P P') →
                    (WFProgram P') →
                    (TrProgram P' (Step S)).

Lemma Preservation_addi
                  : (S:State; H:Heap; R:RegFile; a:AP;
                     rd,rs:Reg; i:int; I:InstrSeq;
                     P,P':Program)
                    P = (H,(R,(a,(iseq (addi rd rs i) I)))) →
```

(*WFProgram P*) →
(*TrProgram P S*) →
(*Eval P P′*) →
(*WFProgram P′*) →
(*TrProgram P′* (*Step S*)).

Lemma *heap_size_eq_label*
: (*h*:(*Map label HeapVal*); *wfh*:(*mWF ? ? h*))
(*ht*:(*Map label Omega*); *wfht*:(*mWF ? ? ht*))
(*l*:*label*)
(*WFHeap* (*wfmap ? ? h wfh*) (*wfmap ? ? ht wfht*)) →
(*msize ? ? h* (*S l*)) →
(*EX hv* | (*EX h′* | *h*=(*melem ? ? l hv h′*))).

Lemma *heap_label_eq_size*
: (*l*:*label*; *hv*:*HeapVal*; *h*:(*Map label HeapVal*))
(*wf*:(*mWF ? ? h*); *wf′*:(*mWF ? ?* (*melem ? ? l hv h*)); *G*,*G′*:*HeapTy*)
(*WFHeap* (*wfmap ? ? h wf*) *G*) →
(*WFHeap* (*wfmap ? ?* (*melem ? ? l hv h*) *wf′*) *G′*) →
(*msize ? ? h l*).

Lemma *tr_extend_layout_aux_eq_preserv*
: (*h*,*h′*:(*Map label HeapVal*); *wfh*:(*mWF ? ? h*); *wfh′*:(*mWF ? ? h′*))
(*l*:*label*; *hv*:*HeapVal*)
h′=(*melem ? ? l hv h*) →
(*msize ? ? h l*) →
(*Layout_aux h l*)=(*Layout_aux h′ l*).

Lemma *tr_extend_layout_aux_preserv*
: (*h*,*h′*:(*Map label HeapVal*); *wfh*:(*mWF ? ? h*); *wfh′*:(*mWF ? ? h′*))
(*l*,*n*:*label*; *hv*:*HeapVal*)
h′=(*melem ? ? l hv h*) →
(*msize ? ? h l*) →
(*lt n l*) →
(*Layout_aux h n*)=(*Layout_aux h′ n*).

Lemma *tr_extend_layout_preserv*
: (*h*,*h′*:(*Map label HeapVal*); *wfh*:(*mWF ? ? h*); *wfh′*:(*mWF ? ? h′*))
(*l*,*n*:*label*; *hv*:*HeapVal*)
(*mextend ? ? h l hv h′*) →
(*msize ? ? h l*) →
(*lt n l*) →
(*Layout* (*wfmap ? ? h′ wfh′*) *n*)=(*Layout* (*wfmap ? ? h wfh*) *n*).

Lemma *tr_extend_heap_uninitwordlist*

: (*Ts*:(*list Omega*); *H*:*Heap*; *M*:*Mem*; *l*:*label*)
(*TrWordValList* (*Layout H*) (*makeUninitTup Ts*) *M l*).

Lemma *tr_extend_instr_preserv*
: (*i*:*Instr*; *I*:*InstrSeq*; *R*:*RegFileTy*; *T*:*APTy*;
_i:*_Instr*; *l*:*label*; *Ts*:(*list Omega*);
h,*h'*:(*Map label HeapVal*))
(*wf*:(*mWF* ? ? *h*); *wf'*:(*mWF* ? ? *h'*))
(*G*,*G'*:*HeapTy*)
h'=(*melem* ? ? *l* (*tuple* (*makeUninitTup Ts*)) *h*) →
(*WFHeap* (*wfmap* ? ? *h wf*) *G*) →
(*WFHeap* (*wfmap* ? ? *h' wf'*) *G'*) →
(*WFInstrSeq G R T* (*iseq i I*)) →
(*TrInstr* (*Layout_aux h*) *i _i*) →
(*TrInstr* (*Layout_aux h'*) *i _i*).

Lemma *tr_extend_instrseq_preserv*
: (*i*:*InstrSeq*; *M*:*Mem*; *l*:*label*; *Ts*:(*list Omega*); *h*,*h'*:(*Map label HeapVal*))
(*wf*:(*mWF* ? ? *h*); *wf'*:(*mWF* ? ? *h'*))
(*m*:*RegFileTy*; *a*:*APTy*)
(*D*:*Word*)
(*G*,*G'*:*HeapTy*)
h'=(*melem* ? ? *l* (*tuple* (*makeUninitTup Ts*)) *h*) →
(*WFHeap* (*wfmap* ? ? *h wf*) *G*) →
(*WFHeap* (*wfmap* ? ? *h' wf'*) *G'*) →
(*WFInstrSeq G m a i*) →
(*TrInstrSeq* (*Layout_aux h*) *i M D*) →
(*TrInstrSeq* (*Layout_aux h'*) *i M D*).

Lemma *wordval_strip_not_wfold*
: (*v*,*v'*:*WordVal*) *v'*=(*stripWV v*) → ~(*EX w* | (*EX t* | *v'*=(*wfold w t*))).

Lemma *tr_wordval_strip*
: (*L*:*LayoutF*; *v*:*WordVal*; *w*:*Word*)
(*TrWordVal L v w*) →
(*EX v'* | *v'*=(*stripWV v*) ∧ (*TrWordVal L v' w*)).

Lemma *tr_wordval_strip_alt*
: (*L*:*LayoutF*; *v*:*WordVal*; *w*:*Word*)
(*TrWordVal L v w*) →
(*TrWordVal L* (*stripWV v*) *w*).

Lemma *tr_strip_wordval*
: (*L*:*LayoutF*; *v*,*v'*:*WordVal*; *w*:*Word*)
v'=(*stripWV v*) →

(*TrWordVal L v' w*) →
(*TrWordVal L v w*).

Lemma *tr_extend_wordval_preserv_aux*
: (*w*:*Word*; *l*:*label*; *Ts*:(*list Omega*); *h*,*h'*:(*Map label HeapVal*))
(*wf*:(*mWF ? ? h*); *wf'*:(*mWF ? ? h'*))
(*wv'*,*wv*:*WordVal*)
(*HT*,*HT'*:*HeapTy*)
h'=(*melem ? ? l* (*tuple* (*makeUninitTup Ts*)) *h*) →
(*WFHeap* (*wfmap ? ? h wf*) *HT*) →
(*WFHeap* (*wfmap ? ? h' wf'*) *HT'*) →
wv'=(*stripWV wv*) →
((*l*,*n*:*nat*) (*stripWV wv*)=(*wl l*) → (*msize ? ?* (*htunwrap HT*) *n*) → (*lt l n*))
→
(*TrWordVal* (*Layout_aux h*) *wv' w*) →
(*TrWordVal* (*Layout_aux h'*) *wv' w*).

Lemma *tr_extend_wordvali_preserv*
: (*w*:*Word*; *l*:*label*; *Ts*:(*list Omega*); *h*,*h'*:(*Map label HeapVal*))
(*wf*:(*mWF ? ? h*); *wf'*:(*mWF ? ? h'*))
(*wv*:*WordVal*; *t*:*Omega*; *i*:*initflag*)
(*G*,*G'*:*HeapTy*)
h'=(*melem ? ? l* (*tuple* (*makeUninitTup Ts*)) *h*) →
(*WFHeap* (*wfmap ? ? h wf*) *G*) →
(*WFHeap* (*wfmap ? ? h' wf'*) *G'*) →
(*WFWordValinit G wv t i*) →
(*TrWordVal* (*Layout_aux h*) *wv w*) →
(*TrWordVal* (*Layout_aux h'*) *wv w*).

Lemma *tr_extend_wordvallist_preserv*
: (*L*:(*list WordVal*); *tl*:(*list Omega*); *il*:(*list bool*))
(*M*:*Mem*; *l*:*label*; *Ts*:(*list Omega*); *h*,*h'*:(*Map label HeapVal*))
(*wf*:(*mWF ? ? h*); *wf'*:(*mWF ? ? h'*))
(*D*:*Word*)
(*G*,*G'*:*HeapTy*)
h'=(*melem ? ? l* (*tuple* (*makeUninitTup Ts*)) *h*) →
(*WFHeap* (*wfmap ? ? h wf*) *G*) →
(*WFHeap* (*wfmap ? ? h' wf'*) *G'*) →
(*WFWordValinitList G L tl il*) →
(*TrWordValList* (*Layout_aux h*) *L M D*) →
(*TrWordValList* (*Layout_aux h'*) *L M D*).

Lemma *tr_extend_heapval_preserv_aux*

: (*M*:*Mem*; *l*:*label*; *Ts*:(*list Omega*); *h*,*h′*:(*Map label HeapVal*))
(*wf*:(*mWF ? ? h*); *wf′*:(*mWF ? ? h′*))
(*n*:*nat*; *hv*:*HeapVal*; *t*:*Omega*)
(*G*,*G′*:*HeapTy*)
h′=(*melem ? ? l* (*tuple* (*makeUninitTup Ts*)) *h*) →
(*WFHeap* (*wfmap ? ? h wf*) *G*) →
(*WFHeap* (*wfmap ? ? h′ wf′*) *G′*) →
(*WFHeapVal G hv t*) →
(*lt n l*) →
(*TrHeapVal* (*Layout_aux h*) *hv M* (*Layout_aux h n*)) →
(*TrHeapVal* (*Layout_aux h′*) *hv M* (*Layout_aux h′ n*)).

Lemma *tr_extend_heapval_preserv*
: (*M*:*Mem*; *l*:*label*; *Ts*:(*list Omega*))
(*H*,*H′*:*Heap*; *G*,*G′*:*HeapTy*)
(*n*:*nat*; *hv*:*HeapVal*; *t*:*Omega*)
(*WFHeap H G*) →
(*WFHeap H′ G′*) →
(*hextend* (*hunwrap H*) *l* (*tuple* (*makeUninitTup Ts*)) (*hunwrap H′*)) →
(*WFHeapVal G hv t*) →
(*lt n l*) →
(*TrHeapVal* (*Layout H*) *hv M* (*Layout H n*)) →
(*TrHeapVal* (*Layout H′*) *hv M* (*Layout H′ n*)).

Lemma *tr_heap_extend_emptytup_aux*
: (*M*:*Mem*; *l*:*label*; *Ts*:(*list Omega*); *h*,*h′*:(*Map label HeapVal*))
(*wf*:(*mWF ? ? h*); *wf′*:(*mWF ? ? h′*);
wf″:(*mWF ? ?* (*melem ? ? l* (*tuple* (*makeUninitTup Ts*)) *h*));
G,*G′*:*HeapTy*)
h′=(*melem ? ? l* (*tuple* (*makeUninitTup Ts*)) *h*) →
(*WFHeap* (*wfmap ? ? h wf*) *G*) →
(*WFHeap* (*wfmap ? ? h′ wf′*) *G′*) →
(*TrHeap* (*Layout* (*wfmap ? ? h wf*)) (*wfmap ? ? h wf*) *M*) →
(*TrHeap* (*Layout* (*wfmap ? ?* (*melem ? ? l* (*tuple* (*makeUninitTup Ts*)) *h*) *wf″*)) (*wfmap ? ?* (*melem ? ? l* (*tuple* (*makeUninitTup Ts*)) *h*) *wf″*) *M*) →
(*TrHeap* (*Layout* (*wfmap ? ? h′ wf′*)) (*wfmap ? ? h′ wf′*) *M*).

Lemma *tr_heap_extend_emptytup*
: (*M*:*Mem*; *l*:*label*; *Ts*:(*list Omega*); *h*:(*Map label HeapVal*))
(*wf*:(*mWF ? ? h*); *wf′*:(*mWF ? ?* (*melem ? ? l* (*tuple* (*makeUninitTup Ts*)) *h*));
G,*G′*:*HeapTy*)
(*WFHeap* (*wfmap ? ? h wf*) *G*) →
(*WFHeap* (*wfmap ? ?* (*melem ? ? l* (*tuple* (*makeUninitTup Ts*)) *h*) *wf′*) *G′*) →

(*TrHeap* (*Layout* (*wfmap* ? ? *h wf*)) (*wfmap* ? ? *h wf*) *M*) →
(*TrHeap* (*Layout* (*wfmap* ? ? (*melem* ? ? *l* (*tuple* (*makeUninitTup Ts*)) *h*) *wf'*))
(*wfmap* ? ? (*melem* ? ? *l* (*tuple* (*makeUninitTup Ts*)) *h*) *wf'*) *M*).

Lemma *heaplookup_wfheapval*
: (*H*:*Heap*; *HT*:*HeapTy*; *l*:*nat*; *h*:*HeapVal*)
(*WFHeap H HT*) →
(*hlookup* (*hunwrap H*) *l h*) →
(*EX t* | (*WFHeapVal HT h t*)).

Lemma *Preservation_alloc*
: (*S*:*State*; *H*:*Heap*; *R*:*RegFile*; *a*:*AP*;
rd:*Reg*; *Ts*:(*list Omega*); *I*:*InstrSeq*;
P,*P'*:*Program*)
P = (*H*,(*R*,(*a*,(*iseq* (*alloc rd Ts*) *I*)))) →
(*WFProgram P*) →
(*TrProgram P S*) →
(*Eval P P'*) →
(*WFProgram P'*) →
(*TrProgram P'* (*Step S*)).

Lemma *Preservation_bgt*
: (*S*:*State*; *H*:*Heap*; *R*:*RegFile*; *a*:*AP*;
rs,*rt*:*Reg*; *l*:*label*; *I*:*InstrSeq*;
P,*P'*:*Program*)
P = (*H*,(*R*,(*a*,(*iseq* (*bgt rs rt l*) *I*)))) →
(*WFProgram P*) →
(*TrProgram P S*) →
(*Eval P P'*) →
(*WFProgram P'*) →
(*TrProgram P'* (*Step S*)).

Lemma *wf_wordlist_len_eq*
: (*HT*:*HeapTy*; *V*:(*list WordVal*); *tl*:(*list Omega*); *ol*:(*list initflag*))
(*WFWordValinitList HT V tl ol*) →
(*length V*)=(*length tl*).

Lemma *wf_tuple_tupty_len_eq*
: (*HT*:*HeapTy*; *V*:(*list WordVal*); *tl*:(*list Omega*); *ol*:(*list initflag*))
(*WFHeapVal HT* (*tuple V*) (*tupty tl ol*)) →
(*length V*)=(*length tl*).

Lemma *layout_gt_eq*
: (*h*:(*Map label HeapVal*); *wfh*:(*mWF* ? ? *h*); *a*:*label*)
((*b*:*label*) (*mindom* ? ? *h b*) → (*lt b a*)) →

(*Layout_aux h a*)=(*Layout_aux h* (*S a*)).

Lemma *wfheap_next_mindom_imp_lt*
: (*l*,*a*,*b*:*label*; *hv*:*HeapVal*; *h*,*H*:(*Map label HeapVal*); *wfh*:(*mWF* ? ? *H*))
(*HT*:*HeapTy*)
(*WFHeap* (*wfmap* ? ? *H wfh*) *HT*) →
H=(*melem* ? ? *l hv h*) →
(*msize* ? ? *h a*) →
(*mindom* ? ? *h b*) → (*lt b a*).

Lemma *wfh_h_size_eq*
: (*h*:(*Map label HeapVal*); *wfh*:(*mWF* ? ? *h*);
ht:(*Map label Omega*); *wfht*:(*mWF* ? ? *ht*); *l*:*nat*)
(*WFHeap* (*wfmap* ? ? *h wfh*) (*wfmap* ? ? *ht wfht*)) →
(*htsize* (*htunwrap* (*wfmap* ? ? *ht wfht*)) *l*) →
(*hsize* (*hunwrap* (*wfmap* ? ? *h wfh*)) *l*).

Lemma *bump_aux*
: (*H*:*Heap*; *HT*:*HeapTy*; *a*,*n*:*nat*; *w*:*Word*; *tl*:(*list Omega*); *ol*:(*list bool*))
(*WFHeap H HT*) →
(*htsize* (*htunwrap HT*) (*S a*)) →
(*Layout H a*)=*w* →
(*WFWordVal HT* (*wl a*) (*tupty tl ol*)) →
n=(*length tl*) →
(*Layout H* (*S a*))=(*plus w n*).

Lemma *Preservation_bump*
: (*S*:*State*; *H*:*Heap*; *R*:*RegFile*; *a*:*AP*;
i:*nat*; *I*:*InstrSeq*;
P,*P'*:*Program*)
P = (*H*,(*R*,(*a*,(*iseq* (*bump i*) *I*)))) →
(*WFProgram P*) →
(*TrProgram P S*) →
(*Eval P P'*) →
(*WFProgram P'*) →
(*TrProgram P'* (*Step S*)).

Lemma *Preservation_fold*
: (*S*:*State*; *H*:*Heap*; *R*:*RegFile*; *a*:*AP*;
rd,*rs*:*Reg*; *t*:*Omega*; *I*:*InstrSeq*;
P,*P'*:*Program*)
P = (*H*,(*R*,(*a*,(*iseq* (*fold rd t rs*) *I*)))) →
(*WFProgram P*) →
(*TrProgram P S*) →

(*Eval P P'*) →
(*WFProgram P'*) →
(*TrProgram P'* (*Step S*)).

Lemma *Preservation_jmp*
: (*St*:*State*; *H*:*Heap*; *R*:*RegFile*; *a*:*AP*; *r*:*Reg*;
P,*P'*:*Program*)
P = (*H*,(*R*,(*a*,(*jmp r*)))) →
(*WFProgram P*) →
(*TrProgram P St*) →
(*Eval P P'*) →
(*WFProgram P'*) →
(*TrProgram P'* (*Step St*)).

Lemma *Preservation_jd*
: (*St*:*State*; *H*:*Heap*; *R*:*RegFile*; *a*:*AP*; *l*:*label*;
P,*P'*:*Program*)
P = (*H*,(*R*,(*a*,(*jd l*)))) →
(*WFProgram P*) →
(*TrProgram P St*) →
(*Eval P P'*) →
(*WFProgram P'*) →
(*TrProgram P'* (*Step St*)).

Lemma *trwordvallist_i_wordval*
: (*L*:*LayoutF*; *M*:*Mem*; *V*:(*list WordVal*); *i*:*int*; *v*:*WordVal*; *w*:*Word*)
(*TrWordValList L V M w*) →
(*ListNth* ? *V i v*) →
(*TrWordVal L v* (*M* (*plus w i*))).

Lemma *Preservation_ld*
: (*S*:*State*; *H*:*Heap*; *R*:*RegFile*; *a*:*AP*;
rd,*rs*:*Reg*; *i*:*int*; *I*:*InstrSeq*;
P,*P'*:*Program*)
P = (*H*,(*R*,(*a*,(*iseq* (*ld rd rs i*) *I*)))) →
(*WFProgram P*) →
(*TrProgram P S*) →
(*Eval P P'*) →
(*WFProgram P'*) →
(*TrProgram P'* (*Step S*)).

Lemma *Preservation_mov*
: (*S*:*State*; *H*:*Heap*; *R*:*RegFile*; *a*:*AP*;
rd,*rs*:*Reg*; *I*:*InstrSeq*;

```
    P,P':Program)
  P = (H,(R,(a,(iseq (mov rd rs) I)))) →
  (WFProgram P) →
  (TrProgram P S) →
  (Eval P P') →
  (WFProgram P') →
  (TrProgram P' (Step S)).

Lemma Preservation_movi
  : (S:State; H:Heap; R:RegFile; a:AP;
     rd:Reg; i:int; I:InstrSeq;
     P,P':Program)
    P = (H,(R,(a,(iseq (movi rd i) I)))) →
    (WFProgram P) →
    (TrProgram P S) →
    (Eval P P') →
    (WFProgram P') →
    (TrProgram P' (Step S)).

Lemma Preservation_movl
  : (S:State; H:Heap; R:RegFile; a:AP;
     r:Reg; l:label; I:InstrSeq;
     P,P':Program)
    P = (H,(R,(a,(iseq (movl r l) I)))) →
    (WFProgram P) →
    (TrProgram P S) →
    (Eval P P') →
    (WFProgram P') →
    (TrProgram P' (Step S)).

Lemma tr_update_layout_preserv
  : (h,h':(Map label HeapVal); l:label; hv,hv':HeapVal)
    (mlookup ? ? h l hv) →
    (mupdate ? ? h l hv' h') →
    (HV_size hv)=(HV_size hv') →
    (Layout_aux h)=(Layout_aux h').

Lemma layout_aux_top_label_plus_nonoverlap
  : (s,l:nat; h:HeapMap; hv:HeapVal)
    (mWF ? ? h) → (OrdHeap h) →
    (msize ? ? h (S s)) →
    (mlookup ? ? h l hv) →
    (lt l s) →
```

(*le* (*plus* (*Layout_aux h l*) (*HV_size hv*)) (*Layout_aux h s*)).

Lemma *layout_aux_label_plus_nonoverlap*
: (*h*:*HeapMap*; *l*,*l'*:*label*; *hv*:*HeapVal*)
(*mWF* ? ? *h*) → (*OrdHeap h*) →
(*mlookup* ? ? *h l hv*) →
(*mindom* ? ? *h l'*) →
(*lt l l'*) →
(*le* (*plus* (*Layout_aux h l*) (*HV_size hv*)) (*Layout_aux h l'*)).

Lemma *layout_non_overlap*
: (*L*:*LayoutF*; *H*:*Heap*; *HT*:*HeapTy*)
(*l*,*l'*:*label*; *hv*,*hv'*:*HeapVal*)
(*WFHeap H HT*) →
L=(*Layout H*) →
(*hlookup* (*hunwrap H*) *l hv*) →
(*hlookup* (*hunwrap H*) *l' hv'*) →
¬*l*=*l'* →
(*le* (*plus* (*L l*) (*HV_size hv*)) (*L l'*))
∨ (*le* (*plus* (*L l'*) (*HV_size hv'*)) (*L l*)).

Lemma *tr_update_wordvallist_preserv*
: (*L*:*LayoutF*; *M*:*Mem*; *l'*:*Word*; *w*:*Word*)
(*V*:(*list WordVal*))
(*l*:*Word*)
(*le* (*plus l* (*length V*)) *l'*) ∨ (*lt l' l*) →
(*TrWordValList L V M l*) →
(*TrWordValList L V* (*updatemem M l' w*) *l*).

Lemma *tr_update_instrseq_preserv*
: (*L*:*LayoutF*; *M*:*Mem*; *l'*:*Word*; *w*:*Word*)
(*IS*:*InstrSeq*)
(*l*:*Word*)
(*le* (*plus l* (*lenInstrSeq IS*)) *l'*) ∨ (*lt l' l*) →
(*TrInstrSeq L IS M l*) →
(*TrInstrSeq L IS* (*updatemem M l' w*) *l*).

Lemma *Preservation_st*
: (*S*:*State*; *H*:*Heap*; *R*:*RegFile*; *a*:*AP*;
rd,*rs*:*Reg*; *i*:*int*; *I*:*InstrSeq*;
P,*P'*:*Program*)
P = (*H*,(*R*,(*a*,(*iseq* (*st rd i rs*) *I*)))) →
(*WFProgram P*) →
(*TrProgram P S*) →

(*Eval P P'*) →
(*WFProgram P'*) →
(*TrProgram P'* (*Step S*)).

Lemma *Preservation_unfold*
: (*S*:*State*; *H*:*Heap*; *R*:*RegFile*; *a*:*AP*;
rd,*rs*:*Reg*; *I*:*InstrSeq*;
P,*P'*:*Program*)
P = (*H*,(*R*,(*a*,(*iseq* (*unfold rd rs*) *I*)))) →
(*WFProgram P*) →
(*TrProgram P S*) →
(*Eval P P'*) →
(*WFProgram P'*) →
(*TrProgram P'* (*Step S*)).

FPCC Preservation theorem (complete)

Theorem *Preservation* : (*S*:*State*) (*Inv S*) → (*Inv* (*Step S*)).

FPCC Safety

Theorem *Safety_aux* : (*S*:*State*) (*n*:*nat*) (*Inv S*) → (*Inv* (*MultiStep n S*)).

Theorem *Safety* : (*S*:*State*) (*Inv S*) → (*Safe S*).

Appendix B

Coq Files for Region-Based TAL and Runtime System

This chapter lists the Coq code for the system described in Chapter 5 of this dissertation. Proofs of some smaller, tedious lemmas in the translation have not yet been completed. These mainly involve straightforward reasoning about updates on memory and machine state.

B.1 The CAP specification layer

Definitions

Definition *addr* := *word*.
Definition *pred* := *state* → *Prop*.
Definition *cdspec* := *word* → *optT* (*prodT nat pred*).
Definition *cmdlist* := *list cmd*.
Definition *wordlist* := *list word*.

Fixpoint *flatten* (*Cs* : *cmdlist*) (*M* : *mem*) (*pc* : *addr*) {*struct Cs*} : *Prop* :=
  match Cs with
  | *nil* ⇒ *True*
  | *c* :: *Cs′* ⇒ *Dc* (*M pc*) = *c* ∧ *flatten Cs′ M* (*S pc*)
  end.

Definition *iscodearea* (*Ct* : *cdspec*) (*w* : *addr*) (*n* : *word*) : *Prop* :=
  ∃ *P*, (∃ *f*, (∃ *s*, (∃ *m*, *Ct f* = *someT* (*pairT s P*) ∧ *w* + *n* ≤ *f* + *s* ∧ *w* = *f* + *m*))).

```
Definition curmem (St : state) := let (X, pc):=St in let (M,R):=X in M.
Definition currf (St : state) := let (X, pc):=St in let (M,R):=X in R.
Definition curpc (St : state) := let (X, pc):=St in let (M,R):=X in pc.
Definition curcmd (St : state) := Dc (curmem St (curpc St)).
Definition curcmdp (St : state) (c:cmd) := Dc (curmem St (curpc St)) = c.
```

Inference rules

```
Section CAPRules.

Variable SP : pred.

(* Well-formed command sequences *)
Inductive WFCapCmds : cdspec → pred → cmdlist → Prop :=
  | wfcapcmd :
      ∀ (Ct : cdspec) (P : pred) c Cs rd rs rt i,
      c = add rd rs rt ∨
      c = addi rd rs i ∨
      c = sub rd rs rt ∨
      c = subi rd rs i ∨
      c = mov rd rs ∨ c = movi rd i ∨ c = ld rd rs i →
      (∀ St : state, P St → curcmdp St c → SP St) →
      (∀ St : state,
        P St →
        curcmdp St c → ∃ Q : pred, Q (Step St) ∧ WFCapCmds Ct Q Cs) →
      WFCapCmds Ct P (c :: Cs)
  | wfcapst :
      ∀ (Ct : cdspec) (P : pred) Cs rd rs i,
      let c := st rd i rs in
      (∀ St : state,
        P St → curcmdp St c → ¬ iscodearea Ct (currf St rd + i) 1) →
      (∀ St : state, P St → curcmdp St c → SP St) →
      (∀ St : state,
        P St →
        curcmdp St c → ∃ Q : pred, Q (Step St) ∧ WFCapCmds Ct Q Cs) →
      WFCapCmds Ct P (c :: Cs)
  | wfcapbgt :
      ∀ (Ct : cdspec) (P : pred) Cs rs rt f,
      let c := bgt rs rt f in
      (∀ St : state, P St → curcmdp St c → SP St) →
      (∀ St : state,
        P St →
```

```
    curcmdp St c →
    currf St rs ¿ currf St rt →
    ∃ n, (∃ R, Ct f = someT (pairT n R) ∧ R (Step St))) →
   (∀ St : state,
    P St →
    curcmdp St c →
    currf St rs ≤ currf St rt →
    ∃ Q : pred, Q (Step St) ∧ WFCapCmds Ct Q Cs) →
   WFCapCmds Ct P (c :: Cs)
| wfcapbgti :
   ∀ (Ct : cdspec) (P : pred) Cs rs i f,
   let c := bgti rs i f in
   (∀ St : state, P St → curcmdp St c → SP St) →
   (∀ St : state,
    P St →
    curcmdp St c →
    currf St rs ≤ i → ∃ Q : pred, Q (Step St) ∧ WFCapCmds Ct Q Cs) →
   (∀ St : state,
    P St →
    curcmdp St c →
    currf St rs ¿ i →
    ∃ n, (∃ R, Ct f = someT (pairT n R) ∧ R (Step St))) →
   WFCapCmds Ct P (c :: Cs)
| wfcapjd :
   ∀ (Ct : cdspec) (P : pred) f,
   (∀ St : state, P St → curcmdp St (jd f) → SP St) →
   (∀ St : state,
    P St →
    curcmdp St (jd f) →
    ∃ n, (∃ Q, Ct f = someT (pairT n Q) ∧ Q (Step St))) →
   WFCapCmds Ct P (jd f :: nil)
| wfcapjmp :
   ∀ (Ct : cdspec) (P : pred) r,
   (∀ St : state, P St → curcmdp St (jmp r) → SP St) →
   (∀ St : state,
    P St →
    curcmdp St (jmp r) →
    ∃ n,
      (∃ Q, Ct (currf St r) = someT (pairT n Q) ∧ Q (Step St))) →
   WFCapCmds Ct P (jmp r :: nil).
```

```
(* Well-formed code specification *)
Definition WFCapcdspec (M : mem) (Ct : cdspec) :
  Prop      :=
  ∀ f n P,
  Ct f = someT (pairT n P) →
  ∃ Cs, flatten Cs M f ∧ length Cs = n ∧ WFCapCmds Ct P Cs.

(* Well-formed machine state *)
Inductive WFCapstate : state → Prop :=
     wfcapstate :
       ∀ M R pc Ct Cs P,
          WFCapcdspec M Ct →
          flatten Cs M pc →
          WFCapCmds Ct P Cs →
          P (M, R, pc) →
          iscodearea Ct pc (length Cs) →
          WFCapstate (M, R, pc).
```

Properties of the CAP system

```
Lemma iscodearea_n_Sn :
 ∀ Ct f n, iscodearea Ct f (S n) → iscodearea Ct (S f) n.

Lemma le_Sm_1 : ∀ f x l, S (f + x) ≤ l → S f ≤ l.

Lemma flatten_noncodeupd :
 ∀ M Ct Cs f x w,
 flatten Cs M f →
 iscodearea Ct f (length Cs) →
 ¬ iscodearea Ct x 1 → flatten Cs (updatemem M x w) f.

Lemma WFCapcdspec_noncodeupd :
 ∀ M Ct x w,
 WFCapcdspec M Ct → ¬ iscodearea Ct x 1 → WFCapcdspec (updatemem M x w) Ct.

Lemma CapPreserv : ∀ St, WFCapstate St → WFCapstate (Step St).

Lemma WFCapstate_SP : ∀ St, WFCapstate St → SP St.

End CAPRules.
```

Definitions and proofs for FPCC package production

```
(* Definition of safety *)
Definition Safe (St : state) (SP : pred) := ∀ n, SP (MultiStep n St).
```

```
(* The most basic safety policy *)
Definition SPbase (St : state) := ¬ curcmdp St ill.

(* CAP well-formedness implies basic safety policy *)
Lemma WFCap_SPbase : ∀ SP St, WFCapstate SP St → SPbase St.

Lemma WFCap_SPadd :
 ∀ (PA SPadd : pred) St,
 WFCapstate PA St → (∀ St, PA St → SPadd St) → SPbase St ∧ SPadd St.

Lemma CapPreservMulti :
 ∀ n (SP : pred) St, WFCapstate SP St → WFCapstate SP (MultiStep n St).

(* CAP well-formedness implies FPCC Safety *)
Theorem WFCap_Safe :
 ∀ (PA SPadd : pred) St,
 WFCapstate PA St →
 (∀ St, PA St → SPadd St) → Safe St (fun S ⇒ SPbase S ∧ SPadd S).
```

B.2 RgnTAL Syntax

Regions

```
(* Region identifiers are identified with integers *)
Definition rgn : Set := nat.

(* A decidable equality on region identifiers *)
Definition beq_rgn : rgn → rgn → bool := beq_nat.

(* Properties of the equality *)
Definition beq_rgn_refl : ∀ p, beq_rgn p p = true
  := fun p ⇒ sym_eq (beq_nat_refl p).
Definition beq_rgn_neq : ∀ p p', p≠p' → beq_rgn p p' = false
  := fun p p' H ⇒ sym_eq (natutil.beq_neq_false p p' H).
Hint Immediate beq_rgn_refl.
```

Capabilities

```
(* Multiplicities are none, unique, or multiple *)
Inductive accap : Set :=
  | noC : accap | uniC : accap | mulC : accap.

Definition beq_accap : accap → accap → bool
  := fun c1 c2 ⇒ match (c1,c2) with
                   | (noC,noC) ⇒ true
```

```
  | (uniC,uniC) ⇒ true
  | (mulC,mulC) ⇒ true
  | _ ⇒ false
end.

(* A set of capabilities is implemented as a (partial) function from region identifiers to multiplicities
*)
Definition capset := rgn → accap.

(* These are syntactic sugar for the Capability Language constructs of Crary, Walker, et al. *)
Definition nullcap : capset := fun r ⇒ noC.
Definition uniqcap : rgn → capset
  := fun rk r ⇒ if (beq_rgn r rk) then uniC else noC.
Definition multcap : rgn → capset
  := fun rk r ⇒ if (beq_rgn r rk) then mulC else noC.
Definition disjcap : capset → capset → Prop :=
  fun A1 A2 ⇒ ∀ p, (A1 p = noC) ∨ (A2 p = noC).
Definition pluscap : ∀ (A1 A2:capset), (disjcap A1 A2) → capset :=
  fun A1 A2 D r ⇒
    if (beq_accap (A1 r) noC) then (A2 r) else
      if (beq_accap (A2 r) noC) then (A1 r) else noC.
Definition barcap : capset → capset
  := fun A r ⇒ match (A r) with uniC ⇒ mulC | _ ⇒ (A r) end.

Notation "A ⋈ B" := (disjcap A B) (at level 0).

Notation "A ⊕ B " C" := (pluscap A B C) (at level 0).
```

Registers and labels

```
Definition label := nat.

Definition beq_label := beq_nat.
Definition beq_label_neq := natutil.beq_neq_false.
Definition beq_label_refl := beq_nat_refl.
Hint Immediate beq_label_refl.

Inductive regt : Set :=
  | r0 : regt | r1 : regt | r2 : regt | r3 : regt
  | r4 : regt | r5 : regt | r6 : regt | r7 : regt.

Load talreg.

Definition beq_regt_refl := beq_regt_true_id.
Definition beq_regt_neq := beq_regt_neq_false.
Hint Immediate beq_regt_refl.
```

Types

```
(* Types with free variables (tracked deBruijn style) *)
Inductive omegaV : nat → Set :=
  | tvvar : ∀ i, omegaV (S i)
  | tvlift : ∀ i, omegaV i → omegaV (S i)

  | tvint : omegaV 0
  | tvhandle : rgn → omegaV 0
  | tvpair : ∀ i, omegaV i → omegaV i → rgn → omegaV i
  | tvcode : ∀ i, capset → (regt → omegaV i) → omegaV i

  | tvabsr : ∀ i, (rgn → omegaV i) → omegaV i
  | tvabsc : ∀ i, (capset → omegaV i) → omegaV i
  | tvabscb : ∀ i, (capset → omegaV i) → capset → omegaV i
  | tvabscd : ∀ i, ∀ (c1 c2:capset), ((c1 ⋈ c2) → omegaV i) → omegaV i
  | tvabst : ∀ i, omegaV (S i) → omegaV i
  | tvrec : ∀ i, omegaV (S i) → omegaV i.

(* Top-level types *)
Inductive omega : Set :=
  | tint : omega
  | thandle : rgn → omega
  | tpair : omega → omega → rgn → omega
  | tcode : capset → (regt → omega) → omega
  | tabsr : (rgn → omega) → omega
  | tabsc : (capset → omega) → omega
  | tabscb : (capset → omega) → capset → omega
  | tabscd : ∀ (c1 c2:capset), ((c1 ⋈ c2) → omega) → omega
  | tabst : (omegaV 1) → omega
  | trec : (omegaV 1) → omega.

(* Utility definitions for reasoning about equality *)
Definition myfequal
  : ∀ (A B : Type) (f : A → B) (x y : A), x = y → f x = f y

Definition myeqaddS : ∀ n m, S n = S m → n = m.

Definition O_S_set : ∀ (A:Set) j, 0=S j → A.

(* Definition of substitution for deBruijn representation *)
Fixpoint subst_aux (i:nat) (t:omegaV i) {struct t}
  : ∀ j, i=(S j) → omegaV j → omegaV j
  := match
       t as X in (omegaV i)
         return (∀ j (p:i=S j) (e':omegaV j), omegaV j) with
```

```
  | tvvar n ⇒ fun j _ e' ⇒ e'
  | tvlift n t' ⇒ fun j (p:S n=S j) _ ⇒
                                  eq_rec n _ t' j (myeqaddS n j p)
  | tvint ⇒ fun j (p:0=S j) _ ⇒ O_S_set _ j p
  | tvhandle _ ⇒ fun j (p:0=S j) _ ⇒ O_S_set _ j p
  | tvpair n t1 t2 p'
                  ⇒ fun j (p:n=S j) e' ⇒
                                  tvpair j (subst_aux n t1 j p e')
                                           (subst_aux n t2 j p e') p'
  | tvcode n A G ⇒ fun j (p:n=S j) e' ⇒
                                  tvcode j A
                                    (fun r ⇒ (subst_aux n (G r) j p e'))
  | tvabsr n Fr ⇒ fun j (p:n=S j) e' ⇒
                                  tvabsr j
                                    (fun p' ⇒ (subst_aux n (Fr p') j p e'))
  | tvabsc n Fc ⇒ fun j (p:n=S j) e' ⇒
                                  tvabsc j
                                    (fun c ⇒ (subst_aux n (Fc c) j p e'))
  | tvabscb n Fc A ⇒ fun j (p:n=S j) e' ⇒
                                  tvabscb j
                                    (fun c ⇒ (subst_aux n (Fc c) j p e')) A
  | tvabscd n c1 c2 Fcd ⇒ fun j (p:n=S j) e' ⇒
                                  tvabscd j c1 c2
                                    (fun D ⇒
                                      (subst_aux n (Fcd D) j p e'))
  | tvabst n t' ⇒ fun j (p:n=S j) e' ⇒
                                  tvabst j
                                    (subst_aux (S n) t' (S j)
                                               (myfequal _ _ S _ _ p)
                                               (tvlift j e'))
  | tvrec n t' ⇒ fun j (p:n=S j) e' ⇒
                                  tvrec j
                                    (subst_aux (S n) t' (S j)
                                               (myfequal _ _ S _ _ p)
                                               (tvlift j e'))
  end.

(* Top-level substitution function *)
Definition substV : omegaV 1 → omegaV 0 → omegaV 0
  := fun T t ⇒ (subst_aux _ T _ (refl_equal 1) t).

(* Converting between [omega] representations *)
```

```
Fixpoint unliftV i (t:omegaV i) {struct t} : 0=i → omega :=
  match t as X in (omegaV i) return (∀ (D:0=i), omega) with
    | tvvar n ⇒ fun D ⇒ (O_S_set _ _ D)
    | tvlift n _ ⇒ fun D ⇒ (O_S_set _ _ D)
    | tvint ⇒ fun _ ⇒ tint
    | tvhandle p ⇒ fun _ ⇒ thandle p
    | tvpair n t1 t2 p ⇒ fun D ⇒ tpair (unliftV n t1 D) (unliftV n t2 D) p
    | tvcode n A G ⇒ fun D ⇒ tcode A (fun r ⇒ (unliftV _ (G r) D))
    | tvabsr n Fr ⇒ fun D ⇒ tabsr (fun p ⇒ (unliftV _ (Fr p) D))
    | tvabsc n Fc ⇒ fun D ⇒ tabsc (fun c ⇒ (unliftV _ (Fc c) D))
    | tvabscb n Fc A ⇒ fun D ⇒ tabscb (fun c ⇒ (unliftV _ (Fc c) D)) A
    | tvabscd n c1 c2 Fcd ⇒ fun D ⇒ tabscd c1 c2(fun Dc
                                                  ⇒ (unliftV _ (Fcd Dc) D))
    | tvabst n t' ⇒ fun D ⇒ tabst (eq_rec _ _ t' 1 (sym_eq (eq_S _ _ D)))
    | tvrec n t' ⇒ fun D ⇒ trec (eq_rec _ _ t' 1 (sym_eq (eq_S _ _ D)))
  end.

Definition unliftV0 : omegaV 0 → omega := fun t ⇒ unliftV _ t (refl_equal _).

Fixpoint lifttoV (t:omega) : omegaV 0 :=
  match t with
  | tint ⇒ tvint
  | thandle p ⇒ tvhandle p
  | tpair t1 t2 p ⇒ tvpair _ (lifttoV t1) (lifttoV t2) p
  | tcode A G ⇒ tvcode _ A (fun r ⇒ (lifttoV (G r)))
  | tabsr Fr ⇒ tvabsr _ (fun p ⇒ (lifttoV (Fr p)))
  | tabsc Fc ⇒ tvabsc _ (fun c ⇒ (lifttoV (Fc c)))
  | tabscb Fc A ⇒ tvabscb _ (fun c ⇒ (lifttoV (Fc c))) A
  | tabscd c1 c2 Fcd ⇒ tvabscd _ _ _ (fun Dc ⇒ (lifttoV (Fcd Dc)))
  | tabst t' ⇒ tvabst _ t'
  | trec t' ⇒ tvrec _ t'
  end.

Definition unfoldV := fun t:(omegaV 1) ⇒ unliftV0 (substV t (tvrec _ t)).

(* Constructors of the type system: regions, capabilities, types *)
Inductive constr : Set :=
  | c_rgn : rgn → constr
  | c_cap : capset → constr
  | c_disj : ∀ c1 c2, c1 ⋈ c2 → constr
  | c_type : omega → constr.

(* Register file type *)
Definition rftype : Set := regt → omega.
```

```
(* Heap region type *)
Definition rgntype : Set := fmap label omega.

(* Data memory type *)
Definition memtype : Set := fmap rgn rgntype.
```

Program state

```
Inductive wordval : Set :=
  | wi : nat → wordval
  | wl : rgn → label → wordval
  | wf : label → wordval
  | wh : rgn → wordval
  | wappr : wordval → rgn → wordval
  | wappc : wordval → capset → wordval
  | wappcd : ∀ (c1 c2:capset), wordval → (c1 ⋈ c2) → wordval
  | wappt : wordval → omega → wordval
  | wfold : wordval → omega → wordval.

Inductive instr : Set :=
  | iadd : regt → regt → regt → instr
  | iaddi : regt → regt → nat → instr
  | imov : regt → regt → instr
  | imovi : regt → nat → instr
  | imovf : regt → label → instr
  | ild : regt → regt → nat → instr
  | ist : regt → nat → regt → instr
  | ibgt : regt → regt → label → instr
  | ibgti : regt → nat → label → instr
  | iappr : regt → rgn → instr
  | iappc : regt → capset → instr
  | iappcd : ∀ c1 c2, regt → (c1 ⋈ c2) → instr
  | iappt : regt → omega → instr
  | ifold : regt → omega → instr
  | iunfold : regt → instr.

Inductive iseq : Set :=
  | icons : instr → iseq → iseq
  | ijd : label → iseq
  | ijmp : regt → iseq.

Inductive codeval : Set :=
  | cvcode : omega → ciseq → codeval
```

```
  | cvstub : omega → codeval.

Inductive heapval : Set :=
  | hvpair : wordval → wordval → heapval.
Notation "[ A , B ]" := (hvpair A B) (at level 0).


(* Heap region *)
Definition heap := fmap label heapval.

(* Data memory *)
Definition datamem := fmap rgn heap.

(* Register file *)
Definition regfile := regt → wordval.

(* Code memory *)
Definition codemem := fmap label codeval.

(* Program state *)
Definition progstate := datamem × regfile × iseq.
```

B.3 RgnTAL Operational Semantics

```
(* Update utility functions *)
Definition rf_upd (R:regfile) (r:regt) (v:wordval) : regfile
  := fun r' ⇒ if (beq_regt r' r) then v else R r'.

Definition hp_upd (H:heap) (l:label) (hv:heapval) : heap
  := (fmapupd beq_label H l hv).

Definition dm_upd (DM:datamem) (p:rgn) (H:heap) : datamem
  := (fmapupd beq_rgn DM p H).

Definition rft_upd (G:rftype) (r:regt) (t:omega) : rftype
  := fun r' ⇒ if (beq_regt r' r) then t else G r'.

(* Utility function for type erasure *)
Fixpoint stripabs (v:wordval) {struct v} : (option label) :=
  match v with
    | wf f ⇒ (Some f)
    | wappr v' _ ⇒ (stripabs v')
    | wappc v' _ ⇒ (stripabs v')
    | wappcd _ _ v' _ ⇒ (stripabs v')
    | wappt v' _ ⇒ (stripabs v')
    | _ ⇒ (None)
```

end.

The Step relation

```
Inductive rt_eval : codemem → progstate → progstate → Prop :=
  | ev_iadd
    : ∀ CM DM R IS rd rs rt s t,
      let i:=(iadd rd rs rt) in
        let R':=(rf_upd R rd (wi (plus s t))) in
          (R rs)=(wi s) →
          (R rt)=(wi t) →
          (rt_eval CM (DM, R, (icons i IS)) (DM, R', IS))
  | ev_iaddi
    : ∀ CM DM R IS rd rs s t,
      let i:=(iaddi rd rs t) in
        let R':=(rf_upd R rd (wi (plus s t))) in
          (R rs)=(wi s) →
          (rt_eval CM (DM, R, (icons i IS)) (DM, R', IS))
  | ev_imov
    : ∀ CM DM R IS rd rs,
      let i:=(imov rd rs) in
        let R':=(rf_upd R rd (R rs)) in
          (rt_eval CM (DM, R, (icons i IS)) (DM, R', IS))
  | ev_imovi
    : ∀ CM DM R IS rd t,
      let i:=(imovi rd t) in
        let R':=(rf_upd R rd (wi t)) in
          (rt_eval CM (DM, R, (icons i IS)) (DM, R', IS))
  | ev_imovf
    : ∀ CM DM R IS rd f,
      let i:=(imovf rd f) in
        let R':=(rf_upd R rd (wf f)) in
          (rt_eval CM (DM, R, (icons i IS)) (DM, R', IS))
  | ev_ild
    : ∀ CM DM R IS rd rs n p l H v0 v1 v,
      let i:=(ild rd rs n) in
        let R':=(rf_upd R rd v) in
          (R rs)=(wl p l) →
          (fmaplook DM p H) →
          (fmaplook H l [v0,v1]) →
          (match n with | 0 ⇒ v=v0 | 1 ⇒ v=v1 | _ ⇒ False end) →
          (rt_eval CM (DM, R, (icons i IS)) (DM, R', IS))
```

```
| ev_ist
  : ∀ CM DM R IS rd n rs p l H v0 v1 v0' v1',
    let i:=(ist rd n rs) in
      let v:=(R rs) in
        let H':=(hp_upd H l [v0',v1']) in
          let DM':=(dm_upd DM p H') in
            (R rd)=(wl p l) →
            (fmaplook DM p H) →
            (fmaplook H l [v0,v1]) →
            (match n with 0 ⇒ v0'=v ∧ v1'=v1
                        | 1 ⇒ v0'=v0 ∧ v1'=v | _ ⇒ False end) →
            (rt_eval CM (DM, R, (icons i IS)) (DM', R, IS))
| ev_ibgt
  : ∀ CM DM R IS rs rt f s t G IS' IS'',
    let i:=(ibgt rs rt f) in
      (R rs)=(wi s) →
      (R rt)=(wi t) →
      (fmaplook CM f (cvcode G IS')) →
      (match (le_gt_dec s t) with left _ ⇒ IS''=IS
                                | _ ⇒ IS''=IS' end) →
      (rt_eval CM (DM, R, (icons i IS)) (DM, R, IS''))
| ev_ibgti
  : ∀ CM DM R IS rs f s t G IS' IS'',
    let i:=(ibgti rs t f) in
      (R rs)=(wi s) →
      (fmaplook CM f (cvcode G IS')) →
      (match (le_gt_dec s t) with left _ ⇒ IS''=IS
                                | _ ⇒ IS''=IS' end) →
      (rt_eval CM (DM, R, (icons i IS)) (DM, R, IS''))
| ev_iappr
  : ∀ CM DM R IS r p,
    let i:=(iappr r p) in
      let R':=(rf_upd R r (wappr (R r) p)) in
        (rt_eval CM (DM, R, (icons i IS)) (DM, R', IS))
| ev_iappc
  : ∀ CM DM R IS r c,
    let i:=(iappc r c) in
      let R':=(rf_upd R r (wappc (R r) c)) in
        (rt_eval CM (DM, R, (icons i IS)) (DM, R', IS))
| ev_iappcd
```

```
  : ∀ CM DM R IS r c1 c2 Dc,
    let i:=(iappcd c1 c2 r Dc) in
      let R':=(rf_upd R r (wappcd c1 c2 (R r) Dc)) in
        (rt_eval CM (DM, R, (icons i IS)) (DM, R', IS))
| ev_iappt
  : ∀ CM DM R IS r t,
    let i:=(iappt r t) in
      let R':=(rf_upd R r (wappt (R r) t)) in
        (rt_eval CM (DM, R, (icons i IS)) (DM, R', IS))
| ev_ifold
  : ∀ CM DM R IS r t,
    let i:=(ifold r t) in
      let R':=(rf_upd R r (wfold (R r) t)) in
        (rt_eval CM (DM, R, (icons i IS)) (DM, R', IS))
| ev_iunfold
  : ∀ CM DM R IS r v t,
    let i:=(iunfold r) in
      let R':=(rf_upd R r v) in
        (R r)=(wfold v t) →
        (rt_eval CM (DM, R, (icons i IS)) (DM, R', IS))
| ev_ijd
  : ∀ CM DM R f G IS,
    (fmaplook CM f (cvcode G IS)) →
    (rt_eval CM (DM, R, (ijd f)) (DM, R, IS))
| ev_ijmp
  : ∀ CM DM R r G IS f,
    (stripabs (R r))=(Some f) →
    (fmaplook CM f (cvcode G IS)) →
    (rt_eval CM (DM, R, (ijmp r)) (DM, R, IS)).
```

B.4 RgnTAL Static Semantics

```
(* Defining capability set equality *)
Definition eqcap (A1 A2:capset) : Prop
  := ∀ p, (A1 p)=(A2 p).

Notation "A =c B" := (eqcap A B) (at level 0).

(* Derivable properties of the equality *)
Lemma eqcap_refl : ∀ A, A =c A.

Lemma eqcap_sym : ∀ A1 A2, A1 =c A2 → A2 =c A1.
```

Lemma *eqcap_trans* : $\forall$ *A1 A2 A3*, *A1* $=_c$ *A2* $\rightarrow$ *A2* $=_c$ *A3* $\rightarrow$ *A1* $=_c$ *A3*.

Lemma *eqcap_barcap_each*
: $\forall$ *A1 A2*, *A1* $=_c$ *A2* $\rightarrow$ (*barcap A1*) $=_c$ (*barcap A2*).

Lemma *eqcap_distrib*
: $\forall$ *A1 A2 D1 D2*,
(*barcap* (*A1* $\oplus$ *A2* " *D1*)) $=_c$ ((*barcap A1*) $\oplus$ (*barcap A2*) " *D2*).

Hint Immediate *eqcap_distrib*.
Hint Immediate *eqcap_refl eqcap_sym eqcap_barcap_each*.

Lemma *eqcap_cong_plus*
: $\forall$ *A1 A2 A1′ A2′ D D′*,
A1 $=_c$ *A1′* $\rightarrow$ *A2* $=_c$ *A2′* $\rightarrow$ (*A1* $\oplus$ *A2* " *D*) $=_c$ (*A1′* $\oplus$ *A2′* " *D′*).

Lemma *eqcap_cong_bar* : $\forall$ *A1 A2*, *A1* $=_c$ *A2* $\rightarrow$ (*barcap A1*) $=_c$ (*barcap A2*).

Lemma *eqcap_null* : $\forall$ *A D*, (*nullcap* $\oplus$ *A* " *D*) $=_c$ *A*.

Lemma *eqcap_comm*
: $\forall$ *A1 A2 D12 D21*, (*A1* $\oplus$ *A2* " *D12*) $=_c$ (*A2* $\oplus$ *A1* " *D21*).

Lemma *eqcap_assoc*
: $\forall$ *A1 A2 A3 D1 D2 D3 D4*,
(*A1* $\oplus$ (*A2* $\oplus$ *A3* " *D1*) " *D2*) $=_c$ ((*A1* $\oplus$ *A2* " *D3*) $\oplus$ *A3* " *D4*).

Lemma *eqcap_dup*
: $\forall$ *A D*, (*barcap A*) $=_c$ ((*barcap A*) $\oplus$ (*barcap A*) " *D*).

Lemma *eqcap_bar_null* : (*barcap nullcap*) $=_c$ *nullcap*.

Lemma *eqcap_flag* : $\forall$ *p*, (*barcap* (*uniqcap p*)) $=_c$ (*multcap p*).

Lemma *eqcap_bar_idem* : $\forall$ *A*, (*barcap* (*barcap A*)) $=_c$ (*barcap A*).

Hint Immediate *eqcap_cong_plus eqcap_cong_bar eqcap_null eqcap_comm*
eqcap_assoc eqcap_dup eqcap_bar_null eqcap_flag eqcap_bar_idem.

(Capability set subtyping *)*
Inductive *subaccap* : *accap* $\rightarrow$ *accap* $\rightarrow$ *Prop* :=
| *subaccap_refl* : $\forall$ *c*, *subaccap c c*
| *subaccap_mult* : *subaccap uniC mulC*.

Definition *subcap* : *capset* $\rightarrow$ *capset* $\rightarrow$ *Prop*
:= *fun A1 A2* $\Rightarrow$ $\forall$ *p*, *subaccap* (*A1 p*) (*A2 p*).

Notation "*A* $\subseteq_c$ *B*" := (*subcap A B*) (*at level* 0).

Lemma *subcap_eq* : $\forall$ *A1 A2*, *A1* $=_c$ *A2* $\rightarrow$ *A1* $\subseteq_c$ *A2*.

Lemma *subcap_trans*

```
  : ∀ A1 A2, A1 ⊆c A2 → ∀ A3, A2 ⊆c A3 → A1 ⊆c A3.

Lemma subcap_cong_bar
  : ∀ A1 A2, A1 ⊆c A2 → (barcap A1) ⊆c (barcap A2).

Lemma subcap_bar : ∀ A, A ⊆c (barcap A).

(* Code type instantiation *)
Inductive instcodetype : omega → list constr → omega → Prop :=
  | inst_code
      : ∀ t,
        instcodetype t (nil) t
  | inst_absr
      : ∀ Fr g Cs t,
        instcodetype (Fr g) Cs t →
        instcodetype (tabsr Fr) ((c_rgn g)::Cs) t
  | inst_absc
      : ∀ Fc c Cs t,
        instcodetype (Fc c) Cs t →
        instcodetype (tabsc Fc) ((c_cap c)::Cs) t
  | inst_abscb
      : ∀ Fc A c Cs t,
        instcodetype (Fc c) Cs t →
        c ⊆c A →
        instcodetype (tabscb Fc A) ((c_cap c)::Cs) t
  | inst_abscd
      : ∀ c1 c2 Fcd Dc Cs t,
        instcodetype (Fcd Dc) Cs t →
        instcodetype (tabscd c1 c2 Fcd) ((c_disj c1 c2 Dc)::Cs) t
  | inst_abst
      : ∀ Ft t Cs t' t'',
        t' = unliftV0 (substV Ft (lifttoV t)) →
        instcodetype t' Cs t'' →
        instcodetype (tabst Ft) ((c_type t)::Cs) t''.

(* Type equality *)
Definition eqtype (t1 t2:omega) : Prop
  := t1=t2.

(* Register file type equality *)
Definition eqrftype (G1 G2:rftype) : Prop := ∀ r, eqtype (G1 r) (G2 r).

(* Properties of type equality *)
Definition eqtype_refl : ∀ t, eqtype t t := fun t ⇒ refl_equal t.
```

```
Definition eqtype_sym : ∀ t1 t2, eqtype t1 t2 → eqtype t2 t1
  := fun t1 t2 D ⇒ (sym_eq D).

Definition eqtype_trans
  : ∀ t1 t2 t3, eqtype t1 t2 → eqtype t2 t3 → eqtype t1 t3
  := fun t1 t2 t3 D1 D2 ⇒ (trans_eq D1 D2).

Hint Immediate eqtype_refl eqtype_sym.

Hint Immediate eqtype_refl eqtype_sym.

(* Memory type lookup *)
Definition memtypeof (MT:memtype) (p:rgn) (l:label) (t:omega) : Prop
  := ∃ g, (fmaplook MT p g) ∧ (fmaplook g l t).

(* Syntactic restrictions on code types *)
Inductive iscodetype : omega → Prop :=
  | isct_tabsr : ∀ F, (∀ x, iscodetype (F x)) → (iscodetype (tabsr F))
  | isct_tabsc : ∀ F, (∀ x, iscodetype (F x)) → (iscodetype (tabsc F))
  | isct_tabscb : ∀ F A, (∀ x, iscodetype (F x)) → (iscodetype (tabscb F A))
  | isct_tabscd : ∀ c1 c2 F, (∀ (D:c1 ⋈ c2), iscodetype (F D)) →
                    (iscodetype (tabscd c1 c2 F))
  | isct_tabst : ∀ F, (∀ x, iscodetype (unliftV0 (substV F (lifttoV x)))) →
                    (iscodetype (tabst F))
  | isct_tcode : ∀ A G, iscodetype (tcode A G).

Definition codevaltype (cv:codeval) : omega :=
  match cv with (cvcode t _) ⇒ t | (cvstub t) ⇒ t end.

(* Well-formed word values *)
Inductive wf_wordval : codemem → memtype → wordval → omega → Prop :=
  | wfv_int : ∀ CM MT i, wf_wordval CM MT (wi i) tint
  | wfv_addr : ∀ CM MT p l t,
                 memtypeof MT p l t →
                 wf_wordval CM MT (wl p l) t
  | wfv_addr_pair
              : ∀ CM MT p l t1 t2,
                 notindomf MT p →
                 wf_wordval CM MT (wl p l) (tpair t1 t2 p)
  | wfv_codeptr
              : ∀ CM MT f cv t,
                 fmaplook CM f cv →
                 eqtype t (codevaltype cv) →
                 wf_wordval CM MT (wf f) t
  | wfv_handle
```

```
                : ∀ CM MT g,
                  wf_wordval CM MT (wh g) (thandle g)
  | wfv_typer
                : ∀ CM MT v g Fr,
                  wf_wordval CM MT v (tabsr Fr) →
                  iscodetype (tabsr Fr) →
                  wf_wordval CM MT (wappr v g) (Fr g)
  | wfv_typec
                : ∀ CM MT v A Fc,
                  wf_wordval CM MT v (tabsc Fc) →
                  iscodetype (tabsc Fc) →
                  wf_wordval CM MT (wappc v A) (Fc A)
  | wfv_typecb
                : ∀ CM MT v A Fc A',
                  wf_wordval CM MT v (tabscb Fc A') →
                  subcap A A' →
                  iscodetype (tabscb Fc A') →
                  wf_wordval CM MT (wappc v A) (Fc A)
  | wfv_typet
                : ∀ CM MT v t Ft t',
                  wf_wordval CM MT v (tabst Ft) →
                  unliftV0 (substV Ft (lifttoV t)) = t' →
                  iscodetype (tabst Ft) →
                  wf_wordval CM MT (wappt v t) t'
  | wfv_fold
                : ∀ CM MT v t,
                  wf_wordval CM MT v (unliftV0 (substV t (tvrec _ t))) →
                  wf_wordval CM MT (wfold v (trec t)) (trec t).

(* Well-formed instruction sequences *)
Inductive wf_iseq : codemem → capset → rftype → iseq → Prop :=
  | wf_iadd
        : ∀ CM A G rd rs rt Is,
          G(rs)=tint →
          G(rt)=tint →
          wf_iseq CM A (rft_upd G rd tint) Is →
          wf_iseq CM A G (icons (iadd rd rs rt) Is)
  | wf_iaddi
        : ∀ CM A G rd rs t Is,
          G(rs)=tint →
          wf_iseq CM A (rft_upd G rd tint) Is →
```

$wf\_iseq\ CM\ A\ G\ (icons\ (iaddi\ rd\ rs\ t)\ Is)$

| wf_imov

: $\forall\ CM\ A\ G\ rd\ rs\ Is,$
$wf\_iseq\ CM\ A\ (rft\_upd\ G\ rd\ (G(rs)))\ Is \rightarrow$
$wf\_iseq\ CM\ A\ G\ (icons\ (imov\ rd\ rs)\ Is)$

| wf_imovi

: $\forall\ CM\ A\ G\ rd\ s\ Is,$
$wf\_iseq\ CM\ A\ (rft\_upd\ G\ rd\ tint)\ Is \rightarrow$
$wf\_iseq\ CM\ A\ G\ (icons\ (imovi\ rd\ s)\ Is)$

| wf_imovf

: $\forall\ CM\ A\ G\ rd\ f\ Is\ cv\ t,$
$fmaplook\ CM\ f\ cv \rightarrow$
$eqtype\ t\ (codevaltype\ cv) \rightarrow$
$wf\_iseq\ CM\ A\ (rft\_upd\ G\ rd\ t)\ Is \rightarrow$
$wf\_iseq\ CM\ A\ G\ (icons\ (imovf\ rd\ f)\ Is)$

| wf_ild

: $\forall\ CM\ A\ G\ rd\ rs\ n\ Is\ t\ t1\ t2\ g,$
$(n{=}0 \wedge t{=}t1) \vee (n{=}1 \wedge t{=}t2) \rightarrow$
$G(rs) = tpair\ t1\ t2\ g \rightarrow$
$subaccap\ (A\ g)\ mulC \rightarrow$
$wf\_iseq\ CM\ A\ (rft\_upd\ G\ rd\ t)\ Is \rightarrow$
$wf\_iseq\ CM\ A\ G\ (icons\ (ild\ rd\ rs\ n)\ Is)$

| wf_ist

: $\forall\ CM\ A\ G\ rd\ n\ rs\ Is\ t\ t1\ t2\ g,$
$(n{=}0 \wedge t{=}t1) \vee (n{=}1 \wedge t{=}t2) \rightarrow$
$G(rd) = tpair\ t1\ t2\ g \rightarrow$
$G(rs) = t \rightarrow$
$subaccap\ (A\ g)\ mulC \rightarrow$
$wf\_iseq\ CM\ A\ G\ Is \rightarrow$
$wf\_iseq\ CM\ A\ G\ (icons\ (ist\ rd\ n\ rs)\ Is)$

| wf_ibgt

: $\forall\ CM\ A\ G\ rs\ rt\ f\ Is\ cv\ ts\ A'\ G',$
$G(rs) = tint \rightarrow$
$G(rt) = tint \rightarrow$
$fmaplook\ CM\ f\ cv \rightarrow instcodetype\ (codevaltype(cv))\ ts\ (tcode\ A'\ G') \rightarrow$
$eqrftype\ G\ G' \rightarrow$
$subcap\ A\ A' \rightarrow$
$wf\_iseq\ CM\ A\ G\ Is \rightarrow wf\_iseq\ CM\ A\ G\ (icons\ (ibgt\ rs\ rt\ f)\ Is)$

| *wf_ibgti*
: ∀ *CM A G rs t f Is cv ts A′ G′*,
G(*rs*) = *tint* →
fmaplook CM f cv → *instcodetype* (*codevaltype*(*cv*)) *ts* (*tcode A′ G′*) →
eqrftype G G′ →
subcap A A′ →
wf_iseq CM A G Is → *wf_iseq CM A G* (*icons* (*ibgti rs t f*) *Is*)
| *wf_iappr*
: ∀ *CM A G r g Is Fr*,
G(*r*) = *tabsr Fr* →
wf_iseq CM A (*rft_upd G r* (*Fr g*)) *Is* →
wf_iseq CM A G (*icons* (*iappr r g*) *Is*)
| *wf_iappc*
: ∀ *CM A G r c Is Fc*,
G(*r*) = *tabsc Fc* →
wf_iseq CM A (*rft_upd G r* (*Fc c*)) *Is* →
wf_iseq CM A G (*icons* (*iappc r c*) *Is*)
| *wf_iappcb*
: ∀ *CM A G r c Is Fc A′*,
G(*r*) = *tabscb Fc A′* →
subcap c A′ →
wf_iseq CM A (*rft_upd G r* (*Fc c*)) *Is* →
wf_iseq CM A G (*icons* (*iappc r c*) *Is*)
| *wf_iappcd*
: ∀ *CM A G r c1 c2 Dc Is Fcd*,
G(*r*) = *tabscd c1 c2 Fcd* →
wf_iseq CM A (*rft_upd G r* (*Fcd Dc*)) *Is* →
wf_iseq CM A G (*icons* (*iappcd c1 c2 r Dc*) *Is*)
| *wf_iappt*
: ∀ *CM A G r t Is Ft t′*,
G(*r*) = *tabst Ft* →
t′ = *unliftV0* (*substV Ft* (*lifttoV t*)) →
wf_iseq CM A (*rft_upd G r t′*) *Is* →
wf_iseq CM A G (*icons* (*iappt r t*) *Is*)
| *wf_ifold*
: ∀ *CM A G r t Is*,
G(*r*) = *unfoldV t* →
wf_iseq CM A (*rft_upd G r* (*trec t*)) *Is* →
wf_iseq CM A G (*icons* (*ifold r* (*trec t*)) *Is*)

```
  | wf_iunfold
        : ∀ CM A G r Is t t',
          G(r) = trec t →
          t' = unfoldV t →
          wf_iseq CM A (rft_upd G r t') Is →
          wf_iseq CM A G (icons (iunfold r) Is)
  | wf_ijd
        : ∀ CM A G f cv ts A' G',
          fmaplook CM f cv →          instcodetype (codevaltype(cv)) ts (tcode A' G') →
          eqrftype G G' →
          subcap A A' →
          wf_iseq CM A G (ijd f)
  | wf_ijmp
        : ∀ CM A G r t ts A' G',
          G(r) = t →
          instcodetype t ts (tcode A' G') →          eqrftype G G' →
          subcap A A' →
          wf_iseq CM A G (ijmp r).

(* Well-formed code heap values *)
Inductive wf_codeval : codemem → codeval → Prop :=
  | wf_cvcode : ∀ CM t Is,
                  iscodetype t →
                  (∀ Ts A G, instcodetype t Ts (tcode A G) →
                                  wf_iseq CM A G Is) →
                  wf_codeval CM (cvcode t Is)
  | wf_cvstub : ∀ CM t,
                  iscodetype t →
                  wf_codeval CM (cvstub t).

(* Well-formed data type values *)
Inductive wf_heapval : codemem → memtype → heapval → rgn → omega → Prop :=
  | wf_hvpair : ∀ CM MT v1 v2 t1 t2 g,
                  wf_wordval CM MT v1 t1 →
                  wf_wordval CM MT v2 t2 →
                  wf_heapval CM MT [v1, v2] g (tpair t1 t2 g).

(* Well-formed heap region *)
Definition wf_heap : codemem → memtype → heap → rgn → rgntype → Prop :=
  fun CM MT H g gt ⇒
    (eqdomf _ _ _ H gt) ∧
    (∀ l hv t, fmaplook H l hv → fmaplook gt l t →
                    wf_heapval CM MT hv g t).
```

```
(* Well-formed data memory *)
Definition wf_datamem : codemem → datamem → memtype → Prop :=
  fun CM M MT ⇒
    (eqdomf _ _ _ M MT) ∧
    (∀ p H gt, fmaplook M p H → fmaplook MT p gt →
                          wf_heap CM MT H p gt).

(* Well-formed register file *)
Definition wf_regfile : codemem → memtype → regfile → rftype → Prop :=
  fun CM MT R G ⇒
    ∀ r, wf_wordval CM MT (R r) (G r).

(* Well-formed code memory *)
Definition wf_codemem : codemem → Prop :=
  fun CM ⇒ ∀ l cv, fmaplook CM l cv → wf_codeval CM cv.

(* Memory type–capability satisfiability *)
Definition sat_cap_memtype : memtype → capset → Prop :=
  fun MT A ⇒
    (∀ g, indomf MT g → subaccap (A g) mulC) ∧
    (∀ g, subaccap (A g) mulC → indomf MT g).

(* Well-formed program state *)
Definition wf_state
  : codemem → datamem → regfile → memtype → capset → rftype → Prop :=
  fun CM M R MT A G ⇒
    wf_codemem CM ∧
    wf_datamem CM M MT ∧
    wf_regfile CM MT R G ∧
    sat_cap_memtype MT A.

(* Well-formed program *)
Inductive wf_program : (codemem × progstate) → Prop :=
  | wf_rgntalprog : ∀ CM M R Is MT A G,
                          wf_state CM M R MT A G →
                          wf_iseq CM A G Is →
                          wf_program (CM, (M, R, Is)).
```

B.5 RgnTAL Soundness Proofs

Lemmas about the types that appear in the data and code memory types

```
Lemma memtype_not_int
  : ∀ CM M MT p l, wf_datamem CM M MT → ¬memtypeof MT p l tint.
```

Lemma *memtype_not_handle*
: ∀ *CM M MT p l g*, *wf_datamem CM M MT* → ¬*memtypeof MT p l* (*thandle g*).

Lemma *memtype_not_rec*
: ∀ *CM M MT p l t*, *wf_datamem CM M MT* → ¬*memtypeof MT p l* (*trec t*).

Lemma *memtype_not_code*
: ∀ *CM M MT p l A G*, *wf_datamem CM M MT* → ¬*memtypeof MT p l* (*tcode A G*).

Lemma *memtype_not_absr*
: ∀ *CM M MT p l Fr*, *wf_datamem CM M MT* → ¬*memtypeof MT p l* (*tabsr Fr*).

Lemma *memtype_not_absc*
: ∀ *CM M MT p l Fc*, *wf_datamem CM M MT* → ¬*memtypeof MT p l* (*tabsc Fc*).

Lemma *memtype_not_abscb*
: ∀ *CM M MT p l Fc A*, *wf_datamem CM M MT* → ¬*memtypeof MT p l* (*tabscb Fc A*).

Lemma *memtype_not_abscd*
: ∀ *CM M MT p l c1 c2 Fc*,
wf_datamem CM M MT → ¬*memtypeof MT p l* (*tabscd c1 c2 Fc*).

Lemma *memtype_not_abst*
: ∀ *CM M MT p l Ft*,
wf_datamem CM M MT → ¬*memtypeof MT p l* (*tabst Ft*).

Lemmas about the syntactic form of code value types

Lemma *iscodetype_not_int* : ∀ *t*, *iscodetype t* → *t* ≠ *tint*.

Lemma *codevaltype_not_int*
: ∀ *CM f cv*, *wf_codemem CM* → *fmaplook CM f cv* → ¬*codevaltype cv=tint*.

Lemma *codevaltype_not_handle*
: ∀ *CM f cv g*, *wf_codemem CM* → *fmaplook CM f cv* → ¬*codevaltype cv=*(*thandle g*).

Lemma *codevaltype_not_rec*
: ∀ *CM f cv t*, *wf_codemem CM* → *fmaplook CM f cv* → ¬*codevaltype cv=trec t*.

Lemma *codevaltype_not_pair*
: ∀ *CM f cv t1 t2 p*, *wf_codemem CM* → *fmaplook CM f cv* → ¬*codevaltype cv=tpair t1 t2 p*.

Lemma *codevaltype_iscodetype*
: ∀ *CM f cv*, *wf_codemem CM* → *fmaplook CM f cv* → *iscodetype* (*codevaltype cv*).

Type erasure with code pointers

Lemma *wf_tabsr_stripabs*
: ∀ *CM M MT v t*,
wf_datamem CM M MT →
wf_wordval CM MT v t →

(∃ *Fr*, *t* = (*tabsr Fr*)) ∨
(∃ *Fc*, *t* = (*tabsc Fc*)) ∨
(∃ *Fc*, ∃ *A*, *t* = (*tabscb Fc A*)) ∨
(∃ *c1*, ∃ *c2*, ∃ *Fc*, *t* = (*tabscd c1 c2 Fc*)) ∨
(∃ *Ft*, *t* = (*tabst Ft*)) →
∃ *f*, *stripabs v* = *Some f*.

Canonical forms lemmas

Lemma *canform_regval_int*
: ∀ *CM M R MT A G r*,
wf_state CM M R MT A G →
G(*r*)=*tint* →
∃ *s*, *R*(*r*)=*wi s*.

Lemma *canform_regval_handle*
: ∀ *CM M R MT A G r g*,
wf_state CM M R MT A G →
G(*r*)=*thandle g* →
R(*r*)=*wh g*.

Lemma *canform_regval_pair*
: ∀ *CM M R MT A G r t1 t2 g*,
wf_state CM M R MT A G →
G(*r*) = *tpair t1 t2 g* →
subaccap (*A g*) *mulC* →
∃ *p*, ∃ *l*, ∃ *H*, ∃ *v1*, ∃ *v2*,
R(*r*) = (*wl p l*) ∧
fmaplook M p H ∧
fmaplook H l [v1, v2].

Lemma *canform_regval_rec*
: ∀ *CM M R MT A G r t*,
wf_state CM M R MT A G →
G(*r*)=*trec t* →
∃ *v*, *R*(*r*)=*wfold v* (*trec t*).

Lemma *canform_regval_code*
: ∀ *CM M R MT A G r A′ G′*,
wf_state CM M R MT A G →
G(*r*)=*tcode A′ G′* →
∃ *f*, *stripabs*(*R*(*r*))=(*Some f*).

Lemma *canform_regval_absr*
: ∀ *CM M R MT A G r Fr*,

wf_state CM M R MT A G →
G(r)=tabsr Fr →
∃ f, stripabs(R(r))=(Some f).

Lemma *canform_regval_absc*
: *∀ CM M R MT A G r F,*
wf_state CM M R MT A G →
G(r)=tabsc F →
∃ f, stripabs(R(r))=(Some f).

Lemma *canform_regval_abscb*
: *∀ CM M R MT A G r F A′,*
wf_state CM M R MT A G →
G(r)=tabscb F A′ →
∃ f, stripabs(R(r))=(Some f).

Lemma *canform_regval_abscd*
: *∀ CM M R MT A G r c1 c2 F,*
wf_state CM M R MT A G →
G(r)=tabscd c1 c2 F →
∃ f, stripabs(R(r))=(Some f).

Lemma *canform_regval_abst*
: *∀ CM M R MT A G r F,*
wf_state CM M R MT A G →
G(r)=tabst F →
∃ f, stripabs(R(r))=(Some f).

Progress lemmas

Definition *isgoodjump* : *codemem → regfile → iseq → Prop* :=
fun CM R Is ⇒
match Is with
| (icons (ibgt _ _ f) _) ⇒
(∃ t′, ∃ Is′, (fmaplook CM f (cvcode t′ Is′)))
| (icons (ibgti _ _ f) _) ⇒
(∃ t′, ∃ Is′, (fmaplook CM f (cvcode t′ Is′)))
| (ijd f) ⇒
(∃ t′, ∃ Is′, (fmaplook CM f (cvcode t′ Is′)))
| (ijmp r) ⇒
∀ f, stripabs(R r)=(Some f) →
(∃ t′, ∃ Is′, (fmaplook CM f (cvcode t′ Is′)))
| _ ⇒ True
end.

Lemma *progress_iadd*

: ∀ *CM M R MT A G rd rs rt Is*,
let curIs := (*icons* (*iadd rd rs rt*) *Is*) *in*
wf_state CM M R MT A G →
wf_iseq CM A G curIs →
∃ *P*, *rt_eval CM* (*M*, *R*, *curIs*) *P*.

Lemma *progress_iaddi*
: ∀ *CM M R MT A G rd rs t Is*,
let curIs := (*icons* (*iaddi rd rs t*) *Is*) *in*
wf_state CM M R MT A G →
wf_iseq CM A G curIs →
∃ *P*, *rt_eval CM* (*M*, *R*, *curIs*) *P*.

Lemma *progress_imov*
: ∀ *CM M R MT A G rd rs Is*,
let curIs := (*icons* (*imov rd rs*) *Is*) *in*
wf_state CM M R MT A G →
wf_iseq CM A G curIs →
∃ *P*, *rt_eval CM* (*M*, *R*, *curIs*) *P*.

Lemma *progress_imovi*
: ∀ *CM M R MT A G rd t Is*,
let curIs := (*icons* (*imovi rd t*) *Is*) *in*
wf_state CM M R MT A G →
wf_iseq CM A G curIs →
∃ *P*, *rt_eval CM* (*M*, *R*, *curIs*) *P*.

Lemma *progress_imovf*
: ∀ *CM M R MT A G rd f Is*,
let curIs := (*icons* (*imovf rd f*) *Is*) *in*
wf_state CM M R MT A G →
wf_iseq CM A G curIs →
∃ *P*, *rt_eval CM* (*M*, *R*, *curIs*) *P*.

Lemma *progress_ild*
: ∀ *CM M R MT A G rd rs n Is*,
let curIs := (*icons* (*ild rd rs n*) *Is*) *in*
wf_state CM M R MT A G →
wf_iseq CM A G curIs →
∃ *P*, *rt_eval CM* (*M*, *R*, *curIs*) *P*.

Lemma *progress_ist*
: ∀ *CM M R MT A G rd rs n Is*,
let curIs := (*icons* (*ist rd n rs*) *Is*) *in*
wf_state CM M R MT A G →

wf_iseq CM A G curIs →
∃ P, rt_eval CM (M, R, curIs) P.

Lemma *progress_ibgt*
: *∀ CM M R MT A G rs rt f Is,*
let curIs := (icons (ibgt rs rt f) Is) in
wf_state CM M R MT A G →
wf_iseq CM A G curIs →
isgoodjump CM R curIs →
∃ P, rt_eval CM (M, R, curIs) P.

Lemma *progress_ibgti*
: *∀ CM M R MT A G rs t f Is,*
let curIs := (icons (ibgti rs t f) Is) in
wf_state CM M R MT A G →
wf_iseq CM A G curIs →
isgoodjump CM R curIs →
∃ P, rt_eval CM (M, R, curIs) P.

Lemma *progress_iappr*
: *∀ CM M R MT A G r g Is,*
let curIs := (icons (iappr r g) Is) in
wf_state CM M R MT A G →
wf_iseq CM A G curIs →
∃ P, rt_eval CM (M, R, curIs) P.

Lemma *progress_iappc*
: *∀ CM M R MT A G r c Is,*
let curIs := (icons (iappc r c) Is) in
wf_state CM M R MT A G →
wf_iseq CM A G curIs →
∃ P, rt_eval CM (M, R, curIs) P.

Lemma *progress_iappcd*
: *∀ CM M R MT A G r c1 c2 Dc Is,*
let curIs := (icons (iappcd c1 c2 r Dc) Is) in
wf_state CM M R MT A G →
wf_iseq CM A G curIs →
∃ P, rt_eval CM (M, R, curIs) P.

Lemma *progress_iappt*
: *∀ CM M R MT A G r t Is,*
let curIs := (icons (iappt r t) Is) in
wf_state CM M R MT A G →
wf_iseq CM A G curIs →

$\exists\ P,\ rt\_eval\ CM\ (M, R, curIs)\ P.$

Lemma *progress_ifold*
: $\forall\ CM\ M\ R\ MT\ A\ G\ r\ t\ Is,$
$let\ curIs := (icons\ (ifold\ r\ t)\ Is)\ in$
$wf\_state\ CM\ M\ R\ MT\ A\ G \rightarrow$
$wf\_iseq\ CM\ A\ G\ curIs \rightarrow$
$\exists\ P,\ rt\_eval\ CM\ (M, R, curIs)\ P.$

Lemma *progress_iunfold*
: $\forall\ CM\ M\ R\ MT\ A\ G\ r\ Is,$
$let\ curIs := (icons\ (iunfold\ r)\ Is)\ in$
$wf\_state\ CM\ M\ R\ MT\ A\ G \rightarrow$
$wf\_iseq\ CM\ A\ G\ curIs \rightarrow$
$\exists\ P,\ rt\_eval\ CM\ (M, R, curIs)\ P.$

Lemma *progress_ijd*
: $\forall\ CM\ M\ R\ MT\ A\ G\ f,$
$let\ curIs := (ijd\ f)\ in$
$wf\_state\ CM\ M\ R\ MT\ A\ G \rightarrow$
$wf\_iseq\ CM\ A\ G\ curIs \rightarrow$
$isgoodjump\ CM\ R\ curIs \rightarrow$
$\exists\ P,\ rt\_eval\ CM\ (M, R, curIs)\ P.$

Lemma *progress_ijmp*
: $\forall\ CM\ M\ R\ MT\ A\ G\ r,$
$let\ curIs := (ijmp\ r)\ in$
$wf\_state\ CM\ M\ R\ MT\ A\ G \rightarrow$
$wf\_iseq\ CM\ A\ G\ curIs \rightarrow$
$isgoodjump\ CM\ R\ curIs \rightarrow$
$\exists\ P,\ rt\_eval\ CM\ (M, R, curIs)\ P.$

Preservation lemmas

Lemma *preserv_iadd*
: $\forall\ CM\ M\ R\ MT\ A\ G\ rd\ rs\ rt\ Is\ M'\ R'\ Is',$
$let\ curIs := (icons\ (iadd\ rd\ rs\ rt)\ Is)\ in$
$let\ P := (M', R', Is')\ in$
$wf\_state\ CM\ M\ R\ MT\ A\ G \rightarrow$
$wf\_iseq\ CM\ A\ G\ curIs \rightarrow$
$rt\_eval\ CM\ (M, R, curIs)\ P \rightarrow$
$\exists\ MT',$
$wf\_state\ CM\ M'\ R'\ MT'\ A\ (rft\_upd\ G\ rd\ tint).$

Lemma *preserv_iaddi*

: ∀ *CM M R MT A G rd rs t Is M′ R′ Is′*,
let curIs := (*icons* (*iaddi rd rs t*) *Is*) *in*
let P := (*M′*, *R′*, *Is′*) *in*
wf_state CM M R MT A G →
wf_iseq CM A G curIs →
rt_eval CM (*M*, *R*, *curIs*) *P* →
∃ *MT′*,
wf_state CM M′ R′ MT′ A (*rft_upd G rd tint*).

Lemma *preserv_imov*
: ∀ *CM M R MT A G rd rs Is M′ R′ Is′*,
let curIs := (*icons* (*imov rd rs*) *Is*) *in*
let P := (*M′*, *R′*, *Is′*) *in*
wf_state CM M R MT A G →
wf_iseq CM A G curIs →
rt_eval CM (*M*, *R*, *curIs*) *P* →
∃ *MT′*,
wf_state CM M′ R′ MT′ A (*rft_upd G rd* (*G rs*)).

Lemma *preserv_imovi*
: ∀ *CM M R MT A G rd t Is M′ R′ Is′*,
let curIs := (*icons* (*imovi rd t*) *Is*) *in*
let P := (*M′*, *R′*, *Is′*) *in*
wf_state CM M R MT A G →
wf_iseq CM A G curIs →
rt_eval CM (*M*, *R*, *curIs*) *P* →
∃ *MT′*,
wf_state CM M′ R′ MT′ A (*rft_upd G rd tint*).

Lemma *preserv_imovf*
: ∀ *CM M R MT A G rd f Is M′ R′ Is′*,
let curIs := (*icons* (*imovf rd f*) *Is*) *in*
let P := (*M′*, *R′*, *Is′*) *in*
wf_state CM M R MT A G →
wf_iseq CM A G curIs →
rt_eval CM (*M*, *R*, *curIs*) *P* →
∃ *MT′*, ∃ *cv*, ∃ *t*,
fmaplook CM f cv ∧ *eqtype t* (*codevaltype cv*) ∧
wf_state CM M′ R′ MT′ A (*rft_upd G rd t*).

Lemma *preserv_ild*
: ∀ *CM M R MT A G rd rs n Is M′ R′ Is′*,
let curIs := (*icons* (*ild rd rs n*) *Is*) *in*

$\mathit{let}\ P := (M', R', Is')\ \mathit{in}$
$\mathit{wf\_state}\ CM\ M\ R\ MT\ A\ G \rightarrow$
$\mathit{wf\_iseq}\ CM\ A\ G\ \mathit{curIs} \rightarrow$
$\mathit{rt\_eval}\ CM\ (M, R, \mathit{curIs})\ P \rightarrow$
$\exists\ MT', \exists\ t, \exists\ t1, \exists\ t2, \exists\ g,$
$(n = 0 \wedge t = t1 \vee n = 1 \wedge t = t2) \wedge (G\ rs = \mathit{tpair}\ t1\ t2\ g) \wedge$
$\mathit{wf\_state}\ CM\ M'\ R'\ MT'\ A\ (\mathit{rft\_upd}\ G\ rd\ t).$

Lemma *preserv_ist*
$: \forall\ CM\ M\ R\ MT\ A\ G\ rd\ rs\ n\ Is\ M'\ R'\ Is',$
$\mathit{let}\ \mathit{curIs} := (\mathit{icons}\ (\mathit{ist}\ rd\ n\ rs)\ Is)\ \mathit{in}$
$\mathit{let}\ P := (M', R', Is')\ \mathit{in}$
$\mathit{wf\_state}\ CM\ M\ R\ MT\ A\ G \rightarrow$
$\mathit{wf\_iseq}\ CM\ A\ G\ \mathit{curIs} \rightarrow$
$\mathit{rt\_eval}\ CM\ (M, R, \mathit{curIs})\ P \rightarrow$
$\exists\ MT',$
$\mathit{wf\_state}\ CM\ M'\ R'\ MT'\ A\ G.$

Lemma *preserv_ibgt*
$: \forall\ CM\ M\ R\ MT\ A\ G\ rs\ rt\ f\ Is\ M'\ R'\ Is',$
$\mathit{let}\ \mathit{curIs} := (\mathit{icons}\ (\mathit{ibgt}\ rs\ rt\ f)\ Is)\ \mathit{in}$
$\mathit{let}\ P := (M', R', Is')\ \mathit{in}$
$\mathit{wf\_state}\ CM\ M\ R\ MT\ A\ G \rightarrow$
$\mathit{wf\_iseq}\ CM\ A\ G\ \mathit{curIs} \rightarrow$
$\mathit{rt\_eval}\ CM\ (M, R, \mathit{curIs})\ P \rightarrow$
$\exists\ MT',$
$\mathit{wf\_state}\ CM\ M'\ R'\ MT'\ A\ G\ .$

Lemma *preserv_ibgti*
$: \forall\ CM\ M\ R\ MT\ A\ G\ rs\ t\ f\ Is\ M'\ R'\ Is',$
$\mathit{let}\ \mathit{curIs} := (\mathit{icons}\ (\mathit{ibgti}\ rs\ t\ f)\ Is)\ \mathit{in}$
$\mathit{let}\ P := (M', R', Is')\ \mathit{in}$
$\mathit{wf\_state}\ CM\ M\ R\ MT\ A\ G \rightarrow$
$\mathit{wf\_iseq}\ CM\ A\ G\ \mathit{curIs} \rightarrow$
$\mathit{rt\_eval}\ CM\ (M, R, \mathit{curIs})\ P \rightarrow$
$\exists\ MT',$
$\mathit{wf\_state}\ CM\ M'\ R'\ MT'\ A\ G\ .$

Lemma *preserv_iappr*
$: \forall\ CM\ M\ R\ MT\ A\ G\ r\ g\ Is\ M'\ R'\ Is',$
$\mathit{let}\ \mathit{curIs} := (\mathit{icons}\ (\mathit{iappr}\ r\ g)\ Is)\ \mathit{in}$
$\mathit{let}\ P := (M', R', Is')\ \mathit{in}$
$\mathit{wf\_state}\ CM\ M\ R\ MT\ A\ G \rightarrow$

wf_iseq CM A G curIs →
rt_eval CM (M, R, curIs) P →
∃ MT', ∃ Fr,
G r = tabsr Fr ∧
wf_state CM M' R' MT' A (rft_upd G r (Fr g)).

Lemma *preserv_iappc*
: *∀ CM M R MT A G r c Is M' R' Is',*
let curIs := (icons (iappc r c) Is) in
let P := (M', R', Is') in
wf_state CM M R MT A G →
wf_iseq CM A G curIs →
rt_eval CM (M, R, curIs) P →
∃ MT',
∃ Fc, (G r = tabsc Fc ∨ ∃ A', G r = tabscb Fc A') ∧
wf_state CM M' R' MT' A (rft_upd G r (Fc c)).

Lemma *preserv_iappt*
: *∀ CM M R MT A G r t Is M' R' Is',*
let curIs := (icons (iappt r t) Is) in
let P := (M', R', Is') in
wf_state CM M R MT A G →
wf_iseq CM A G curIs →
rt_eval CM (M, R, curIs) P →
∃ MT', ∃ A', ∃ G',
wf_state CM M' R' MT' A' G' .

Lemma *preserv_ifold*
: *∀ CM M R MT A G r t Is M' R' Is',*
let curIs := (icons (ifold r t) Is) in
let P := (M', R', Is') in
wf_state CM M R MT A G →
wf_iseq CM A G curIs →
rt_eval CM (M, R, curIs) P →
∃ MT', ∃ A', ∃ G',
wf_state CM M' R' MT' A' G' .

Lemma *preserv_iunfold*
: *∀ CM M R MT A G r Is M' R' Is',*
let curIs := (icons (iunfold r) Is) in
let P := (M', R', Is') in
wf_state CM M R MT A G →
wf_iseq CM A G curIs →

```
rt_eval CM (M, R, curIs) P →
∃ MT', ∃ A', ∃ G',
  wf_state CM M' R' MT' A' G' .
```

B.6 Translating RgnTAL to CAP

Notation for separation logic primitives

```
Definition mempred := fmapPred word word.

Definition truemp := @truefp word word.
Definition falsemp : mempred := fun _ ⇒ False.
Definition emptymp : mempred := @emptyfp word word.

Notation "P1 & P2" := (andfp P1 P2) (at level 0).
Notation "' P" := (liftfp P) (at level 0).
Notation "'ex P" := (@existsfp _ _ _ P) (at level 0).
Notation "a |-> w" := (singfp a w) (at level 0).
Notation "a |-> ?" := ('ex fun w ⇒ (a |-> w)) (at level 0).

Definition appliesto (M:fmap word word) (MM:mem) : Prop :=
  ∀ a w, fmaplook M a w → MM a = w.
```

The runtime system

```
Section rgntal2tis.

Definition rgnsize := 10.
Definition nilptr := 0.

Definition heaplyt := fmap label word.
Definition datamemlyt := fmap rgn heaplyt.
Definition rgnlyt := fmap rgn word.
Definition extcodety := word → optT (list cmd ** pred).

Variable ExtCode : extcodety.

Variable DMLyt : datamemlyt.
Variable RLyt : rgnlyt.
Variable CMLyt : heaplyt.

(* Translating a data pointer using the layout functions *)
Definition tr_dataptr : rgn → label → option word
  := fun p l ⇒ match (DMLyt p) with
                 | Some HL ⇒ match (HL l) with
                               | Some w ⇒ Some w
                               | _ ⇒ None
```

```
                                    end
                              | _ ⇒ None
                          end.

Fixpoint iseq_size (Is : iseq) : nat :=
  match Is with
    | icons _ Is' ⇒ S (iseq_size Is')
    | _ ⇒ 1
  end.

Fixpoint wordsinmem (ws : wordlist) (w : word) {struct ws} : mempred :=
  match ws with
  | nil ⇒ emptymp
  | w' :: ws' ⇒ ((w |-> w') && (wordsinmem ws' (S w)))
  end.

Definition cmdsinmem (Cs : cmdlist) w : mempred :=
  'ex fun Ws ⇒ '(map Dc Ws = Cs) & (wordsinmem Ws w).

Fixpoint tailnth (n : nat) A (Ls : list A) {struct n} :
 option (list A) :=
  match n with
  | O ⇒ Some Ls
  | S m ⇒ match Ls with
              | nil ⇒ None
              | _ ⇒ tailnth m _ (tail Ls)
              end
  end.
```

Describing the free list

```
Fixpoint ptrtoanylist (a sz:word) {struct sz} : mempred :=
  match sz with
    | 0 ⇒ emptymp
    | (S n) ⇒ ((a |-> ?) && (ptrtoanylist (a+1) n))
  end.

Definition freeblock (a sz nxt:word) : mempred :=
  '(a ≠ nilptr) &
  ((a |-> sz) &&
  (((a+1) |-> ?) && (((a+2) |-> nxt) && (ptrtoanylist (a+3) sz)))).

Definition newblock (a sz:word) : mempred :=
  '(a ≠ nilptr) &
  ((a |-> sz) &&
  (((a+1) |-> 0) && (((a+2) |-> ?) && (ptrtoanylist (a+3) sz)))).

Fixpoint fblist (n a:word) {struct n} : mempred :=
```

```
  match n with
    | 0 ⇒ '(a=nilptr) & emptymp
    | (S n') ⇒ '(a≠nilptr)
                  & ('ex fun sz ⇒
                      'ex fun a' ⇒ ((freeblock a sz a') && (fblist n' a')))
  end.

Definition freelist (a:word) : mempred :=
  'ex fun n ⇒ (fblist n a).
```

Translation relations

```
(* Word Values and Data Heap Values *)

Inductive tr_wordval : wordval → word → Prop :=
  | tr_wi : ∀ n, tr_wordval (wi n) n
  | tr_wl : ∀ p l w,
                tr_dataptr p l = Some w →
                tr_wordval (wl p l) w
  | tr_wlnop : ∀ p l w,
                notindomf DMLyt p →
                tr_wordval (wl p l) w
  | tr_wlnol : ∀ p l HLyt w,
                fmaplook DMLyt p HLyt →
                notindomf HLyt l →
                tr_wordval (wl p l) w
  | tr_wf : ∀ f w,
                fmaplook CMLyt f w →
                tr_wordval (wf f) w
  | tr_wh : ∀ p w,
                fmaplook RLyt p w →
                tr_wordval (wh p) w
  | tr_wappr : ∀ v p w,
                tr_wordval v w →
                tr_wordval (wappr v p) w
  | tr_wappc : ∀ v c w,
                tr_wordval v w →
                tr_wordval (wappc v c) w
  | tr_wappcd : ∀ v c1 c2 Dc w,
                tr_wordval v w →
                tr_wordval (wappcd c1 c2 v Dc) w
  | tr_wappt : ∀ v t w,
```

```
                    tr_wordval v w →
                    tr_wordval (wappt v t) w
  | tr_wfold : ∀ v t w,
                    tr_wordval v w →
                    tr_wordval (wfold v t) w.

(* Utility functions for maps *)
Fixpoint limmapfoldr
  B1 B2
  (Lim:nat)
  (H:natfmap B1)
  (f:B1 → B2 → B2)
  (b2:B2) {struct Lim} : B2 :=
  match Lim with
    | 0 ⇒ b2
    | (S m) ⇒ match (H Lim) with
                  | None ⇒ limmapfoldr _ _ m H f b2
                  | Some b1 ⇒ f b1 (limmapfoldr _ _ m (fmapdelN _ H Lim) f b2)
                end
  end.

Fixpoint limmapstarfp
  B1 B2
  (Lim:nat)
  (H:natfmap B1)
  (eP : nat → B1 → (fmapPred nat B2)) {struct Lim}
                                          : fmapPred nat B2 :=
  match Lim with
    | 0 ⇒ ('(nulldomf H) & (@emptyfp _ _))
    | (S m) ⇒ match (H Lim) with
                  | None ⇒ limmapstarfp _ _ m H eP
                  | Some b ⇒ ((eP Lim b) &&
                                  (limmapstarfp _ _ m (fmapdelN _ H Lim) eP))
                end
    end.

Definition tr_heapval (p:rgn) (l:nat) (a:word) (hv:heapval) : mempred :=
  match hv with [v0,v1] ⇒
    'ex fun w0 ⇒
      'ex fun w1 ⇒
          '(tr_dataptr p l = (Some a)) &
          '(tr_wordval v0 w0) &
```

```
        'tr_wordval v1 w1 &
        ( a |-> w0 && (a+1) |-> w1))
  end.

Fixpoint tr_hvs (p:rgn) (a:word) (Lim:word) (H:heap) {struct Lim} : mempred :=
  match Lim with
    | 0 => ('nulldomf H & @emptyfp _ _)
    | S m => match H Lim with
                 | None => tr_hvs p a m H
                 | Some hv => (tr_heapval p Lim a hv &&
                                  tr_hvs p (2+a) m (fmapdelN _ H Lim)))
               end
  end.

Definition hvlist_size (Lim:nat) (H:heap) : nat :=
  limmapfoldr _ _ Lim H (fun hv sz => 2+sz) 0.

Definition tr_heap : rgn -> heap -> mempred :=
  fun p H =>
    'ex fun a =>
    'ex fun hplim =>
    'ex fun rsize =>
    'ex fun diff =>
      'a <> nilptr &
      'fmaplook RLyt p a &
      '@limitf heapval H hplim &
      '(hvlist_size hplim H)+diff = rsize &
      ((a |-> rsize) &&
        ((a+1) |-> hvlist_size hplim H) &&
        (a+2) |-> ?)) &&
        (tr_hvs p (a+3) hplim H) &&
        ptrtoanylist (a+3+(hvlist_size hplim H)) diff))).

(* Instructions and code values *)

Definition regt2reg (r : regt) : reg := nat2reg (regt2nat r).
Coercion regt2reg : regt >-> reg.

Inductive tr_instr : cmdlist -> instr -> cmdlist -> Prop :=
  | tr_iadd : forall Cs rd rs rt,
                tr_instr Cs (iadd rd rs rt) (add rd rs rt :: Cs)
  | tr_iaddi : forall Cs rd rs t,
                tr_instr Cs (iaddi rd rs t) (addi rd rs t :: Cs)
  | tr_imov : forall Cs rd rs,
                tr_instr Cs (imov rd rs) (mov rd rs :: Cs)
```

```
  | tr_imovi : ∀ Cs rd s,
                tr_instr Cs (imovi rd s) (movi rd s :: Cs)
  | tr_imovf : ∀ Cs rd f w,
                fmaplook CMLyt f w →
                tr_instr Cs (imovf rd f) (movi rd w :: Cs)
  | tr_ild : ∀ Cs rd rs n,
                tr_instr Cs (ild rd rs n) (ld rd rs n :: Cs)
  | tr_ist : ∀ Cs rd n rs,
                tr_instr Cs (ist rd n rs) (st rd n rs :: Cs)
  | tr_ibgt : ∀ Cs rs rt f w,
                fmaplook CMLyt f w →
                tr_instr Cs (ibgt rs rt f) (bgt rs rt w :: Cs)
  | tr_ibgti : ∀ Cs rs t f w,
                fmaplook CMLyt f w →
                tr_instr Cs (ibgti rs t f) (bgti rs t w :: Cs)
  | tr_iappr : ∀ Cs r x,
                tr_instr Cs (iappr r x) (mov r r :: Cs)
  | tr_iappc : ∀ Cs r x,
                tr_instr Cs (iappc r x) (mov r r :: Cs)
  | tr_iappcd : ∀ Cs r c1 c2 Dc,
                tr_instr Cs (iappcd c1 c2 r Dc) (mov r r :: Cs)
  | tr_iappt : ∀ Cs r x,
                tr_instr Cs (iappt r x) (mov r r :: Cs)
  | tr_ifold : ∀ Cs r x,
                tr_instr Cs (ifold r x) (mov r r :: Cs)
  | tr_iunfold : ∀ Cs r,
                tr_instr Cs (iunfold r) (mov r r :: Cs).

Inductive tr_iseq : iseq → cmdlist → Prop :=
  | tr_icons : ∀ i Is cs Cs,
                tr_iseq Is Cs → tr_instr Cs i cs → tr_iseq (icons i Is) cs
  | tr_ijd : ∀ f w,
                fmaplook CMLyt f w →
                tr_iseq (ijd f) (jd w :: nil)
  | tr_ijmp : ∀ r,
                tr_iseq (ijmp r) (jmp r :: nil).

Inductive tr_codeval : codeval → wordlist → Prop :=
  | tr_cvcode : ∀ G Is Cs Ws,
                  tr_iseq Is Cs →
                  Cs = map Dc Ws →
```

```
                    tr_codeval (cvcode G Is) Ws
  | tr_cvstub : ∀ G, tr_codeval (cvstub G) nil.

(* Translating the data and code heaps *)
Section sizevars.

Variables ec_min ec_size cm_min cm_size : word.

Definition tr_datamem_aux (Lim:nat) (DM:datamem) : mempred
  := limmapstarfp _ _ Lim DM tr_heap.

Definition tr_datamem (DM:datamem) dm_min dm_size : mempred :=
    'ex fun dmlim ⇒
    'ex fun fp ⇒          (fun M ⇒ coversfnat _ M dm_min (dm_min+dm_size)) &
      '(limitf _ DM dmlim) &
      ((dm_min |-> dm_size) &&
        (((dm_min+1) |-> fp) && ((tr_datamem_aux dmlim DM) && (freelist fp)))).

Definition tr_datamem_exc_freelist (DM:datamem) dm_min dm_size : mempred :=
    'ex fun dmlim ⇒
      '(limitf _ DM dmlim) &
        ((dm_min |-> dm_size) && (tr_datamem_aux dmlim DM)).

Definition tr_codemem (CM:codemem) : mempred :=
  fun M ⇒
    (coversfnat _ M cm_min (cm_min+cm_size)) ∧
    (∀ f cv,
      fmaplook CM f cv →
      ∃ Ws, ∃ a, fmaplook CMLyt f a ∧ tr_codeval cv Ws
                                    ∧ ((wordsinmem Ws a) && truemp) M).

Definition extcode_in_ec : Prop :=
  (∀ a Cs Pcs,
    ExtCode a = someT (Cs,*Pcs) →
      (ec_min ≤ a ∧ a+(length Cs) ¡ ec_min+ec_size)).

Definition tr_extcode : codemem → mem → Prop :=
  fun CM MM ⇒
    ∃ M,
      (extcode_in_ec) ∧
      (∀ f w G Is,
        fmaplook CMLyt f w → fmaplook CM f (cvcode G Is)
                                    → ExtCode w = noneT _) ∧
      (coversfnat _ M ec_min (ec_min+ec_size)) ∧
      (∀ a Cs Pcs, ExtCode a = someT (Cs,*Pcs) → cmdsinmem Cs a M) ∧
      (appliesto M MM).
```

```
Definition tr_memstate_aux (CM:codemem) (DM:datamem) : mempred :=
  'ex fun dm_min ⇒ 'ex fun dm_size ⇒
      '(eqdomf _ _ _ CMLyt CM)
    & '(eqdomf _ _ _ DMLyt DM)
    & (((0 |-> ?)
    && ((1 |-> cm_min)
    && ((2 |-> dm_min))))
    && ((tr_codemem CM)
    && (tr_datamem DM dm_min dm_size)))
.

Definition tr_memstate (CM:codemem) (DM:datamem) (MM:mem) : Prop :=
  ∃ M,
    tr_memstate_aux CM DM M
    ∧ (appliesto M MM)
    ∧ True (*(∃ mlim, ∀ a, a ≥ mlim → notindomf M a)*).

(* Register file and current program counter *)
Definition tr_regfile (Rf:regfile) (R:rfile) : Prop :=
  ∀ r v w, Rf r = v → R r = w → tr_wordval v w.

(* Tail subsets of instruction sequences *)
Inductive subseqis : iseq → iseq → nat → Prop :=
  | isubd0 : ∀ Is, subseqis Is Is 0
  | isubds : ∀ Is Is' i n, subseqis Is' Is n → subseqis Is' (icons i Is) (S n).

Definition tr_pc (CM:codemem) (Is:iseq) (pc:word) : Prop :=
  ∃ f, ∃ G', ∃ Is', ∃ w, ∃ n,
    fmaplook CM f (cvcode G' Is') ∧ subseqis Is Is' n ∧ fmaplook CMLyt f w ∧ pc = n + w.

(* The top-level translation relation between RgnTAL programs and machine states *)

Definition maxrgnprop (CM:codemem) (DM:datamem) (R:regfile) :=
  ∃ maxp,
    ∀ p, (rgn_in_codemem CM p ∨
                rgn_in_datamem DM p ∨
                rgn_in_regfile R p) → le p maxp.

Inductive tr_program : codemem → progstate → state → Prop :=
  | tr_prog :
      ∀ CM DM R Is MM RR pc,
        tr_extcode CM MM →
        maxrgnprop CM DM R →
        tr_memstate CM DM MM →
        tr_regfile R RR →
```

tr_pc CM Is pc →
tr_program CM (DM,R,Is) (MM,RR,pc).

End *sizevars*.
End *rgntal2tis*.

B.7 Correctness of RgnTAL to CAP Translation

Section *Proofs*.

Variable *ExtCode* : *extcodety*.
Variable *CMLyt* : *heaplyt*.
Variables *ec_min ec_size cm_min cm_size* : *word*.

A custom safety policy

```
Definition MySP (St:state) :=
  match St with (M,R,pc) ⇒
    let dm_min := (M 2) in
    let dm_size := (M dm_min) in
      match (curcmd St) with
        | ld rd rs n ⇒ let a:=(R rs)+n in dm_min ¡ a ¡ (dm_min+dm_size)
                                  ∨ (ec_min ≤ pc ¡ (ec_min+ec_size))
        | st rd n rs ⇒ let a:=(R rd)+n in dm_min ¡ a ¡ (dm_min+dm_size)
                                  ∨ (ec_min ≤ pc ¡ (ec_min+ec_size))
        | _ ⇒ True
      end
  end.
```

Definition of CpInv

```
Definition cpinv (CM:codemem) (T:omega) : pred :=
  fun St ⇒ match St with (MM,RR,pc) ⇒
        ∃ DLyt, ∃ RLyt, ∃ DM, ∃ RF, ∃ MT, ∃ Ts, ∃ A, ∃ G,
          instcodetype T Ts (tcode A G) ∧
          wf_state CM DM RF MT A G ∧
          tr_memstate DLyt RLyt CMLyt cm_min cm_size CM DM MM ∧
          tr_regfile DLyt RLyt CMLyt RF RR ∧
          maxrgnprop CM DM RF
    end.
```

Constraints on CpGen

```
Definition ext_in_ct (CT:cdspec) : Prop :=
  ∀ w Cs P,
    ExtCode w = someT (Cs ,* P) →
    CT w = someT (length Cs ,* P).
```

```
Definition cmcode_in_ct (CT:cdspec) (CM:codemem) : Prop :=
  ∀ f G Is w,
    fmaplook CM f (cvcode G Is) →
    fmaplook CMLyt f w →
    CT w = someT (iseq_size Is ,* cpinv CM G).

Definition cmstub_in_ext (CM:codemem) : Prop :=
  ∀ f G w,
    fmaplook CM f (cvstub G) →
    fmaplook CMLyt f w →
    ∃ Cs, ∃ P, ExtCode w = someT (Cs ,* P).

Definition cmlyt_in_ct (CT:cdspec) : Prop :=
  ∀ f w,
    fmaplook CMLyt f w →
    ∃ P, CT w = someT P.

Definition ct_from_ext_or_cm (CT:cdspec) (CM:codemem) : Prop :=
  ∀ w n P,
    CT w = someT (n ,* P) →
    (∃ Cs, ExtCode w = someT (Cs ,* P) ∧ n = length Cs)
    ∨
    (∃ f, ∃ G, ∃ Is,
      fmaplook CM f (cvcode G Is) ∧
      fmaplook CMLyt f w ∧
      n = iseq_size Is ∧
      P = cpinv CM G).

Definition ext_interf_corr (CM:codemem) : Prop :=
  ∀ w Cs P,
    ExtCode w = someT (Cs ,* P) →
    (∀ f G,
      fmaplook CM f (cvstub G) →
      fmaplook CMLyt f w →
      (∀ St, cpinv CM G St →
                    ec_min ≤ (curpc St) →
                    (curpc St)+(length Cs) ¡ ec_min + ec_size →
                    P St)).

Definition iscpgen (CT:cdspec) (CM:codemem) : Prop :=
    onetoonef CMLyt ∧
    ext_in_ct CT ∧
    cmcode_in_ct CT CM ∧
    cmstub_in_ext CM ∧
```

cmlyt_in_ct CT ∧
ct_from_ext_or_cm CT CM ∧
ext_interf_corr CM.

Utility lemmas

Lemma *regt_neq_reg_neq*
: ∀ *r r'*, *r* ≠ *r'* → *regt2reg r* ≠ *regt2reg r'*.

Lemma *limmapstarfp_eqf*
: ∀ *B1 B2 Lim* (*H H'*:*natfmap B1*) *eP M*,
eqf H H' →
limmapstarfp B1 B2 Lim H eP M →
limmapstarfp B1 B2 Lim H' eP M.

Lemma *limmapstarfp_fmaplook_and_limmapstar_fmapdel* :
∀ *B1 B2 flim F P M n b*,
limmapstarfp B1 B2 flim F P M →
fmaplook F n b →
((*P n b*) && (*limmapstarfp B1 B2 flim* (*fmapdelN* _ *F n*) *P*)) *M*.

Lemma *limmapstarfp_fmaplook_and_truefp* :
∀ *B1 B2 flim F P M n b*,
limmapstarfp B1 B2 flim F P M →
fmaplook F n b →
((*P n b*) && (@*truefp* _ _)) *M*.

Lemma *tr_datamem_aux_tr_heap_M* :
∀ *DLyt RLyt dmlim DM M p H*,
tr_datamem_aux DLyt RLyt CMLyt dmlim DM M →
(*fmaplook DM p H*) →
((*tr_heap DLyt RLyt CMLyt p H*) && (*truemp*)) *M*.

Lemma *tr_heapval_tr_wordval* :
∀ *DLyt RLyt hplim H M a p l v0 v1*,
tr_hvs DLyt RLyt CMLyt p a hplim H M →
(*fmaplook H l* [v0,v1]) →
∃ *a*,
(‘*ex* (*fun w0* ⇒ ‘*ex* (*fun w1* ⇒
‘(*tr_dataptr DLyt p l* = *Some a*) &
‘(*tr_wordval DLyt RLyt CMLyt v0 w0*) &
‘(*tr_wordval DLyt RLyt CMLyt v1 w1*) &
((*a* |–> *w0*) && ((*a*+1) |–> *w1*)))) && *truemp*) *M*.

Lemma *tr_heap_tr_wordval* :

∀ *DLyt RLyt H M p l v0 v1*,
tr_heap DLyt RLyt CMLyt p H M →
fmaplook H l [v0,v1] →
∃ *a*,
(‘*ex* (*fun w0* ⇒ ‘*ex* (*fun w1* ⇒
‘(*tr_dataptr DLyt p l* = *Some a*) &
‘(*tr_wordval DLyt RLyt CMLyt v0 w0*) &
‘(*tr_wordval DLyt RLyt CMLyt v1 w1*) &
((*a* |−> *w0*) && ((*a*+1) |−> *w1*)))) && *truemp*) *M*.

Lemma *tr_memstate_tr_wordval* :
∀ *DLyt RLyt CM DM MM*
RF RR p H l v0 v1 r,
tr_memstate DLyt RLyt CMLyt cm_min cm_size CM DM MM →
tr_regfile DLyt RLyt CMLyt RF RR →
fmaplook DM p H →
fmaplook H l [v0, v1] →
RF r = *wl p l* →
tr_wordval DLyt RLyt CMLyt v0 (*MM* (*RR r* + 0)) ∧
tr_wordval DLyt RLyt CMLyt v1 (*MM* (*RR r* + 1)).

Lemma *appliesto_eqf*
: ∀ *M MM M1 M2* (*x*:*disjf M1 M2*),
appliesto M MM →
eqf (*appendf x*) *M* →
appliesto M1 MM.

Lemma *appliesto_pappendf_left* :
∀ *MM M M1 M2*,
appliesto M MM →
pappendf M1 M2 M →
appliesto M1 MM.

Lemma *appliesto_pappendf_right* :
∀ *MM M M1 M2*,
appliesto M MM →
pappendf M1 M2 M →
appliesto M2 MM.

Lemma *cmdsin_flatten*
: ∀ *Cs M MM w*,
cmdsinmem Cs w M → *appliesto M MM* → *flatten Cs MM w*.

Lemma *triseq_size_eq* :
∀ *Lyt Is Cs*, *tr_iseq Lyt Is Cs* → *length Cs* = *iseq_size Is*.

Lemma *tr_iseq_subseq* :
∀ *n Is Is' Ws MM a*,
subseqis Is Is' n →
tr_iseq CMLyt Is' (*map Dc Ws*) →
flatten (*map Dc Ws*) *MM a* →
∃ *Ws'*,
tr_iseq CMLyt Is (*map Dc Ws'*) ∧ *flatten* (*map Dc Ws'*) *MM* (*n*+*a*).

Lemma *not_listnth_nil* :
∀ *A n p*, ~(*listnth A nil n*)=(*Some p*).

Main lemmas and proofs

Preservation of CpInv for RgnTAL instructions

Lemma *cpinv_preserv_add* :
∀ *CM A G rd rs rt Is St*
(*D0* : *cpinv CM* (*tcode A G*) *St*)
(*D1* : *wf_iseq CM A G* (*icons* (*iadd rd rs rt*) *Is*))
(*D2* : *curcmd St* = (*add rd rs rt*)),
cpinv CM (*tcode A* (*rft_upd G rd tint*)) (*Step St*).

Lemma *cpinv_preserv_addi* :
∀ *CM A G rd rs t Is St*
(*D0* : *cpinv CM* (*tcode A G*) *St*)
(*D1* : *wf_iseq CM A G* (*icons* (*iaddi rd rs t*) *Is*))
(*D2* : *curcmd St* = (*addi rd rs t*)),
cpinv CM (*tcode A* (*rft_upd G rd tint*)) (*Step St*).

Lemma *cpinv_preserv_mov* :
∀ *CM A G rd rs Is St*
(*D0* : *cpinv CM* (*tcode A G*) *St*)
(*D1* : *wf_iseq CM A G* (*icons* (*imov rd rs*) *Is*))
(*D2* : *curcmd St* = (*mov rd rs*)),
cpinv CM (*tcode A* (*rft_upd G rd* (*G rs*))) (*Step St*).

Lemma *cpinv_preserv_movi* :
∀ *CM A G rd s Is St*
(*D0* : *cpinv CM* (*tcode A G*) *St*)
(*D1* : *wf_iseq CM A G* (*icons* (*imovi rd s*) *Is*))
(*D2* : *curcmd St* = (*movi rd s*)),
cpinv CM (*tcode A* (*rft_upd G rd tint*)) (*Step St*).

Lemma *cpinv_preserv_movf* :
∀ *CM A G rd f Is St a*

```
    (D0 : cpinv CM (tcode A G) St)
    (D1 : wf_iseq CM A G (icons (imovf rd f) Is))
    (D2 : curcmd St = (movi rd a))
    (D3 : fmaplook CMLyt f a),
    ∃ cv, ∃ t,
      fmaplook CM f cv ∧
      eqtype t (codevaltype cv) ∧
      cpinv CM (tcode A (rft_upd G rd t)) (Step St).

Lemma cpinv_preserv_ld :
  ∀ CM A G rd rs n Is St
    (D0 : cpinv CM (tcode A G) St)
    (D1 : wf_iseq CM A G (icons (ild rd rs n) Is))
    (D2 : curcmd St = (ld rd rs n)),
    ∃ t, ∃ t1, ∃ t2, ∃ g,
      (n = 0 ∧ t = t1 ∨ n = 1 ∧ t = t2) ∧
      G rs = tpair t1 t2 g ∧
      cpinv CM (tcode A (rft_upd G rd t)) (Step St).

Lemma tr_memstate_heap_update0 :
  ∀ DLyt RLyt CM DM MM
        RF RR p H l v0 v1 rd rs,
    tr_memstate DLyt RLyt CMLyt cm_min cm_size CM DM MM →
    tr_regfile DLyt RLyt CMLyt RF RR →
    fmaplook DM p H →
    fmaplook H l [v0, v1] →
    RF rd = wl p l →
    tr_memstate DLyt RLyt CMLyt cm_min cm_size CM
      (dm_upd DM p (hp_upd H l [RF rs, v1]))
      (updatemem MM (RR rd) (RR rs)).

Lemma tr_memstate_heap_update1 :
  ∀ DLyt RLyt CM DM MM
        RF RR p H l v0 v1 rd rs,
    tr_memstate DLyt RLyt CMLyt cm_min cm_size CM DM MM →
    tr_regfile DLyt RLyt CMLyt RF RR →
    fmaplook DM p H →
    fmaplook H l [v0, v1] →
    RF rd = wl p l →
    tr_memstate DLyt RLyt CMLyt cm_min cm_size CM
      (dm_upd DM p (hp_upd H l [v0, RF rs]))
      (updatemem MM (RR rd + 1) (RR rs)).
```

Lemma *cpinv_preserv_st* :
∀ *CM A G rd n rs Is St*
(*D0* : *cpinv CM* (*tcode A G*) *St*)
(*D1* : *wf_iseq CM A G* (*icons* (*ist rd n rs*) *Is*))
(*D2* : *curcmd St* = (*st rd n rs*)),
cpinv CM (*tcode A G*) (*Step St*).

Lemma *cpinv_preserv_appr* :
∀ *CM A G r p Is St*
(*D0* : *cpinv CM* (*tcode A G*) *St*)
(*D1* : *wf_iseq CM A G* (*icons* (*iappr r p*) *Is*))
(*D2* : *curcmd St* = (*mov r r*)),
∃ *Fr*,
G r = *tabsr Fr* ∧
cpinv CM (*tcode A* (*rft_upd G r* (*Fr p*))) (*Step St*).

Load and store RgnTAL instructions respect the custom safety policy

Lemma *wf_iseq_ld_MySP* :
∀ *DLyt RLyt CM DM MM*
RF RR pc MT A G rd rs n Is,
wf_state CM DM RF MT A G →
tr_memstate DLyt RLyt CMLyt cm_min cm_size CM DM MM →
tr_regfile DLyt RLyt CMLyt RF RR →
wf_iseq CM A G (*icons* (*ild rd rs n*) *Is*) →
curcmdp (*MM*, *RR*, *pc*) (*ld rd rs n*) →
MySP (*MM*, *RR*, *pc*).

Lemma *wf_iseq_st_MySP* :
∀ *DLyt RLyt CM DM MM*
RF RR pc MT A G rd rs n Is,
wf_state CM DM RF MT A G →
tr_memstate DLyt RLyt CMLyt cm_min cm_size CM DM MM →
tr_regfile DLyt RLyt CMLyt RF RR →
wf_iseq CM A G (*icons* (*ist rd n rs*) *Is*) →
curcmdp (*MM*, *RR*, *pc*) (*st rd n rs*) →
MySP (*MM*, *RR*, *pc*).

RgnTAL-CAP instruction safety

Lemma *rgntal2wfcapcmds* :
∀ *CT CM Is T Ws Cs*
(*D0* : *iscpgen CT CM*)
(*D1* : *extcode_in_ec ExtCode ec_min ec_size*)
(*D2* : *tr_codeval CMLyt* (*cvcode T Is*) *Ws*)

(*D3* : *Cs* = *map Dc Ws*)
(*D4* : ∀ *Ts A G*, *instcodetype T Ts* (*tcode A G*)
→ *wf_iseq CM A G Is*),
WFCapCmds MySP CT (*cpinv CM T*) *Cs*.

RgnTAL-CAP code heap safety

Lemma *rgntal2wfcapcdspec* :
∀ *DLyt RLyt CT CM DM R Is MM RR pc*
(*D0*:*iscpgen CT CM*)
(*D1*:*wf_program* (*CM*, (*DM*,*R*,*Is*)))
(*D2*:*tr_program ExtCode DLyt RLyt CMLyt ec_min ec_size*
cm_min cm_size CM (*DM*,*R*,*Is*) (*MM*,*RR*,*pc*))
(*extcode_wf*: ∀ *f Cs P*,
ExtCode f = *someT* (*Cs* ,* *P*) →
∀ *CT'*, *iscpgen CT' CM* → *WFCapCmds MySP CT' P Cs*),
WFCapcdspec MySP MM CT.

RgnTAL-CAP safety theorem

Theorem *rgntal2cap* :
∀ *DLyt RLyt CT CM DM R Is St*
(*D0*:*iscpgen CT CM*)
(*D1*:*wf_program* (*CM*, (*DM*,*R*,*Is*)))
(*D2*:*tr_program ExtCode DLyt RLyt CMLyt ec_min ec_size*
cm_min cm_size CM (*DM*,*R*,*Is*) *St*)
(*extcode_wf*: ∀ *f Cs P*,
ExtCode f = *someT* (*Cs* ,* *P*) →
∀ *CT'*, *iscpgen CT' CM* → *WFCapCmds MySP CT' P Cs*),
WFCapstate MySP St.

End *Proofs*.

B.8 RgnTAL Runtime System

The free region library function

Definition *freergn_cmds* : *cmdlist* :=
(*movi rI* 2) :: (*ld rI rI* 0) :: (*ld rJ rI* 1) :: (*st rA* 2 *rJ*) :: (*st rI* 1 *rA*) :: (*jmp rH*) :: (@*nil cmd*).

Definition *talregs* : *list reg* := *rA*::*rB*::*rC*::*rD*::*rE*::*rF*::*rG*::*rH*::(*nil*).

Fixpoint *eqonregs* (*rs*:*list reg*) : *rfile* → *rfile* → *Prop* :=
fun R R' ⇒
match rs with

```
    | (r'::rs') ⇒ (R r') = (R' r') ∧ (eqonregs rs' R R')
    | _ ⇒ True
  end.

Section Proofs.

Variables ec_min ec_size cm_min cm_size : word.

Definition freergn_jmp_req
  (CT:cdspec) RR (PmemA PmemB:mempred) dmmin dmsize
  : Prop :=
  ∃ n, ∃ Q, CT(RR rH) = someT (n,*Q) ∧
    ∀ (M:fmap word word) (MM:mem) RR' fp,
        (appliesto M MM) ∧
        (PmemA && (2 |-> dmmin)) &&
        ((fun Md ⇒ (∀ a, indomf Md a → ~(iscodearea CT a 1))) &
         (fun Md ⇒ (coversfnat _ Md dmmin (dmmin+dmsize))) &
         (PmemB && (dmmin |-> dmsize)
                && ((dmmin+1) |-> fp)
                && (freelist fp)))
            M ∧
        (eqonregs talregs RR RR')
        → Q(MM,RR',RR rH).

Definition freergn_req
  : cdspec → pred → Prop :=
  fun CT Pfree ⇒
    ∀ St, Pfree(St) →
      match St with ((MM,RR),pc) ⇒
        ec_min ≤ pc ∧ (pc + (length freergn_cmds)) ¡ ec_min+ec_size
        ∧
        ∃ M, ∃ PmemA, ∃ PmemB,
          ∃ dmmin, ∃ dmsize, ∃ fp, ∃ rsize, ∃ nxt,
            (appliesto M MM) ∧
            (PmemA && (2 |-> dmmin)) &&
            ((fun Md ⇒ (∀ a, indomf Md a → ~(iscodearea CT a 1))) &
             (fun Md ⇒ (coversfnat _ Md dmmin (dmmin+dmsize))) &
             (PmemB && (freeblock (RR rA) rsize nxt)
                    && (dmmin |-> dmsize)
                    && ((dmmin+1) |-> fp)
                    && (freelist fp)))
                M ∧
            (freergn_jmp_req CT RR PmemA PmemB dmmin dmsize)
```

```
    end.

Lemma coversfnat_fmapupd
  : ∀ B M min size a b,
    coversfnat B M min size →
    indomf M a →
    coversfnat B (fmapupd beq_nat M a b) min size.

Lemma singfp_fmapupdN
  : ∀ B a b' b (M:fmap nat B),
    a |-> b' M →
    a |-> b (fmapupd beq_nat M a b).
```

Proving CAP safety of the library code

```
Lemma freergn_wfcap
  : ∀ CT Pfree
    (D:freergn_req CT Pfree),
    WFCapCmds (MySP ec_min ec_size) CT (Pfree) (freergn_cmds).
```

RgnTAL type for the free region library call

```
Definition freergn_type : omega :=
  tabsr (fun p ⇒
  tabsc (fun c ⇒
  tabscd (uniqcap p) c (fun Djcp ⇒
    tabst (
(* a1 = 5 *)
    tvabst _ (
(* a2 = 4 *)
    tvabst _ (
(* a3 = 3 *)
    tvabst _ (
(* a4 = 2 *)
    tvabst _ (
(* a5 = 1 *)
    tvabst _ (
(* a6 = 0 *)
    tvcode 6 (pluscap (uniqcap p) c Djcp)
             (fun r ⇒
               match r with
                 | r0 ⇒ (tvlift _ (tvlift _ (tvlift _ (tvlift _
                          (tvlift _ (tvlift 0 (tvhandle p)))))))
                 | r1 ⇒ (tvvar 5)
                 | r2 ⇒ (tvlift _ (tvvar 4))
```

```
                    | r3 ⇒ (tvlift _ (tvlift _ (tvvar 3)))
                    | r4 ⇒ (tvlift _ (tvlift _ (tvlift _ (tvvar 2))))
                    | r5 ⇒ (tvlift _ (tvlift _ (tvlift _ (tvlift _ (tvvar 1)))))
                    | r6 ⇒ (tvlift _ (tvlift _ (tvlift _
                              (tvlift _ (tvlift _ (tvvar 0))))))
                    | _ ⇒ (tvabst 6 (
(* a0 = 1 *)
                                tvabst 7 (
(* a7 = 0 *)
                                tvcode 8 (c)
                                        (fun r ⇒
                                          match r with
                                            | r0 ⇒ (tvlift _ (tvlift _ (tvlift _
                                                    (tvlift _ (tvlift _ (tvlift _
                                                      (tvvar 1)))))))
                                            | r1 ⇒ (tvvar 7)
                                            | r2 ⇒ (tvlift _ (tvvar 6))
                                            | r3 ⇒ (tvlift _ (tvlift _ (tvvar 5)))
                                            | r4 ⇒ (tvlift _ (tvlift _
                                                    (tvlift _
                                                      (tvvar 4))))
                                            | r5 ⇒ (tvlift _ (tvlift _
                                                    (tvlift _ (tvlift _
                                                      (tvvar 3)))))
                                            | r6 ⇒ (tvlift _ (tvlift _
                                                    (tvlift _ (tvlift _ (tvlift _
                                                      (tvvar 2))))))
                                            | r7 ⇒ (tvlift _ (tvlift _ (tvlift _
                                                    (tvlift _ (tvlift _ (tvlift _
                                                    (tvlift _ (tvvar 0))))))))
                                          end)
                                )))
                    end))))))))))).

Definition cpinvshort :=
  fun CMLyt CM ⇒
    (cpinv CMLyt cm_min cm_size CM).

Definition cpinv' :=
  fun CMLyt CM t (cmds:cmdlist) ⇒
    fun (St:state) ⇒
      let (p,pc) := St in let (MM,RR) := p in
```

(*ec_min* ≤ *pc*) ∧
(*pc* + *length cmds* ¡ *ec_min* + *ec_size*) ∧
(*cpinvshort CMLyt CM t St*).

Definition *iscpgen'* :=
fun EC CMLyt CT CM ⇒
(*iscpgen EC CMLyt ec_min ec_size cm_min cm_size CT CM*).

Lemmas

Lemma *trdatamem_fmapdel_trheap*
: ∀ *DLyt RLyt CMLyt dm_min dm_size DM M g h*,
fmaplook DM g h →
tr_datamem DLyt RLyt CMLyt DM dm_min dm_size M →
(*'ex fun dmlim* ⇒ *'ex fun fp* ⇒
(*fun M* ⇒ *coversfnat _ M dm_min* (*dm_min*+*dm_size*)) &
'(*limitf _ DM dmlim*) &
((*dm_min* |−> *dm_size*)
&& ((*dm_min*+1) |−> *fp*)
&& (*freelist fp*)
&& (*tr_datamem_aux* (*fmapdelN _ DLyt g*)
RLyt CMLyt dmlim (*fmapdelN _ DM g*))
&& (*tr_heap DLyt RLyt CMLyt g h*)
)) *M*.

Lemma *tr_hvs_ptrtoanylist*
: ∀ *DLyt RLyt CMLyt hplim g a h M*,
tr_hvs DLyt RLyt CMLyt g a hplim h M →
ptrtoanylist a (*hvlist_size hplim h*) *M*.

Lemma *ptrtoanylist_eqf*
: ∀ *n a M1 M2*,
eqf M1 M2 →
ptrtoanylist a n M1 →
ptrtoanylist a n M2.

Lemma *ptrtoanylist_concat*
: ∀ *n a m M M1 M2*,
pappendf M1 M2 M →
ptrtoanylist a n M1 →
ptrtoanylist (*a*+*n*) *m M2* →
ptrtoanylist a (*n*+*m*) *M*.

Lemma *tr_hvs_plus_ptrtoanylist*

```
  : ∀ DLyt RLyt CMLyt M M1 M2 g a hplim h n rsize,
      (hvlist_size hplim h) + n = rsize →
      pappendf M1 M2 M →
      tr_hvs DLyt RLyt CMLyt g (a + 3) hplim h M2 →
      ptrtoanylist (a + 3 + hvlist_size hplim h) n M1 →
      ptrtoanylist (a + 3) rsize M.

Lemma memtype_tpair
  : ∀ CM M MT p l t,
      wf_datamem CM M MT →
      memtypeof MT p l t →
      ∃ t1, ∃ t2, t = (tpair t1 t2 p).

Lemma memtype_gc_wf_wordval
  : ∀ CM M MT v t g,
      wf_datamem CM M MT →
      wf_wordval CM MT v t →
      wf_wordval CM (fmapdelN _ MT g) v t.

Lemma memtype_gc_wf_regfile
  : ∀ CM M MT RF G g,
      wf_datamem CM M MT →
      wf_regfile CM MT RF G →
      wf_regfile CM (fmapdelN _ MT g) RF G.

Lemma memtype_gc_sat_cap_memtype
  : ∀ MT A g c D,
      eqcap A (pluscap (uniqcap g) c D) →
      sat_cap_memtype MT A →
      sat_cap_memtype (fmapdelN _ MT g) c.

Lemma tr_wordval_fmapdelheaplyt
  : ∀ DLyt RLyt CMLyt v w g,
      tr_wordval DLyt RLyt CMLyt v w →
      tr_wordval (fmapdelN heaplyt DLyt g) RLyt CMLyt v w.

Lemma tr_regfile_fmapdelheaplyt_eqregs
  : ∀ DLyt RLyt CMLyt RF RR RR' g,
      tr_regfile DLyt RLyt CMLyt RF RR →
      eqonregs talregs RR RR' →
      tr_regfile (fmapdelN heaplyt DLyt g) RLyt CMLyt RF RR'.

Fixpoint typevarsof (v:wordval) : list constr → list constr :=
  fun ts ⇒
    match v with
```

```
    | wappr w p ⇒ typevarsof w ((c_rgn p)::ts)
    | wappc w c ⇒ typevarsof w ((c_cap c)::ts)
    | wappcd c1 c2 w Dj ⇒ typevarsof w ((c_disj _ _ Dj)::ts)
    | wappt w t ⇒ typevarsof w ((c_type t)::ts)
    | _ ⇒ ts
  end.

Lemma memtype_not_codetype
  : ∀ CM M MT p l t,
    wf_datamem CM M MT →
    memtypeof MT p l t →
    ¬iscodetype t.

Lemma instcodetype_cons_r
  : ∀ t ts r t' F,
    instcodetype t ts t' →
    eqtype t' (tabsr F) →
    instcodetype t (ts ++ (c_rgn r::nil)) (F r).

Lemma instcodetype_cons_c
  : ∀ t ts c t' F,
    instcodetype t ts t' →
    eqtype t' (tabsc F) →
    instcodetype t (ts ++ (c_cap c::nil)) (F c).

Lemma instcodetype_cons_cb
  : ∀ t ts c A t' F,
    instcodetype t ts t' →
    subcap c A →
    eqtype t' (tabscb F A) →
    instcodetype t (ts ++ (c_cap c::nil)) (F c).

Lemma instcodetype_cons_t
  : ∀ t t' ts o t'' F,
    instcodetype t ts t' →
    eqtype t' (tabst F) →
    (unliftV0 (substV F (lifttoV o))) = t'' →
    instcodetype t (ts ++ (c_type o::nil)) t''.

Lemma typevarsof_app
  : ∀ v t ts,
    typevarsof v (t :: ts) = (typevarsof v nil) ++ (t :: ts).

Lemma instcodetype_trans
  : ∀ T ts t,
```

instcodetype T ts t →
∀ ts′ t′,
instcodetype t ts′ t′ →
instcodetype T (ts ++ ts′) t′.

Lemma *wf_wordval_codetype_inst*
: *∀ CM M MT v t (Dwf:wf_datamem CM M MT),*
wf_wordval CM MT v t →
iscodetype t →
∃ f, ∃ cv, ∃ cvt,
stripabs(v) = Some f ∧
fmaplook CM f cv ∧
instcodetype (codevaltype cv) (typevarsof v nil) cvt ∧
eqtype cvt t.

(Finally, prove that the RgnTAL type satisfies the pre- and post-condition requirements for the* `freergn` *library function *)*

Definition *freergn_pred*
: *heaplyt → codemem → pred*
:= *fun CMLyt CM St ⇒ (cpinv′ CMLyt CM freergn_type freergn_cmds St).*

Theorem *freergn_type_req*
: *∀ CT EC CMLyt CM (Dx:extcode_in_ec EC ec_min ec_size),*
iscpgen′ EC CMLyt CT CM →
freergn_req CT (freergn_pred CMLyt CM).

End *Proofs*.

Appendix C

Computing Fibonacci Numbers in RgnTAL

In this chapter, I give an example program written in RgnTAL to demonstrate the language and the use of the runtime library functions. The program will be one to compute the n^{th} pair of Fibonacci numbers. In ML, the code would look like the following:

```
fun fib 0 (a,b) = (a,b)
  | fib n (a,b) = fib (n-1) (b, a+b);
```

This function recurses on an integer parameter `n` and a pair of Fibonacci numbers `(a,b)`. This high-level code hides the details of how memory for the pairs is allocated and deallocated. A version of the same program at the C level of abstraction would be:

```
pair* fib(int n, pair* f /* in p */, rgnblock* p) {
  if (n == 0) return f;
  int a = f.snd;
  int b = f.fst + f.snd;
  rgnblock* p' = newrgn();
  freergn(p);
  pair* f' = alloc(p', a, b);
  return fib(n-1, f', p');
}
```

This code uses the data structures and associated operations defined in Figure 5.14 to create and delete regions and allocate data. Note the overlapping lifetimes of regions

p and p' and also the fact that the region pointers are explicitly passed along function calls. Of course, we are interested in a version of this fib code compiled to a well-typed RgnTAL program.

For our RgnTAL version, we first define the interfaces for the region library primitives:

```
freergn =
  stub[ρ, ϵ, α1, α2, α3, α4, α5, α6]
      ( {ρ^1} ⊕ ϵ,
        { r0 : ρ handle, r1 : α1, ..., r6 : α6,
          r7 :∀[α0, α7](ϵ, {r0 : α0, ..., r[7] : α7})
        }).∅

newrgn =
  stub[ϵ, α0, α1, α2, α3, α4, α5, α6]
      ( ϵ,
        {r0 : α0, ..., r6 : α6,
          r7 :∀[ρ, α7]({ρ^1} ⊕ ϵ, {r0 : ρ handle, r1 : α1, ..., r7 : α7})
        }).∅


alloc =
  stub[ρ, ϵ, ϵ0 ≤{ρ^+} ⊕ ϵ, α3, α4, α5, α6]
      ( ϵ0,
       {r0 : ρ handle, r1 : int, r2 : int, r3 : α3, ..., r6 : α6,
       r7:∀[α2, α7](ϵ0, {r0:ρ handle, r1:⟨int × int⟩ at ρ, r2 : α2, ..., r7 : α7})
        }).∅
```

Recall, the syntax for RgnTAL stub values is stub $[\Delta](A, \Gamma).\emptyset$ where Δ is a list of constructor variables, parameterizing over region names, capabilities, or types; A is the set of capabilities that the code needs in order to execute; and Γ is the type that must be satisfied by the register file. freergn above requires a unique capability for a given region, ρ, and also a handle to that region in register r0. Since the capability for that region is unique, the type system can ensure that there are no other region variables or names active in the program that alias ρ – thus it will be safe to free it and remove that capability from the set of capabilities ϵ in the continuation (in register r7).

newrgn can be invoked with any current capability set ϵ and a continuation pointer in register r7 that expects a new, unique region to be added to ϵ. The alloc function requires a capability set ϵ_0 in which the region ρ is accessible, whether it is aliased or not

```
halt_type =
  ∀[ρ, ϵ, α2, α3, α4, α5, α6, α7]
    ({ρ^1} ⊕ ϵ, { r0 : ⟨int × int⟩ at ρ, r1 : ρ handle, r2 : α2, ..., r7 : α7 })


fib_entry =
 code[ρ, ϵ, α3, α4, α5, α6]
     ({ρ^1} ⊕ ϵ,
      { r0 : int, r1 : ⟨int × int⟩ at ρ, r2 : ρ handle, r3 : α3, ..., r6 : α6, r7 :halt_type })
   bgti r0, 0, fib_loop_addr[ρ, ϵ, α3, α4, α5, α6]
   mov r0, r1
   mov r1, r2
   jmp r7[ρ, ϵ, ρ handle, α3, α4, α5, α6, halt_type]



fib_loop =
 code[ρ, ϵ, α3, α4, α5, α6]
     ({ρ^1} ⊕ ϵ,
      { r0 : int, r1 : ⟨int × int⟩ at ρ, r2 : ρ handle, r3 : α3, ..., r6 : α6, r7 :halt_type })
   subi r0, r0, 1                 //  n--
   ld r3, r1, 1                   //  r3 = f.snd
   ld r4, r1, 0                   //  r4 = f.fst
   add r4, r4, r3                 //  r4 = f.fst + f.snd
   mov r5, r0                     //  r5 = n  --- save n
   mov r6, r7                     //  save TAL ret ptr
   movf r7, fib_free_addr
   tapp r7 [ρ]
   tapp r7 [ϵ]
   jd newrgn_addr[{ρ^1} ⊕ ϵ, int, ⟨int × int⟩ at ρ, ρ handle, int, int, int, halt_type]



fib_free =
 code[ρ, ϵ, ρ0, α7]
     ({ρ0^1} ⊕ ({ρ^1} ⊕ ϵ),
      { r0 : ρ0 handle, r1 : ⟨int × int⟩ at ρ, r2 : ρ handle, r3 : int, r4 : int, r5 : int,
        r6 :halt_type, r7 : α7 })
   mov r1, r0                    // r1 = ρ0 handle --- save it
   mov r0, r2                    // r0 = ρ handle --- to be freed
   movf r7, fib_alloc_addr
   tapp r7 [ρ0]
   tapp r7 [ϵ]
   tapp r7 [ρ handle]            // will be unusable afterwards
   jd freergn_addr[ρ, {ρ0^1} ⊕ ϵ, ρ0 handle, ρ handle, int, int, int, halt_type]
```

Figure C.1: RgnTAL `fib` function blocks (1 of 2).

```
fib_alloc =
 code[ρ, ϵ, α2, α0, α7]
     ({ρ1} ⊕ ϵ,
      {r0 : α0, r1 : ρ handle, r2 : α2, r3 : int, r4 : int, r5 : int, r6 :halt_type, r7 : α7})
   mov r0, r1                    // r0 = ρ handle
   mov r1, r3                    // r1 = f.snd
   mov r2, r4                    // r2 = f.fst + f.snd
   mov r7, fib_ret_addr
   tapp r7 [ρ]
   tapp r7 [ϵ]
   tapp r7 [int]
   tapp r7 [int]
   jd alloc_addr[ρ, ϵ, {ρ1} ⊕ ϵ, int, int, int, halt_type]


fib_ret =
 code[ρ, ϵ, α3, α4, α2, α7]
     ({ρ1} ⊕ ϵ,
      {r0 : ρ handle, r1 : ⟨int × int⟩ at ρ, r2 : α2, r3 : α3, r4 : α4, r5 : int,
        r6 :halt_type, r7 : α7})
   mov r2, r0                    // r2 = ρ handle
   mov r0, r5                    // restore n to r0
   mov r7, r6                    // restore TAL ret ptr
   jd fib_entry_addr[ρ, ϵ, α3, α4, int, halt_type]


halt =
 code[ρ, ϵ, α2, α3, α4, α5, α6, α7]
     ({ρ1} ⊕ ϵ,
      {r0 : ⟨int × int⟩ at ρ, r1 : ρ handle, r2 : α2, ..., r7 : α7})
   jd halt_addr[ρ, ϵ, α2, α3, α4, α5, α6, α7]
```

Figure C.2: RgnTAL `fib` function blocks (2 of 2).

```
Definition fib_loop_type : omega :=
  tabsr (fun p =>                    (* \rho *)
  tabsc (fun c =>                    (* \epsilon *)
  tabscd (uniqcap p) c (fun Djcp =>  (* p and c disjoint *)
    tabst (                          (* \alpha_3 *)
      (tvabst _                      (* \alpha_4 *)
      (tvabst _                      (* \alpha_5 *)
      (tvabst _                      (* \alpha_6 *)
        (tvcode 4 ((uniqcap p) (+) c \ Djcp)
             (fun r => match r with
                 | r0 => tvlift4 (tvint)
                 | r1 => tvpair _ (tvlift4 (tvint)) (tvlift4 (tvint)) p
                 | r2 => tvlift4 (tvhandle p)
                 | r3 => V 3
                 | r4 => tvlift (V 2)
                 | r5 => tvlift2 (V 1)
                 | r6 => tvlift3 (V 0)
                 | r7 =>
                   (tvabsr _ (fun p' =>
                   (tvabsc _ (fun c' =>
                   (tvabscd _ (uniqcap p') c' (fun Djcp' =>
                   (tvabst _          (* \alpha_2 *)
                   (tvabst _          (* \alpha_7 *)
                     (tvcode 6 ((uniqcap p') (+) c' \ Djcp')
                       (fun r =>
                         match r with
                           | r0 => tvpair _ (tvlift6 tvint) (tvlift6 tvint) p'
                           | r1 => tvlift6 (tvhandle p')
                           | r2 => tvlift4 (V 1)
                           | r3 => tvlift6 tvint
                           | r4 => tvlift6 tvint
                           | r5 => tvlift6 tvint
                           | r6 => tvlift6 tvint
                           | r7 => tvlift5 (V 0) end))))))))))) end))))))))).

Definition fib_loop_cmds : ciseq :=
  ciabsr (fun p => ciabsc (fun c => ciabsd (uniqcap p) c (fun Djcp =>
  ciabst (fun a3 => ciabst (fun a4 => ciabst (fun a5 => ciabst (fun a6 =>
  cibase (icons (isubi r0 r0 1)         (* n-- *)
         (icons (ild r3 r1 1)           (* r3 = f.snd *)
         (icons (ild r4 r1 0)           (* r4 = f.fst *)
         (icons (iadd r4 r4 r3)         (* r4 = f.fst + f.snd *)
         (icons (imov r5 r0)            (* r5 = n    // save n *)
         (icons (imov r6 r7)            (* save TAL retptr *)
         (icons (imovf r7 fib_free_addr)
         (icons (iappr r7 p)
         (icons (iappc r7 c)
         (icons (iappcd _ _ r7 Djcp)
                (ijd newrgn_addr))))))))))))))))))).

Definition fib_loop_cv : codeval := cvcode fib_loop_type fib_loop_cmds.
```

Figure C.3: RgnTAL `fib_loop` block in Coq

– hence the use of the bounded quantification. `alloc` will create a new, initialized (using the values of r1 and r2) pair of data in that region and pass a pointer to the new pair on to the continuation pointer.

We can now write the main program. Because RgnTAL programs are written in continuation-passing style (CPS), the `fib` function is actually split into several blocks, each of which perform some statements and then call one of the region library primitives, passing along a continuation to the next block. The RgnTAL code is listed in Figures C.1 and C.2. `fib_entry` tests if the parameter `n` (stored in r0) is zero, jumping to the `halt` continuation pointer if so. The `halt` code expects a pointer to the final pair of Fibonacci numbers in r0 and a unique handle to the region in r1.

If the base case has not been reached, the `fib_entry` code branches to the body of a recursive loop, `fib_loop`, which decrements `n`, loads the current Fibonacci pair into registers and computes the next pair, and then makes a call to the library function, `newrgn`, passing the address of the next block, `fib_free`, as a continuation.

`fib_free` prepares to free the region in which the old pair of Fibonacci numbers is stored. Notice that both regions are accessible in the capability of `fib_free`. The continuation passed to the `freergn` library call is the next block, `fib_alloc`, which prepares for the allocation and initialization of a new pair in the newly acquired region. Having allocated a pair, `fib_ret` restores the data of registers that had been shuffled around and jumps back to the main entry point, `fib_entry`.

This code, along with the CAP implementations of the region library primitives, has been defined and simulated on my prototype CAP machine. The type checking and certification, however, is only partially complete at the time of this writing, due to the tedious nature of working with even such moderately complex programs in the Coq proof assistant, as discussed in the main text of the dissertation (*e.g.* Sections 6.1–6.3). To illustrate, the Coq definition of the `fib_loop` code block is given in Figure C.3. This uses the syntax encoding for RgnTAL types and code blocks defined in Figures 5.3 and 5.4, and the deBruijn representation of types with free variables defined on page 143.

Bibliography

[1] A. D. Gordon and T. Melham. Five axioms of alpha-conversion. In J. Von Wright, J. Grundy, and J. Harrison, editors, *Proceedings 9th International Conference on Theorem Proving in Higher Order Logics TPHOLs '96*, volume 1125 of *LNCS*, pages 173–190. Springer-Verlag, 1996.

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

[3] A. W. Appel. Foundational proof-carrying code. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–258, June 2001.

[4] A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proceedings 27th ACM Symposium on Principles of Programming Languages*, pages 243–253. ACM Press, 2000.

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

[6] A. W. Appel, N. G. Michael, A. Stump, , and R. Virga. A trustworthy proof checker. In *Journal of Automated Reasoning*, volume 31, pages 231–260, 2003.

[7] B. Barras. *Auto-validation d'un systéme de preuves avec familles inductives*. PhD thesis, Université Paris 7, 1999.

[8] G. Barthe, P. Courtieu, G. Dufay, and S. Sousa. Tool-assisted specification and verification of the JavaCard platform. In *Proceedings 9th International Conference on Algebraic Methodology and Software Technology (AMAST '02)*, volume 2422 of *LNCS*, pages 41–59. Springer-Verlag, 2002.

[9] A. Bernard and P. Lee. Temporal logic for proof-carrying code. In A. Voronkov, editor, *Proceedings of 18th International Conference on Automated Deduction (CADE)*, volume 2392 of *LNCS*, pages 31–46. Springer-Verlag, 2002.

[10] L. Birkedal, N. Torp-Smith, and J. C. Reynolds. Local reasoning about a copying garbage collector. In *Proceedings 31st ACM Symposium on Principles of Programming Languages*, pages 220–231. ACM Press, 2004.

[11] R. S. Boyer and Y. Yu. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM*, 43(1):166–192, Jan. 1996.

[12] R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 23–50. Edinburgh University Press, Edinburgh, Scotland, 1972.

[13] C. Calcagno, S. Ishtiaq, and P. W. O'Hearn. Semantic analysis of pointer aliasing, allocation and disposal in Hoare logic. In M. Gabbrielli and F. Pfenning, editors, *Proceedings 2nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'00)*, pages 190–201. ACM Press, 2000.

[14] J. Chen, D. Wu, A. W. Appel, and H. Fang. A provably sound TAL for back-end optimization. In *Proceedings 2003 ACM Conference on Programming Language Design and Implementationn*, pages 208–219. ACM Press, 2003.

[15] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. In *Proceedings 2000 ACM Conference on Programming Language Design and Implementation*, pages 95–107. ACM Press, 2000.

[16] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[17] T. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings Colog'88*, volume 417 of *LNCS*. Springer-Verlag, 1990.

[18] K. Crary. Toward a foundational typed assembly language. In *Proceedings 30th ACM Symposium on Principles of Programming Languages*, pages 198–211. ACM Press, Jan. 2003.

[19] K. Crary and S. Sarkar. A metalogical approach to foundational certified code. Technical Report CMU-CS-03-108, School of Computer Science, Carnegie Mellon University, Pittsburg, PA, Jan. 2003.

[20] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings 26th ACM Symposium on Principles of Programming Languages*, pages 262–275. ACM Press, 1999.

[21] F.-N. Demers and J. Malenfant. Reflection in Logic, Functional and Object-Oriented Programming: a Short Comparative Study. In *Workshop on Reflection and Metalevel Architectures and their Applications in AI (IJCAI'95)*, pages 29–38, August 1995.

[22] J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-order abstract syntax in Coq. In *Proceedings 1995 International Conference on Typed Lambda Calculi and Applications (TLCA '95)*, volume 902 of *LNCS*, pages 124–138. Springer-Verlag, Apr. 1995.

[23] J. Despeyroux and A. Hirschowitz. Higher-order syntax and induction in Coq. In *Proceedings 5th International Conference on Logic Programming and Automated Reasoning (LPAR '94)*, volume 822 of *LNAI*, pages 159–173. Springer-Verlag, July 1994.

[24] J. Despeyroux, F. Pfenning, and C. Schurmann. Primitive recursion for higher-order

abstract syntax. In *Proceedings 3rd International Conference on Typed Lambda Calculi and Applications (TLCA '97)*, LNCS, pages 147–163. Springer-Verlag, 1997.

[25] A. Felty. Semantic models of types and machine instructions for proof-carrying code. Talk presented at 2000 PCC Workshop, June 2000.

[26] A. P. Felty. Two-level meta-reasoning in Coq. In *Proceedings 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs '02)*, volume 2410 of *LNCS*, pages 198–213. Springer-Verlag, Aug. 2002.

[27] J.-C. Filliâtre. The WHY certification tool, tutorial and reference manual. `http://why.lri.fr/`, July 2002.

[28] J.-C. Filliâtre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4):709–745, 2003.

[29] R. W. Floyd. Assigning meanings to programs. In A. M. Society, editor, *Proceedings of the Symposium on Applied Math. Vol. 19*, pages 19–31, 1967.

[30] G. Gillard. A formalization of a concurrent object calculus up to alpha-conversion. In D. A. McAllester, editor, *Proceedings 17th International Conference on Automated Deduction (CADE)*, volume 1831 of *LNCS*, pages 417–432. Springer-Verlag, 2000.

[31] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proceedings 2002 ACM Conference on Programming Language Design and Implementationn*, pages 282–293. ACM Press, 2002.

[32] N. A. Hamid and Z. Shao. Interfacing hoare logic and type systems for foundational proof-carrying code. In *Proceedings 17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3223 of *LNCS*, pages 118–135. Springer-Verlag, Sept. 2004.

[33] N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach

to foundational proof carrying-code. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*, pages 89–100. IEEE Computer Society, July 2002.

[34] N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof carrying-code. *Journal of Automated Reasoning (Special issue on Proof-Carrying Code)*, 31(3-4):191–229, Dec. 2003.

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

[36] M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Safe and flexible memory management in Cyclone. Technical Report CS-TR-4514, University of Maryland, College Park, MD, July 2003.

[37] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, Oct. 1969.

[38] F. Honsell, M. Miculan, and I. Scagnetto. An axiomatic approach to metareasoning on systems in higher-order abstract syntax. In *Proceedings of ICALP'01*, volume 2076 of *LNCS*, pages 963–978. Springer-Verlag, 2001.

[39] F. Honsell, M. Miculan, and I. Scagnetto. The theory of contexts for first order and higher order abstract syntax. In M. Lenisa and M. Miculan, editors, *Electronic Notes in Theoretical Computer Science*, volume 62. Elsevier, 2002.

[40] W. A. Howard. The formulae-as-types notion of constructions. In *To H.B.Curry: Essays on Computational Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

[41] G. Huet. Residual theory in λ-calculus: a formal development. *Journal of Functional Programming*, 4(3):371–394, 1994.

[42] S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Proceedings 28th ACM Symposium on Principles of Programming Languages*, pages 14–26. ACM Press, 2001.

[43] T. Jim, G. Morrisett, D. Grossman, and M. Hicks. Cyclone: A safe dialect of C. In *Usenix Annual Technical Conference*, June 2002.

[44] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[45] C. League, Z. Shao, and V. Trifonov. Precision in practice: A type-preserving Java compiler. In *Proceedings 12th International Conference on Compiler Construction*, volume 2622 of *LNCS*, pages 106–120. Springer-Verlag, Apr. 2003.

[46] J. McCarthy. Towards a mathematical theory of computation. In *Proc. IFIP Congress 62*, pages 21–28. North-Holland, 1963.

[47] J. McKinna. *Deliverables: a categorical approach to program development in type theory*. PhD thesis, University of Edinburgh, UK, 1992.

[48] N. Michael and A. Appel. Machine instruction syntax and semantics in higher order logic. In *Proceedings 17th International Conference on Automated Deduction*, volume 1831 of *LNCS*, pages 7–24. Springer-Verlag, June 2000.

[49] M. Miculan. Developing (meta)theory of lambda-calculus in the theory of contexts. Technical Report TR 2001/26, University of Leicester, Siena, 2001.

[50] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: a realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, May 1999.

[51] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. In X. Leroy and A. Ohori, editors, *Proceedings 1998 International Workshop on Types in Compilation*, volume 1473 of *LNCS*, pages 28–52. Springer-Verlag, March 1998.

[52] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *Proceedings 25th ACM Symposium on Principles of Programming Languages*, pages 85–97. ACM Press, Jan. 1998.

[53] NASA. Mars exploration rover mission press releases. http://marsrovers.jpl.nasa.gov, 2004.

[54] G. C. Necula. Proof-carrying code. In *Proceedings 24th ACM Symposium on Principles of Programming Languages*, pages 106–119. ACM Press, Jan. 1997.

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

[56] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings 2nd USENIX Symp. on Operating System Design and Impl.*, pages 229–243, 1996.

[57] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proceedings 1998 ACM Conference on Programming Language Design and Implementation*, pages 333–344. ACM Press, 1998.

[58] G. C. Necula and P. Lee. Safe, untrusted agents using proof-carrying code. In G. Vigna, editor, *Special Issue on Mobile Agent Security*, volume 1419 of *LNCS*, pages 61–91. Springer-Verlag, Mar. 1998.

[59] G. C. Necula and R. R. Schneck. Proof-carrying code with untrusted proof rules. In M. Okada, B. C. Pierce, A. Scedrov, H. Tokuda, and A. Yonezawa, editors, *Software Security – Theories and Systems, Mext-NSF-JSPS International Symposium (ISSS)*, volume 2609 of *LNCS*, pages 283–298. Springer-Verlag, 2003.

[60] G. C. Necula and R. R. Schneck. A sound framework for untrusted verification-condition generators. In *Proceedings 18th Annual IEEE Symposium on Logic in Computer Science (LICS'03)*, pages 248–260, June 2003.

[61] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *Computer Science Logic*, volume 2142 of *LNCS*, pages 1–19. Springer-Verlag, 2001.

[62] C. Paulin-Mohring. Inductive definitions in the system Coq—rules and properties. In M. Bezem and J. Groote, editors, *Proceedings TLCA*, volume 664 of *LNCS*. Springer-Verlag, 1993.

[63] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings 1988 ACM Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, 1988.

[64] F. Pfenning and C. Paulin-Mohring. Inductively defined types in the Calculus of Constructions. In *Proceedings 5th Conference on the Mathematical Foundations of Programming Semantics*, volume 442 of *LNCS*, pages 209–228. Springer-Verlag, 1990. technical report CMU-CS-89-209.

[65] F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In *Proc. 16th International Conference on Automated Deduction*, volume 1632 of *LNCS*, pages 202–206. Springer-Verlag, July 1999.

[66] J. C. Reynolds. Lectures on reasoning about shared mutable data structures. IFIP Working Group 2.3 School/Seminar on State-of-the-Art Program Design Using Logic, Tandil, Argentina, September 6-13, 2000.

[67] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*, pages 55–74. IEEE Computer Society, July 2002.

[68] I. Scagnetto. *Reasoning about names in Higher-Order Abstract Syntax*. PhD thesis, Università degli Studi di Udine, Udine, Mar. 2002.

[69] R. R. Schneck. *Extensible Untrusted Code Verification*. PhD thesis, University of California, Berkeley, May 2004.

[70] R. R. Schneck and G. C. Necula. A gradual approach to a more trustworthy, yet scalable, proof-carrying code. In A. Voronkov, editor, *Proceedings of 18th International Conference on Automated Deduction (CADE)*, volume 2392 of *LNCS*, pages 47–62. Springer-Verlag, 2002.

[71] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, Feb. 2000.

[72] C. Schurmann, J. Despeyroux, and F. Pfenning. Primitive recursion for higher-order abstract syntax. *Theoretical Computer Science*, 266(1-2):1–57, 2001.

[73] Z. Shao. An overview of the FLINT/ML compiler. In *Proceedings 1997 ACM SIGPLAN Workshop on Types in Compilation*, June 1997.

[74] Z. Shao, B. Saha, V. Trifonov, and N. Papaspyrou. A type system for certified binaries. In *Proceedings 29th ACM Symposium on Principles of Programming Languages*, pages 217–232. ACM Press, Jan. 2002.

[75] F. Smith, D. Walker, and G. Morrisett. Alias types. In *Proceedings 9th European Symposium on Programming (ESOP 2000)*, volume 1782 of *LNCS*, pages 366–381. Springer-Verlag, June 2000.

[76] K. N. Swadi and A. W. Appel. Typed machine language and its semantics. Preliminary version available at `www.cs.princeton.edu/~appel/papers/tml.pdf`, July 2001.

[77] G. Tan, A. W. Appel, K. N. Swadi, and D. Wu. Construction of a semantic model for a typed assembly language. In *Proceedings 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '04)*, volume 2937 of *LNCS*, pages 30–43. Springer-Verlag, Jan. 2004.

[78] The Coq Development Team. The Coq proof assistant reference manual. Coq release v7.1, 2001.

[79] The Coq Development Team. The Coq proof assistant reference manual. Coq release v8.0, 2004.

[80] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Proceedings 21st ACM Symposium on Principles of Programming Languages*, pages 188–201. ACM Press, 1994.

[81] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[82] V. Trifonov, B. Saha, and Z. Shao. Fully reflexive intensional type analysis. In *Proceedings 2000 ACM International Conference on Functional Programming*, pages 82–93. ACM Press, Sept. 2000.

[83] J. C. Vanderwaart and K. Crary. A typed interface for garbage collection. In *Proceedings 2003 International workshop on Types in Languages Design and Implementation (TLDI '03)*, pages 109–122. ACM Press, 2003.

[84] D. Walker. *Typed Memory Management*. PhD thesis, Cornell University, Ithaca, NY, Jan. 2001.

[85] D. Walker, K. Crary, and G. Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, 2000.

[86] D. Walker and G. Morrisett. Alias types for recursive data structures. In R. Harper, editor, *Proceedings 3rd Workshop on Types in Compilation (TIC 2000)*, volume 2071 of *LNCS*, pages 177–206. Springer-Verlag, June 2001.

[87] T. R. Weiss. Out-of-memory problem caused Mars rover's glitch. Computerworld. http://www.computerworld.com, February 4 2004.

[88] B. Werner. *Une Théorie des Constructions Inductives*. PhD thesis, A L'Université Paris 7, Paris, France, 1994.

[89] M. Wildmoser and T. Nipkow. Certifying machine code safety: Shallow versus deep embedding. In *Proceedings 17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3223 of *LNCS*, pages 305–320. Springer-Verlag, Sept. 2004.

[90] M. Wildmoser, T. Nipkow, G. Klein, and S. Nanz. Prototyping proof carrying code. In *Proceedings 3rd IFIP International Conference on Theoretical Computer Science (TCS 2004)*, 2004.

[91] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings International Workshop on Memory Management (ICMM'92)*, volume 637 of *LNCS*, pages 1–42. Springer-Verlag, 1992.

[92] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings International Workshop on Memory Management*, volume 986 of *LNCS*, pages 1–116. Springer-Verlag, 1995.

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

[94] D. Wu, A. W. Appel, and A. Stump. Foundational proof checkers with small witnesses. In *Proceedings 5th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 264–274, Aug. 2003.

[95] H. Xi and R. Harper. A dependently typed assembly language. In *Proceedings 2001 ACM International Conference on Functional Programming*, pages 169–180. ACM Press, Sept. 2001.

[96] D. Yu. *Safety Verification of Low-Level Code*. PhD thesis, Yale University, New Haven, CT, Dec. 2004.

[97] D. Yu, N. A. Hamid, and Z. Shao. Building certified libraries for PCC: Dynamic storage allocation. In *Proceedings 2003 European Symposium on Programming (ESOP'03)*, volume 2618 of *LNCS*, pages 363–379. Springer-Verlag, Apr. 2003.

[98] D. Yu, N. A. Hamid, and Z. Shao. Building certified libraries for PCC: Dynamic storage allocation. *Science of Computer Programming*, 50(1-3):101–127, 2004.

[99] D. Yu and Z. Shao. Verification of safety properties for concurrent assembly code. In *Proceedings 2004 ACM International Conference on Functional Programming*, September 2004.

[100] Y. Yu. *Automated proofs of object code for a widely used microprocessor*. PhD thesis, University of Texas at Austin, Austin, TX, 1992.