Abstract

Certifying Virtual Memory Manager Using Multiple Abstraction Levels

Alexander Leo Vaynberg

2013

Abstraction is the main tool that makes the creation of complex software systems tractable. However, software verification using the Hoare-logic approach has only a few methods in its arsenal to make use of abstraction, with separation logic being the most popular. Combined with recursive predicates, separation logic can be used for defining abstract data types (ADTs) and information hiding in verification. But there are limits to its application: separation logic cannot be used to define abstractions that rely on the alteration of operational semantics of primitive operations. One example of such abstraction is virtual memory: in the concrete model, any memory access goes through address translation, but in abstract model, memory accesses operate on a virtual data store that hides a particular address translation. Other examples of non-ADT abstractions include compilation, transactions, and time-sharing. Currently, these abstractions can not be implemented by current Hoare-logic verification techniques.

In our thesis, we present an alternative approach to handling abstraction in software verification that allows us to define the abstractions not expressible in separation logic. Instead of using a single machine model and a complex logic for verification, our new software verification framework makes use of multiple machine models and common static semantics expressed in simple logic. For every module of a complex software system, our framework enables us to define a new machine model with the abstract primitives natural for that module, thereby simplifying the verification of code. These machine models are then connected to each other by abstraction relations, from which our framework generates the refinements. Using these refinements, we link the modules verified at different levels of abstraction. The final result is that though we define and specify software modules at their natural levels of abstraction, we still get the proof that all modules linked together are sound with respect to the most concrete machine model. In other words, our framework merges the techniques of abstract machines and refinement with Hoare-logic style verification.

To show that this approach is effective, we have used our framework to deal with the virtual address space abstraction. For a large portion of the OS kernel, the abstract address space model is more natural than the concrete address translation model, making it preferable to certify the kernel using address space primitives. However, the virtual memory manager, a complicated and error-prone part of the OS, relies on the address translation primitives to operate, and therefore can only be verified using the concrete model. Instead of using the address translation model to verify the entire kernel, which would make the verification more complicated, we use our framework to link the virtual memory manager and the rest of the kernel, both verified in their natural models. The verified linking guarantees that the entire kernel will execute correctly on the machine with address translation. Using Coq Proof Assistant, we have machine-checked the proofs of both the framework and the certification of address space abstraction to ensure correctness.

The work presented in the thesis raises the state-of-art in the field of formal software verification frameworks. Our framework is both entirely language independent and can handle self-modifying code without relying on separation logic. However, in our opinion, the main contribution of this work lies in modularity and reusability of certified code. Software verification tends to be all-or-nothing deal. To be reused in a different context verified modules require new specifications, and thus new proofs. Our framework enables their refinement into the new context without re-proving them. It is our hope that our work will spur the development of reusable certified libraries and creation of ever larger certified software systems.

Certifying Virtual Memory Manager Using Multiple Abstraction Levels

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

by
Alexander Leo Vaynberg

Dissertation Director: Zhong Shao

May 2013

Contents

List of Figures

Acknowledgements

I would like to thank my advisor Zhong Shao for his guidance throughout my graduate career. He has always pushed me to get actual results and has never let me get away with imprecision and speculation. This work would never be as detailed and as rigorous without his assistance.

I would like to thank my thesis readers, Bryan Ford, Paul Hudak, and Xinyu Feng for reading my dissertation and providing thoughtful comments. I am also grateful to the anonymous reviewers of my submitted papers for their comments that improved my work.

I am indebted to Xinyu Feng, Hongxu Cai, and Wei Wang for helping me find the direction in my research. I also owe many thanks to Rodrigo Ferreira, Antonis Stampoulis, and other members of FLINT research team for numerous informative conversations. They have never refused any request for help, and have always set an example for me and my work.

This dissertation is dedicated to my parents, Leo and Natalya, and to my wife Olga. They have always given me their support and encouragement, enabling me to do my best.

This research is based on work supported in part by DARPA grants FA8750-10-2-0254 and FA8750-12-2-0293, ONR grant N000141210478, and NSF grants 0910670 and 1065451. 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

Writing software is a complex engineering task, in many ways more difficult than other forms of engineering, as the number of ways software can fail is not limited to several well-known modes of failure. Even the smallest error can lead to catastrophic results such as complete failure of USS Yorktown[45]. Failure to account for all possibilities can result in software being exploited[11, 30]. Testing software is a way to reduce the number of such errors, but it can not eliminate them completely.

For these reasons, the field of software certification evolved, taking several related paths including type theory, model-checking, and Hoare-logic[20]. Although all of those approaches are related, the research presented in this thesis is closer to the Hoare-logic approaches. The idea behind Hoare logic is that for any programming language or a machine, we can construct a model of it. This model allows to formally analyze any program running on the machine. The analysis allows us to formally show that a program has a specific behavior (called a specification) that it is logically guaranteed to follow. This approach to reasoning about the programs has yielded several frameworks for analyzing program safety and correctness such as proof-carrying code[34, 33, 3] and certified software[48, 13, 35], the latter actually being closer to the spirit of the original Hoare-logic, aiming at reduction of the number of axioms and at the simplification of the correctness proofs. The prior research on certified software has followed the ideas of foundational PCC, and has constructed frameworks for verification of real programs running on real computers, mostly focusing on the semantics of assembly languages[48, 14, 6]. Using these frameworks, our group can, at least in theory, certify all kinds of software.

However, formal certification of software using our approach is done with great difficulty, and usually on small examples. Our colleagues, using slightly different approaches, have verified larger amounts of code, but not without spending a lot more effort, while still having problems of their own. The most famous verification project, L4.verified[10, 47, 23], has managed to verify a microkernel[27] using a top down approach, where the kernel is first defined with high-level specifications, and then these specifications are refined down to more precise logical objects, which are then proven to correspond to actual data structures maintained by the code. Their approach relies on parts of their implementation, such as memory allocator and initialization, to be trusted. They have also avoided abstracting some of the complicated features of the OS, such as virtual memory, passing these towards the user programs to handle safely. The Verisoft project[40, 1] has also verified a large portion of the kernel. Their approach involves defining a machine that represents the instruction set architecture, and a machine that represents the high-level semantics from the point of view of the user program. The verification of the kernel is a large proof that the semantics of the high-level machine are correctly simulated by the underlying machine and the microkernel running on top of it. This proof has taken many years of work to develop. Our worry about their method is that, unless they have taken great care in constructing their proof, their proof may be brittle. There is no formal definition of modularity within their microkernel, and thus a change in one definition may cause changes throughout the entire simulation proof.

Although formal software certification is already useable, its use has not gained traction outside the closely-knit research community. Real-world programmers consider complete formal certification to be too difficult, not worth the cost, and therefore impractical. Although formal verification will never be as simple as mind-checking, since explicitly writing down specifications and proofs takes more work than quick consideration, if the verification became easier and more natural, the benefits would begin to outweigh the costs, and more programmers will start verifying their software.

There are several reasons why formal verification is too hard. First, the current state of formal verification forces the programmer to think about the program in a way that does not correspond to the programmer's own understanding of the program, as the formal model of the behavior of the language may not correspond to how the programmer imagines it. Second, there is a danger that verification proofs are not fully modular or reusable. It is extremely frustrating to have to rework verification proofs for some module just because of a small change in another module that alters some global invariant that should have been abstracted away. And third, the programmers rely on already crafted and debugged libraries when building their programs, which greatly speeds up their development. However, there are no such reusable libraries of verified code, meaning that programmers have to do verification from scratch.

All these problems are especially visible when verifying systems software, such as operating system kernels. As they have large codebases, they are written as a set of fairly independent modules, carefully separated, each considered at a separate layer of abstraction, so that it can be understood by a person. It is impossible for any person to keep track of all the invariants present in a modern kernel. Thus if there is any hope of verifying a complete kernel, then the verification itself must be modular and reusable, so that all modifications can be isolated and contained, and not create the need to re-verify everything.

What makes the abstraction in OS kernels particularly challenging is the fact that many abstractions modify the way that the kernel seems to operate - that is they seemingly alter the operational semantics of the machine. Examples of such abstractions include time-sharing, interrupt handling, and virtual memory, just to name a few. In this thesis, we focus on the problem of virtual memory abstraction.

As the OS kernel is written in C, it is natural to think that the programmer uses a C model of computation to reason about the programs. This is true, but not completely. The C model of computation does not have a single definition of how the memory is accessed. As we are dealing with OS kernels, we should assume that the memory works as hardware defines it. This means that every memory access may go through a complicated process called address translation, where an address is broken into parts, and each part as an index into page tables, which are program-controlled data tables that define how the address is translated. However, this is not how much of the kernel is reasoned about. The address translation system of the hardware is very detailed, and only a small portion of the kernel, namely the virtual memory manager, will deal with all the details of the address translation. That portion of the kernel will then provide a less-detailed and more intuitive interface to the rest of the kernel, called the address space model. The address space model hides away all the complicated address translation, instead presenting the memory as though it was a single store indexed by the virtual addresses. The programmer can then forget about the details of address translation, and only consider the memory accesses. Thus, reasoning about the software using the address space model is not only much simpler than trying to reason about address translation, but also is independent of the hardware-dependent address translation mechanism. Thus the abstract model of memory makes reasoning more machine-independent and modular.

What this means is that the programmer reasons with the address translation model when thinking about the virtual memory manager, and the address space model when thinking about the other parts of the kernel. This presents a problem for PCC approaches, as they are defined only for one machine model. If we try to verify the kernel on the machine model that uses address spaces - then we can not certify the virtual memory manager module. If the model is address translation, then everything but VMM becomes harder to certify.

To combat similar problems, quite a bit research focused on making PCC be expressive enough to allow some abstraction within the confines of a single machine model, by using advanced approaches involving separation logic[41] and frame rules[38] to define high-level views of the machine model, such as first-class abstract data types[39]. Separation logic allows for a huge improvement in ability to reason modularly and abstractly, but it does have its limits. Its main power lies in its ability to reason about the state of the model fragmented into separate and disjoint pieces without having the need to know where each piece belongs. This approach essentially defines a model where a sub-state (a partial state) is also a valid state. The frame rule shows that a program valid under a strong specification (one that uses a very small partial state) is also valid with a larger state.

Unfortunately, separation logic is not adequate in this case - it can not be used to define the virtual address space abstraction. A more powerful, and more complex Mapped Separation Logic[24] can express some of the high-level properties of virtual memory, as it defines a separation over mappings. However, this is a specialized solution for reasoning over heaps with virtual addresses. It does not allow one to reason about the code that updates the hardware-specific address translation tables, as these can not be linked to the mappings on which the logic relies. Nor can it be used to reason about the initialization of virtual memory, when no such mappings are present.

We do not want to dismiss or belittle separation logic, and, in fact, we think it is great. However, we do want to bring attention to the fact that it does not give us an ability to perform the kinds of abstraction needed. Neither is it completely natural to use - it forces the machine model into a special form where it can be described as pieces, which might not be a natural way to think about the behavior of the machine, and thus it will require more effort from the programmer to formalize his reasoning.

In this thesis, we develop a new framework for certifying code that allows the use complex abstractions. The new framework works by allowing the creation of separate machine models for each code module, each machine model providing the appropriate level of abstraction to make the certification of the code module simple. The framework also includes a method by which the machine models can be related. The related machine models automatically define a refinement rule by which the certified modules can be safely linked across the abstraction boundary, resulting in a proof of soundness of the entire software system.

For the example of virtual memory, we could define two machine models: the address translation model to verify the VMM, and the address space model to verify the rest of the kernel. Then our framework would help us define a refinement that will convert the abstract certification of the high-level kernel into the certification of that same code over the address translation model. Furthermore, our framework will allow us to conclude that the high-level kernel will safely link with the certified VMM code.

Currently the field of formal software verification does not have a formal framework for verifying software using multiple abstract machine models. One of our own lines of work, OCAP[12], has created a way to reason about separate modules using separate logics. For example, one module can be certified using typed assembly language[32, 31], the other using XCAP[36], and it is possible to show that the two will work together. However, this work does not allow different machine models, but only different logics that analyze code using the same machine model.

There are several works that use multiple machine models in verification of the software. However, these machine models were defined and linked in an ad-hoc way, no general framework for doing this is explored or defined. One of these works is a certified garbage collector by McCreight *et al.*[29] Although its approaches have served as an inspiration for this thesis, the work itself is limited; the two machines used in the certification of the GC use the same operational semantics - only the state is abstracted. The proofs used apply only to the particular code and machine models, making the approach not reusable for other certified software. It also can not handle calls from the concrete layer into the abstract layer, meaning that it can not handle initialization, which is an important and tricky component of the software. The other example of multi-machine verification is the already mentioned Verisoft project, which links its machine models using a simulation argument. It also does not try to establish a general approach to multi-model verification, and is limited to proving linkages between the particular models used in the project.

In our thesis, we will show that our multi-machine verification framework is an improvement on the ad-hoc examples above. First, it makes the definition of machine models a faster task by allowing machine templates, and by providing common static semantics that are sound for all machines. Second, the framework provides a way to link code certified using multiple machine models in a simpler way, with smaller proof obligations than are needed by ad-hoc approaches. Third, unlike other software certification projects that make use of multi-machine verification, our framework allows for general use of upcalls, e.g. when a module certified using a more concrete machine makes calls into the more abstract module. The upcalls allow us to cleanly reason about initialization code, without the need for special cases in the relations between machines.

This new verification framework forms the first half of this thesis. The second half is devoted to applying this framework to completely verify a small virtual memory manager using multiple abstract machines, showing that our approach is effective at simplifying the verification of code that is difficult to handle traditional certification approaches. It also hints at how our framework can be used to verify other complex problems present in OS kernel verification.

1.1 Contributions

In this thesis, we have developed a new approach to software verification by allowing the complicated software to be verified not only in a modular fashion, but also in a way that each module of software can be verified using its own abstract view of the world. More specifically we have made the following contributions to the field of software verification.

1. We have developed a new framework for verification of software. This framework is novel in that it is completely machine-independent. This means that it can be instantiated with a model of any machine that works as a state-transition system, and the framework will provide a way to create specifications to the software for these machines, and to provide a way to check that the software does indeed conform to these specifications. The framework is proven sound and guarantees partial correctness (can be extended to total correctness) for any machine

definition.

2. The framework is an improvement over its predecessors, SCAP[15] and GCAP[6]. It supports both stack-based control workflow and self-modifying code, without requiring complex data structures and predicates to keep track of stack and code modification.

3. The definition of operational semantics in our framework does not use inductive definitions, but instead is a data structure presented as a mapping from operations to specifications. This means that a machine definition can be manipulated. In our example with kernels, there are two C machines, each with its own memory model. Because the machine definition is a data structure - we can define our C machine parametric over the memory model, and thus quickly generate C machines, reusing many definitions.

4. Our framework provides a formal definition of the refinement between machines. Because the verification system is the same for all machines, we can create various useful refinements from just a few properties about the machines. For example, we have shown that a representation refinement (seen in work on certified garbage collection) is one of refinements definable in our system. We have also extended the representation refinement with invariants. Such addition allows us to use the refinement for frame rules and information hiding, which until now required the use of separation logic.

5. We have applied our framework to verify a virtual memory manager. We have shown how to split the VMM into multiple modules, verify each one in its own natural level of abstraction, and then link them together into one completely verified code. We have also shown that any kernel verified over the abstract address space model will correctly link with our virtual memory, guaranteeing safety over the actual model of C with address translated model of memory. This result is one of the first real verification results toward the verification of CertiKOS[17], a secure hypervisor kernel.

6. We have formalized our entire framework, including refinements, in Coq proof assistant. This allows us to have confidence in the soundness and correctness of our framework.

1.2 Thesis Outline

The rest of this thesis will be structured as follows. In Chapter 2, we will give a non-technical overview of our framework and how it can be used. Then, we will proceed with a full technical presentation of the precise definition of abstract machine, and how the verification framework uses them in Chapter 3. Chapter 4 is devoted entirely to linking and refinements. This is where we give a technical definition of what a refinement is, as well as define automatic ways of creating these refinements. Here we also try to argue that refinements are both simple and powerful enough to be used in place of separation logic. Then, in Chapter 5, we move away from the theoretical contributions, and instead focus on the certification of a small virtual memory manager designed for simplified hardware. We show how we can extend the certification of the virtual memory manager to be more realistic and practical in Chapter 6. Chapter 7 is the guide to the Coq implementation of the framework and the verification of the VMM. Then we give a quick overview of related works and conclude in Chapter 8.

Chapter 2

Overview

One of the most important features of the hardware is that it provides a mechanism of address translation. This feature allows the operating systems to have an indirect addressing scheme for its data, which in turn enables the kernel to provide memory protection as well as various other features. A simplified example of such address translation mechanism is shown in Figure 2.1. This model of the hardware (and its more realistic counterparts) use a portion of the memory, selected by control registers, to keep an address translation table. When address translation is turned on, all memory accesses, e.g. load and stores, are defined in terms of virtual addresses, which the hardware translates to physical addresses before carrying out the operation.

This somewhat complicated model is designed with two goals in mind: to be easy to implement in hardware, and to be as general as possible, so that software is unrestricted by the design choices of the hardware. The downside of this model is that it pushes the complexity to the software,

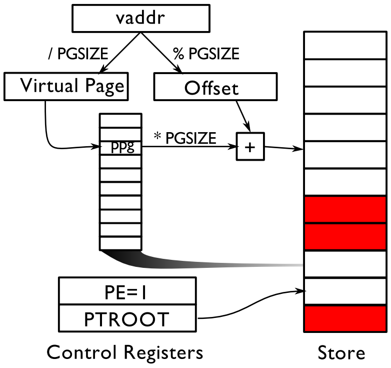
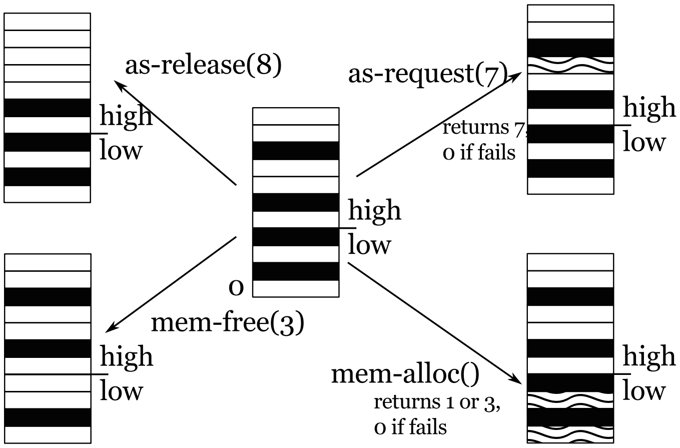


Figure 2.1: Address Translation



```
uint64_t mem_alloc(); // allocate some page in the low memory area
void mem_free(uint64_t page); // free an allocated page in the low memory area
uint64_t as_request(uint64_t page); // allocate the specific page in the high memory area
void as_release(uint64_t page); // release a page allocated by as_request
```

Figure 2.2: Example of API of the Virtual Memory Manager

which must now properly control and protect the translation tables, and to isolate the complexity of managing these data structures.

To handle this task, a typical OS kernel is split into two parts: the low level portions, to which we refer as the *virtual memory manager*, and the rest of the kernel, which we refer to as the *high-level kernel*. The virtual memory manager is aware of the specific details of the hardware's address translation model, and is responsible for managing the translation tables and the control registers. Then it abstracts away these details behind a simpler abstract interface for managing the indirect addresses, which it presents to the high-level kernel.

The high-level interface of the virtual memory manager is specific to the particular operating system, and will be different depending on the needs of the kernel. However, for most kernels, the abstract interface has a common feature: it makes virtual addresses appear as though they are actual memory cells. This is usually complemented by some API for managing these virtual areas. An example of such an abstract interface and its API can be seen in Figure 2.2. The high-level kernel no longer needs to know the specifics of the hardware, and manages the indirect addresses only through the abstract interface.

The abstract interface allows the programmer to have a machine-independent, abstract model of the virtual address space in mind when designing the high-level kernel. Reasoning with this model is much simpler as the programmer no longer has to consider the translation mechanism, nor the memory requires to contain the virtual memory, nor the possibility of modifying the translation tables, expect through a well-defined interface. This also has an added benefit that the kernel becomes machine independent, and given different hardware address translation implementations, will work on any of them, as long as a suitable implementation of virtual memory manager is provided.

When we try to formally certify an operating system kernel, we want to make use of this abstraction. After all, if the abstract memory model makes it easier for the programmer to mentally reason about the high-level kernel, it should be easier to formally specify and prove it, as well. However, if we try to make use of separation logic and information hiding to define such an abstraction, we will discover that we can not do it.

Consider the operational semantics of the memory store on a machine with address translation.

$$\{\{pl \rightsquigarrow ?\}\} \ \texttt{*l:=v} \ \{\{pl \rightsquigarrow v\}\}$$
$$\text{where } pl = \begin{cases} R(PTROOT) + Pg(l) * 8 & \text{if } R(PE) = 1 \\ l & \text{otherwise} \end{cases}$$

The specification requires us to follow the entire hardware translation mechanism to get the state that results from performing the store, which is independent of the particular implementation of the virtual memory manager. Suppose now that we have implemented a virtual memory manager, and we are trying to simplify the verification of stores within the high-level code. The natural approach would be to create an abstract data type for the page tables and allocation tables that the virtual memory manager needs. Then we could try to define a lemma that modifies the specifications of the store to make use of the abstracted translation data. The result would be something like the following specification:

$$\begin{array}{l} \exists pl, AT, PM \\ \{ValidAllocation(AT) \wedge Allocated(AT, Pg(pl)) * ValidPM(PM) \wedge Trans(PM, l, pl) * \{pl \rightsquigarrow ?\}\} \\ \texttt{*l:=v} \\ \{ValidAllocation(AT) * ValidPM(PM) * \{pl \rightsquigarrow v\}\} \end{array}$$

This specification is definitely more abstract, as it managed to hide the details of the translation mechanism, as well as exact locations of the page tables. However, we are still stuck with having to carry all the page tables and allocation tables in all the specifications, as every memory access is going to require it. This means that the high-level kernel is still aware of the underlying physical memory, and that to actually reason about memory operations in the high-level kernel, we still have to translate addresses and access the physical location. No matter how hard we try, we would not be able to use separation logic to create an abstract data type that gives us the abstract model of address spaces that we are aiming for, and therefore, we can never achieve the ideal form of the specification of a store in the presence of virtual memory:

$$\{l \leadsto ?\}\ \texttt{*l:=v}\ \{l \leadsto v\}$$

The easiest way to achieve the above semantics is to simply define new semantics that incorporate virtual memory and the API as a primitive. Then we can use these new semantics to verify the high-level kernel, and use the original semantics to verify the virtual memory manager. However, we will then face the challenge of how to connect the two pieces. It is for this purpose we have created a multi-machine verification framework that we are about to describe.

2.1 Multi-Machine Verification of the Virtual Memory Manager

The goal of this thesis is to show how to verify a large software system, by breaking it up into modules, verifying each module on the abstract machine that is ideal for it, and then linking all the verified modules together in a way that guarantees the soundness of the whole system. We will demonstrate this technique on our particular implementation of a virtual memory manager that is used by some high-level kernel the details of which are not important. A diagram of an implementation of such a software system can be seen in Figure 2.3, which shows the functions of the virtual memory manager, how they interact, and how they provide an API for the high-level kernel.

In the previous section, we have seen that the verification of the high-level kernel can be simplified if we were to analyze it against a higher-level machine model, one that assumes virtual memory and its API to be a primitive. When we do this, we get the design diagram in Figure 2.4. The new design shows a new machine model (the green block labeled *Abstract AS Model*) over which the kernel is certified. This new model is connected to the code of the virtual memory managers by

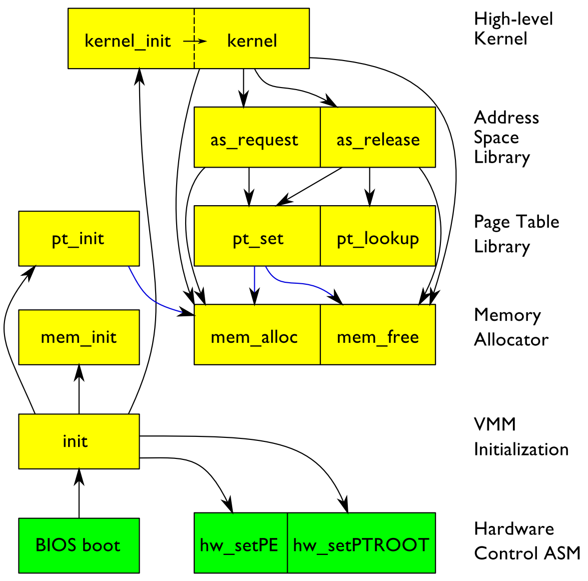


Figure 2.3: Diagram of Kernel Code

lines with circles on the end, indicating that the particular model feature is being implemented by code at the lower level of abstraction.

In a sense, what we are advocating is a meta-linguistic approach to abstraction and verification. This means that we create abstractions by defining a new language (machine model) with new operational semantics that exactly defines the abstraction in use. Such an abstraction mechanism is extremely powerful, as there is no real limitation on what the abstract language can be. Furthermore, it is a "hard" form of abstraction, meaning that there is no way to write a specification or create code that breaks the boundary of the semantics of the language. This hard abstraction guards against abstraction leaks that can be a common source of bugs in complex systems.

Although the separate verification of components in abstract machines may be easier, it is worthless if we can not guarantee the safety of the entire system when it is linked together. To link the two components, we use the notion of refinement, diagrammed in Figure 2.5. The programmer defines a relation between the abstract model and the actual hardware model, from which our framework creates a translation function that will refine the kernel, meaning that it will convert the proof of the correctness of the kernel from the abstract model to the proof of correctness of the kernel over the concrete model. However, this proof will be dependent on the compatibility of the VMM implementation, which the programmer must show. This compatibility proof is constructed from the

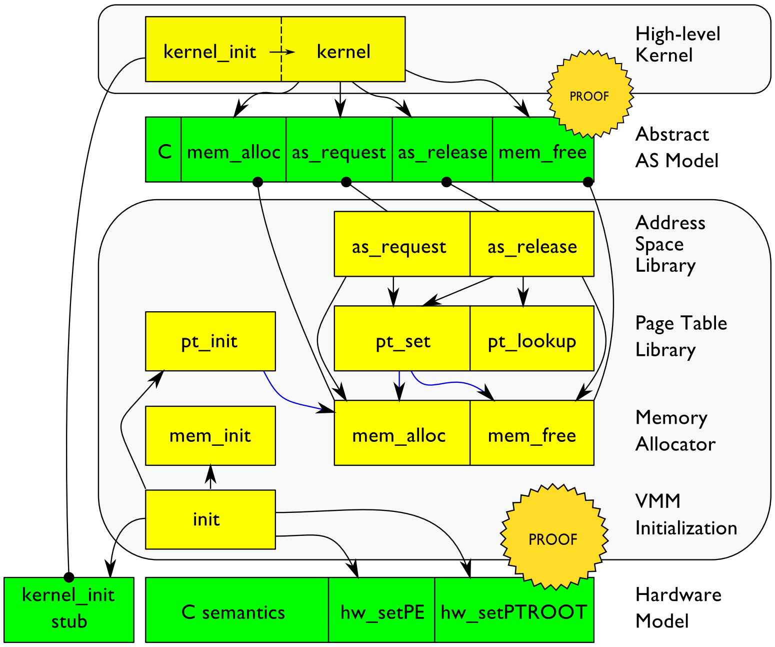


Figure 2.4: Multi-Machine Verification of the Kernel

fact that the VMM implementation correctly implements the primitives of the abstract machine, e.g. those lines with circles that we have seen in our kernel verification plan. Thus we get a proof of correctness of the high-level kernel and the VMM implementation linked together, meaning that they are safe to execute on the actual hardware.

However, there is one problem: we have not shown that the init can safely call the kernel. Our refinement approach to multi-machine verification handles such call (which we refer to as "upcall") as shown in Figure 2.6. In the most general sense, it is just a compatibility proof that works backwards. Instead of showing that the refined abstract primitive is compatible with the actual implementation, we show that the refined specification of the actual code is compatible with the specification with which the upcall was certified. By showing this, we guarantee that the when the kernel is refined to the hardware model, the refined specification of kernel-init will be exactly what init expects.

Our approach to defining the multi-machine verification using refinement has an additional benefit: it is chainable. This means we can use more than two machines, but layer the abstraction as we see fit. Thus, we can design the certification as a series of gradual refinements, each one being relatively easy to define. In fact, our virtual memory implementation is complex enough that it

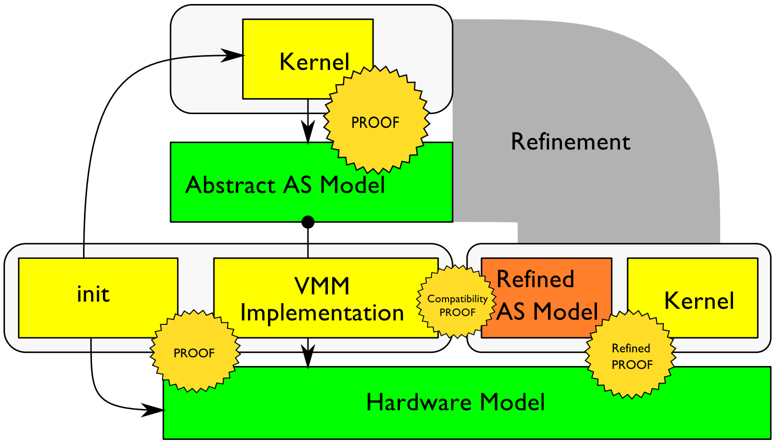


Figure 2.5: The Workings of Refinement

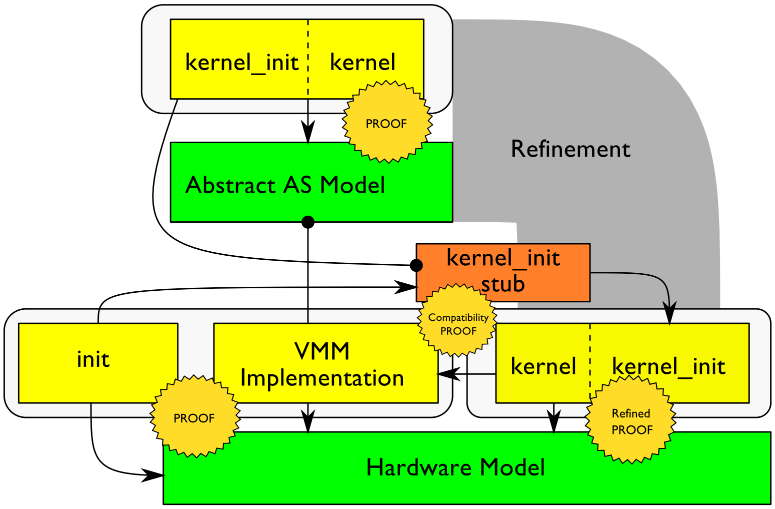


Figure 2.6: The Workings of Refinement (fixed for upcall)

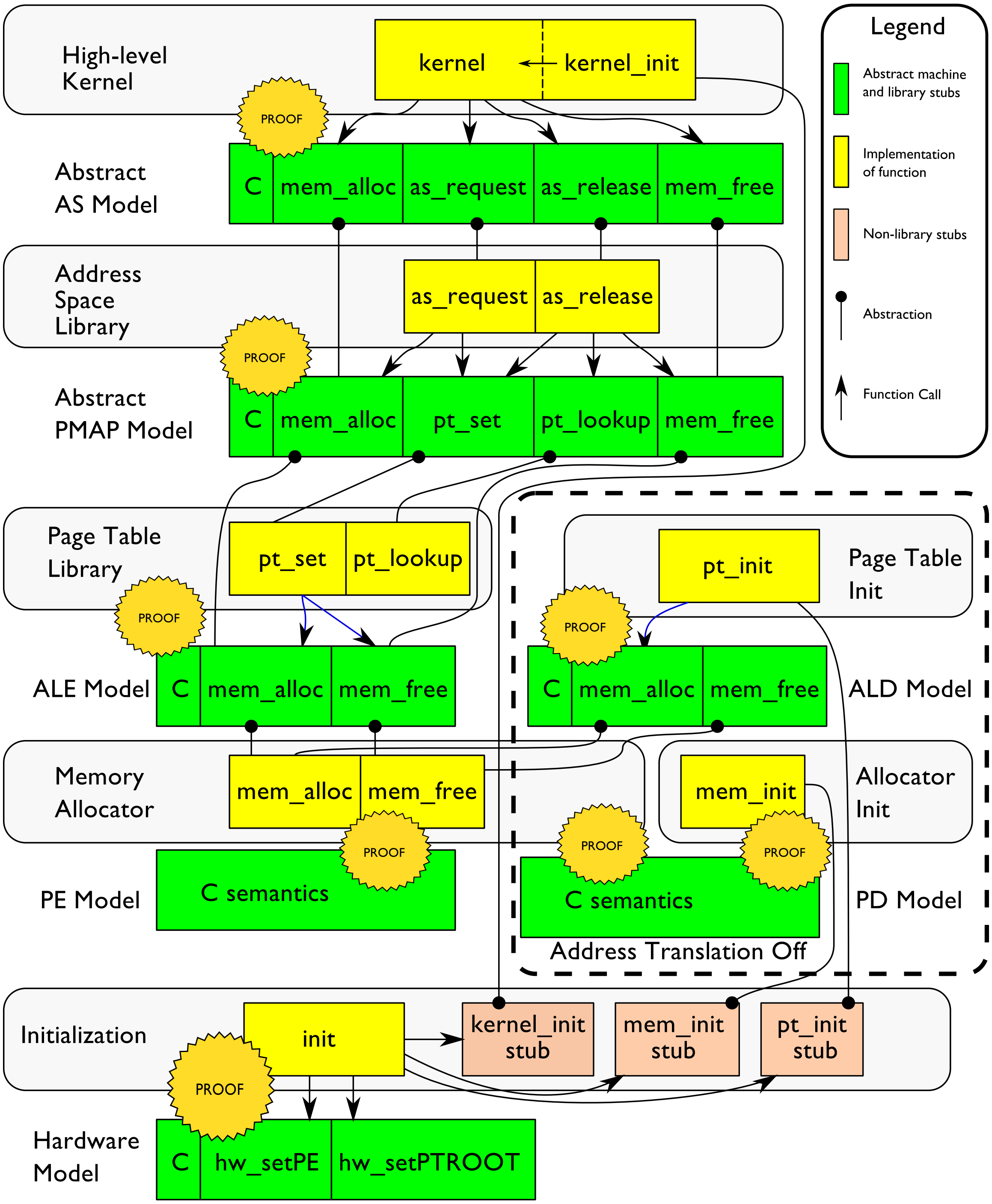


Figure 2.7: Complete Plan for VMM Certification

becomes beneficial to split the verification into several abstract machines, resulting our complete verification plan shown in Figure 2.7, which is the final plan that we will use to verify our virtual memory implementation.

At this point we have explained the high-level plan for multi-machine verification of our kernel. In the next section, we will give an overview of the framework that makes our approach work.

2.2 The Framework for Multi-Layer Verification

The multi-machine approach to abstraction, in its naive implementation, suffers from several complications. First, the programmer must define numerous machines. Each machine must come with a sound verification system, meaning that the programmer must define static semantics and prove the soundness and correctness of the semantics. Second, the refinement and linking of code written in different machines, when done naively, ends up as an ad-hoc affair handled by meta-logical proof that connects the two static semantics. It is no wonder that most verification attempts tend to rely on a single machine model, enhanced by separation logic.

One of the goals of this thesis is to develop a framework that enables the multi-machine verification without suffering the amount of work that the naive approach seems to require. There are two problems that we are trying to solve. One is to allow the programmer to quickly define the machines that are used for verification, and the other is to allow easier definitions of refinements between machines. Our framework answers both issues.

First, our framework creates a machine-independent static semantics for any language that can be represented as a state transition system. This is done by creating a meta-language that takes a definition of a machine as a parameter. The definition supplies our meta-language with type of state and the set of operations that modify the state, with the meta-language supplying all the control flow primitives such as sequence, call, and branch. When the meta-language is instantiated with a particular machine model, the meta-language becomes a complete language, which can be used to create and analyze programs written for that machine. Because the control flow operations are defined independently from the particular machine definition, we can provide a single set of semantics for our meta-language, and prove that the semantics are sound and correct for any machine with which our meta-language is instantiated.

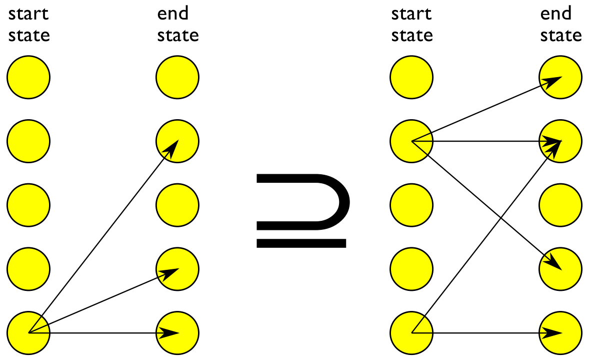


Figure 2.8: Graphical Representation of an Action

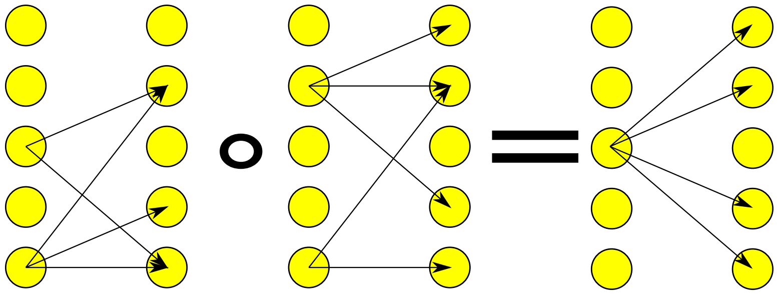


Figure 2.9: Action Concatenation

Thus a machine definition is just a set of states (of an arbitrary type), and a set of transitions between states, which we call *actions*. An example of two actions (related by a weaker-than relation) are shown graphically in Figure 2.8. The actions are the key feature of our framework. They are used to define the operational semantics of a particular machine. The effect of executing a program can be defined as an action, which consists of chaining of actions of individual operations of the program. Finally, the specifications are also actions.

An action means the following: if a state is in the domain of an action, that means that this state is safe with respect to this action - the computer executing this action from this state will not crash. The co-domain of an action is the set of possible states, one of which will be the one that machine will reach once the action completes.

Our framework defines several key operations over actions. For example, the same Figure 2.8 shows the meaning of the action on the left being weaker than the one on the right. This relation is critical for specifications, as a weaker action is a valid specification for the stronger one. Informally, one action is weaker than the other when it defines fewer valid starting states, and may result in more final states. In the case of specifications, this can be seen clearly - the specifications can not allow additional starting states that the program can not handle, and the specifications may indicate that certain final states can occur, while the program may not be capable of producing these states. Other operations on actions, such as action composition, e.g. safely executing one action after another (Figure 2.9), and a choice/branching operation (not shown graphically) are also defined in our framework.

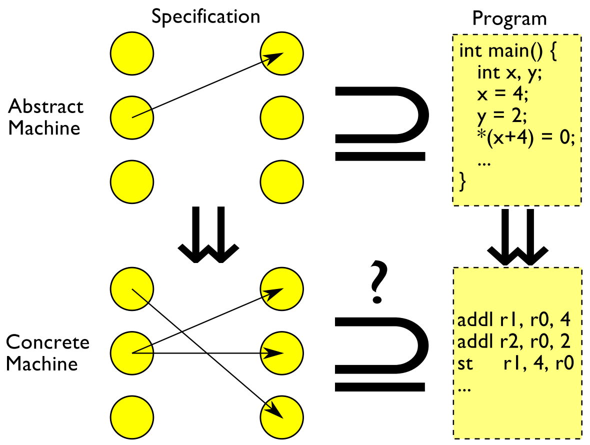
The operational semantics of a machine $\mathcal{M}$ is just a named set of these actions. The names are the instructions or operations of the machine, which we refer to as ι. Thus for example, in our definition of the MIPS machine, $\mathcal{M}_{MIPS}(\texttt{addiu}\; r_d, r_s, w)$ will be an action over the state of the MIPS processor that corresponds to the addition operation. In high-level languages such as C, operations will correspond to higher-level commands, such as variable assignment or setting up a function call.

What is important is that many languages can be expressed as a set of possible atomic transitions in the state space, all of them can be described using our actions and machines. Every single program in such languages can then be converted into a tiny meta-language that we define.

$$\begin{aligned} (\textit{Proc}) \quad & \mathbb{I} \; ::= \texttt{nil} \mid \iota \mid [\texttt{l}] \mid \mathbb{I}_1 ; \mathbb{I}_2 \mid \mathbb{I}_1 + \mathbb{I}_2 \\ (\textit{Proc Heap}) \quad & \mathbb{C} ::= \{\texttt{l} \rightsquigarrow \mathbb{I}\}^* \end{aligned}$$

This language is defined by two things - procedures ($\mathbb{I}$), which are non-looping chains of machine commands and calls into other procedures in the procedure heap ($\mathbb{C}$). This language can be used to define any imperative programs, as it includes composition, choice, and recursion, as well as the ability to include any atomic operation from the definition of the machine. In our thesis, we will use this meta-language to verify programs written in MIPS assembly, IMP (a simple imperative language), and a simplified version of C. Our framework verifies programs by asking the programmer to give specifications to all procedures, and then to prove that these specifications are indeed weaker than the actions of the actual programs. If all procedures in a module have been checked in this way, we call such a module of code a certified module.

The other benefit of using the meta-language is that programs in all languages are verified in the same static semantics. This means that a single proof of soundness of the static semantics of our meta-language guarantees that the program verification is sound for all machines with which the meta-machine can be instantiated. No separate proofs of soundness for different machines are



```
int main() {
  int x, y;
  x = 4;
  y = 2;
  *(x+4) = 0;
  ...
}
```

```
addl r1, r0, 4
addl r2, r0, 2
st    r1, 4, r0
...
```

Figure 2.10: Definition of Refinement

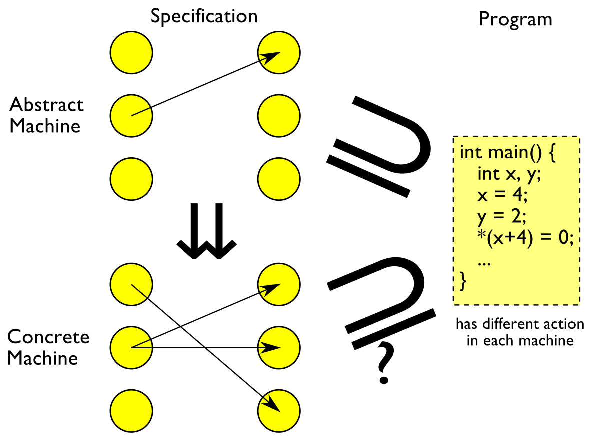
needed.

Thus the programmer benefits by having an ability to define the machines in a simpler way, and by not having to ever redo the soundness and correctness proof for the verification system, no matter which machine is being used. Moreover, we can use this commonality of semantics to assist in the generation of refinements that are needed to support multi-machine verification.

2.3 Refinement within the Framework

Using the meta-language allows us to verify modules in their own layer of abstraction. To connect the separate certified modules that exist at different levels of abstraction, our framework formally defines the notion of refinement. Suppose that we have two machines, the abstract one $\mathcal{M}_A$, and the concrete one $\mathcal{M}_C$. In the most general definition of the refinement, outlined in Figure 2.10, the programmer provides two functions: an action translation function that converts abstract specifications into concrete specifications, and a program translation function to convert an abstract program into a concrete program. These are marked by arrows in the diagram. Then the programmer provides a proof that if the abstract program is certified (abstract specification is weaker than the action of the abstract program) then the concrete program is also certified. Together these functions and the proof establish a refinement.

The diagram shows the refinement being created for a specific set of specifications and programs. However, it is possible to define these functions and proof to work not only for one specific



```
int main() {
   int x, y;
   x = 4;
   y = 2;
   *(x+4) = 0;
   ...
}
```

Figure 2.11: Code-Preserving Refinement

program, but for a large set of programs. In doing so, the refinement becomes general, as it can be reused for any number of programs written in the abstract machine. Such an approach is no longer ad-hoc, since the soundness of the refinement is proven once, rather that once for every program refined. However, creating these general refinements can be a lot of difficult work. As a part of our framework, we have shown several ways to automatically generate these general refinements from simpler definitions by utilizing features of our framework, such as the common meta-language and common static semantics.

2.3.1 Code-Preserving Refinements

When we consider what it would take to refine a high-level kernel from an abstract memory model to the address translated memory model, we can make a very important observation: the code of the program does not change, only the meaning of individual operations does. For example, if a high-level kernel stores a value in memory, then the refined kernel also performs the same state operation. The only difference is that in the abstract semantics, the kernel accesses a virtual page while in concrete semantics, the kernel does an address translation, followed by a store in the relevant physical page. However, the program is still exactly same.

This has several important consequences for the way we define refinements. First, for our VMM verification, since we currently are not aiming at compilation, we can define refinement without the need for relation of abstract and concrete programs. They are completely the same. Thus a general refinement can be defined by defining a function that converts abstract specifications into concrete

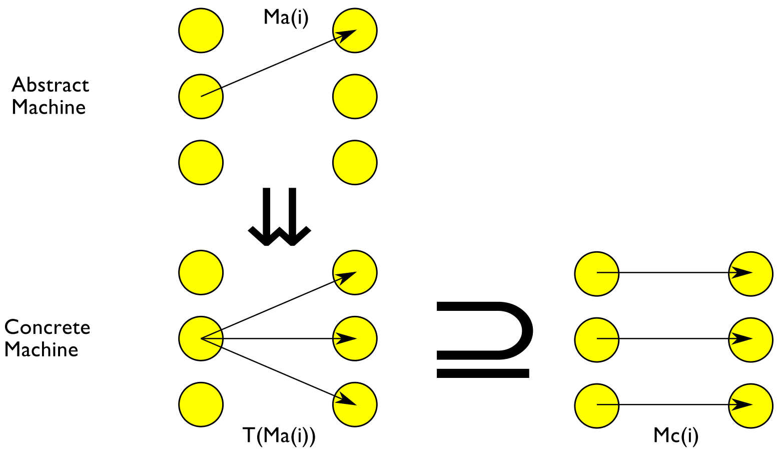


Figure 2.12: Relation Between Actions of Machines

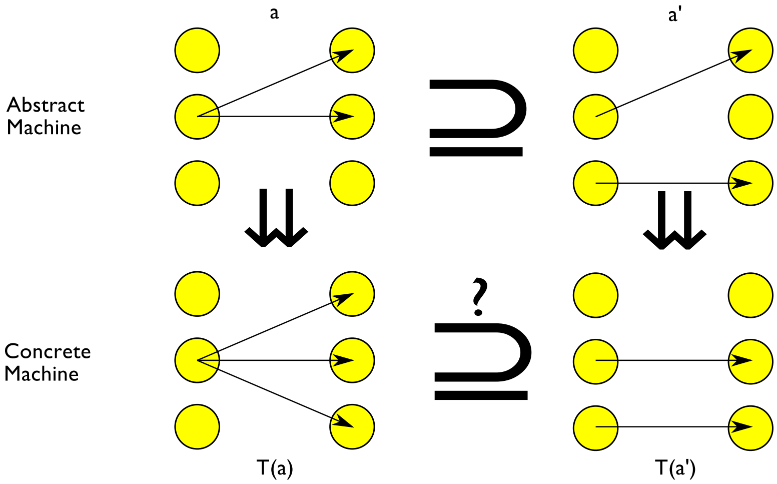


Figure 2.13: Action Translation Preserves Weaker-than Relation

specifications, and a proof that the same program certified with abstract spec in the abstract machine will also be certified with the converted spec in the concrete machine (Figure 2.11).

More so, the fact that the program does not change will allow us to automatically generate the proof as well, as long as a few conditions are met. The first such condition is that the specification conversion function must work for all individual operations in the machine. In other words, if we take the semantics (action) of any operation of the abstract machine, and convert it, the resulting action must be weaker than the action defined by the concrete machine's operational semantics for that same instruction (see Figure 2.12). This guarantees that we can always refine instructions into themselves.

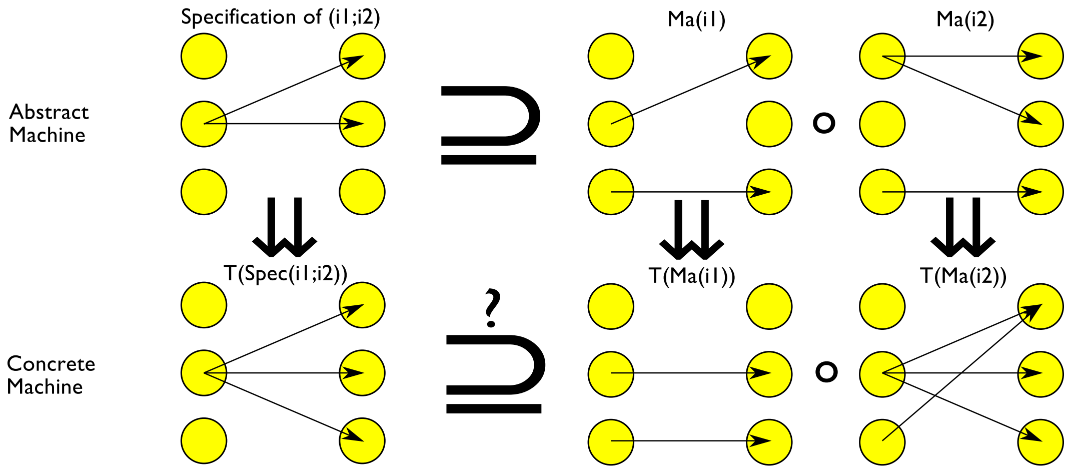


Figure 2.14: Action Translation Preserves Weaker-than Relation for Concatenation

We also need the conversion of actions to have several other properties as well. For example, if we take any two abstract actions where one is weaker than another, then the conversion function must preserve the weaker-than relation between them (see Figure 2.13). Similarly, if we have an action that is weaker than a chaining of two actions, then translated action must be weaker than the chaining of two translated actions (see Figure 2.14). There are a few other properties that the translation function must follow.

What is important is that if we show that an action translation obeys these few properties, then there is a proof that any program certified in the abstract machine is certified under the translated specification in the concrete machine. Thus we have succeeded in defining a refinement by defining a function that translates specifications and showing that it satisfies a few properties; our framework does the rest.

2.3.2 Generating Refinements from Relations of Machine State

So far, we have not assumed that we know anything about the internal structures of the machines, relying only on the fact that they run the same programs, and leaving the definition of the specification translation function to the programmer.

However, suppose we assume that there is a particular representation relation (`repr`) between the abstract and the concrete machine states. In this case, we are able to automatically define a general specification translation function based on this relation. The way we define this function is

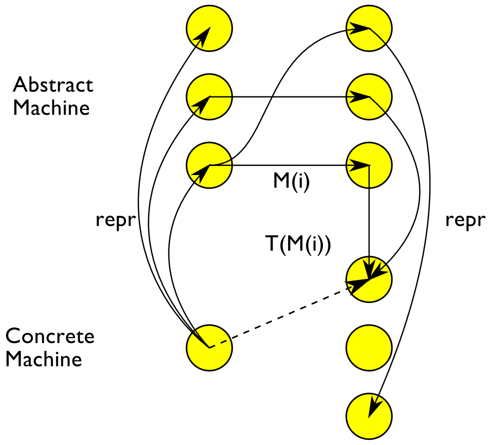


Figure 2.15: Representation-based Specification Translation

similar to the way that McCreight[29] has defined it in his ad-hoc approach to modular verification of the garbage collector. The conversion function creates a concrete action by looking up related abstract states, running the abstract action on it, and returning the intersection of all possible results (Figure 2.15).

It is not important to understand how this conversion function works at this point. What is important is that such a conversion function always obeys all the properties we need for the conversion function to have, except one. We still need to show that for each operation in the machine, if we take the abstract operational semantics for that operation and convert it, the result must be weaker than the action defined by the operational semantics of the concrete machine. Thus, our framework automatically generates refinements that work for any program from the relation between the states of the machines and the proof of the property that the relation between the machines is preserved by every operation.

2.3.3 Other Refinements

In this thesis, we also show several other ways to generate refinements. For example, if we know that an abstract machine is a projection of a concrete machine, we could automatically generate a much simpler action translation function than the very general one produced using `repr`. Another refinement generator we define is an extension of the `repr` with an invariant, that allows us to guarantee that the abstract machine preserves state information that is not contained in the abstract state.

Such a refinement allows us to use our refinements for information hiding, something that was impossible with McCreight's original `repr`. Other ways to generate refinements were imagined, such as refinements that allow a single process program to be embedded into a multiprocessor environment, etc., but these were not fully developed as they were not needed for the VMM verification. Still, our framework is general enough that it should be able to integrate these future refinements as well.

2.4 Conclusion of Overview

In the overview, we have presented the motivation for verifying the virtual memory manager and the high-level kernel that uses virtual memory using multiple abstract machines. We have presented our approach for linking these separately certified pieces, and we defined shown an informal description of the framework that makes it possible. In the rest of the thesis, we will show all the technical details needed for such verification. However, the explanation of the technical detail will start with the framework, build up the machinery needed for verification, and then give the detailed explanation of VMM certification.

We now proceed to the technical content.

Chapter 3

The Verification Framework

Many software verification frameworks, including our own previous work [15, 6] are practical for purposes of software verification. However, these frameworks are always tailored to the specific machine or a language. As the aim of our work is to enable verification of software at multiple abstraction layers, we need a verification system capable of handling multiple machine models. We accomplish this by making the verification framework parametric over the definitions of machines.

3.1 General Machine Definitions and Actions

There are two things that any machine or language has: some notion of context or state, and a set of operations over such state. In other words, a machine can be defined as data structure, which is similar to how finite state machines[44] or abstract state machines[18] are usually defined.

The exact details of the data structure that describes a machine are given in Figure 3.1. The ma-

$$
\begin{array}{rl}
(\mathit{State}) & \mathbb{S} \in \Sigma \\
(\mathit{Operation}) & \iota \in \Delta \\
(\mathit{Conditional}) & b \in \beta \\
(\mathit{Cond\ Interp}) & \Upsilon \in \beta \rightarrow \Sigma \rightarrow \mathit{Prop} \\
(\mathit{Action}) & \mathtt{a} \in \Sigma \rightharpoonup \mathcal{P}(\Sigma) \\
(\mathit{Operational\ Semantics}) & \mathtt{OS} \in \{\iota \rightsquigarrow \mathtt{a}\}^{*} \\
(\mathit{Language\,/\,Machine}) & \mathcal{M} \in (\Sigma, \Delta, \beta, \Upsilon, \mathtt{OS})
\end{array}
$$

$$\text{where } \mathcal{M}(\iota) \triangleq \mathcal{M}.\mathtt{OS}(\iota) \text{ and } \mathcal{M}(b) := \mathcal{M}.\Upsilon(b)$$

Figure 3.1: Abstract State Machine

chine is a tuple containing several type definitions, functions, and mappings. The most important part of the machine definition is the type of the state (Σ). The type of the state determines all the information that the code may access about the machine. The machine also contains a set of operations (Δ), which are names of all possible atomic operations that the machine can be performed. What the operations do is encoded in the operational semantics of the machine ($\mathtt{OS}$), which map the individual operations to their actions. The action ($\mathtt{a}$) is a state transformation relation/function which defines a particular transition from a state into a set of possible states (more on this in Section 3.1.1).

The machine definition also includes the notion of conditional type (β) which describes the type of expressions that can be used for determining that branching conditions in the particular machines, which the semantics converts into predicates by using the interpreter function Υ.

3.1.1 The Language of Actions

In our verification system, the actions are used for three things:

1. To define the operational semantics of a particular machine. The definition of $\mathcal{M}.\mathtt{OS}$ is that particular use.

2. To define the behavior of a particular program. Any program can be imagined as a sequence of operations, and thus a sequence of actions, which itself is an action.

3. To define the specifications of a program.

These uses require that our actions be expressive enough to describe certain behaviors, which include

- **Non-determinism.** Suppose that action $\mathtt{a}$, when applied to a particular state $\mathbb{S}$, does not fail and will result in one of the several states, although we do not know which. To express this, we have defined an action type to have a set of states as a codomain. Thus the set of states defined by ($\mathtt{a}\ \mathbb{S}$) is a set of possible final states of that action. This does not mean that the operation with this actions has to eventually result in all the states it defines. In fact, it is possible that the operation always results in a particular state within the set (or does not terminate), but the action does not provide this information.

- **Infinite loop.** As we will only talk about partial correctness in this work, we will assume that any operation may infinite loop at any time. However, there are cases when an operation will not fail, but will not terminate either. To indicate this case, an action can be defined that for that particular starting state, the result will be an empty set, indicating no possible final states, nor failure.

- **Possible Failure.** Action $\mathtt{a}$ may fail on a particular $\mathbb{S}$. This can be indicated by defining that $\mathbb{S} \notin \mathtt{dom}(\mathtt{a})$. This means that if the action may potentially fail on the state, then we reduce the meaning of that actions to failure. This is the reason why the type of an action is a partial function, rather than a total function.

Because the type of actions is somewhat unwieldy, we have defined several combinators that allow us to work with this type in a concise manner. These are given in Figure 3.2.

Informally, the meaning of the combinators of the action calculus are as follows:

- *id* An identity combinator is a shorthand for a no-operation action. It simply states that for any state, the action indicates a transition to the singleton set consisting of the starting state.

- *fail* This combinator is a shorthand for an action that fails on every state. Thus the $fail$ action has an empty domain.

- *loop* This combinator encodes a safe action that never terminates. It can only represent an unconditional infinite loop.

- $\mathtt{p}?$ If the starting state satisfies the predicate, this action will behave as an identity. But if the starting state does not satisfy the predicate, the action will fail on that particular state. This combinator is useful in defining actions that serve as preconditions or guards.

- $\mathtt{a} \circ \mathtt{a}'$ This combinator composes the two actions. However, this composition is more complex than function composition, as the actions must result in sets of possible final states, and must fail if there is any possibility of failure. This means that if for some state $\mathbb{S}$, there is a possible final state $\mathbb{S}' \in \mathtt{a}\,\mathbb{S}$, and $\mathtt{a}'$ fails on that state, then the composition of the actions must fail on state $\mathbb{S}$. The result of the composed action must produce the set of all possibilities

Useful combinators for actions

Combinator	Notation	Definition using type $(\Sigma \rightharpoonup \mathcal{P}(\Sigma))$
Precondition	$\mathtt{p}?$	$\lambda \mathbb{S}. \begin{cases} \{\mathbb{S}\} & \text{if } \mathtt{p}\ \mathbb{S} \\ \text{undefined} & \text{otherwise} \end{cases}$
Identity	*id*	$\lambda \mathbb{S}. \{\mathbb{S}\}$
Failure	*fail*	$\lambda \mathbb{S}. \text{undefined}$ *(function with empty domain)*
Infinite loop	*loop*	$\lambda \mathbb{S}. \{\}$
Composition	$\mathtt{a} \circ \mathtt{a}'$	$\lambda \mathbb{S}. \begin{cases} \text{undef} & \text{if } \mathbb{S} \notin \mathtt{dom(a)} \\ \text{undef} & \text{if } \exists \mathbb{S}' \in \mathtt{a}\ \mathbb{S}.\ \mathbb{S}' \notin \mathtt{dom(a')} \\ \bigcup\limits_{\mathbb{S}' \in \mathtt{a}\ \mathbb{S}} (\mathtt{a}'\ \mathbb{S}') & \text{otherwise} \end{cases}$
Choice	$\mathtt{a} \oplus \mathtt{a}'$	$\lambda \mathbb{S}. \begin{cases} \text{undef} & \text{if } \mathbb{S} \notin \mathtt{dom(a)} \land \mathbb{S} \notin \mathtt{dom(a')} \\ \mathtt{a}\ \mathbb{S} & \text{if } \mathbb{S} \in \mathtt{dom(a)} \land \mathbb{S} \notin \mathtt{dom(a')} \\ \mathtt{a}'\ \mathbb{S} & \text{if } \mathbb{S} \notin \mathtt{dom(a)} \land \mathbb{S} \in \mathtt{dom(a')} \\ \mathtt{a}\ \mathbb{S} \cap \mathtt{a}'\ \mathbb{S} & \text{otherwise} \end{cases}$
Branch	$(\mathtt{p}?\ \mathtt{a} \oplus \mathtt{a}')$	$\lambda \mathbb{S}. \begin{cases} \mathtt{a}\ \mathbb{S} & \text{if } \mathbb{S} \in \mathtt{dom(p)} \\ \mathtt{a}'\ \mathbb{S} & \text{if } \mathbb{S} \notin \mathtt{dom(p)} \end{cases}$
Conjunction	$\mathtt{a} \land \mathtt{a}'$	$\lambda \mathbb{S}. \begin{cases} \mathtt{a}\ \mathbb{S} \cap \mathtt{a}'\ \mathbb{S} & \mathtt{if}\ \ \mathbb{S} \in \mathtt{dom(a)} \land \mathbb{S} \in \mathtt{dom(a')} \\ \text{undef} & \text{otherwise} \end{cases}$
Disjunction	$\mathtt{a} \lor \mathtt{a}'$	$\lambda \mathbb{S}. \begin{cases} \mathtt{a}\ \mathbb{S} \cup \mathtt{a}'\ \mathbb{S} & \mathtt{if}\ \ \mathbb{S} \in \mathtt{dom(a)} \lor \mathbb{S} \in \mathtt{dom(a')} \\ \text{undef} & \text{otherwise} \end{cases}$
One Of	$\bigvee\limits_{\sigma} f$	$\lambda \mathbb{S}. \{\mathbb{S}' \mid \exists v : \sigma.\ \mathbb{S}' \in f\ v\ \mathbb{S}\}$

The actions have a "weaker-than" partial order:

Combinator	Notation	Definition
Weaker-than	$\mathtt{a} \sqsupseteq \mathtt{a}'$	$\forall \mathbb{S}. \begin{cases} \mathbb{S} \notin \mathtt{dom(a)} & \text{if } \mathbb{S} \notin \mathtt{dom(a')} \\ \mathbb{S} \notin \mathtt{dom(a)} \lor (\mathtt{a}\ \mathbb{S} \sqsupseteq \mathtt{a}'\ \mathbb{S}) & \text{if } \mathbb{S} \in \mathtt{dom(a')} \end{cases}$
Equivalence	$\mathtt{a} \cong \mathtt{a}'$	$\mathtt{a} \sqsupseteq \mathtt{a}' \land \mathtt{a}' \sqsupseteq \mathtt{a}$

Figure 3.2: Combinators and Properties of Actions

reachable through the actions. In other words if from state $\mathbb{S}$, action $\mathtt{a}$ can result in states $\mathbb{S}_1$ and $\mathbb{S}_2$, and if running action $\mathtt{a}'$ on state $\mathbb{S}_1$ may result in $\mathbb{S}_{11}$ and $\mathbb{S}_{12}$ and on state $\mathbb{S}$ results in $\mathbb{S}_{21}$ and $\mathbb{S}_{22}$, then running action $\mathtt{a} \circ \mathtt{a}'$ on state $\mathbb{S}$ can result in any of $\mathbb{S}_{11}, \mathbb{S}_{12}, \mathbb{S}_{21}, \mathbb{S}_{22}$.

- ($\mathtt{p}?\ \mathtt{a} \oplus \mathtt{a}'$) The branch combinator is equivalent to

  $$(\text{if } \mathtt{p} \text{ then } \mathtt{a} \text{ else } \mathtt{a}' \text{ end if})$$

  The predicate is used to determine whether the left or right action is to be used.

- $\mathtt{a} \oplus \mathtt{a}'$ This combinator represents non-deterministic choice between two actions. Unlike the branch combinator, it does not have any conditionals which determine which of the two branches will execute. The easiest way to understand this is to imagine the domains of $\mathtt{a}$ and $\mathtt{a}'$ as predicates that determine which side is taken. The only possible issue is if both actions are valid for a particular state. In this case, both actions must properly describe the possible resulting states, and therefore the set of possible states is the intersection of the two sets.

- $\mathtt{a} \wedge \mathtt{a}'$ This combinator can be used to define an action where both are $\mathtt{a}$ and $\mathtt{a}'$ have to be valid at the same time.

- $\mathtt{a} \vee \mathtt{a}'$ This combinator can be used to create an action which consists of nondeterministic join of both actions. This differs from choice in that when a state is valid for both $\mathtt{a}$ and $\mathtt{a}'$, the result is a union, and not the intersection of the results of the actions.

- $\bigvee_{\sigma} f$ This combinator is a form of disjunction. However, instead of joining two arbitrary actions, this form joins together a family of actions indexed by values of some type σ. This combinator is useful to define actions whose non-determinism is based a single value. For example, this combinator could be used to define an action that depends on a random number.

The most important feature of the actions is that we can define a "weaker-than" preorder ($\mathtt{a} \supseteq \mathtt{a}'$) over them. An action $\mathtt{a}$ is weaker than $\mathtt{a}'$ if for every state $\mathbb{S}$

- If $\mathtt{a}'$ fails on $\mathbb{S}$, then $\mathtt{a}$ fails on $\mathbb{S}$ as well

- If $\mathtt{a}$ applied to $\mathbb{S}$ produces a set of possible states R, then $\mathtt{a}'$ applied to $\mathbb{S}$ will produce a subset of R.

Property	Definition
Reflexivity	$\forall \mathtt{a}.\, \mathtt{a} \supseteq \mathtt{a}$
Transitivity	$\forall \mathtt{a}, \mathtt{a}', \mathtt{a}''.\, \mathtt{a} \supseteq \mathtt{a}' \rightarrow \mathtt{a}' \supseteq \mathtt{a}'' \rightarrow \mathtt{a} \supseteq \mathtt{a}''$
Precondition Weakening	$\forall \mathtt{p}, \mathtt{p}'.\, (\mathtt{p} \rightarrow \mathtt{p}') \rightarrow (\mathtt{p}? \supseteq \mathtt{p}'?)$
Precondition Weakening 2	$\forall \mathtt{p}, \mathtt{a}.\, (\mathtt{p}? \circ \mathtt{a}) \supseteq \mathtt{a}$
Strongest Action	$\forall \mathtt{a}.\, \mathtt{a} \supseteq \textit{loop}$
Composition Associativity	$\mathtt{a} \circ (\mathtt{a}' \circ \mathtt{a}'') \cong (\mathtt{a} \circ \mathtt{a}') \circ \mathtt{a}''$
Composition Weakening	if $\mathtt{a}_1 \supseteq \mathtt{a}'_1$ and $\mathtt{a}_2 \supseteq \mathtt{a}'_2$, then $\mathtt{a}_1 \circ \mathtt{a}_2 \supseteq \mathtt{a}'_1 \circ \mathtt{a}'_2$
Choice Weakening	if $\mathtt{a}_1 \supseteq \mathtt{a}'_1$ and $\mathtt{a}_2 \supseteq \mathtt{a}'_2$, then $\mathtt{a}_1 \oplus \mathtt{a}_2 \supseteq \mathtt{a}'_1 \oplus \mathtt{a}'_2$
Branch Weakening	if $\mathtt{a}_1 \supseteq \mathtt{a}'_1$ and $\mathtt{a}_2 \supseteq \mathtt{a}'_2$, then $\forall \mathtt{p}.\ (\mathtt{p}?\ \mathtt{a}_1 \oplus \mathtt{a}_2) \supseteq \left(\mathtt{p}?\ \mathtt{a}'_1 \oplus \mathtt{a}'_2\right)$
Composition Equivalence	if $\mathtt{a}_1 \cong \mathtt{a}'_1$ and $\mathtt{a}_2 \cong \mathtt{a}'_2$, then $\mathtt{a}_1 \circ \mathtt{a}_2 \cong \mathtt{a}'_1 \circ \mathtt{a}'_2$
Choice Equivalence	if $\mathtt{a}_1 \cong \mathtt{a}'_1$ and $\mathtt{a}_2 \cong \mathtt{a}'_2$, then $\mathtt{a}_1 \oplus \mathtt{a}_2 \cong \mathtt{a}'_1 \oplus \mathtt{a}'_2$
Branch Equivalence	if $\mathtt{a}_1 \cong \mathtt{a}'_1$ and $\mathtt{a}_2 \cong \mathtt{a}'_2$, then $\forall \mathtt{p}.\ (\mathtt{p}?\ \mathtt{a}_1 \oplus \mathtt{a}_2) \cong \left(\mathtt{p}?\ \mathtt{a}'_1 \oplus \mathtt{a}'_2\right)$
Conjunction Weakening	$\forall \mathtt{a}, \mathtt{a}', \mathtt{a}''.\, \mathtt{a} \supseteq \mathtt{a}' \rightarrow (\mathtt{a} \wedge \mathtt{a}'') \supseteq (\mathtt{a}' \wedge \mathtt{a}'')$

Figure 3.3: Properties of Action Combinators

Informally, that means that $\mathtt{a}$ describes more possibilities than $\mathtt{a}'$. When if we have two actions, where both $\mathtt{a} \supseteq \mathtt{a}'$ and $\mathtt{a}' \supseteq \mathtt{a}$, then we know that the two actions are equivalent, that is $\mathtt{a} \cong \mathtt{a}'$, and for actions, being equivalent is the same as being point-wise equal.

All these action combinators have certain mathematical properties which becomes useful in our proofs. These are listed in Figure 3.3, which include the weaker-than relation defining a preorder, and thus has the property of reflexivity and transitivity. There are several other properties over actions and their combinators that become useful. For example the fact that action concatenation is associative is quite natural to the informal understanding of what actions are, but formally, we still have to give a proof of this fact to use it. The less obvious properties are *Composition Weakening* and *Choice Weakening*, which show that given any two actions weaker than some other two actions, their combination by composition or choice will be weaker than the combination of the stronger pair.

The proofs of these properties are given in Appendix A.

3.1.2 Alternative Definition of Actions

Defining actions as $\Sigma \rightharpoonup \mathcal{P}(\Sigma)$ makes them easy to understand and to imagine; it seems to match our understanding of what it means to execute a program. However, this type can be difficult to

Combinator	Definition
$\mathrm{p}?$	$(\lambda\mathbb{S}.\,\mathrm{p}, \quad \lambda\mathbb{S},\mathbb{S}'.\,\mathbb{S} = \mathbb{S}')$
id	$(\lambda\mathbb{S}.\,\mathsf{True}, \quad \lambda\mathbb{S},\mathbb{S}'.\,\mathbb{S} = \mathbb{S}')$
$fail$	$(\lambda\mathbb{S}.\,\mathsf{False}, \quad \lambda\mathbb{S},\mathbb{S}'.\,\mathsf{False})$
$(\mathrm{p},\mathrm{r}) \circ (\mathrm{p}',\mathrm{r}')$	$(\lambda\mathbb{S}.\,\mathrm{p}\ \mathbb{S} \wedge \forall\mathbb{S}''.\,\mathrm{r}\ \mathbb{S}\ \mathbb{S}'' \rightarrow \mathrm{p}'\ \mathbb{S}'',$ $\lambda\mathbb{S},\mathbb{S}'.\,\exists\mathbb{S}''.\ \mathrm{r}\ \mathbb{S}\ \mathbb{S}'' \wedge \mathrm{r}'\ \mathbb{S}''\ \mathbb{S}')$
$(\mathrm{p},\mathrm{r}) \oplus (\mathrm{p}',\mathrm{r}')$	$(\lambda\mathbb{S}.\,\mathrm{p}\ \mathbb{S} \vee \mathrm{p}'\ \mathbb{S},$ $\lambda\mathbb{S},\mathbb{S}'.\,(\mathrm{p}\ \mathbb{S} \rightarrow \mathrm{r}\ \mathbb{S}\ \mathbb{S}') \wedge (\mathrm{p}'\ \mathbb{S} \rightarrow \mathrm{r}'\ \mathbb{S}\ \mathbb{S}'))$
$(\mathrm{p}''?\ (\mathrm{p},\mathrm{r}) \oplus (\mathrm{p}',\mathrm{r}'))$	$(\lambda\mathbb{S}.\,\mathrm{p}\ \mathbb{S} \wedge \mathrm{p}''\ \mathbb{S} \vee \mathrm{p}'\ \mathbb{S} \wedge \neg\mathrm{p}''\ \mathbb{S},$ $\lambda\mathbb{S},\mathbb{S}'.\,(\mathrm{p}\ \mathbb{S} \rightarrow \mathrm{p}''\ \mathbb{S} \rightarrow \mathrm{r}\ \mathbb{S}\ \mathbb{S}') \wedge (\mathrm{p}'\ \mathbb{S} \rightarrow \neg\mathrm{p}''\ \mathbb{S} \rightarrow \mathrm{r}'\ \mathbb{S}\ \mathbb{S}'))$
$(\mathrm{p},\mathrm{r}) \wedge (\mathrm{p}',\mathrm{r}')$	$(\lambda\mathbb{S}.\,\mathrm{p}\ \mathbb{S} \wedge \mathrm{p}'\ \mathbb{S}, \quad \lambda\mathbb{S},\mathbb{S}'.\,\mathrm{r}\ \mathbb{S}\ \mathbb{S}' \wedge \mathrm{r}'\ \mathbb{S}\ \mathbb{S}')$
$(\mathrm{p},\mathrm{r}) \vee (\mathrm{p}',\mathrm{r}')$	$(\lambda\mathbb{S}.\,\mathrm{p}\ \mathbb{S} \vee \mathrm{p}'\ \mathbb{S}, \quad \lambda\mathbb{S},\mathbb{S}'.\,\mathrm{r}\ \mathbb{S}\ \mathbb{S}' \vee \mathrm{r}'\ \mathbb{S}\ \mathbb{S}')$
$(\mathrm{p},\mathrm{r}) \supseteq (\mathrm{p}',\mathrm{r}')$	$(\forall\mathbb{S}.\,\mathrm{p}\ \mathbb{S} \rightarrow \mathrm{p}'\ \mathbb{S}) \wedge$ $(\forall\mathbb{S},\mathbb{S}'.\,\mathrm{p}\ \mathbb{S} \rightarrow \mathrm{r}'\ \mathbb{S}\ \mathbb{S}' \rightarrow \mathrm{r}\ \mathbb{S}\ \mathbb{S}')$
$(\mathrm{p},\mathrm{r}) \cong (\mathrm{p}',\mathrm{r}')$	$(\mathrm{p},\mathrm{r}) \supseteq (\mathrm{p}',\mathrm{r}') \wedge (\mathrm{p}',\mathrm{r}') \supseteq (\mathrm{p},\mathrm{r})$

Figure 3.4: Combinators for (p,r)-style Actions

encode in formal logic. Instead, when dealing with proof assistants such Coq, we find that it is much easier to define actions as relations. For these reasons, we use the following type as an action in our formalized proofs:

$$(\mathrm{p} : \Sigma \rightarrow \mathit{Prop}, \mathrm{r} : \Sigma \rightarrow \Sigma \rightarrow \mathit{Prop})$$

This version is defined using a state predicate and a relation of states. The state predicate p (precondition) determines whether the action has a possibility of failure for the specific state, e.g. it defines the domain of the action. The state relation (r) defines which states are the possible end states of the specific action, thereby defining the codomain of the function. It is important to note that if $\mathbb{S}$ is in the domain of p, but r $\mathbb{S}$ is empty, the meaning of the action for this state is an infinite loop, and not failure.

Figure 3.4 shows some of the combinators for this definition of actions. Although different in implementation, these definitions of combinators are isomorphic to the original definitions. Although we will not show this fact in this thesis, all the same properties that were true of the original action definitions still hold. In fact, all formal verification proofs that are done in Coq use the (p,r) actions defined here.

$$
\begin{array}{rlll}
(\textit{Meta-program}) & \mathbb{P} & ::= & (\mathbb{C},\mathbb{I}) \\
(\textit{Proc}) & \mathbb{I} & ::= & \texttt{nil} \mid \iota \mid [\texttt{l}] \mid \mathbb{I}_1;\mathbb{I}_2 \mid \mathbb{I}_1 + \mathbb{I}_2 \\
 & & & \mid (b?\ \mathbb{I}_1 + \mathbb{I}_2)\ \textit{(alternatively)} \\
(\textit{Proc Heap}) & \mathbb{C} & ::= & \{\texttt{l} \rightsquigarrow \mathbb{I}\}^* \\
(\textit{Labels}) & \texttt{l} & ::= & n \text{ (natural numbers)} \\
(\textit{Spec Heap}) & \mathcal{L},\Psi & ::= & \{\texttt{l} \rightsquigarrow \texttt{a}\}^*
\end{array}
$$

$$
\begin{array}{lll}
\llbracket \mathbb{C}, \texttt{a} \rrbracket^0_{\mathcal{M},\mathcal{L}} & := \mathit{loop} & \\
\llbracket \mathbb{C}, \texttt{nil} \rrbracket^n_{\mathcal{M},\mathcal{L}} & := \mathit{id} & \\
\llbracket \mathbb{C}, \iota \rrbracket^n_{\mathcal{M},\mathcal{L}} & := (\mathcal{M}(\iota)) & \\
\llbracket \mathbb{C}, [\texttt{l}] \rrbracket^n_{\mathcal{M},\mathcal{L}} & := (\mathcal{L}(\texttt{l})) & \text{if } \texttt{l} \in \texttt{dom}(\mathcal{L}) \\
\llbracket \mathbb{C}, [\texttt{l}] \rrbracket^n_{\mathcal{M},\mathcal{L}} & := \llbracket \mathbb{C}, \mathbb{C}(\texttt{l}) \rrbracket^{n-1}_{\mathcal{M},\mathcal{L}} & \text{if } \texttt{l} \in \texttt{dom}(\mathbb{C}) \\
\llbracket \mathbb{C}, \mathbb{I};\mathbb{I}' \rrbracket^n_{\mathcal{M},\mathcal{L}} & := \llbracket \mathbb{C}, \mathbb{I} \rrbracket^n_{\mathcal{M},\mathcal{L}} \circ \llbracket \mathbb{C}, \mathbb{I} \rrbracket^n_{\mathcal{M},\mathcal{L}} & \\
\llbracket \mathbb{C}, \mathbb{I}_1 + \mathbb{I}_2 \rrbracket^n_{\mathcal{M},\mathcal{L}} & := \llbracket \mathbb{C}, \mathbb{I}_1 \rrbracket^n_{\mathcal{M},\mathcal{L}} \oplus \llbracket \mathbb{C}, \mathbb{I}_2 \rrbracket^n_{\mathcal{M},\mathcal{L}} & \\
\llbracket \mathbb{C}, (b?\ \mathbb{I}_1 + \mathbb{I}_2) \rrbracket^n_{\mathcal{M},\mathcal{L}} & := \left(\mathcal{M}(b)?\ \llbracket \mathbb{C}, \mathbb{I}_1 \rrbracket^n_{\mathcal{M},\mathcal{L}} \oplus \llbracket \mathbb{C}, \mathbb{I}_2 \rrbracket^n_{\mathcal{M},\mathcal{L}}\right) &
\end{array}
$$

Figure 3.5: Syntax and Semantics of the Meta-Language

3.2 Meta-language

In our definition of the machine, a machine is just a named set of operations. It does not have its own semantics, nor does it have any notion of computation. To actually perform computations on our machines, we have built a simulation meta-machine (meta-language), which uses the definition of the machine ($\mathcal{M}$) to make computational steps over programs designed for $\mathcal{M}$. Thus our meta-language is a form of a universal simulator (at least for the machines that can be encoded as $\mathcal{M}$).

To simulate any program on machine $\mathcal{M}$, we lift the program into our meta-language, resulting in a meta-program. This meta-program can now be analyzed using the standard semantic approaches. Since all of the semantics of the meta-language are parametric over the definition of $\mathcal{M}$, we no longer have to define new semantics for every single machine we use. We will discuss whether this approach is actually equivalent to defining actual machines in Section 3.6.5, but first we will give a full description of the meta-language.

3.2.1 Meta-Language Syntax

The syntax of our meta-machine / meta-language is given in Figure 3.5. The keystone definition of the meta-language is a *Proc* ($\mathbb{I}$), which defines a computational procedure. The procedure can be the following:

- *nil*. An empty identity procedure that does nothing.

- An operation ι provided by machine $\mathcal{M}$. This procedure indicates that this single operation will be simulated.

- A meta-call ([`l`]). Indicates that a procedure in the procedure heap ($\mathbb{C}$) at label `l` will be executed till completion. If the meta-call is a part of a larger procedure, once the execution of the called procedure completes, the original execution will resume.

- A sequence of two procedures ($\mathbb{I}_1;\mathbb{I}_2$). Indicates a sequential execution of procedures.

- A branch ($b?\ \mathbb{I}_1+\mathbb{I}_2$). This is an if-then-else statement in our language. Notice how it uses the conditional expression to define the branching condition.

- A non-deterministic choice of procedures ($\mathbb{I}_1+\mathbb{I}_2$). The simplest explanation is that this is a generalization of a branch statement, without a particular condition. We non-deterministically run both subprocedures, and see which one succeeds. In practice, the incorrect branch will fail quickly, as the machine operations usually include checks to make sure that they are following the correct workflow.

It is not necessary to have both the branch and the choice in the language. As long as the language includes either, it is suitable for our needs. For clarity, we will mostly use the deterministic branch.

In addition to the syntax of the language itself, we also define *specification heap* (Ψ) to hold the specifications of the all the procedures in the heap. When the specification heap contains stubs, i.e. specification of procedures that are not actually present in the current code because they will either be linked in later, or the procedure is actually a primitive, then these specification heaps are referred to as libraries, and are usually marked with ($\mathcal{L}$).

Because this language does not really execute to a particular value, but rather deals with possible states, we think that the denotational semantics is the best approach for defining the semantics of this language. The semantics described in the lower half of Figure 3.5 define an action as the meaning of a procedure.

Most of the semantics is fairly intuitive and follow the syntax. For example, the meaning of a primitive operation (ι) is just the operational semantics for that operation defined in the machine $\mathcal{M}$

$$\frac{\exists \mathsf{a}. \quad \mathcal{M},\mathcal{L} \vdash \mathbb{C} : \Psi \quad \mathcal{M},\mathcal{L} \cup \Psi \vdash \mathbb{I} : \mathtt{a} \quad \mathbb{S} \in \mathsf{dom}(\mathsf{a})}{\mathcal{M},\mathcal{L} \vdash (\mathbb{C},\mathbb{I},\mathbb{S})} \; (\textsc{top})$$

$$\frac{\forall \mathtt{l} \in \mathtt{dom}(\mathbb{C}).\, \mathcal{M},\Psi \cup \mathcal{L} \vdash \mathbb{C}(\mathtt{l}) : \Psi(\mathtt{l})}{\mathcal{M},\mathcal{L} \vdash \mathbb{C} : \Psi} \; (\textsc{wf-code})$$

$$\frac{\mathcal{M},\mathcal{L} \vdash \mathbb{I}' : \mathsf{a}' \quad \mathcal{M},\mathcal{L} \vdash \mathbb{I}'' : \mathsf{a}''}{\mathcal{M},\mathcal{L} \vdash (b?\, \mathbb{I}' + \mathbb{I}'') : (\mathcal{M}(b)?\, \mathsf{a}' \oplus \mathsf{a}'')} \; (\textsc{wf-branch})$$

$$\frac{\mathcal{M},\mathcal{L} \vdash \mathbb{I}' : \mathsf{a}' \quad \mathcal{M},\mathcal{L} \vdash \mathbb{I}'' : \mathsf{a}''}{\mathcal{M},\mathcal{L} \vdash \mathbb{I}' + \mathbb{I}'' : \mathsf{a}' \oplus \mathsf{a}''} \; (\textsc{wf-choice}) \qquad \frac{\mathcal{M},\mathcal{L} \vdash \mathbb{I}' : \mathsf{a}' \quad \mathcal{M},\mathcal{L} \vdash \mathbb{I}'' : \mathsf{a}''}{\mathcal{M},\mathcal{L} \vdash \mathbb{I}';\mathbb{I}'' : (\mathsf{a}' \circ \mathsf{a}'')} \; (\textsc{wf-seq})$$

$$\frac{\mathcal{M},\mathcal{L} \vdash \mathbb{I} : \mathsf{a}' \quad \mathsf{a} \supseteq \mathsf{a}'}{\mathcal{M},\mathcal{L} \vdash \mathbb{I} : \mathsf{a}} \; (\textsc{wf-weak}) \qquad \frac{}{\mathcal{M},\mathcal{L} \vdash \iota : \mathcal{M}(\iota)} \; (\textsc{wf-op})$$

$$\frac{}{\mathcal{M},\mathcal{L} \vdash [\mathtt{l}] : \mathcal{L}(\mathtt{l})} \; (\textsc{wf-call}) \qquad \frac{}{\mathcal{M},\mathcal{L} \vdash \mathtt{nil} : \mathit{id}} \; (\textsc{wf-nil})$$

Figure 3.6: Static Semantics of the Meta-Language

in which the program is running. However, the case of the call ([l]) needs additional explanation. The call can mean two things for a set of procedures: if a procedure exists in the module then it is procedure inclusion. However, if the procedure is not in a module it might be part of the trusted or assumed library $\mathcal{L}$. In this case, we treat the call the same way we treat a primitive operation - we assume that the action of the callee is just the specification that we have for it. This additional behavior allows us expand the machine definitions with additional trusted primitives, an important mechanism in our multi-machine verification.

3.3 Static Semantics of the Meta-Language

Although our work tends to follow a traditional Hoare-logic approach, we have begun to deviate from using the standard pre and post-conditions for specification of our programs. The main reason for this is that Hoare triples frequently require the use of auxiliary variables, which take additional machinery to specify. In our SCAP and subsequent work, we have been relying on the (p,r) relation pair as a substitute for the Hoare triple. Since this work shows that this pair is essentially isomorphic to our actions, we therefore use actions as the specifications language for our programs.

The static semantics for our meta-language is given in Figure 3.6. A quick look will indicate that the rules used in the static semantics are very similar to the denotational semantics. For example, in the case of a primitive operation the meaning of the primitive operation is $[\![\mathbb{C}, \iota]\!]_{\mathcal{M}} = \mathcal{M}(\iota)$, while the specification of the operation is $\mathcal{M}, \Psi \vdash \iota : \mathcal{M}(\iota)$. In other words, both the meaning of ι, and the specification of ι are $\mathcal{M}(\iota)$, the action which corresponds to the operation itself. Similarly, both the meaning and the specification of $\mathbb{I}_1;\mathbb{I}_2$ are meanings or specification of both sides connected by $\circ$, respectively.

However, there is a difference in the case of [l]. The denotational semantics substitutes the call into the procedure with the body of the procedure, decreasing the index. This approach can not be used for the static semantics because of circularity. We handle this by having a *specification heap* that contains the actions that define the approximations of program meanings, with the requirement that the approximation is always weaker than the meaning. Then the specification of the procedure call is just the approximation of meaning of the procedure that we retrieve from the specification heap. The properties of the combinators that we use ensure that if we use approximations, then the specification for the procedure we are analyzing also results in a valid approximation. Thus our static semantics essentially comes down to picking approximations of the meaning of programs, and then making sure that those approximations are sound.

These individual well-formed procedures are packaged together using the WF-CODE rule. The code rule also shows how the recursion is handled. To certify a recursive procedure $\mathbb{I}$ at a label l, we would simply assume the fixed point specification of $\mathbb{I}$, and then show that under that assumption $\mathbb{I}$ is indeed well-formed. The rule accomplishes this by placing the specification of $\mathbb{I}$ into Ψ(l), and using it in the verification of $\mathbb{I}$, and then discharging that assumption. The rule then gives rise to the well-formed code heap, which we will refer to as *certified module*.

The TOP rule simply shows that if a program, which consists of a module with a code heap $\mathbb{C}$ and a top-level procedure $\mathbb{I}$, is sound, and the state is in the domain of a procedure specification, then the whole program is sound. That means that this program is safe to run starting with the given state.

3.3.1 Automatic Generation of Strongest Specification

An individual procedure has no cycles in itself. It may call other procedures, but the static semantics require that we satisfy the specifications that are defined in the specification heap. Thus it is possible to automatically generate the strongest action that will satisfy the static semantics. In other words, we can create genspec such that the following is true.

If $\mathbb{I} =$	then $\texttt{genspec}_{\mathcal{M},\mathcal{L}}(\mathbb{I}) =$
`nil`	id
ι	$\mathcal{M}(\iota)$
`[l]`	$\mathcal{L}(\texttt{l})$
$\mathbb{I}_1;\mathbb{I}_2$	$\texttt{genspec}_{\mathcal{M},\mathcal{L}}(\mathbb{I}_1) \circ \texttt{genspec}_{\mathcal{M},\mathcal{L}}(\mathbb{I}_2)$
$\mathbb{I}_1 + \mathbb{I}_2$	$\texttt{genspec}_{\mathcal{M},\mathcal{L}}(\mathbb{I}_1) \oplus \texttt{genspec}_{\mathcal{M},\mathcal{L}}(\mathbb{I}_2)$
$(b?\, \mathbb{I}_1 + \mathbb{I}_2)$	$\big(\mathcal{M}(b)?\, \texttt{genspec}_{\mathcal{M},\mathcal{L}}(\mathbb{I}_1) \oplus \texttt{genspec}_{\mathcal{M},\mathcal{L}}(\mathbb{I}_2)\big)$

Figure 3.7: Automatic Generation of Strongest Specification

Lemma 3.3.1 (`genspec` **valid)**

$\mathcal{M},\mathcal{L} \vdash \mathbb{I} : \texttt{genspec}_{\mathcal{M},\mathcal{L}}(\mathbb{I})$

Pf. By induction on $\mathbb{I}$. There are several cases:

- $\mathbb{I} = \texttt{nil}$ Then $\mathcal{M},\mathcal{L} \vdash \texttt{nil} : id$ by wf-nil rule.
- $\mathbb{I} = \iota$ Then $\mathcal{M},\mathcal{L} \vdash \iota : \mathcal{M}(\iota)$ by wf-op rule.
- $\mathbb{I} = \texttt{[l]}$ Then $\mathcal{M},\mathcal{L} \vdash \texttt{[l]} : \mathcal{L}(\texttt{l})$ by wf-call rule.
- $\mathbb{I} = \mathbb{I}_1;\mathbb{I}_2$
  By IH, $\mathcal{M},\mathcal{L} \vdash \mathbb{I}_1 : \texttt{genspec}_{\mathcal{M},\mathcal{L}}(\mathbb{I}_1)$ and $\mathcal{M},\mathcal{L} \vdash \mathbb{I}_2 : \texttt{genspec}_{\mathcal{M},\mathcal{L}}(\mathbb{I}_2)$.
  By wf-seq rule, $\mathcal{M},\mathcal{L} \vdash \mathbb{I}_1;\mathbb{I}_2 : \texttt{genspec}_{\mathcal{M},\mathcal{L}}(\mathbb{I}_1) \circ \texttt{genspec}_{\mathcal{M},\mathcal{L}}(\mathbb{I}_2)$.
  By definition of `genspec`, $\mathcal{M},\mathcal{L} \vdash \mathbb{I}_1;\mathbb{I}_2 : \texttt{genspec}_{\mathcal{M},\mathcal{L}}(\mathbb{I}_1;\mathbb{I}_2)$
- $\mathbb{I} = \mathbb{I}_1 + \mathbb{I}_2$
  By IH, $\mathcal{M},\mathcal{L} \vdash \mathbb{I}_1 : \texttt{genspec}_{\mathcal{M},\mathcal{L}}(\mathbb{I}_1)$ and $\mathcal{M},\mathcal{L} \vdash \mathbb{I}_2 : \texttt{genspec}_{\mathcal{M},\mathcal{L}}(\mathbb{I}_2)$.
  By wf-choice rule, $\mathcal{M},\mathcal{L} \vdash \mathbb{I}_1 + \mathbb{I}_2 : \texttt{genspec}_{\mathcal{M},\mathcal{L}}(\mathbb{I}_1) \oplus \texttt{genspec}_{\mathcal{M},\mathcal{L}}(\mathbb{I}_2)$.
  By definition of `genspec`, $\mathcal{M},\mathcal{L} \vdash \mathbb{I}_1 + \mathbb{I}_2 : \texttt{genspec}_{\mathcal{M},\mathcal{L}}(\mathbb{I}_1 + \mathbb{I}_2)$
- $\mathbb{I} = (b?\, \mathbb{I}_1 + \mathbb{I}_2)$
  By IH, $\mathcal{M},\mathcal{L} \vdash \mathbb{I}_1 : \texttt{genspec}_{\mathcal{M},\mathcal{L}}(\mathbb{I}_1)$ and $\mathcal{M},\mathcal{L} \vdash \mathbb{I}_2 : \texttt{genspec}_{\mathcal{M},\mathcal{L}}(\mathbb{I}_2)$.
  By wf-branch rule, $\mathcal{M},\mathcal{L} \vdash (b?\, \mathbb{I}_1 + \mathbb{I}_2) : \big(\mathcal{M}(b)?\, \texttt{genspec}_{\mathcal{M},\mathcal{L}}(\mathbb{I}_1) \oplus \texttt{genspec}_{\mathcal{M},\mathcal{L}}(\mathbb{I}_2)\big)$.
  By definition of `genspec`, $\mathcal{M},\mathcal{L} \vdash (b?\, \mathbb{I}_1 + \mathbb{I}_2) : \{\texttt{genspec}_{\mathcal{M},\mathcal{L}}((b?\, \mathbb{I}_1 + \mathbb{I}_2))$

□

The definition of such function is given in Figure 3.7. The algorithm does not use any weakening rules, meaning that it generates the strongest specification it can using the library $\mathcal{L}$. This means that any specification we can assign to the procedure has to be weaker than the one generated by `genspec`. This allows us to simplify the process of verification. Instead of using the syntax driven static semantics to verify that a procedure follows some specification, we could do it with the following corollary:

Lemma 3.3.2 (proc)

For all $\mathcal{M},\mathcal{L},\mathbb{I},\mathsf{a}$, if $\mathsf{a} \supseteq \texttt{genspec}_{\mathcal{M},\mathcal{L}}(\mathbb{I})$, then $\mathcal{M},\mathcal{L} \vdash \mathbb{I} : \mathsf{a}$

Pf. By using Lemma 3.3.1 followed by the WF-WEAK rule.

□

With this corollary, the process of verification a procedure is a matter of choosing a specification, generating the strongest specification, and then showing that the target specification is weaker than than the generated one.

This algorithm relies on the presence of specifications in $\mathcal{L}$ for all procedures that are called. Because procedures may be recursive, this algorithm can not be used to populate $\mathcal{L}$, and can not be used to determine the strongest specification of fixed points.

3.4 Safety and Partial Correctness

Using denotational semantics, it is easy to define the safety and the partial correctness of the program with specification $\mathtt{a}$. The definition of partial correctness is simply:

$$\forall n.\, \mathtt{a} \supseteq [\![\mathbb{C}, \mathbb{I}]\!]^{n}_{\mathcal{M},\mathcal{L}}$$

Then safety can be ensured by checking that the starting state $\mathbb{S}$ is in the domain of $\mathtt{a}$, and thereby in the domain of $[\![\mathbb{C}, \mathbb{I}]\!]^{n}_{\mathcal{M},\mathcal{L}}$. This definition also guarantees that from any state $\mathbb{S}$, the set of possible states that can result when the program terminates ($[\![\mathbb{C}, \mathbb{I}]\!]^{n}_{\mathcal{M},\mathcal{L}}\ \mathbb{S}$) will be smaller than the resulting set described by the specification ($\mathtt{a}\ \mathbb{S}$).

To establish correctness, we need to do an induction over the index of the denotational semantics. This fact is made more complicated as procedures may depend on each other. To handle this issue, we assume that we have already shown that the specification in the specification heap are weaker than the (n-1) approximations of actions of the programs in the code heap. With this assumption we show that any well-formed specification for a procedure $\mathbb{I}$ is weaker than the n-approximation of the action of the procedure.

Theorem 3.4.1 (Partial Correctness of Procedures)

For all $n > 0$, if $\mathcal{M}, \Psi \cup \mathcal{L} \vdash \mathbb{I} : \mathtt{a}$ and $\mathcal{M}, \mathcal{L} \vdash \mathbb{C} : \Psi$ and $\forall \mathtt{f} \in \mathsf{dom}(\mathbb{C}).\, \Psi(\mathtt{f}) \supseteq [\![\mathbb{C}, \mathbb{C}(\mathtt{f})]\!]^{n-1}_{\mathcal{M},\mathcal{L}}$, then $\mathtt{a} \supseteq [\![\mathbb{C}, \mathbb{I}]\!]^{n}_{\mathcal{M},\mathcal{L}}$.

Pf. Proof is by induction on the derivation of $\mathcal{M}, \Psi \cup \mathcal{L} \vdash \mathbb{I} : \mathtt{a}$.

- Case $\mathcal{M}, \Psi \cup \mathcal{L} \vdash \mathtt{nil} : id$
  By reflexivity of $\supseteq$, $id \supseteq id$
  By definition of $\llbracket\rrbracket$, $\llbracket \mathbb{C}, nil \rrbracket^n_{\mathcal{M},\mathcal{L}} = id$
  Thus, $id \supseteq \llbracket \mathbb{C}, nil \rrbracket^n_{\mathcal{M},\mathcal{L}}$
- Case $\mathcal{M}, \Psi \cup \mathcal{L} \vdash \iota : \mathcal{M}(\iota)$
  The meaning of of the program is the following: $\llbracket \mathbb{C}, \iota \rrbracket^n_{\mathcal{M},\mathcal{L}} = \mathcal{M}(\iota)$
  By reflexivity of $\supseteq$: $\mathcal{M}(\iota) \supseteq \mathcal{M}(\iota)$
- Case $\mathcal{M}, \Psi \cup \mathcal{L} \vdash \mathbb{I}_1 ; \mathbb{I}_2 : \mathtt{a}_1 \circ \mathtt{a}_2$
  By inversion, $\mathcal{M}, \Psi \cup \mathcal{L} \vdash \mathbb{I}_1 : \mathtt{a}_1$ and $\mathcal{M}, \Psi \cup \mathcal{L} \vdash \mathbb{I}_2 : \mathtt{a}_2$.
  By induction hypothesis, $\mathtt{a}_1 \supseteq \llbracket \mathbb{C}, \mathbb{I}_1 \rrbracket^n_{\mathcal{M},\mathcal{L}}$ and $\mathtt{a}_2 \supseteq \llbracket \mathbb{C}, \mathbb{I}_2 \rrbracket^n_{\mathcal{M},\mathcal{L}}$.
  By composition weakening, $\mathtt{a}_1 \circ \mathtt{a}_2 \supseteq \llbracket \mathbb{C}, \mathbb{I}_1 \rrbracket^n_{\mathcal{M},\mathcal{L}} \circ \llbracket \mathbb{C}, \mathbb{I}_2 \rrbracket^n_{\mathcal{M},\mathcal{L}}$.
  By definition of $\llbracket\rrbracket$, $\llbracket \mathbb{C}, \mathbb{I}_1 ; \mathbb{I}_2 \rrbracket^n_{\mathcal{M},\mathcal{L}} = \llbracket \mathbb{C}, \mathbb{I}_1 \rrbracket^n_{\mathcal{M},\mathcal{L}} \circ \llbracket \mathbb{C}, \mathbb{I}_2 \rrbracket^n_{\mathcal{M},\mathcal{L}}$
  Thus, $\mathtt{a}_1 \circ \mathtt{a}_2 \supseteq \llbracket \mathbb{C}, \mathbb{I}_1 ; \mathbb{I}_2 \rrbracket^n_{\mathcal{M},\mathcal{L}}$
- Case $\mathcal{M}, \Psi \cup \mathcal{L} \vdash \mathbb{I}_1 + \mathbb{I}_2 : \mathtt{a}_1 \oplus \mathtt{a}_2$
  By inversion $\mathcal{M}, \Psi \cup \mathcal{L} \vdash \mathbb{I}_1 : \mathtt{a}_1$ and $\mathcal{M}, \Psi \cup \mathcal{L} \vdash \mathbb{I}_2 : \mathtt{a}_2$
  By induction hypothesis, $\mathtt{a}_1 \supseteq \llbracket \mathbb{C}, \mathbb{I}_1 \rrbracket^n_{\mathcal{M},\mathcal{L}}$ and $\mathtt{a}_2 \supseteq \llbracket \mathbb{C}, \mathbb{I}_2 \rrbracket^n_{\mathcal{M},\mathcal{L}}$
  By choice weakening, $\mathtt{a}_1 \oplus \mathtt{a}_2 \supseteq \llbracket \mathbb{C}, \mathbb{I}_1 \rrbracket^n_{\mathcal{M},\mathcal{L}} \oplus \llbracket \mathbb{C}, \mathbb{I}_2 \rrbracket^n_{\mathcal{M},\mathcal{L}}$
  By definition of $\llbracket\rrbracket$, $\llbracket \mathbb{C}, \mathbb{I}_1 + \mathbb{I}_2 \rrbracket_{\mathcal{M},\mathcal{L}} = \llbracket \mathbb{C}, \mathbb{I}_1 \rrbracket^n_{\mathcal{M},\mathcal{L}} \oplus \llbracket \mathbb{C}, \mathbb{I}_1 \rrbracket^n_{\mathcal{M},\mathcal{L}}$
  Thus, $\mathtt{a}_1 \oplus \mathtt{a}_2 \supseteq \llbracket \mathbb{C}, \mathbb{I}_1 + \mathbb{I}_2 \rrbracket^n_{\mathcal{M},\mathcal{L}}$
- Case $\mathcal{M}, \Psi \cup \mathcal{L} \vdash (b?\ \mathbb{I}_1 + \mathbb{I}_2) : (\mathcal{M}(b)?\ \mathtt{a}_1 \oplus \mathtt{a}_2)$
  By inversion $\mathcal{M}, \Psi \cup \mathcal{L} \vdash \mathbb{I}_1 : \mathtt{a}_1$ and $\mathcal{M}, \Psi \cup \mathcal{L} \vdash \mathbb{I}_2 : \mathtt{a}_2$
  By induction hypothesis, $\mathtt{a}_1 \supseteq \llbracket \mathbb{C}, \mathbb{I}_1 \rrbracket^n_{\mathcal{M},\mathcal{L}}$ and $\mathtt{a}_2 \supseteq \llbracket \mathbb{C}, \mathbb{I}_2 \rrbracket^n_{\mathcal{M},\mathcal{L}}$
  By branch weakening, $(\mathcal{M}(b)?\ \mathtt{a}_1 \oplus \mathtt{a}_2) \supseteq \left(\mathcal{M}(b)?\ \llbracket \mathbb{C}, \mathbb{I}_1 \rrbracket^n_{\mathcal{M},\mathcal{L}} \oplus \llbracket \mathbb{C}, \mathbb{I}_2 \rrbracket^n_{\mathcal{M},\mathcal{L}}\right)$
  By definition of $\llbracket\rrbracket$, $\llbracket \mathbb{C}, (b?\ \mathbb{I}_1 + \mathbb{I}_2) \rrbracket_{\mathcal{M},\mathcal{L}} = \left(\mathcal{M}(b)?\ \llbracket \mathbb{C}, \mathbb{I}_1 \rrbracket^n_{\mathcal{M},\mathcal{L}} \oplus \llbracket \mathbb{C}, \mathbb{I}_1 \rrbracket^n_{\mathcal{M},\mathcal{L}}\right)$
  Thus, $(b?\ \mathtt{a}_1 + \mathtt{a}_2) \supseteq \llbracket \mathbb{C}, \mathbb{I}_1 + \mathbb{I}_2 \rrbracket^n_{\mathcal{M},\mathcal{L}}$
- Case $\mathcal{M}, \Psi \cup \mathcal{L} \vdash [\mathtt{f}] : (\Psi \cup \mathcal{L})(\mathtt{f})$.
  If $\mathtt{f} \in \mathtt{dom}(\mathcal{L})$, then $\llbracket \mathbb{C}, [\mathtt{f}] \rrbracket^n_{\mathcal{M},\mathcal{L}} = \mathcal{L}(\mathtt{f})$
  By reflexivity, $(\Psi \cup \mathcal{L})(\mathtt{f}) \supseteq \llbracket \mathbb{C}, [\mathtt{f}] \rrbracket^n_{\mathcal{M},\mathcal{L}}$.
  Else, $\mathtt{f} \in \mathtt{dom}(\Psi)$, then $\mathtt{f} \in \mathtt{dom}(\mathbb{C})$.
  By premise, $\Psi(\mathtt{f}) \supseteq \llbracket \mathbb{C}, \mathbb{C}(\mathtt{f}) \rrbracket^{n-1}_{\mathcal{M},\mathcal{L}}$
  By definition of $\llbracket\rrbracket$, $\llbracket \mathbb{C}, [\mathtt{f}] \rrbracket^n_{\mathcal{M}} = \llbracket \mathbb{C}, \mathbb{C}(\mathtt{f}) \rrbracket^{n-1}_{\mathcal{M},\mathcal{L}}$
  Thus, $\Psi(\mathtt{f}) \supseteq \llbracket \mathbb{C}, [\mathtt{f}] \rrbracket^n_{\mathcal{M},\mathcal{L}}$
- Case WF-WEAK on $\mathcal{M}, \Psi \cup \mathcal{L} \vdash \mathbb{I} : \mathtt{a}$
  By inversion, $\mathcal{M}, \Psi \cup \mathcal{L} \vdash \mathbb{I} : \mathtt{a}'$ and $\mathtt{a} \supseteq \mathtt{a}'$
  By IH, $\mathtt{a}' \supseteq \llbracket \mathbb{C}, \mathbb{I} \rrbracket^n_{\mathcal{M},\mathcal{L}}$
  By transitivity over $\supseteq$, $\mathtt{a} \supseteq \llbracket \mathbb{C}, \mathbb{I} \rrbracket^n_{\mathcal{M},\mathcal{L}}$

□

By applying the above theorem to every single procedure in the procedure heap, we can inductively increase the index of the approximation of the codeheap. Thus a well-formed codeheap is valid under any n-approximation.

Theorem 3.4.2 (Partial Correctness for Modules)

If $\mathcal{M}, \mathcal{L} \vdash \mathbb{C} : \Psi$, then for any index n, for every label $\mathtt{f} \in \mathsf{dom}(\mathbb{C})$, $\Psi(\mathtt{f}) \supseteq \llbracket \mathbb{C}, \mathbb{C}(\mathtt{f}) \rrbracket^n_{\mathcal{M},\mathcal{L}}$.

Pf. By induction on n.

- Case $n = 0$.
  Then $[\![\mathbb{C}, \mathbb{C}(\mathtt{f})]\!]^{0}_{\mathcal{M},\mathcal{L}} = \mathit{loop}$.
  By definition of $\mathcal{M}, \mathcal{L} \vdash \mathbb{C} : \Psi$, there exists $\mathtt{a}$, such that $\Psi(\mathtt{f}) = \mathtt{a}$
  By Strongest Action, $\mathtt{a} \supseteq \mathit{loop}$.

- If $n > 0$.
  Choose any label $\mathtt{f} \in \mathsf{dom}(\mathbb{C})$.
  By inversion on $\mathcal{M}, \mathcal{L}, \vdash \mathbb{C} : \Psi$, $\mathcal{M}, \mathcal{L} \cup \Psi \vdash \mathbb{C}(\mathtt{f}) : \Psi(\mathtt{f})$.
  By IH, $\forall \mathtt{f}'. \Psi(\mathtt{f}') \supseteq [\![\mathbb{C}, \mathbb{C}(\mathtt{f}')]\!]^{n-1}_{\mathcal{M},\mathcal{L}}$.
  By theorem 3.4.1, $\Psi(\mathtt{f}) \supseteq [\![\mathbb{C}, \mathbb{C}(\mathtt{f})]\!]^{n}_{\mathcal{M},\mathcal{L}}$
  Since $\mathtt{f}$ can be any label $\forall \mathtt{f} \in \mathsf{dom}(\mathbb{C}). \Psi(\mathtt{f}) \supseteq [\![\mathbb{C}, \mathbb{C}(\mathtt{f})]\!]^{n}_{\mathcal{M},\mathcal{L}}$.

□

The correctness proofs are the interesting result. Though, just to be thorough, we define the safety corollary, which shows how the partial correctness guarantees safety of the executing code:

Corollary 3.4.3 (Safety)

If $\mathcal{M}, \mathcal{L} \vdash (\mathbb{C}, \mathbb{I}, \mathbb{S})$, then $\mathbb{S} \in \mathsf{dom}([\![\mathbb{C}, \mathbb{I}]\!]^{n}_{\mathcal{M},\mathcal{L}})$.

Pf. Invert on the TOP rule, then use partial correctness theorems.

□

The above theorems show that our static semantics are sound for any machine $\mathcal{M}$ and a library of primitives $\mathcal{L}$. Thus we can use the static semantics in any machine to certify any module, and have a guarantee of partial correctness without doing any additional proofs.

3.5 Notation for Writing Down Actions

One of the problems of this thesis is to present actions and specifications in a concise and clear way. The type of actions is fairly complex, and generally requires lengthy definitions. We have already seen how we can shorten actions by using combinators to quickly combine several actions into one.

However, combinators do not help us shorten the actions that perform modifications on the particular state of the machines, and as state types can become quite complex, so are the definitions. To reduce the mess, we are using special notation to define these accesses.

The first major component of this notation is the partial state update, which may look like the following:

$$(\mathbb{S}.r1 := \mathbb{S}.r2.m5 + \mathbb{S}.r2.m3)$$

This notation defines an action which updates just a part of the state with the computation based on the starting state. In the cases where the state type has unique names, the prefixes could be dropped. For example if in the above statement, $r1$, $m5$, and $m3$ are unique names, the above could be written as

$$r1 := m5 + m3$$

Another notation we will use is that we will write down several state updates at once, for example

$$r1 := m5 + m3, r2 := 0$$

This would mean that parts of the state that are uniquely identified by $r1$ and $r2$ are updated with the appropriate values, while the rest are kept the same. This of course can create conflicts in that the same part of the state can be updated twice. We disallow such notations.

Yet another state update that we will use is the forgetting of a part of the state, for example

$$\bcancel{r1}$$

This indicates that the set of resulting states can include all states with all possible values for $r1$. Of course, this forgetting can not conflict with other updates of the state.

The last portion of notation is the precondition check, which has a pattern of something like

$$(r1 = 0)?$$

This action works nearly the same as the p? combinator. It makes the action invalid if the condition is not satisfied. The difference between this and the combinator is that this notation is combinable with the state updates we have described above.

A little syntactic sugar that we will use is the "if" combinator within this notation. For example, we could write

$$(r1 := 5, \text{if } r2 = 0 \text{ then } r3 := 5 \text{ else } r3 := 6 \text{ end if})$$

This sugar works by pushing all the outside updates into the combinator. The result of the above would be something like this:

$$(r2 = 0?\ (r1 := 5, r3 := 5) + (r1 := 5, r3 := 6))$$

Unfortunately, this notation is defined very informally, and in fact we do not use it in any of our formal Coq proofs, where we write out all our specifications in full. The notation is just used to simplify the presentation in the write-up. We hope that this notation can be converted fairly intuitively into the full specification when needed. Ideally, a complete specification language should be defined for writing down actions, or an already existing language, such as Spec#[4, 5], can be modified for this purpose.

3.6 Examples of Machine Definitions and Verification

To show that our system can be used to define and simulate actual languages, we will provide several examples of machines and certifications. The first machine will be an implementation of the MIPS architecture, one having a separated code heap, an approach commonly used to prevent reasoning over self-modifying code. Using this machine, we will show a complete certification of the Fibonacci function, and explain how the certification in our framework is different from the previous frameworks.

The second machine will be a MIPS machine without a separate code heap. Our framework flexible enough to support self-modifying code, and we will use this machine to show a quick example.

The third machine that we will demonstrate is a machine that implements IMP, a simple imperative language. This shows how our verification system allows us to perform verification on languages with high-level primitives such as loops and conditionals.

3.6.1 MIPS with a Fixed Code Heap

The first machine that we will demonstrate is the MIPS machine with the additional requirement that the code heap is fixed. Having fixed code heap is not a requirement of our verification system, but it simplifies the verification as information about the current state of the code is immutable.

$$\begin{array}{rlll}
(\textit{Codeheap}) & \mathbb{Z} & ::= & \{\mathtt{f} \leadsto w\}^* \\
(\textit{State}) & \mathbb{S} & ::= & (M, R, \mathtt{ip}) \\
(\textit{Memory}) & M & ::= & \{\mathtt{l} \leadsto w\ \}^* \\
(\textit{Registers}) & R & ::= & \{r_0 \leadsto 0\} \cup \{r_k \leadsto w\}^{k \in (1 \ldots 31)} \\
(\textit{Instruction Ptr}) & \mathtt{ip} & ::= & \mathtt{f} \\
(\textit{Labels}) & \mathtt{f}, \mathtt{l} & ::= & i \text{ (natural numbers)} \\
(\textit{Words}) & w & ::= & n \text{ (integers)} \\
(\textit{Operation}) & \iota & ::= & \mathtt{addu}\ r_d^*, r_s, r_t \mid \mathtt{addiu}\ r_d^*, r_s, w \mid \mathtt{lw}\ r_d^*, w, r_s \mid \mathtt{sw}\ r_t, w, r_s \\
 & & & \mid \mathtt{beq}\ r_s, r_d, \mathtt{f} \mid \mathtt{j\ f} \mid \mathtt{jal\ f} \mid \mathtt{jr}\ r_s \\
(\textit{Cond Expression}) & b & ::= & r_s = r_d
\end{array}$$

* register can not be r_0.

Operation if ι =	Action then $\mathcal{M}(\iota) = (\mathbb{Z}(\mathtt{ip}) = \iota)?$,
ADDU r_d, r_s, r_t	$r_d := (r_s + r_t), \mathtt{ip} := \mathtt{ip} + 4$
ADDIU r_d, r_s, w	$r_d := (r_s + w), \mathtt{ip} := \mathtt{ip} + 4$
LW r_d, w, r_t	$r_d := M(r_t + w), \mathtt{ip} := \mathtt{ip} + 4$
SW r_s, w, r_t	$M(r_t + w) := r_s, \mathtt{ip} := \mathtt{ip} + 4$
BEQ $r_s, r_d, \mathtt{f}$	$(r_s = r_d ?\ \mathtt{ip} := \mathtt{f} \oplus \mathtt{ip} := \mathtt{ip} + 4)$
J $\mathtt{f}$	$\mathtt{ip} := \mathtt{f}$
JAL $\mathtt{f}$	$r_{31} := (\mathtt{ip} + 4), \mathtt{ip} := \mathtt{f}$
JR r_s	$\mathtt{ip} := r_{31}$

$$\mathcal{M}.\Upsilon(r_s = r_d) \triangleq \lambda\mathbb{S}.\,\mathbb{S}.R(r_s) = \mathbb{S}.R(r_d)$$

Figure 3.8: MIPS Machine with Fixed Code

The speciication of the MIPS machine with the fixed code heap is given in Figure 3.8. The specifications are tailored precisely to create a machine that fits the required pair consisting of the state type and a set of instructions.

In this machine, the state consists of a triple of memory, registers, and an instruction pointer. The memory is represented as a partial mapping from addresses into integers. The registers consist of 31 general purpose registers, with the first one r_0 being reserved, its value fixed to 0, and there are no valid instructions that can assign to it. The instruction pointer is a label that points to a code heap ($\mathbb{Z}$) that is a constant in the machine (and not a part of the state). The fact that code heap is a constant means that this machine is uniquely tailored to a specific program.

The actions that correspond to instructions are quite self-describing, and are very easy to read due to our notation. The action of every instruction includes $(\mathbb{Z}(\mathtt{ip}) = \iota)?$. This restriction on the domain does two things: it guaratees that the instruction that we are considering is in fact the instruction that the machine should execute, and it also guarantees that only one instruction is valid at the same time, since only one instruction exists at the current location.

It is important to note that there are no special cases for instructions that affect control flow - in our verification system, the actions of the instructions only alter the state. We use the procedures to determine what the next instruction should be, and then we verify it against the machine to make sure that the code in the code heap matches our procedures.

To demonstrate how such verification works, we will certify an iterative implementation of the Fibonacci function, i.e. the function that returns the nth Fibonacci number. The code of this function is given in Figure 3.9. The function expects the parameter in register 1, and using registers 2-4 it uses a loop to compute the nth number, placing it in register 1, and then jumps to the address contained in regiser 31, the register that is used by convention to store return pointers.

The code itself consists of three basic blocks with associated labels. Fib sets the initial values, fibloop actually runs the computation, and then fibdone finalizes the result.

To certify this code using our verification system, we first need to translate the fib program into the meta-language program. While our verification system can not define this translation automatically for every language(as each language may be different), we can define the translation from MIPS basic blocks into procedures using the definition in Figure 3.10.

The gen-proc procedure converts every basic block into a procedure in the meta-language. Each

```
# r1 - parameter ( >= 0) - also result
# r2 - first,  r3 - second,  r4 - tmp
100   fib:      addiu r2, r0, 0
104             addiu r3, r0, 1
108             j     fibloop
112   fibloop:  beq   r1, r0, fibdone
116             addi  r4, r2, r3
120             addiu r2, r3, 0
124             addiu r3, r4, 0
128             addiu r1, r1, (-1)
132             j     fibloop
136   fibdone:  addiu r1, r2, 0
140             jr    r31
```

Figure 3.9: MIPS Code of the Fibonacci Function ($\mathbb{Z}$)

If $B =$	then gen-proc(B) =
$\mathtt{beq}\ r_s, r_d, \mathtt{f}; B'$	$\mathtt{beq}\ r_s, r_d, \mathtt{f};$ $(r_s = r_d?\ ([\mathtt{f}]) + (\text{gen-proc}(B')))$
$\iota; B'$	$\iota; \text{gen-proc}(B')$
$\mathtt{jal}\ \mathtt{f}; B'$	$\mathtt{jal}\ \mathtt{f}; [\mathtt{f}]; \text{gen-proc}(B')$
$\mathtt{j}\ \mathtt{f}$	$\mathtt{j}\ \mathtt{f}; [\mathtt{f}]$
$\mathtt{jr}\ r_s$	$\mathtt{jr}\ r_s$

Figure 3.10: Automatic Translation of MIPS Code into Meta-language

$\mathbb{C}_{fib}(\text{fib})$:	$\mathbb{C}_{fib}(\text{fibloop})$:	$\mathbb{C}_{fib}(\text{fibdone})$:
$\texttt{addiu}\ r_2, r_0, 0;$ $\texttt{addiu}\ r_3, r_0, 1;$ $[\text{fibloop}]$	$\texttt{beq}\ r_1, r_0, \texttt{fibdone};$ $\left(r_1 = r_0?\ ([\text{fibdone}]) + \left(\begin{array}{l} \texttt{addu}\ r_4, r_2, r_3; \\ \texttt{addiu}\ r_2, r_3, 0; \\ \texttt{addiu}\ r_3, r_4, 0; \\ \texttt{addiu}\ r_1, r_1, -1; \\ \texttt{j fibloop}; \\ [\text{fibloop}] \end{array} \right) \right)$	$\texttt{addiu}\ r_1, r_2, 0;$ $\texttt{jr}\ r_{31}$

Figure 3.11: Fibonacci Function Translated into Meta-language ($\mathbb{C}_{fib}$)

$$\begin{array}{rl} \Psi_{fib}(fib) & := ((\texttt{ip} = \texttt{fib})?, (r_1 >= 0)?, r_1 := fib(r_1), \cancel{r_2}, \cancel{r_3}, \cancel{r_4}, \texttt{ip} := r_{31}) \\ \Psi_{fib}(fibloop) & := \left(\begin{array}{l} (r_1 >= 0)?, \texttt{ip} = \texttt{fibloop}?, \\ \bigvee_m (r_2 = fib(m)?, r_3 = fib(m+1)?, r_1 := fib(m + r_1), \cancel{r_2}, \cancel{r_3}, \cancel{r_4}, \texttt{ip} := r_{31}) \end{array} \right) \\ \Psi_{fib}(fibdone) & := ((\texttt{ip} = \texttt{fibdone})?, r_1 := r_2, \texttt{ip} := r_{31}) \end{array}$$

Figure 3.12: Specifications for the Fibonacci Procedures

procedure will only include instructions that are present in the basic block. However, these basic blocks will make calls to other basic blocks, as can be seen in the example of the direct jump (`j f`). The resulting meta-language program will have procedures that are tail-recursive up to the indirect jump ($\texttt{jr}\ r_s$), which implies the end of the function in MIPS. This means that whenever the basic block ends on a direct jump, the procedure for the basic block will include a call into the next block. This means that whenever we will consider the action of the procedure corresponding to the basic block, we will have to consider its effects all the way to the end of the function. We will see how this is reflected in the verification soon.

The result of running this gen-proc on the fib mips code results in a meta-program given in Figure 3.11. This is what we will certify using our framework. The specifications for the procedures of the Fibonacci function are given in Figure 3.12.

To certify the procedures using the specifications in Ψ_{fib}, we will use our lemma, and generate the strongest specification for our code using `genspec`. These automatically generated strongest specifications are given in Figure 3.13.

To finalize the verification of the code, we just have to show that our specifications are weaker than the generated specifications:

Lemma 3.6.1 (fib certified)

$$\texttt{genspec}(\text{fib}): \begin{pmatrix} ((\mathbb{Z}(\texttt{ip}) = \texttt{addiu}\ r_2, r_0, 0)?, r_2 := r_0 + 0, \texttt{ip} := \texttt{ip} + 4) \circ \\ ((\mathbb{Z}(\texttt{ip}) = \texttt{addiu}\ r_3, r_0, 1)?, r_3 := r_0 + 1, \texttt{ip} := \texttt{ip} + 4) \circ \\ \Psi_{fib}(\text{fibloop-spec}) \end{pmatrix}$$

$$\texttt{genspec}(\text{fibdone}): \begin{pmatrix} ((\mathbb{Z}(\texttt{ip}) = \texttt{addiu}\ r_1, r_2, 0)?, r_1 := r_2 + 0, \texttt{ip} := \texttt{ip} + 4) \circ \\ ((\mathbb{Z}(\texttt{ip}) = \texttt{jr}\ r_{31})?, \texttt{ip} := r_{31}) \end{pmatrix}$$

$$\texttt{genspec}(\text{fibloop}): \begin{pmatrix} \begin{pmatrix} (\mathbb{Z}(\texttt{ip}) = \texttt{beq}\ r_1, r_0, \texttt{fibdone})?, \\ (\mathcal{M}(r_1 = r_0)?\ \texttt{ip} := \texttt{fibdone} \oplus \texttt{ip} := \texttt{ip} + 4) \end{pmatrix} \circ \\ \begin{pmatrix} \mathcal{M}(r_1 = r_0)?\ \Psi_{fib}(\text{fibdone-spec}) \oplus \begin{pmatrix} \begin{pmatrix} \mathbb{Z}(\texttt{ip}) = \texttt{addu}\ r_4, r_2, r_3)?, \\ r_4 := r_2 + r_3, \texttt{ip} := \texttt{ip} + 4 \end{pmatrix} \circ \\ \begin{pmatrix} (\mathbb{Z}(\texttt{ip}) = \texttt{addiu}\ r_2, r_3, 0)?, \\ r_2 := r_3 + 0, \texttt{ip} := \texttt{ip} + 4 \end{pmatrix} \circ \\ \begin{pmatrix} (\mathbb{Z}(\texttt{ip}) = \texttt{addiu}\ r_3, r_4, 0)?, \\ r_3 := r_4 + 0, \texttt{ip} := \texttt{ip} + 4 \end{pmatrix} \circ \\ \begin{pmatrix} (\mathbb{Z}(\texttt{ip}) = \texttt{addiu}\ r_1, r_1, -1)?, \\ r_1 := r_1 + (-1), \texttt{ip} := \texttt{ip} + 4 \end{pmatrix} \circ \\ \begin{pmatrix} (\mathbb{Z}(\texttt{ip}) = \texttt{j fibloop})?, \\ \texttt{ip} := \texttt{fibloop} \end{pmatrix} \circ \\ \Psi_{fib}(\text{fibloop-spec}) \end{pmatrix} \end{pmatrix} \end{pmatrix}$$

Figure 3.13: Automatically Generated Strong Action for Fibonacci Function

The translated fib program is well-specified ($\mathcal{M}_{CMIPS}, \emptyset \vdash \mathbb{C}_{fib} : \Psi_{fib}$)

Pf. We prove the following action weakenings:

$$\begin{aligned} &\Psi_{fib}(\text{fib}) \supseteq \texttt{genspec}(\text{fib-spec}) \\ &\Psi_{fib}(\text{fibloop}) \supseteq \texttt{genspec}(\text{fibloop-spec}) \\ &\Psi_{fib}(\text{fibdone}) \supseteq \texttt{genspec}(\text{fibdone-spec}) \end{aligned}$$

We do not show the actual proofs for above, as they are simple, but tedious.
By lemma 3.3.2, we conclude that

$$\begin{aligned} &\mathcal{M}_{CMIPS}, \Psi_{fib} \vdash \mathbb{C}_{fib}(\text{fib}) : \Psi_{fib}(\text{fib}) \\ &\mathcal{M}_{CMIPS}, \Psi_{fib} \vdash \mathbb{C}_{fib}(\text{fibloop}) : \Psi_{fib}(\text{fibloop}) \\ &\mathcal{M}_{CMIPS}, \Psi_{fib} \vdash \mathbb{C}_{fib}(\text{fibdone}) : \Psi_{fib}(\text{fibdone}) \end{aligned}$$

By CODE rule, $\mathcal{M}_{CMIPS}, \emptyset \vdash \mathbb{C}_{fib} : \Psi_{fib}$.

□

The verification above used a generated meta-program, which is similar to SCAP in that its specifications are tail-recursive to the end of the function. However, one benefit of using our framework is that our meta-programs can be organized differently. For example, another way to reason about the Fibonacci is to treat the fibloop as a separate block which does not include the behavior of fibdone, and make fibdone a part of the fib code.

$$\begin{array}{l|l} \text{fib:} & \text{fibloop:} \\ \hline & \texttt{beq}\ r_1, r_0, \texttt{fibdone}; \\ \begin{array}{l} \texttt{addiu}\ r_2, r_0, 0; \\ \texttt{addiu}\ r_3, r_0, 1; \\ [\text{fibloop}]; \\ \texttt{addiu}\ r_1, r_2, 0; \\ \texttt{jr}\ r_{31} \end{array} & \left(r_1 = r_0?\ \text{nil} + \left(\begin{array}{l} \texttt{addu}\ r_4, r_2, r_3; \\ \texttt{addiu}\ r_2, r_3, 0; \\ \texttt{addiu}\ r_3, r_4, 0; \\ \texttt{addiu}\ r_1, r_1, -1; \\ \texttt{j fibloop}; \\ [\text{fibloop}] \end{array} \right) \right) \end{array}$$

$$\begin{array}{rl} \Psi_{fib}(fib) & := (\texttt{ip} = \texttt{fib})?, (r_1 >= 0)?, r_1 := fib(r_1), \cancel{r_2}, \cancel{r_3}, \cancel{r_4}, \texttt{ip} := r_{31} \\ \Psi_{fib}(fibloop) & := \left(\begin{array}{l} (r_1 >= 0)?, \texttt{ip} = \texttt{fibloop}?, \\ \bigvee_m \left(\begin{array}{l} r_2 = fib(m)?, r_3 = fib(m+1)?, \\ r_1 := 0, r_2 := fib(m+r_1), r_3 := fib(m+n+r_1), \cancel{r_4}, \texttt{ip} := \texttt{fibdone} \end{array} \right) \end{array} \right) \end{array}$$

Figure 3.14: Alternate Translation of Fibonacci Code and its Spec

The meta-program that corresponds to this is given in Figure 3.14. Because the code of fibdone is now included in fib, there is no specification for fibdone as it is not needed. Because fibloop no longer includes the call to the fibdone procedure, its specification no longer includes the behavior of fibdone. Thus the specification of fibloop only includes the behavior of the loop, and not the code that follows. Given that the meta-program now looks different, we would need to regenerate the strongest specification of these blocks, and redo the proof.

Another effect of performing the verification in this way is that the verification of fib no longer depends on the code of fibloop, but only on its specification. This approach modularized the fib function into two separately verifiable components.

One of the benefits of having multiple ways to reason about the same program is that we are no longer obligated to treat regular jumps as tail-calls. We are allowed to separate our understanding of code from the code itself, something SCAP frameworks are not able to do. It is now our choice whether to consider a regular jump as a function call, a tail call, or even a return. It is our choice to think of some part of the block as a separate procedure. This in turn gives us great flexibility in how we can reason about our code, which is something that was not possible with previous, language-driven verification systems.

$$
\begin{array}{rll}
(\mathit{State}) & \mathbb{S} & ::= (M, R, \texttt{ip}) \\
(\mathit{Memory}) & M & ::= \{\texttt{l} \rightsquigarrow w\}^{*} \\
(\mathit{Registers}) & R & ::= \{r_0 \rightsquigarrow 0\} \cup \{r_k \rightsquigarrow w\}^{k \in (1 \ldots 31)} \\
(\mathit{Instruction\ Ptr}) & \texttt{ip} & ::= \texttt{f} \\
(\mathit{Labels}) & \texttt{f}, \texttt{l} & ::= i \text{ (natural numbers)} \\
(\mathit{Words}) & w & ::= n \text{ (integers)} \\
(\mathit{Operation}) & \iota & ::= \texttt{addu}\ r_d^{*}, r_s, r_t \mid \texttt{addiu}\ r_d^{*}, r_s, w \mid \texttt{lw}\ r_d^{*}, w, r_s \mid \texttt{sw}\ r_t, w, r_s \\
 & & \quad \mid \texttt{beq}\ r_s, r_d, \texttt{f} \mid \texttt{j f} \mid \texttt{jal f} \mid \texttt{jr}\ r_s \\
(\mathit{Cond.\ Expr.}) & b & ::= r_s = r_d \\
(\mathit{Encoding}) & \mathit{En} & ::= \iota \to w \text{ (encoding function)}
\end{array}
$$

* register can not be r_0.

Operation if $\iota =$	Action then $\mathcal{M}(\iota) = \boxed{(M(\texttt{ip}) = \mathit{En}(\iota))?}\ \circ$
$\texttt{addu}\ r_d, r_s, r_t$	$r_d := (r_s + r_t), \texttt{ip} := \texttt{ip} + 4$
$\texttt{addiu}\ r_d, r_s, w$	$r_d := (R(r_s) + w), \texttt{ip} := \texttt{ip} + 4$
$\texttt{lw}\ r_d, w, r_t$	$r_d := M(r_t + w), \texttt{ip} := \texttt{ip} + 4$
$\texttt{sw}\ r_s, w, r_t$	$M(r_t + w) := r_s, \texttt{ip} := \texttt{ip} + 4$
$\texttt{beq}\ r_s, r_d, \texttt{f}$	$(r_s = r_d ?\ \texttt{ip} := \texttt{f} \oplus \texttt{ip} := \texttt{ip} + 4)$
$\texttt{j f}$	$\texttt{ip} := \texttt{f}$
$\texttt{jal f}$	$r_{31} := (\texttt{ip} + 4), \texttt{ip} := \texttt{f}$
$\texttt{jr}\ r_s$	$\texttt{ip} := r_{31}$

$$\mathcal{M}.\Upsilon(r_s = r_d) \triangleq \lambda \mathbb{S}.\, \mathbb{S}.R(r_s) = \mathbb{S}.R(r_d)$$

Figure 3.15: MIPS machine

3.6.2 MIPS with Self-Modifying Code

To support self-modifying code, all we need to do is to push the actual code into the state of the machine, by understanding it as instructions encoded in memory. To do this, we assume that there is an encoding function which maps the instruction to an integer. Then each instruction, instead of checking the immutable code heap, checks that its encoding is in memory at the location pointed to by the instruction pointer.

The full formal definition of this MIPS architecture is given in Figure 3.15. The only difference from the machine in Figure 3.8 is that every operation in the language definition now checks the presence of instuction in the memory, rather than in the constant instruction store, which also means that the immutable code store $\mathbb{Z}$ is no longer present. This change from the CMIPS machine is highlighted by a box in the figure. However, this small change has serious consequences. Because in

```
.data   # data declaration section

100   new:     addi  r2, r2, 1

.text   # code declaration section

200   main:    beq   r2, r4, modify
204   target:  addiu r2, r4, 0

220   modify:  lw    r9, new, r0
224            sw    r9, target, r0
228            j     target
```

Figure 3.16: Single Instruction Replacement Code

the previous example the code heap was immutable, it was not necessary to include any information about the code in the specifications, nor was it possible to modify code in place. In this version of the machine, code modification is possible, albeit at the expense of having to include the presense of code in the specifications.

To show that this machine handles self-modifying code, we will show how to certify code that features a single instruction replacement. This code is given in Figure 3.16. The program defined by this code serves as a demonstration of self-modifying code, but is not very useful. It checks that r_2 and r_4 is equal. If they are, then the code jumps to modify, which replaces the next instruction with an increment of r_2, and then the code jump back the modified instruction, and executes it. The other possibility is if r_2 and r_4 are not equal, then it does not change the next instruction before executing it. Normally, this means that if $r_2 \neq r_4$, then we would copy the value in r_4 into r_2. However, if the instruction has already been modified by a previous run, then we would still have the increment r_2 in place. Thus, assuming that no other modification take place (and we will have to include that fact in specifications), there are three possibilities of execution of this code.

It is important to note that we can not automatically generate a meta-program from the code of this program. Because the code is self-modifying - the execution determines the actual program that will run. Thus we must generate the meta-program for self-modifying code by hand. For this small example, we can produce two possible meta-programs, one in Figure 3.17 and another in Figure 3.18.

The first meta-program is mostly intuitive. It shows that there are two choices that can be made:

```
main                                                  | modify
------------------------------------------------------+---------------------------
beq $r_2, r_4$, modify;                               |
($r_2 = r_4$? [modify] + nil);                        |
if $M(\texttt{ip}) = En(\texttt{addiu}\ r_2, r_4, 0)$ then | lw $r_9$, new, $r_0$;
    addiu $r_2, r_4, 0$                               | sw $r_9$, target, $r_0$;
else                                                  | j target
    addiu $r_2, r_2, 1$                               |
end if                                                |
```

Figure 3.17: Single Instruction Replacement Meta-program ($\mathbb{C}_{ir}$)

```
main                                                  | modify
------------------------------------------------------+---------------------------
beq $r_2, r_4$, modify;                               |
if $r_2 = r_4$ then                                   |
    [modify];                                         |
    addiu $r_2, r_2, 1$                               |
else                                                  | lw $r_9$, new, $r_0$;
    if $M(\texttt{ip}) = En(\texttt{addiu}\ r_2, r_4, 0)$ then | sw $r_9$, target, $r_0$;
        addiu $r_2, r_4, 0$                           | j target
    else                                              |
        addiu $r_2, r_2, 1$                           |
    end if                                            |
end if                                                |
```

Figure 3.18: Single Instruction Replacement Meta-program (alternate) ($\mathbb{C}_{ira}$)

$\texttt{genspec(main)}:$

$$M(\texttt{ip}) = En(\texttt{beq}\ r_2, r_4, \texttt{modify})?, (r_2 = r_4?\ \texttt{ip} := \texttt{modify} \oplus \texttt{ip} := \texttt{ip} + 4) \circ$$
$$(r_2 = r_4?\ \Psi(\texttt{modify}) \oplus id) \circ$$
$$\left(M(\texttt{ip}) = En(\texttt{addiu}\ r_2, r_4, 0)? \begin{pmatrix} M(\texttt{ip}) = En(\texttt{addiu}\ r_2, r_4, 0)?, \\ r_2 := r_4 + 0, \texttt{ip} := \texttt{ip} + 4 \end{pmatrix} \oplus \begin{pmatrix} M(\texttt{ip}) = En(\texttt{addiu}\ r_2, r_2, 1)?, \\ r_2 := r_2 + 1, \texttt{ip} := \texttt{ip} + 4 \end{pmatrix} \right)$$

$\texttt{genspec(modify)}:$

$$(M(\texttt{ip}) = En(\texttt{lw}\ r_9, \texttt{new}, r_0)?, r_9 := M(\texttt{new}), \texttt{ip} := \texttt{ip} + 4) \circ$$
$$(M(\texttt{ip}) = En(\texttt{sw}\ r_9, \texttt{target}, r_0)?, M(\texttt{target}) := r_9, \texttt{ip} := \texttt{ip} + 4) \circ$$
$$(M(\texttt{ip}) = En(\texttt{j target})?, \texttt{ip} := \texttt{target})$$

Figure 3.19: Automatically Generated Strongest Specification

to run modify or not, and then there is a choice of the instruction to be executed. Thus, this meta-program suggests that there are four possibilities, when in fact one of them is impossible. We can not execute `modify` and then have the move instruction at 204. This case will be shown to be impossible by any valid specification. The alternate version of the program makes the impossibility explicit, by removing this case outright. The choice of the meta-program to be used is left to the programmer. To stay a little closer to the code, we decided to follow through with the first version.

Although we can not longer automatically generate meta-programs, the automatic strongest specification generation remains valid. If we run it on $\mathbb{C}_{ir}$, we would get the specification listed in Figure 3.19. Since the `modify` block is non-recursive, the result is a complete and valid specification all by itself.

However, this specification hides a lot of details. The precondition of this specification is not easy to compute, nor does it figure in the contents of $M(\texttt{new})$, making it impossible to compute ahead of time that if $r_2 = r_4$ then the effect of this code would be an increment of r_2. Thus, to be more clear, we can craft a weaker, yet much cleaner, specification by hand. A possible specification, called Ψ_{ir}, is given in Figure 3.20.

This weaker specification is a lot easier to read. The requirements on instructions located in memory (except instruction at 204) are brought upfront. There is an assumption of what is located at address 100. The flow of reasoning is much cleaner. For example, if the registers are equal or the instruction was copied over already then the only choice is to increment r_2 and copy the

$\Psi_{ir}(\texttt{main})$	$\Psi_{ir}(\texttt{modify})$
$M(100) = En(\texttt{addiu}\ r_2, r_2, 1)?$ $M(200) = En(\texttt{beq}\ r_2, r_4, \texttt{modify})?$ $M(220) = En(\texttt{lw}\ r_9, \texttt{new}, r_0)?$ $M(224) = En(\texttt{sw}\ r_9, \texttt{target}, r_0)?$ $M(228) = En(\texttt{j target})?$ $\texttt{ip} = \texttt{main}?$ $\cancel{r_9}, \texttt{ip} := 208,$ $\left(\begin{array}{l} \texttt{if}\ \ r_2 = r_4 \vee M(204) = M(100)\ \ \texttt{then} \\ \quad M(204) := M(100), r_2 := r_2 + 1 \\ \texttt{else} \\ \quad (M(204) := En(\texttt{addiu}\ r_2, r_4, 0))? \\ \quad r_2 := r_4 \end{array}\right)$	$(M(220) = \texttt{lw}\ r_9, \texttt{new}, r_0)?$ $(M(224) = \texttt{sw}\ r_9, \texttt{target}, r_0)?$ $(M(228) = \texttt{j target})?$ $(\texttt{ip} = 220)?$ $\cancel{r_9}, M(204) := M(100), \texttt{ip} := 204$

Figure 3.20: Single Instruction Replacement Spec (Ψ_{ir})

instruction alread if it was not already. However, if the registers are not equal, then we must check that the instruction is a move of r_4 to r_2, as our meta-program did not handle other possibilities. The specification also shows possible destruction of the r_9, and the fact that no matter what happens, the ip will be set at 208 upon completion of the program. The specification for modify is similar to the actual specification, except that we bring the preconditions about the code upfront, and we merge the load and store into a single copy and destruction of r_9.

To show that this specification is valid, we would simply need to show that $\Psi_{ir}(\texttt{main}) \supseteq \texttt{genspec}(\mathbb{C}_{ir}(\texttt{main}))$ as usual.

3.6.3 Simple Imperative Machine

To show that our system supports more than assembly, we also define a high-level language, which is a version of Reynold's Simple Imperative Language, IMP, in terms of our semantics. The formal definition of the actual language is given in Figure 3.21.

The model presented in the figure, shows IMP as a machine definition ($\mathcal{M}_{IMP}$) compatible with our meta-machine. The interesting bit here is that in this version, IMP only has one operation - the assignment. The loops, conditional, and sequences are all provided by the meta-machine.

However, we are not used to IMP programs being represented in such a strange manner. To alleviate this issue, we have defined a regular program, given by the construction r. This construction features all the usual definitions of IMP.

$$
\begin{array}{rll}
(\mathit{State}) & \mathbb{S} & ::= (M) \\
(\mathit{Memory}) & M & ::= \{\mathtt{v} \rightsquigarrow w\}^* \\
(\mathit{Operation} & \iota & ::= \mathtt{v} := \mathtt{e} \\
(\mathit{Program}) & r & ::= \mathtt{skip} \mid r;r \mid \mathtt{v} := \mathtt{e} \mid \mathtt{if}\ b\ \mathtt{then}\ r_1\ \mathtt{else}\ r_2 \mid \mathtt{while}\ b\ \mathtt{do}\ r' \\
(\mathit{Expression}) & \mathtt{e} & ::= \mathtt{n}(\text{integers}) \mid \mathtt{v} \mid \mathtt{e}_1 + \mathtt{e}_2 \\
(\mathit{Condition}) & b & ::= \mathsf{True} \mid \mathsf{False} \mid b \wedge b \mid \neg b \mid \mathtt{e}_1 = \mathtt{e}_2 \mid \mathtt{e}_1 < \mathtt{e}_2 \\
(\mathit{Variables}) & \mathtt{v} & ::= (\text{strings})
\end{array}
$$

Operational Semantics:

$$\mathcal{M}(\mathtt{v} := \mathtt{e}) \triangleq M(\mathtt{v}) := \mathit{eval}_M(\mathtt{e})$$

where

$$\mathit{eval}_M(\mathtt{e}) \triangleq \begin{cases} n & \text{if } \mathtt{e} = n \\ M(\mathtt{v}) & \text{if } \mathtt{e} = \mathtt{v} \\ \mathit{eval}_M(\mathtt{e}_1) + \mathit{eval}_M(\mathtt{e}_2) & \text{if } \mathtt{e} = \mathtt{e}_1 + \mathtt{e}_2 \end{cases}$$

$$\mathit{evalc}_M(b) \triangleq \begin{cases} \mathsf{True} & \text{if } b = \mathsf{True} \\ \mathsf{False} & \text{if } b = \mathsf{False} \\ \mathit{evalc}_M(b_1) \wedge \mathit{evalc}_M(b_2) & \text{if } b = b_1 \wedge b_2 \\ \neg \mathit{evalc}_M(b') & \text{if } b = \neg b' \\ \mathit{eval}_M(\mathtt{e}_1) = \mathit{eval}_M(\mathtt{e}_2) & \text{if } b = (\mathtt{e}_1 = \mathtt{e}_2) \\ \mathit{eval}_M(\mathtt{e}_1) < \mathit{eval}_M(\mathtt{e}_2) & \text{if } b = (\mathtt{e}_1 < \mathtt{e}_2) \end{cases}$$

Figure 3.21: Simple Imperative Language (IMP)

if $r =$	then $\mathit{conv}(r, \mathbb{C}) =$
`skip`	$(\mathtt{nil}, \mathbb{C})$
$r_1; r_2$	$\mathtt{let}\ (\mathbb{I}_1, \mathbb{C}_1) := \mathit{conv}(r_1, \mathbb{C})\mathtt{in}$ $\mathtt{let}\ (\mathbb{I}_2, \mathbb{C}_2) := \mathit{conv}(r_2, \mathbb{C}_1)\mathtt{in}$ $(\mathbb{I}_1; \mathbb{I}_2, \mathbb{C}_2)$
$(\mathtt{v} := \mathtt{e})$	$(\mathtt{v} := \mathtt{e}, \mathbb{C})$
$\mathtt{if}\ b\ \mathtt{then}\ r_1\ \mathtt{else}\ r_2$	$\mathtt{let}\ (\mathbb{I}_1, \mathbb{C}_1) := \mathit{conv}(r_1, \mathbb{C})\mathtt{in}$ $\mathtt{let}\ (\mathbb{I}_2, \mathbb{C}_2) := \mathit{conv}(r_2, \mathbb{C}_1)\mathtt{in}$ $(((\mathit{evalc}_M(b) = \mathsf{True})?\ \mathbb{I}_1 + \mathbb{I}_2), \mathbb{C}_2)$
$\mathtt{while}\ b\ \mathtt{do}\ r_1$	$\mathtt{let}\ (\mathbb{I}_1, \mathbb{C}_1) := \mathit{conv}(r_1, \mathbb{C})\ \mathtt{in}$ $([\mathit{loop}], \mathbb{C} \cup \{\mathit{loop} \rightsquigarrow ((\mathit{evalc}_M(b) = \mathsf{True})?\ (\mathbb{I}_1; [\mathit{loop}]) + \mathtt{nil})\})$

Figure 3.22: Conversion of IMP Programs into Meta-programs

Regular IMP	Converted Program	
$n := 15;$ $first := 0;$ $second := 1;$ $while(n > 0)($ $tmp := first + second;$ $first := second;$ $second := tmp;$ $n := n - 1);$ $result := first$	`fib:` $n := 15;$ $first := 0;$ $second := 1;$ $[fibloop];$ $result := first$	`fibloop:` if $evalc(n > 0)$ = True then $tmp := first + second;$ $first := second;$ $second := tmp;$ $n := n - 1;$ $[fibloop]$ end if

Figure 3.23: Fibonacci written in IMP and converted to $\mathcal{M}_{IMP}$

However, to reason about these programs, they have to be converted into programs for $\mathcal{M}_{IMP}$ by an algorithm defined in Figure 3.22. The conversion is quite intuitive. Skips get converted into empty procedures. Sequences get converted into sequences. The if statement gets converted into a branch. The only interesting case is the while. The entire while is simply replaced with a call to a fresh procedure in the codeheap $\mathbb{C}$, marked with a fresh label *loop*. The procedure contains a branch, which is the test portion of the while loop. If the test succeeds, then the left side contains the body of the while loop followed by the recursive call into the loop. If the test fails, the right side contains an empty procedure, meaning that failing the test exits the loop procedure. Thus while loops are onverted into recursive procedures.

For example, Figure 3.23 shows an implementation of Fibonacci in regular IMP code on the left side. If we apply our conversion algorithm, we end up with the procedures suitable for verification on $\mathcal{M}_{IMP}$.

However, we may question that fact the if we certify the converted program, we are actually certifying an IMP program. We will give an argument for this in section 3.6.5.

3.6.4 Other Machines and Languages

It is possible to express any language that has small step operational semantics in a way that corresponds to our machine definition. However, this does not guarantee that the programs written in those languages will be as clear when converted to our meta-language. Most procedural code expressed in terms of our meta-machines will be similar to the original code, as our machine provides procedural semantics. However, if we would try to express a functional language in our semantics, it might not be simple or pretty. First class continuations do not quite fit the pattern of our meta language.

We have considered extending our meta-language with lambda and application, which has produced promising results for using this system to apply Hoare-logic to functional languages, but this avenue has not been pursued as it was a direction that did not correspond to our immediate research interests.

Another directions that was studied in parallel with this research is the ability to reason about programs with code pointers. While this meta-machine does not include any explicit support for reasoning with code pointers, it does not preclude it. Other approaches for code pointer support, such as those used in the XCAP[35] framework, may also be introduced into our verification framework.

3.6.5 Adequacy

One last thing that we need to address in our approach to verification is the question of adequacy. The main issue is whether our machines, which are reasoned about via the meta-machine, are equivalent to the real machines. If we can not make the claim that our verification approach is equivalent to actual languages then our verification system is, arguably, useless.

There are several ways to address the issue. The easiest one is to sweep it under the rug. Our definition of machine semantics is no different than any other machine semantics. Most operational semantics are not shown to be adequate in any formal way, but are simply stated without any guarantee of correspondence to the actual physical machines or languages. A person has to check whether the model corresponds to his understanding of the actual machine. Since, our approach simply requires stating all semantics as transition actions rather than inductive definitions, the model may appear different, but a person can still check that this model corresponds to the actual machine. Just because our models look unusual is not a reason to dismiss them.

The second approach that can be taken is to formally show that the ”natural” machine is equivalent to one described as a transition system. Then it is possible to make an argument that the original program using the original semantics would produce the same result as translated programs using transition semantics.

For example, to argue that $\mathcal{M}_{IMP}$ is as good a specification of IMP as the original specification, we can take a model of actual IMP, given in Figure 3.24, and show the following lemma.

$$\frac{}{(M, \mathtt{skip}) \Downarrow M} \qquad \frac{(M, r_1) \Downarrow M'' \qquad (M'', r_2) \Downarrow M'}{(M, r_1; r_2) \Downarrow M'}$$

$$\frac{eval_M(\mathtt{e}) = x}{(M, \mathtt{v} := \mathtt{e}) \Downarrow M[\mathtt{v}/x]}$$

$$\frac{evalc_M(b) = \mathsf{True} \qquad (M, r_1) \Downarrow M'}{(M, \mathtt{if}\ b\ \mathtt{then}\ r_1\ \mathtt{else}\ r_2) \Downarrow M'} \qquad \frac{evalc_M(b) = \mathsf{False} \qquad (M, r_2) \Downarrow M'}{(M, \mathtt{if}\ b\ \mathtt{then}\ r_1\ \mathtt{else}\ r_2) \Downarrow M'}$$

$$\frac{evalc_M(b) = \mathsf{True} \qquad (M, r'; \mathtt{while}\ b\ \mathtt{do}\ r') \Downarrow M'}{(M, \mathtt{while}\ b\ \mathtt{do}\ r') \Downarrow M'} \qquad \frac{evalc_M(b) = \mathsf{False}}{(M, \mathtt{while}\ b\ \mathtt{do}\ r') \Downarrow M}$$

$$\text{if none of the rules above apply, } (M, r) \Downarrow \textbf{crash}$$

Figure 3.24: Standard Definition of Operational Semantics of IMP

Lemma 3.6.2 ($\mathcal{M}_{IMP}$ adequacy)

For any IMP program r, and any starting codeheap $\mathbb{C}_0$, such that $conv(r, \mathbb{C}_0) = (\mathbb{I}, \mathbb{C})$,

$$M' \in ([\![(\mathbb{C}, \mathbb{I})]\!]\, M) \text{ if and only if } ((M, r) \Downarrow M')$$

$$\text{and}$$

$$M \notin \mathtt{dom}([\![\mathbb{C}, \mathbb{I}]\!]) \text{ if and only if } ((M, r) \Downarrow \textbf{crash})$$

We will not show the proof, as it is not interesting, and just amounts to a simulation argument. However, we do want to consider the implications of having such a lemma. This lemma guarantees that if we take an IMP program r, convert it into $\mathbb{C}, \mathbb{I}$, and find some specification $\mathtt{a}$, such that it is sound $\mathtt{a} \supseteq [\![\mathbb{C}, \mathbb{I}]\!]$, then for any $M \in \mathtt{dom}(\mathtt{a})$, we will know that $(M, r) \Downarrow$ **crash** is not possible, and if there is a M' such that $(M, r) \Downarrow M'$, the it must be the case that $M' \in (\mathtt{a}\ M)$. Thus we have shown that our specifications are valid for the original IMP as well.

If we are being pedantic, this requirement of formally showing adequacy seems to be problematic, as these arguments are not simple. But consider that

- Not every machine or language that is defined has a ”natural” semantics that is different from ours. Since our system is built to handle domain-specific languages/machines, most of those languages have no existing specification. Thus there is no reason why giving specification using our model is a worse approach.

- Because our system is designed for verifying with multiple abstract machines, most intermediate machines are used as a way to specify and check specifications. The actual program

$\mathbb{C}_{main}(\texttt{main}):$	$\Psi_{main}(\texttt{main})$
$\texttt{sw}\ r_{31}, 0, r_{17};$ $\texttt{addiu}\ r_1, r_0, 13;$ $\texttt{jal fib};$ $\texttt{lw}\ r_{31}, 0, r_{17};$ $\texttt{jr}\ r_{31}$	$(\texttt{ip} = \texttt{main})?$ $r_1 := fib(13), \cancel{r_2 \ldots r_{16}}, \cancel{\mathcal{M}(r_{17})}, \texttt{ip} := r_{31}$

Figure 3.25: Code and Spec of the Main Module

is verified to run on a specific concrete machine, and it is important that only that specific machine is equivalent to the real one. The other machines are only used for reasoning, and do not need to correspond to any actual languages.

Thus, we are not worried that adequacy presents a serious issue to our framework.

3.7 Certified Modules and Linking

When we verify code, we always produce a certified code module given by

$$\mathcal{M}, \mathcal{L} \vdash \mathbb{C} : \Psi$$

This definition means that the procedure heap $\mathbb{C}$ is verified under the specification heap Ψ, assuming the correctness of stubs in the library $\mathcal{L}$. The purpose of this section is to show how to link modules so that we can correctly replace the stubs in $\mathcal{L}$ with actual functions.

First, let us give an example. We have already created a MIPS function fib, which computes the Fibonacci number. Suppose that we currently do not know how to implement it, but assume that it has the following specification:

$$\mathcal{L}_{fiblib} := \{fib \rightsquigarrow ((\texttt{ip} = fib)?, (r_1 > 0)?, r_1 := Fib(r_1), \cancel{r_2 \ldots r_{16}}, \texttt{ip} := r_{31})\}$$

In this specification, we have assumed that the fib computes the number, as well as follows the most general calling convention in that it destroys registers 2-16, with the answer in register 1. Then we can write and specify function `main` that uses `fib` as shown in Figure 3.25. The result of such certification will be a certified module with the following definition:

$$\mathcal{M}_{MIPS}, \mathcal{L}_{fiblib} \vdash \mathbb{C}_{main} : \Psi_{main}$$

The *fib* function here is a stub, as we only assume its specification, and do not know the code. However, using this specification, we can show that the main module is well-formed.

Before we get into linking, we must make a brief detour and explain what it means to union codeheaps and specifications. It is easy to understand what a union is when the heaps are disjoint. However, when they are not disjoint, we will need to make sure that the matching labels have matching procedures and specifications. For this purpose, we have defined the following predicates that will be used in our linking lemmas.

$$\begin{aligned} \mathbb{C}_1 \perp \mathbb{C}_2 \quad &\triangleq \forall \mathtt{l} \in \mathtt{dom}(\mathbb{C}_1).\ (\mathtt{l} \notin \mathtt{dom}(\mathbb{C}_2) \vee \mathbb{C}_1(\mathtt{l}) = \mathbb{C}_2(\mathtt{l})) \\ \Psi_1 \perp \Psi_2 \quad &\triangleq \forall \mathtt{l} \in \mathtt{dom}(\Psi_1).\ (\mathtt{l} \notin \mathtt{dom}(\Psi_2) \vee \Psi_1(\mathtt{l}) = \Psi_2(\mathtt{l})) \end{aligned}$$

Now we can define the general theorem for linking of two modules.

Theorem 3.7.1 (Linking)

$$\frac{\mathcal{M}, \mathcal{L}_1 \vdash \mathbb{C}_1 : \Psi_1 \quad \mathcal{M}, \mathcal{L}_2 \vdash \mathbb{C}_2 : \Psi_2 \quad \mathbb{C}_1 \perp \mathbb{C}_2 \quad \mathcal{L}_1 \perp \Psi_2 \quad \mathcal{L}_2 \perp \Psi_1 \quad \mathcal{L}_1 \perp \mathcal{L}_2}{\mathcal{M}, ((\mathcal{L}_1 \cup \mathcal{L}_2) \setminus (\Psi_1 \cup \Psi_2)) \vdash \mathbb{C}_1 \cup \mathbb{C}_2 : \Psi_1 \cup \Psi_2} \ (\textsc{link})$$

Pf.

1. By inversion, $\forall \mathtt{l} \in \mathbb{C}_1 . \mathcal{M}, \mathcal{L}_1 \cup \Psi_1 \vdash \mathbb{C}_1(\mathtt{l}) : \Psi_1(\mathtt{l})$
2. By inversion, $\forall \mathtt{l} \in \mathbb{C}_2 . \mathcal{M}, \mathcal{L}_2 \cup \Psi_2 \vdash \mathbb{C}_2(\mathtt{l}) : \Psi_2(\mathtt{l})$
3. By lemma 3.7.3, $\forall \mathtt{l} \in \mathbb{C}_1 . \mathcal{M}, (\mathcal{L}_1 \cup \Psi_1 \cup \mathcal{L}_2 \cup \Psi_2) \vdash \mathbb{C}_1(\mathtt{l}) : \Psi_1(\mathtt{l})$
4. By lemma 3.7.3, $\forall \mathtt{l} \in \mathbb{C}_2 . \mathcal{M}, (\mathcal{L}_1 \cup \Psi_1 \cup \mathcal{L}_2 \cup \Psi_2) \vdash \mathbb{C}_2(\mathtt{l}) : \Psi_2(\mathtt{l})$
5. Since $\mathbb{C}_1$ and $\mathbb{C}_2$ are disjoint and Ψ_1 and Ψ_2 are disjoint, $\forall \mathtt{l} \in (\mathbb{C}_1 \cup \mathbb{C}_2) . \mathcal{M}, (\mathcal{L}_1 \cup \Psi_1 \cup \mathcal{L}_2 \cup \Psi_2) \vdash (\mathbb{C}_1 \cup \mathbb{C}_2)(\mathtt{l}) : (\Psi_1 \cup \Psi_2)(\mathtt{l})$
6. By $\textsc{wf-code}$, $\mathcal{M}, ((\mathcal{L}_1 \cup \mathcal{L}_2 \cup \Psi_1 \cup \Psi_2) \setminus (\Psi_1 \cup \Psi_2)) \vdash (\mathbb{C}_1 \cup \mathbb{C}_2) : (\Psi_1 \cup \Psi_2)$
7. Which simplifies to $\mathcal{M}, ((\mathcal{L}_1 \cup \mathcal{L}_2) \setminus (\Psi_1 \cup \Psi_2)) \vdash (\mathbb{C}_1 \cup \mathbb{C}_2) : (\Psi_1 \cup \Psi_2)$

□

The linking theorem shows that we can take two certified modules, one with code heap $\mathbb{C}_1$, the other with code heap $\mathbb{C}_2$, and from them produce a new certified module that contains the code from both code heaps. The specifications of this module is simply the union of specifications. The new stub library is the union of the libraries of the two modules minus those stubs that are now provided some procedure of the linked module. In other words, module 1 may supply procedures that fill in stubs for module 2, and vice-versa. Only those stubs that were not filled in by either module remain as stubs in the new module.

Although the theorem above is very general, it requires a lot of information, and thus could be somewhat hard to understand. To simplify the linking process for common scenarios, we provide the following corollary.

Corollary 3.7.2 (Simple Library Linking)

$$\frac{\mathcal{M}, \emptyset \vdash \mathbb{C}_L : \Psi_L \qquad \mathcal{M}, \Psi_L \vdash \mathbb{C}_C : \Psi_C \qquad \mathtt{dom}(\mathbb{C}_L) \cap \mathtt{dom}(\mathbb{C}_C) = \emptyset}{\mathcal{M}, \emptyset \vdash \mathbb{C}_L \cup \mathbb{C}_C : \Psi_L \cup \Psi_C} \;(\textsc{lib-link})$$

Pf. A special case of the linking theorem where $\mathcal{L}_1 = \emptyset$ and $\mathcal{L}_2 = \Psi_1$.

□

The above corollary assumes that module with the code heap $\mathbb{C}_L$ does not have any stubs, and that module $\mathbb{C}_C$ has stubs precisely matched by the first module. The result is just the union of two codeheaps and two specifications, with no stubs. More practically, we can imagine $\mathbb{C}_L$ as a standard library, and $\mathbb{C}_C$ as the code that uses the standard library, and thus the union of the two result in a complete program.

One issue with these linking theorems is that they are excessively strict in that they require the specification heaps to be precisely equal. However, when dealing with modules developed modularly (and therefore separately), it is very likely that $\mathcal{L}_1$ may assume a slightly different specification than Ψ_2 for some label. The linking theorem rules such cases out.

Let us get back to our example, and link our main module with the actual fib library. The fib library is a certified module with the following definition

$$\mathcal{M}_{MIPS}, \emptyset \vdash \mathbb{C}_{fib} : \Psi_{fib}$$

The precise definitions of $\mathbb{C}_{fib}$ and Ψ_{fib} are in Figure 3.12, but we will reproduce the value of $\Psi_{fib}(\texttt{fib})$ here

$$(\texttt{ip} = \texttt{fib})?, (r_1 \geq 0)?, r_1 := Fib(r_1), \cancel{r_2}, \cancel{r_3}, \cancel{r_4}, \texttt{ip} := r_{31}$$

As you can see $\Psi_{fib}(\mathit{fib})$ is not the same as $\mathcal{L}_{fiblib}(\mathit{fib})$, and therefore $\Psi_{fib} \perp \Psi_{fiblib}$ is not true. Thus we can not link these libraries directly by using the linking lemma. To deal with this issue, we have provided several lemmas that allow us to strengthen the specifications of stubs to allow them to match.

Lemma 3.7.3 (Library Strengthening (Proc))

If $\mathcal{M}, \mathcal{L} \vdash \mathbb{I} : \mathtt{a}$, then for any $\mathcal{L}'$ such that $\forall \mathtt{l} \in \mathtt{dom}(\mathcal{L}). \mathcal{L}(\mathtt{l}) \supseteq \mathcal{L}'(\mathtt{l})$, $\mathcal{M}, \mathcal{L}' \vdash \mathbb{I} : \mathtt{a}$.

Pf. By induction on derivation of $\mathcal{M}, \mathcal{L} \vdash \mathbb{I} : \mathtt{a}$.

- Case $\overline{\mathcal{M}, \mathcal{L} \vdash \texttt{nil} : id}$ By WF-NIL, $\mathcal{M}, \mathcal{L}' \vdash \texttt{nil} : id$.

- Case $\overline{\mathcal{M}, \mathcal{L} \vdash \iota : \mathcal{M}(\iota)}$ By WF-OP, $\mathcal{M}, \mathcal{L}' \vdash \iota : \mathcal{M}(\iota)$.

- Case $\overline{\mathcal{M}, \mathcal{L} \vdash [\mathtt{l}] : \mathcal{L}(\mathtt{l})}$
  By WF-CALL, $\mathcal{M}, \mathcal{L}' \vdash [\mathtt{l}] : \mathcal{L}'(\mathtt{l})$.
  By premises, $\mathcal{L}(\mathtt{l}) \supseteq \mathcal{L}'(\mathtt{l})$.
  By WF-WEAK, $\mathcal{M}, \mathcal{L}' \vdash [\mathtt{l}] : \mathcal{L}(\mathtt{l})$.

- Case $\dfrac{\mathcal{M}, \mathcal{L} \vdash \mathbb{I}_1 : \mathtt{a}_1 \qquad \mathcal{M}, \mathcal{L} \vdash \mathbb{I}_2 : \mathtt{a}_2}{\mathcal{M}, \mathcal{L} \vdash \mathbb{I}_1 ; \mathbb{I}_2 : \mathtt{a}_1 \circ \mathtt{a}_2}$
  By IH, $\mathcal{M}, \mathcal{L}' \vdash \mathbb{I}_1 : \mathtt{a}_1$ and $\mathcal{M}, \mathcal{L}' \vdash \mathbb{I}_2 : \mathtt{a}_2$.
  By WF-SEQ, $\mathcal{M}, \mathcal{L}' \vdash \mathbb{I}_1 ; \mathbb{I}_2 : \mathtt{a}_1 \circ \mathtt{a}_2$

- Case $\dfrac{\mathcal{M}, \mathcal{L} \vdash \mathbb{I}_1 : \mathtt{a}_1 \qquad \mathcal{M}, \mathcal{L} \vdash \mathbb{I}_2 : \mathtt{a}_2}{\mathcal{M}, \mathcal{L} \vdash \mathbb{I}_1 + \mathbb{I}_2 : \mathtt{a}_1 \oplus \mathtt{a}_2}$
  By IH, $\mathcal{M}, \mathcal{L}' \vdash \mathbb{I}_1 : \mathtt{a}_1$ and $\mathcal{M}, \mathcal{L}' \vdash \mathbb{I}_2 : \mathtt{a}_2$.
  By WF-CHOICE, $\mathcal{M}, \mathcal{L}' \vdash \mathbb{I}_1 + \mathbb{I}_2 : \mathtt{a}_1 \oplus \mathtt{a}_2$

- Case $\dfrac{\mathcal{M}, \mathcal{L} \vdash \mathbb{I}_1 : \mathtt{a}_1 \qquad \mathcal{M}, \mathcal{L} \vdash \mathbb{I}_2 : \mathtt{a}_2}{\mathcal{M}, \mathcal{L} \vdash (b?\, \mathbb{I}_1 + \mathbb{I}_2) : (\mathcal{M}(b)?\, \mathtt{a}_1 \oplus \mathtt{a}_2)}$
  By IH, $\mathcal{M}, \mathcal{L}' \vdash \mathbb{I}_1 : \mathtt{a}_1$ and $\mathcal{M}, \mathcal{L}' \vdash \mathbb{I}_2 : \mathtt{a}_2$
  By WF-BRANCH, $\mathcal{M}, \mathcal{L}' \vdash (b?\, \mathbb{I}_1 + \mathbb{I}_2) : (\mathcal{M}(b)?\, \mathtt{a}_1 \oplus \mathtt{a}_2)$

- Case $\dfrac{\mathcal{M},\mathcal{L} \vdash \mathbb{I} : \mathtt{a}' \qquad \mathtt{a} \supseteq \mathtt{a}'}{\mathcal{M},\mathcal{L} \vdash \mathbb{I} : \mathtt{a}}$
  By IH, $\mathcal{M},\mathcal{L}' \vdash \mathbb{I} : \mathtt{a}'$.
  By WF-WEAK. $\mathcal{M},\mathcal{L}' \vdash \mathbb{I} : \mathtt{a}$.

□

Now that we can strengthen the library of any procedure, we can use the lemma to prove an equivalent property for the certified modules.

Theorem 3.7.4 (Library Strengthening)

If $\mathcal{M},\mathcal{L} \vdash \mathbb{C} : \Psi$, then for any $\mathcal{L}'$ s.t. $\forall \mathtt{l} \in \mathcal{L}.\, \mathcal{L}(\mathtt{l}) \supseteq \mathcal{L}'(\mathtt{l})$ and $\mathtt{dom}(\mathcal{L}') \cap \mathtt{dom}(\Psi) = \emptyset$, $\mathcal{M},\mathcal{L}' \vdash \mathbb{C} : \Psi$.

Pf.

By inversion on WF-CODE, $\forall \mathtt{l} \in \mathbb{C}.\, \mathcal{M},\mathcal{L} \cup \Psi \vdash \mathbb{C}(\mathtt{l}) : \Psi(\mathtt{l})$.
Then $\forall \mathtt{l} \in (\mathcal{L} \cup \Psi).\, (\mathcal{L} \cup \Psi)(\mathtt{l}) \supseteq (\mathcal{L}' \cup \Psi)(\mathtt{l})$.
By lemma 3.7.3, $\forall \mathtt{l} \in \mathbb{C}.\, \mathcal{M},\mathcal{L}' \cup \Psi \vdash \mathbb{C}(\mathtt{l}) : \Psi(\mathtt{l})$.
By WF-CODE, $\mathcal{M},\mathcal{L}' \vdash \mathbb{C}(\mathtt{l}) : \Psi(\mathtt{l})$.

□

This theorem would then allow us to strengthen the library stubs to match precisely the specification of functions provided by the other module. Then we can apply the linking lemma to put the modules together.

In our example, that means that we could use theorem 3.7.4 (simple library linking) and the fact that $\Psi_{fiblib}(\mathtt{fib}) \supseteq \Psi_{fib}(\mathtt{fib})$ to get the following result:

$$\frac{\mathcal{M}_{MIPS}, \mathcal{L}_{fiblib} \vdash \mathbb{C}_{main} : \Psi_{main}}{\mathcal{M}_{MIPS}, \Psi_{fib} \vdash \mathbb{C}_{main} : \Psi_{main}}$$

Now, we can easily apply the simple library linking lemma to get the following:

$$\frac{\mathcal{M}_{MIPS}, \emptyset \vdash \mathbb{C}_{fib} : \Psi_{fib} \qquad \mathcal{M}_{MIPS}, \Psi_{fib} \vdash \mathbb{C}_{main} : \Psi_{main}}{\mathcal{M}_{MIPS}, \emptyset \vdash (\mathbb{C}_{fib} \cup \mathbb{C}_{main}) : (\Psi_{fib} \cup \Psi_{main})}$$

And thus we have certified the entire program by linking main with fib.

This approach to linking is quite elegant. It does not require any special inference rules, and since the entire system is machine independent, it will work for any language whatsoever. However, this approach can only be used as long as all linked modules use the exact same machine. We will discuss the multi-machine approach to linking in the next chapter.

Chapter 4

Cross-Abstraction Linking

The previous chapter discussed the fact that our verification framework is parameterized over machine definitions, which allows a verifier to come up with a more abstract language to perform the verification of software. However, without an ability to refine this certification to the actual machine on which the program will run, the certification is not a useful one. How does one know that the high-level primitives that the program uses are actually the ones established by another library.

This chapter solves this problem by introducing the general approach to cross-abstraction (or cross-machine) linking, which is the main theoretical contribution of this work.

4.1 Linking Across Abstractions

Our technique for certified linking of modules that assume different levels of abstraction is refinement. Although the use of this technique is common, it is used in an ad-hoc way. Our aim is to give a thorough and consistent framework for using refinement with code certification. In this section, we will give a definition of certified refinement, show what it means, and then define a framework for using it with our system.

Refinement in the context of certified modules means the following. Suppose that we have a certified module that runs on an abstract machine $\mathcal{M}_A$.

$$\mathcal{M}_A, \mathcal{L}_A \vdash \mathbb{C}_A : \Psi_A$$

We would like to come up with a general way to convert this module into a certified module in machine $\mathcal{M}_C$. Clearly, there are many ways to this, including vacuous ways that convert an abstract module into non-sense. However, we are looking for a general description of such a process. This general description is what we call a certified refinement.

Definition 4.1.1 (Certified Refinement)

A certified refinement from machine $\mathcal{M}_A$ to machine $\mathcal{M}_C$ is a pair of relations $(T_{\mathbb{C}}, T_{\Psi})$ and a predicate Acc, such that the following holds

$$\frac{\mathcal{M}_A, \mathcal{L}_A \vdash \mathbb{C}_A : \Psi_A \quad T_{\mathbb{C}}(\mathbb{C}_A, \mathbb{C}_C) \quad T_{\Psi}(\Psi_A, \Psi_C) \quad T_{\Psi}(\mathcal{L}_A, \mathcal{L}_C) \quad Acc(\mathcal{M}_A, \mathcal{L}_A \vdash \mathbb{C}_A : \Psi_A)}{\mathcal{M}_C, \mathcal{L}_C \vdash \mathbb{C}_C : \Psi_C} \text{ REFINE}$$

for all $\mathbb{C}_A, \mathcal{L}_A, \Psi_A, \mathbb{C}_C, \mathcal{L}_C, \Psi_C$.

This definition is not a certified refinement itself, but rather a template for certified refinements in general. To define a particular refinement, we would have to provide a specific code relation $T_{\mathbb{C}}$, specification relation T_{Ψ}, and a predicate Acc that checks if a particular certified module can be refined. These predicates form a complete refinement if we can show that REFINE judgement rule holds for them, as required by the template. Once we have a particular refinement, meaning that we were supplied with $T_{\mathbb{C}}$, T_{Ψ}, Acc, and the proof of the rule, then we can use the rule to translate a particular abstract certified module into a concrete certified module.

For example, let's get back to our abstract certified module, $\mathcal{M}_A, \mathcal{L}_A \vdash \mathbb{C}_A : \Psi_A$. First, we will check that the certified module is accepted by the refinement, which means that it satisfies the Acc predicate. Then, we will need to find the $\mathbb{C}_C$, $\mathcal{L}_C$, and Ψ_C such that $T_{\mathbb{C}}(\mathbb{C}_A, \mathbb{C}_C)$, $T_{\Psi}(\mathcal{L}_A, \mathcal{L}_C)$, and $T_{\Psi}(\Psi_A, \Psi_C)$. Then by the the fact that we have a proof of the REFINE rule in our refinement, we instantly know that $\mathcal{M}_C, \mathcal{L}_C \vdash \mathbb{C}_C : \Psi_C$.

This definition is just a precise description of what refinements are in the system, and gives us a general way to encode refinements for certified software using our framework. In other words, the definition of refinement is just a template for what refinement is. The template does not provide any specific insight or additional benefits for creating refinements. However, by relying on such template, we can now develop certified refinements that are specific enough to significantly reduce the amount of work it takes to create them, yet general enough that they apply to large classes of machines. In the rest of the chapter, we will show several such refinements, and show how quickly one can use them to certify code at multiple levels of abstraction.

4.2 Refinement Generation

In section 4.1, we have given a general framework of establishing refinements of certified modules from one machine to another. However, the framework is just a template for defining refinements, and the hard work of actually finding the relations between code and specifications, as well as making sure that the REFINE rule holds for those relations is left entirely up to the specific cases.

However, we can save ourselves a lot of trouble if we make assumptions about the changes in the code between the machines. In this section, we will make the following assumptions, which will allow us to reduce the complexity of showing that the REFINE rule holds for the specific program. The assumptions we will make are the following:

- That refining an abstract program preserves the procedures that make up that program. This will allow us to make specific assumptions about the calls between procedures, and thus will allow us to prove refinements per procedure.

- An assumption that states that the refinement preserves the structure of the procedure, and that only the atomic operations may be replaced. This assumption allows us to prove properties of refinement by only considering the transformations of specific operations, and not the code.

- Finally, we will make the assumption that the operations themselves do not change. This means that the code of both the abstract and concrete programs must be exactly the same, although the semantics of the individual operations may change. This assumption will be useful as we will only need to consider the changes in semantics, rather than code.

As these assumptions build on top of one another, we will develop these simplifications in that order.

4.2.1 Per-Procedure Refinement

When considering the template for refinements, one may notice a bit of extra machinery that in many cases can be unnecessary. The relation $T_{\mathbb{C}}$ relates heaps of procedures, T_{Ψ} relates the specification heaps, and *Acc* is a predicate over the entire module. If we assume that the labels of the procedures and the labels of specifications do not change, then we can see that the relations over procedure and specification heaps can be reduced to relations over individual items in the heap, namely $T_{\mathbb{I}}$ - a relation over procedures, $T_{\mathtt{a}}$ - relation over actions, and $Acc_{\mathbb{I}}$ - a predicate over procedures, respectively. Making use of this restriction, we can define a new template for creating refinements with the following definition:

Definition 4.2.1 (Per-Procedure Refinement Template)

A per-procedure refinement from machine $\mathcal{M}_A$ to machine $\mathcal{M}_C$ is a pair of relations $(T_{\mathbb{I}}, T_{\mathtt{a}})$, and a predicate $Acc_{\mathbb{I}}$, such that for all $\mathcal{L}_A, \mathcal{L}_C, \mathbb{I}_A, \mathbb{I}_C, \mathtt{a}_A, \mathtt{a}_C$ the following holds:

$$\frac{\begin{array}{c}\mathcal{M}_A, \mathcal{L}_A \vdash \mathbb{I}_A : \mathtt{a}_A \qquad Acc_{\mathbb{I}}(\mathcal{M}_A, \mathcal{L}_A \vdash \mathbb{I}_A : \mathtt{a}_A) \\ \forall \mathtt{l} \in \mathtt{dom}(\mathcal{L}_A).\, T_{\mathtt{a}}(\mathcal{L}_A(\mathtt{l}), \mathcal{L}_C(\mathtt{l})) \qquad T_{\mathbb{I}}(\mathbb{I}_A, \mathbb{I}_C) \qquad T_{\mathtt{a}}(\mathtt{a}_A, \mathtt{a}_C)\end{array}}{\mathcal{M}_C, \mathcal{L}_C \vdash \mathbb{I}_C : \mathtt{a}_C}\ \text{PROC-REFINE}$$

□

This definition by itself can not be used as a refinement for the modules - it only works over individual procedures. However, if we are given $T_{\mathbb{I}}$, $T_{\mathtt{a}}$, $Acc_{\mathbb{I}}$ and the proof of the PROC-REFINE rule, then we can generate the predicates that will satisfy the template for refinements. We can do so by defining the $T_{\mathbb{C}}$, T_{Ψ} and *Acc* as follows:

$$\begin{array}{lcl}
T_{\mathbb{C}}(\mathbb{C}_A, \mathbb{C}_C) & := & \forall \mathtt{l} \in \mathtt{dom}(\mathbb{C}_C).\, T_{\mathbb{I}}(\mathbb{C}_A(\mathtt{l}), \mathbb{C}_C(\mathtt{l})) \wedge \mathtt{dom}(\mathbb{C}_A) = \mathtt{dom}(\mathbb{C}_C) \\
T_{\Psi}(\Psi_A, \Psi_C) & := & \forall \mathtt{l} \in \mathtt{dom}(\Psi_C).\, T_{\mathtt{a}}(\Psi_A(\mathtt{l}), \Psi_C(\mathtt{l})) \\
Acc(\mathcal{M}_A, \Psi'_A \vdash \mathbb{C}_A : \Psi_A) & := & \forall \mathtt{l} \in \mathtt{dom}(\mathbb{C}_C).\, Acc_{\mathbb{I}}(\mathcal{M}_A, \Psi'_A \cup \Psi_A \vdash \mathbb{C}_A(\mathtt{l}) : \Psi_A(\mathtt{l}))
\end{array}$$

From these definitions, it is easy to see that they simply represent a pointwise rebuilding of per-module predicates present in the definition of general refinement from per-procedure predicates present in the definition of the per-procedure refinement. Using these definitions together with the proof of PROC-REFINE, we can show that REFINE rule holds for these these automatically constructed predicates.

Theorem 4.2.2 (Certified Per-Procedure Refinement)

Given a per-procedure refinement consisting of $T_{\mathbb{I}}$, $T_{\mathtt{a}}$, $Acc_{\mathbb{I}}$, and the proof of PROC-REFINE, then the relations $T_{\mathbb{C}}$ and T_{Ψ}, and predicate Acc defined as a above, are a valid refinement (meaning that REFINE rule holds).

Pf. We need to show that the REFINE rule holds, which using the definitions above, has the following form:

$$\frac{\begin{array}{l}\mathcal{M}_A,\mathcal{L}_A \vdash \mathbb{C}_A : \Psi_A \\ \forall \mathtt{l} \in \mathtt{dom}(\mathbb{C}_C).\, Acc_{\mathbb{I}}(\mathcal{M}_A,\mathcal{L}_A \vdash \mathbb{C}_A(\mathtt{l}) : \Psi_A(\mathtt{l})) \\ \forall \mathtt{l} \in \mathtt{dom}(\mathbb{C}_C).\, T_{\mathbb{I}}(\mathbb{C}_A(\mathtt{l}),\mathbb{C}_C(\mathtt{l})) \wedge \mathtt{dom}(\mathbb{C}_A) = \mathtt{dom}(\mathbb{C}_C) \\ \forall \mathtt{l} \in \mathtt{dom}(\Psi_C).\, T_{\mathtt{a}}(\Psi_A(\mathtt{l}),\Psi_C(\mathtt{l})) \qquad \forall \mathtt{l} \in \mathtt{dom}(\mathcal{L}_C).\, T_{\mathtt{a}}(\mathcal{L}_A(\mathtt{l}),\mathcal{L}_C(\mathtt{l}))\end{array}}{\mathcal{M}_C,\mathcal{L}_C \vdash \mathbb{C}_C : \Psi_C}$$

By inversion on $\mathcal{M}_A,\mathcal{L}_A \vdash \mathbb{C}_A : \Psi_A$ we know that

$$\forall \mathtt{l} \in \mathtt{dom}(\mathbb{C}_A).\, \mathcal{M}_A,\mathcal{L}_A \cup \Psi_A \vdash \mathbb{C}_A(\mathtt{l}) : \Psi_A(\mathtt{l})$$

Pick a label $\mathtt{l} \in \mathtt{dom}(\mathbb{C}_A)$, and specialize our given conditions on that label. Then, we know that

$$\begin{array}{c}\mathcal{M}_A,\mathcal{L}_A \cup \Psi_A \vdash \mathbb{C}_A(\mathtt{l}) : \Psi_A(\mathtt{l}) \qquad Acc_{\mathbb{I}}(\mathcal{M}_A,\mathcal{L}_A \cup \Psi_A \vdash \mathbb{C}_A(\mathtt{l}) : \Psi_A(\mathtt{l})) \\ T_{\mathbb{I}}(\mathbb{C}_A(\mathtt{l}),\mathbb{C}_C(\mathtt{l})) \qquad T_{\mathtt{a}}(\Psi_A(\mathtt{l}),\Psi_C(\mathtt{l}))\end{array}$$

By using PROC-REFINE, we then are able to conclude that

$$\mathcal{M}_C,\mathcal{L}_C \cup \Psi_C \vdash \mathbb{C}_C(\mathtt{l}) : \Psi_C(\mathtt{l})$$

Since $\mathtt{l}$ was arbitrarily chosen, we know that

$$\forall \mathtt{l} \in \mathtt{dom}(\mathbb{C}_C).\, \mathcal{M}_C,\mathcal{L}_C \cup \Psi_C \vdash \mathbb{C}_C(\mathtt{l}) : \Psi_C(\mathtt{l})$$

Which by CODE rule gives us

$$\mathcal{M}_C,\mathcal{L}_C \vdash \mathbb{C}_C : \Psi_C$$

Which is exactly what we needed to show.

□

This theorem and this approach to defining refinement creates a bit of confusion. What the per-procedure refinement template does is allow an easier definition of the refinement. To show that the refinement is valid, the person defining the refinement needs to provide the $T_{\mathbb{I}}$, $T_{\mathtt{a}}$, $Acc_{\mathbb{I}}$ and the proof of PROC-REFINE. From this, our framework automatically generates the $T_{\mathbb{C}}$, T_{Ψ}, Acc and the proof of the REFINE rule for these relations.

To actually use the refinement, we will rely on the generated relations ($T_{\mathbb{C}}, T_{\Psi}, Acc$) to compare the code heaps and the specification heaps of the abstract and concrete modules. However, the proof of the REFINE rule, we get as a consequence of the PROC-REFINE. In other words, we still have the same work to link the modules, but from now on, we can save some work when creating a refinement -

which we think is the hard part in creation of the these multi-abstraction systems.

The savings will become more apparent when we will show refinements that are automatically generated from even less information.

4.2.2 Order-Preserving Refinement

We can make the creation of refinements even simpler if we consider the cases where the abstract code and the concrete code is related in a way where the order of related operations is preserved, or in other words, the structure of the procedures remain unchanged.

Under such restrictions, we can relate the code by relating the abstract operations with the concrete ones, and also relating the branch conditions. We refer to these relations as $T_\iota : \mathcal{M}_A.\Delta \to \mathcal{M}_C.\Delta \to Prop$, and $T_b : \mathcal{M}_A.\beta \to \mathcal{M}_C.\beta \to Prop$. These relations are strong enough to be a complete relation between procedures, and thus codeheaps. To produce the actual relation of the code heaps, we will reconstruct $T_{\mathbb{C}}$. To do so, we first construct the procedure relation ($T_{\mathbb{I}}$) that we have seen in the per-procedure refinement. To reconstruct it, we simply follow the structure of the procedure.

$$\frac{}{T_{\mathbb{I}}(\texttt{nil},\texttt{nil})} \qquad \frac{T_\iota(\iota_A,\iota_C)}{T_{\mathbb{I}}(\iota_A,\iota_C)} \qquad \frac{}{T_{\mathbb{I}}([\texttt{l}],[\texttt{l}])} \qquad \frac{T_{\mathbb{I}}(\mathbb{I}_{1A},\mathbb{I}_{1C}) \qquad T_{\mathbb{I}}(\mathbb{I}_{2A},\mathbb{I}_{2C})}{T_{\mathbb{I}}((\mathbb{I}_{1A};\mathbb{I}_{2A}),(\mathbb{I}_{1C};\mathbb{I}_{2C}))}$$

$$\frac{T_{\mathbb{I}}(\mathbb{I}_{1A},\mathbb{I}_{1C}) \qquad T_{\mathbb{I}}(\mathbb{I}_{2A},\mathbb{I}_{2C}) \qquad T_b(b_A,b_C)}{T_{\mathbb{I}}(((b_A ?\, \mathbb{I}_{1A} + \mathbb{I}_{2A})),((b_C ?\, \mathbb{I}_{1C} + \mathbb{I}_{2C})))} \qquad \frac{T_{\mathbb{I}}(\mathbb{I}_{1A},\mathbb{I}_{1C}) \qquad T_{\mathbb{I}}(\mathbb{I}_{2A},\mathbb{I}_{2C})}{T_{\mathbb{I}}((\mathbb{I}_{1A} + \mathbb{I}_{2A}),(\mathbb{I}_{1C} + \mathbb{I}_{2C}))}$$

What is important to note in this definition is that the labels of the abstract code must match the labels in the concrete code, and we use T_ι relation to match up the abstract and the concrete operations. The rest other rules simply reconstruct the structure of the procedures. Using this definition of $T_{\mathbb{I}}$, we can construct $T_{\mathbb{C}}$ just like we did in the per-procedure refinement, namely,

$$T_{\mathbb{C}}(\mathbb{C}_A,\mathbb{C}_C) \quad := \quad \forall \texttt{l} \in \texttt{dom}(\mathbb{C}_C).\, T_{\mathbb{I}}(\mathbb{C}_A(\texttt{l}),\mathbb{C}_C(\texttt{l})) \wedge \texttt{dom}(\mathbb{C}_A) = \texttt{dom}(\mathbb{C}_C)$$

The generated $T_{\mathbb{C}}$ relation makes it clear that the abstract program and the concrete program must be the same in structure - only operations are substituted one for another. Because the structure of the abstract and the concrete programs are the same, we no longer have to produce a proof of PROC-REFINE to construct a refinement. Instead, we will require that several properties defined in

$$\frac{T_\iota(\iota_A,\iota_C) \qquad T_{\mathsf{a}}(\mathsf{a}_A,\mathsf{a}_C) \qquad \mathsf{a}_A \supseteq \mathcal{M}_A(\iota_A)}{\mathsf{a}_C \supseteq \mathcal{M}_C(\iota_C)}\ \textsc{op-compat} \qquad\qquad \frac{}{\forall \mathsf{a}_A.\,\exists \mathsf{a}_C.T_{\mathsf{a}}(\mathsf{a}_A,\mathsf{a}_C)}\ \textsc{act-exist}$$

$$\frac{T_{\mathsf{a}}(\mathsf{a}_A,\mathsf{a}_C) \qquad T_{\mathsf{a}}(\mathsf{a}'_A,\mathsf{a}'_C) \qquad \mathsf{a}_A \supseteq \mathsf{a}'_A}{\mathsf{a}_C \supseteq \mathsf{a}'_C}$$

$$\frac{T(\mathsf{a}_A,\mathsf{a}_C) \qquad \mathsf{a}_A \supseteq id_A}{\mathsf{a}_C \supseteq id_C} \qquad\qquad \frac{T_{\mathsf{a}}(\mathsf{a}_A,\mathsf{a}_C) \qquad T_{\mathsf{a}}(\mathsf{a}'_A,\mathsf{a}'_C) \qquad T_{\mathsf{a}}((\mathsf{a}_A+\mathsf{a}'_A),\mathsf{a}''_C)}{\mathsf{a}''_C \supseteq (\mathsf{a}_C+\mathsf{a}'_C)}$$

$$\frac{T_{\mathsf{a}}(\mathsf{a}_A,\mathsf{a}_C) \qquad T_{\mathsf{a}}(\mathsf{a}'_A,\mathsf{a}'_C) \qquad T_b(b_A,b_C) \qquad T_{\mathsf{a}}((b_A?\ \mathsf{a}_A \oplus \mathsf{a}'_A),\mathsf{a}''_C)}{\mathsf{a}''_C \supseteq (b_C?\ \mathsf{a}_C \oplus \mathsf{a}'_C)}$$

$$\frac{T_{\mathsf{a}}(\mathsf{a}_A,\mathsf{a}_C) \qquad T_{\mathsf{a}}(\mathsf{a}'_A,\mathsf{a}'_C) \qquad T_{\mathsf{a}}((\mathsf{a}_A \circ \mathsf{a}'_A),\mathsf{a}''_C)}{\mathsf{a}''_C \supseteq (\mathsf{a}_C \circ \mathsf{a}'_C)}$$

Figure 4.1: Requirements for Relation of Actions

Figure 4.1 hold on T_{a}.

The easiest explanation of these rules is that they guarantee that a weaker-than relation is preserved by the refinement, and this must be true for all constructors used by the well-formedness rules. For example, we must be certain that the refinement of $\mathsf{a} \circ \mathsf{a}'$ is weaker than the composition of refined actions, and similarly for other combinators.

There are two rules that do not follow this pattern: the OP-COMPAT rule shows that the refinement of the action of an abstract operation must be weaker that the action of the corresponding concrete operation. This is needed to guarantee that the new concrete code will be more precise than what is expected by the abstract code. The ACT-EXIST rule requires that there is always a concrete action for a particular abstract action - a property which is needed for a proof to guarantee that the intermediate steps of the refinement are definable.

The idea is that if T_{a} satisfies the above properties, we can produce a proof of PROC-REFINE. However, we will later encounter cases where we would like to define T_{a} that can not satisfy the above properties for arbitrary actions. Luckily, we only need to have these properties hold for actions that we encounter in the certified modules that we will be translating. If we can not show that T_{a} holds for arbitrary modules, we can restrict the modules by definiing $Acc_{\mathbb{I}}(\mathcal{M}_A, \mathcal{L} \vdash \mathbb{C} : \mathsf{a})$ in such a way that the properties will hold for all modules that satisfy this restriction.

Now that we have given the construction of $T_{\mathbb{C}}$, and $Acc_{\mathbb{I}}$, and knowing that $T_{\mathtt{a}}$ has certain properties, we can give a proof of the PROC-REFINE rule. This is summed up in the following theorem.

Lemma 4.2.3 (Order-Preserving Refinement)

Given any abstract machine $\mathcal{M}_A$ and a concrete machine $\mathcal{M}_C$, a relation of the abstract operations to the concrete operations T_ι, and a relation from abstract actions to concrete actions $T_{\mathtt{a}}$, if $T_{\mathtt{a}}$ satisfies the properties in Figure 4.1, then using the definition of $T_{\mathbb{I}}$ above, the PROC-REFINE rule (shown below) holds.

$$\frac{\begin{array}{c}\mathcal{M}_A,\mathcal{L}_A \vdash \mathbb{I}_A : \mathtt{a}_A \qquad \forall \mathtt{l} \in \mathtt{dom}(\mathbb{C}_A).\, Acc_{\mathbb{I}}(\mathcal{M}_A,\mathcal{L}_A \vdash \mathbb{I}_A : \mathtt{a}_A)\\ \forall \mathtt{l} \in \mathtt{dom}(\mathcal{L}_A).\, T_{\mathtt{a}}(\mathcal{L}_A(\mathtt{l}),\mathcal{L}_C(\mathtt{l})) \qquad T_{\mathbb{I}}(\mathbb{I}_A,\mathbb{I}_C) \qquad T_{\mathtt{a}}(\mathtt{a}_A,\mathtt{a}_C)\end{array}}{\mathcal{M}_C,\mathcal{L}_C \vdash \mathbb{I}_C : \mathtt{a}_C}\ \textsc{proc-refine}$$

Pf. We will show that order-preserving refinement is a special case of per-process refinement, which we know to be valid.

Assume that $\mathcal{M}_A,\mathcal{L}_A \vdash \mathbb{I}_A : \mathtt{a}_A$ and $\forall \mathtt{l} \in \mathtt{dom}(\mathcal{L}_A).\, T_{\mathtt{a}}(\mathcal{L}_A(\mathtt{l}),\mathcal{L}_C(\mathtt{l}))$ and $T_{\mathbb{I}}(\mathbb{I}_A,\mathbb{I}_C)$ and $T_{\mathtt{a}}(\mathtt{a}_A,\mathtt{a}_C)$. We need to show that $\mathcal{M}_C,\mathcal{L}_C \vdash \mathbb{I}_C : \mathtt{a}_C$.

By induction on the derivation of $\mathcal{M}_A,\mathcal{L}_A \vdash \mathbb{I}_A : \mathtt{a}_A$. There are 7 cases.

1. Case WF-NIL.
   Then $\mathbb{I}_A = \mathtt{nil}$ and $\mathtt{a}_A = id_A$
   Thus $\mathcal{M}_A,\mathcal{L}_A \vdash \mathtt{nil} : id$ and $T_{\mathbb{I}}(\mathtt{nil},\mathbb{I}_C)$
   By inversion on $T_{\mathbb{I}}$, $\mathbb{I}_C = \mathtt{nil}$.
   By properties over transformations, $\mathtt{a}_C \supseteq id_C$.
   By WF-NIL, $\mathcal{M}_C,\mathcal{L}_C \vdash \mathtt{nil} : id$
   By WF-WEAK, $\mathcal{M}_C,\mathcal{L}_C \vdash \mathtt{nil} : \mathtt{a}_C$

2. Case WF-OP
   Then $\mathbb{I}_A = \iota_A$ and $\mathtt{a}_A = \mathcal{M}_A(\iota_A)$
   Thus $\mathcal{M}_A,\mathcal{L}_A \vdash \iota_A : \mathcal{M}_A(\iota_A)$ and $T_{\mathbb{I}}(\iota_A,\mathbb{I}_C)$
   By inversion on $T_{\mathbb{I}}$, $\mathbb{I}_C = \iota_C$
   By properties, $\mathtt{a}_C \supseteq \mathcal{M}_C(\iota_C)$.
   By WF-PERF, $\mathcal{M}_C,\mathcal{L}_C \vdash \iota_C : \mathcal{M}_C(\iota_C)$
   by WF-WEAK, $\mathcal{M}_C,\mathcal{L}_C \vdash \iota_C : \mathtt{a}_C$

3. Case WF-CALL
   Then $\mathbb{I}_A = [\mathtt{l}]$ and $\mathtt{a}_A = \mathcal{L}_A(\mathtt{l})$
   Thus $\mathcal{M}_A,\mathcal{L}_A \vdash [\mathtt{l}] : \mathcal{L}_A(\mathtt{l})$ and $T_{\mathbb{I}}([\mathtt{l}],\mathbb{I}_C)$
   By inversion on $T_{\mathbb{I}}$, $\mathbb{I}_C = [\mathtt{l}]$
   By properties, $\mathtt{a}_C \supseteq \mathcal{L}_C(\mathtt{l})$
   By WF-CALL, $\mathcal{M}_C,\mathcal{L}_C \vdash [\mathtt{l}] : \mathcal{L}_C(\mathtt{l})$
   By WF-WEAK, $\mathcal{M}_C,\mathcal{L}_C \vdash [\mathtt{l}] : \mathtt{a}_C$

4. Case WF-SEQ
   Then $\mathbb{I}_A = \mathbb{I}'_A;\mathbb{I}''_A$ and $\mathtt{a}_A = \mathtt{a}'_A \circ \mathtt{a}''_A$
   Thus $\mathcal{M}_A,\mathcal{L}_A \vdash \mathbb{I}'_A;\mathbb{I}''_A : \mathtt{a}'_A \circ \mathtt{a}''_A$ and $T_{\mathbb{I}}(\mathbb{I}'_A;\mathbb{I}''_A,\mathbb{I}_C)$
   By inversion on $T_{\mathbb{I}}$ there exists $\mathbb{I}'_C$ and $\mathbb{I}''_C$ such that $\mathbb{I}_C = \mathbb{I}'_C;\mathbb{I}''_C$, $T_{\mathbb{I}}(\mathbb{I}'_A,\mathbb{I}'_C)$ and $T_{\mathbb{I}}(\mathbb{I}''_A,\mathbb{I}''_C)$

By property, there exists a'_C such that $T_{\mathsf{a}}(\mathsf{a}'_A, \mathsf{a}'_C)$
By property, there exists a''_C such that $T_{\mathsf{a}}(\mathsf{a}''_A, \mathsf{a}''_C)$
By IH, $\mathcal{M}_C, \mathcal{L}_C \vdash \mathbb{I}'_C : \mathsf{a}'_C$
By IH, $\mathcal{M}_C, \mathcal{L}_C \vdash \mathbb{I}''_C : \mathsf{a}''_C$
By wf-seq, $\mathcal{M}_C, \mathcal{L}_C \vdash (\mathbb{I}'_C ; \mathbb{I}''_C) : (\mathsf{a}'_C \circ \mathsf{a}''_C)$
By properties, $\mathsf{a}_C \supseteq \mathsf{a}'_C \circ \mathsf{a}''_C$
By wf-weak, $\mathcal{M}_C, \mathcal{L}_C \vdash \mathbb{I}'_C ; \mathbb{I}''_C : \mathsf{a}_C$

5. Case wf-choice
   Then $\mathbb{I}_A = \mathbb{I}'_A + \mathbb{I}''_A$ and $\mathsf{a}_A = \mathsf{a}'_A \oplus \mathsf{a}''_A$
   Thus $\mathcal{M}_A, \mathcal{L}_A \vdash \mathbb{I}'_A + \mathbb{I}''_A : \mathsf{a}'_A \oplus \mathsf{a}''_A$ and $T_{\mathbb{I}}(\mathbb{I}'_A + \mathbb{I}''_A, \mathbb{I}_C)$
   By inversion on $T_{\mathbb{I}}$ there exists $\mathbb{I}'_C$ and $\mathbb{I}''_C$ such that $\mathbb{I}_C = \mathbb{I}'_C + \mathbb{I}''_C$, $T_{\mathbb{I}}(\mathbb{I}'_A, \mathbb{I}'_C)$ and $T_{\mathbb{I}}(\mathbb{I}''_A, \mathbb{I}''_C)$
   By property, there exists a'_C such that $T_{\mathsf{a}}(\mathsf{a}'_A, \mathsf{a}'_C)$
   By property, there exists a''_C such that $T_{\mathsf{a}}(\mathsf{a}''_A, \mathsf{a}''_C)$
   By IH, $\mathcal{M}_C, \mathcal{L}_C \vdash \mathbb{I}'_C : \mathsf{a}'_C$
   By IH, $\mathcal{M}_C, \mathcal{L}_C \vdash \mathbb{I}''_C : \mathsf{a}''_C$
   By wf-choice, $\mathcal{M}_C, \mathcal{L}_C \vdash (\mathbb{I}'_C + \mathbb{I}''_C) : (\mathsf{a}'_C \oplus \mathsf{a}''_C)$
   By properties, $\mathsf{a}_C \supseteq \mathsf{a}'_C \oplus \mathsf{a}''_C$
   By wf-weak, $\mathcal{M}_C, \mathcal{L}_C \vdash \mathbb{I}'_C + \mathbb{I}''_C : \mathsf{a}_C$

6. Case wf-branch
   Then $\mathbb{I}_A = (b?\ \mathbb{I}'_A + \mathbb{I}''_A)$ and $\mathsf{a}_A = (\mathcal{M}_A(b)?\ \mathsf{a}'_A \oplus \mathsf{a}''_A)$
   Thus $\mathcal{M}_A, \mathcal{L}_A \vdash (b?\ \mathbb{I}'_A + \mathbb{I}''_A) : (\mathcal{M}_A(b)?\ \mathsf{a}'_A \oplus \mathsf{a}''_A)$ and $T_{\mathbb{I}}((\mathcal{M}_A(b)?\ \mathbb{I}'_A \oplus \mathbb{I}''_A), \mathbb{I}_C)$
   By inversion on $T_{\mathbb{I}}$ there exists $\mathbb{I}'_C$ and $\mathbb{I}''_C$ such that $\mathbb{I}_C = (b?\ \mathbb{I}'_C + \mathbb{I}''_C)$, $T_{\mathbb{I}}(\mathbb{I}'_A, \mathbb{I}'_C)$ and $T_{\mathbb{I}}(\mathbb{I}''_A, \mathbb{I}''_C)$
   By property, there exists a'_C such that $T_{\mathsf{a}}(\mathsf{a}'_A, \mathsf{a}'_C)$
   By property, there exists a''_C such that $T_{\mathsf{a}}(\mathsf{a}''_A, \mathsf{a}''_C)$
   By IH, $\mathcal{M}_C, \mathcal{L}_C \vdash \mathbb{I}'_C : \mathsf{a}'_C$
   By IH, $\mathcal{M}_C, \mathcal{L}_C \vdash \mathbb{I}''_C : \mathsf{a}''_C$
   By wf-branch, $\mathcal{M}_C, \mathcal{L}_C \vdash (b?\ \mathbb{I}'_C + \mathbb{I}''_C) : (\mathcal{M}_C(b)?\ \mathsf{a}'_C \oplus \mathsf{a}''_C)$
   By properties, $\mathsf{a}_C \supseteq (\mathcal{M}_C(b)?\ \mathsf{a}'_C \oplus \mathsf{a}''_C)$
   By wf-weak, $\mathcal{M}_C, \mathcal{L}_C \vdash (b?\ \mathbb{I}'_C + \mathbb{I}''_C) : \mathsf{a}_C$

7. Case wf-weak
   Then $\mathcal{M}_A, \mathcal{L}_A \vdash \mathbb{I}_A : \mathsf{a}'_A$ and $\mathsf{a}_A \supseteq \mathsf{a}'_A$ and $T_{\mathsf{a}}(\mathsf{a}_A, \mathsf{a}_C)$
   By property, there exists a'_C such that $T_{\mathsf{a}}(\mathsf{a}'_A, \mathsf{a}'_C)$
   By IH, $\mathcal{M}_C, \mathcal{L}_C \vdash \mathbb{I}_C : \mathsf{a}'_C$
   By property, $\mathsf{a}_C \supseteq \mathsf{a}'_C$
   By wf-weak, $\mathcal{M}_C, \mathcal{L}_C \vdash \mathbb{I}_C : \mathsf{a}_C$

□

Using the above theorem, we conclude that from any tuple $(T_\iota, T_b, T_{\mathsf{a}})$ that respects the requirements, we can automatically generate a pair $(T_{\mathbb{I}}, T_{\mathsf{a}})$ and a proof of the proc-refine rule, which means that we have satisfied the conditions to form a procedure-preserving refinement. Then by the theorems about procedure-preserving refinements, we can produce $T_{\mathbb{C}}$, and T_{Ψ} as well as the proof of the refine rule.

What this means is that we can now generate a refinement, including $T_{\mathbb{C}}$ and T_{Ψ}, by simply picking T_ι and T_{a} and showing the specifications. We can then use the generated $T_{\mathbb{C}}$ and T_{Ψ} to link modules in different layers of abstraction.

4.2.3 Code-Preserving Refinement

In the cases that have encountered in our VMM verification, all of our refinements do not alter the actual program, but only the meaning of the program. As the program is refined, its semantics become more and more detailed, but the code of the individual procedures does not change. The relations between the instructions (T_ι) and the conditional expressions (T_b) in our cases become the identity relations. By the definitions of $T_{\mathbb{I}}$ and $T_{\mathbb{C}}$, all the relations between the code of the abstract and the refined modules becomes identity, and thus can be removed from consideration. We refer to such refinements as code-preserving, as only the specifications are the things being refined.

The result of this simplification is that we can remove all references to code translation. Thus for any two machines, we can create a code-preserving refinement between machines $\mathcal{M}_A$ and $\mathcal{M}_C$, by defining $T_{\mathtt{a}}$ which satisfies all the properties in Figure 4.1, and we will automatically get a refinement defined by the following lemma:

Lemma 4.2.4 (Code-Preserving Refinement Valid) For any abstract certified module $\mathcal{M}_A, \mathcal{L}_A \vdash \mathbb{I} : \mathtt{a}_A$ and concrete machine $\mathcal{M}_C$ and concrete library $\mathcal{L}_C$, and an action relation $T_{\mathtt{a}}$ that obeys the properties in Figure 4.1, the following holds

$$\frac{\mathcal{M}_A, \mathcal{L}_A \vdash \mathbb{I} : \mathtt{a}_A \qquad \forall \mathtt{l} \in \mathtt{dom}(\mathbb{C}_A).\, Acc_{\mathbb{I}}(\mathcal{M}_A, \mathcal{L}_A \vdash \mathbb{I} : \mathtt{a}_A) \qquad \forall \mathtt{l} \in \mathtt{dom}(\mathcal{L}_A).\, T_{\mathtt{a}}(\mathcal{L}_A(\mathtt{l}), \mathcal{L}_C(\mathtt{l})) \qquad T_{\mathtt{a}}(\mathtt{a}_A, \mathtt{a}_C)}{\mathcal{M}_C, \mathcal{L}_C \vdash \mathbb{I} : \mathtt{a}_C} \textsc{ code-refine}$$

Pf. Special case of Order-Preserving Refinement.

This rule is exactly what is needed for the Per-Procedure Refinement, which will define T_Ψ and *Acc* using the per-procedure definition, while the $T_{\mathbb{C}}$ is identity and therefore not present.

$$T_\Psi(\Psi_A, \Psi_C) := \forall \mathtt{l} \in \mathtt{dom}(\Psi_A).\, T_{\mathtt{a}}(\Psi_A(\mathtt{l}), \Psi_C(\mathtt{l}))$$
$$Acc(\mathcal{M}_A, \mathcal{L}_A \vdash \mathbb{C} : \Psi) := \forall \mathtt{l} \in \mathtt{dom}(\Psi_A).\, Acc_{\mathbb{I}}(\mathcal{M}_A, \mathcal{L}_A \cup \Psi \vdash \mathbb{C}(\mathtt{l}) : \Psi(\mathtt{l}))$$

Thus, we can use the proof of Per-Procedure Refinement to generate the proof of the rule that we will actually use for the refinement itself.

$$\frac{}{T_{\mathsf{a}}(\mathcal{M}_A(\iota)) \supseteq \mathcal{M}_C(\iota)}\ \textsc{op-compat} \qquad \frac{}{T_{\mathsf{a}}(id_A) \supseteq id_C} \qquad \frac{\mathsf{a}_A \supseteq \mathsf{a}'_A}{T_{\mathsf{a}}(\mathsf{a}_A) \supseteq T_{\mathsf{a}}(\mathsf{a}'_A)}$$

$$\frac{}{T_{\mathsf{a}}\,(\mathcal{M}_A(b)?\ \mathsf{a}_A \oplus \mathsf{a}'_A) \supseteq (\mathcal{M}_C(b)?\ T_{\mathsf{a}}(\mathsf{a}_A) \oplus T_{\mathsf{a}}(\mathsf{a}'_A))}$$

$$\frac{}{T_{\mathsf{a}}(\mathsf{a}_A \oplus \mathsf{a}'_A) \supseteq (T_{\mathsf{a}}(\mathsf{a}_A) \oplus T_{\mathsf{a}}(\mathsf{a}'_A))} \qquad \frac{}{T_{\mathsf{a}}(\mathsf{a}_A \circ \mathsf{a}'_A) \supseteq (T_{\mathsf{a}}(\mathsf{a}_A) \circ T_{\mathsf{a}}(\mathsf{a}'_A))}$$

Figure 4.2: Requirements for a Function-Based Code-Preserving Refinement

$$\frac{\mathcal{M}_A, \mathcal{L}_A \vdash \mathbb{C} : \Psi_A \qquad T_\Psi(\mathcal{L}_A, \mathcal{L}_C) \qquad T_\Psi(\Psi_A, \Psi_C) \qquad Acc(\mathcal{M}_A, \mathcal{L}_A \vdash \mathbb{C} : \Psi_A)}{\mathcal{M}_C, \mathcal{L}_C \vdash \mathbb{C} : \Psi_C}$$

Thus we have a rule that allows us to quickly define refinements of certified modules by defining only the T_{a} that obeys the specific rules.

4.2.4 Using Functions for Action Refinement

Both the order-preserving and code-preserving refinements make use of the relation T_{a} to link the abstract and the related concrete actions between the machine. However, in many cases such a conversion is deterministic, and thus T_{a} can be defined as a function. The fact that T_{a} is a function can greatly simplify the properties that have to be proven about T_{a}, and the new requirements for the refining of actions is given in Figure 4.2. The simplifications allowed us to remove various existentials from the rules, as well as to remove the ACT-EXIST rule in entirety, as it became trivially true due to T_{a} being a total function.

Because the relation between abstract and concrete actions is now a function, we no longer have to pick a concrete specification, as the specifications can now be automatically generated. This results in an even simpler refinement rule for this restrictive case:

$$\frac{\mathcal{M}_A, \mathcal{L} \vdash \mathbb{C} : \Psi \qquad Acc(\mathcal{M}_A, \mathcal{L} \vdash \mathbb{C} : \Psi)}{\mathcal{M}_C, T_\Psi(\mathcal{L}) \vdash \mathbb{C} : T_\Psi(\Psi)}$$

Both the order and the code-preserving refinements can be re-proven using this more restrictive form of action translation. However, as we will see, this restriction is too restrictive for certain uses, and we will use the relational version to define those refinements. However, as both are valid, the refinements where the transformation of actions can be defined as a function, we will use the more restrictive version, as it simplifies proofs.

4.2.5 The Usefulness of Code-Preserving Refinements

It may be tempting to dismiss order-preserving and code-preserving refinements as too restrictive, and thus not useful. We believe that this is not the case. While the order of the operations is not allowed to change between the abstract and the concrete version, the fact that the meaning of each operation and procedure can be different when refined allows us to express significant changes in abstraction between the code executing in the abstract machine and the code executing in the concrete machine.

However, we do not want to make the claim that all refinements can be defined as order-preserving. In fact, it is easy to find counterexamples, the most classic of which is an example of a compiler that uses instruction reordering optimizations. If such compiler is a certified compiler, then by definition it performs some sort of a refinement, but because it reorders instructions, we can never make it fit the order-preserving pattern.

However, in this thesis, we do not aim to build certifying compilers. We instead will focus on automatic construction of several other refinements, all of which do fit the order-preserving pattern.

4.3 Specialized Refinements

The order-preserving and code-preserving refinements assume no information about the state and any of the semantics of the machine. The refinements are defined in terms of an arbitrary $T_{\mathtt{a}}$ which must satisfy several general requirements in order to build a particular refinement.

However, if more information about the state of the machines and their semantics is available, we would be able to generate both the $T_{\mathtt{a}}$ and the proofs of properties of that relation from even less information, thus making it easier to define the refinements. In this section we will define several such refinements.

4.3.1 Embedding Refinement

One of the simplest refinements that we can define is the embedding refinement, which assumes that the abstract machine is just a subset of the concrete machine. Such refinement can be defined by stating a state transformation function from the concrete state to the abstract state. We call this transformation a projection function, which has the following type.

$$pr : \mathcal{M}_C.\Sigma \to \mathcal{M}_A.\Sigma$$

The fact that the concrete and abstract states are related by the *pr* function means that a concrete state can always be linked to a precise abstract state, though an abstract state could be an abstraction of multiple concrete states.

As the abstract state contains less information, not all of the operations of the concrete machine are valid in the abstract machine. Thus the abstract machine must define a set of operations that is the subset of operations of the concrete machine, and moreover these concrete specifications of these operations must have the same semantics over the abstract subset of the state. If these conditions are true, then we can construct a function that lifts the abstract actions into the concrete actions:

$$\uparrow_{pr} (\mathtt{a}) \triangleq \lambda \mathbb{S}_C. \{ \mathbb{S}'_C \mid pr(\mathbb{S}'_C) \in \mathtt{a}\ pr(\mathbb{S}_C) \}$$

The lifted abstract action is simply a pullback of the abstract action over the projection function. We can use this lifting operation as the relation between the abstract and concrete actions.

$$T_{\mathtt{a}}(\mathtt{a}_A, \mathtt{a}_C) \triangleq \left(\mathtt{a}_C \cong \uparrow_{pr} (\mathtt{a}_A) \right)$$

This definition shows that an abstract action is related to any concrete action that is equivalent to the lifting of the former.

Because the set of operations is the same, we will define the embedding refinement as a special case of the code-preserving refinement. The only bit of information that we will need to define an embedding invariant is a proof that the following fact holds over the abstract and concrete machines:

$$\frac{}{\forall \iota \in \mathtt{dom}(\mathcal{M}_A).\ \ \uparrow_{pr} (\mathcal{M}_A(\iota)) \supseteq \mathcal{M}_C(\iota)} \ \textsc{embed-refine}$$

We can show that the $T_{\mathtt{a}}$ that we have defined satisfies the properties of the function-based code-preserving refinement, and thus defines a valid refinement. This is shown by the following lemma.

Theorem 4.3.1 (Certified Embedding Refinement)

For any machine $\mathcal{M}_C$ and its subset machine $\mathcal{M}_A$ defined by the projection function *pr*, given the proof of EMBED-REFINE rule, the $T_{\mathtt{a}}$ defined as above satisfies the properties of the function based code-preserving refinement (as listed in Figure 4.2).

Pf. We need to show that $T_{\mathtt{a}}$ supports properties in Figure 4.1 (equivalent to showing that $\uparrow_{pr}$ follows properties in Figure 4.2). The proofs are fairly simple, please see the Coq proof for details.

□

What this proof means is that we can generate a $T_{\mathtt{a}}$ from any projection function *pr*, and we know that it will satisfy the properties needed to be used in an code-preserving refinement, thus making the embedding refinement a special case of the code-preserving refinement. The code-preserving refinement already defines a way to generate T_{Ψ} that we can use to actually refine modules from $\mathcal{M}_A$ to $\mathcal{M}_C$. Thus by defining the projection function and checking that EMBED-REFINE rule holds, we automatically generate a refinement between the machines.

4.3.2 Representation Refinement

One of the issues with the embedding refinement is that to create one, we have to supply an embedding function from the concrete state to the abstract state. However, it is not always the case that the relation between the concrete and abstract states is a function, where a concrete state is always represented by at most one abstract state. It is possible to imagine an abstract machine where a single concrete state can correspond to several abstract states, i.e. multiple abstract data structures that have the same concrete encoding, etc.

Thus, we would like to construct an analog of the embedding refinement, except instead of the embedding function *pr*, we would like to have a general representation relation, which we will call `repr`.

$$\mathtt{repr} : \mathcal{M}_A.\Sigma \to \mathcal{M}_C.\Sigma \to \mathit{Prop}$$

The `repr` defines a many-to-many relation between abstract and concrete states. What is more important is how the `repr` relation can be used to convert abstract actions into concrete ones. The

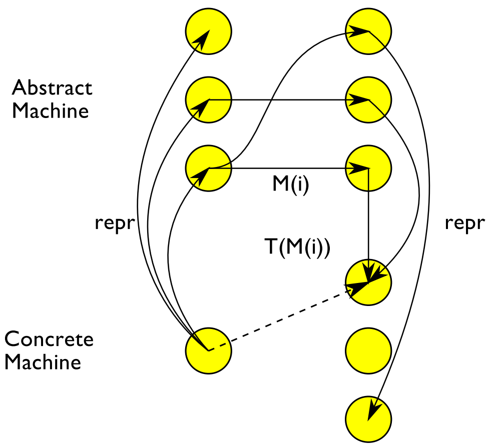


Figure 4.3: Diagram of `repr`-refinement of Actions

idea is that the concrete meaning of the abstract action can be defined as an intersection of all abstract actions suitable for a particular concrete state. For example, if we pick a particular concrete state, then this state will relate to several abstract states, each of these abstract states is a plausible abstraction of the concrete state. Then we will apply the abstract action to all of these states. Some of these abstract states are not valid, and the action will fail, but for all of them on which the action succeeds, we will consider all of them correct, and thus the set of resulting abstract states is the intersection of resulting states of the action from all suitable starting states. Then we will apply the relation to convert the resulting abstract states to get all possible concrete states. This describes the concrete refinement of an abstract action across the `repr` relation. This action is described graphically in Figure 4.3, and mathematically as a total function by the following:

$$\uparrow_{\mathtt{repr}}(\mathsf{a}) \triangleq \lambda \mathbb{S}_C . \left\{ \mathbb{S}'_C \;\middle|\; \left(\begin{array}{l} \forall \mathbb{S}_A . \mathtt{repr}(\mathbb{S}_A, \mathbb{S}_C) \rightarrow \\ \mathbb{S}_A \in \mathtt{dom}(\mathsf{a}) \rightarrow \\ \exists \mathbb{S}'_A . \mathtt{repr}(\mathbb{S}'_A, \mathbb{S}'_C) \wedge \mathbb{S}'_A \in \mathsf{a}\ \mathbb{S}_A \end{array} \right) \right\} \text{ if } (\exists \mathbb{S}_A .\ \mathtt{repr}(\mathbb{S}_A, \mathbb{S}_C) \wedge \mathbb{S}_A \in \mathtt{dom}(\mathsf{a}))$$

$$T_{\mathsf{a}}(\mathsf{a}) \quad \triangleq \uparrow_{\mathtt{repr}}(\mathsf{a})$$

It is a bit difficult to understand why such a definition works for translating abstract actions into concrete ones. If this is the case, consider the restricted case where a concrete state can correspond to at most one abstract state, and the T_{a} for this case will make much more sense.

The next step is to that the $T_{\mathtt{a}}$ we defined forms a valid code-preserving refinement. To prove this, we will need to show that $T_{\mathtt{a}}$ satisfies the order-preserving properties for functions (Figure 4.2). We are allowed to use the simpler properties since we have defined $T_{\mathtt{a}}$ to be a total function. However, of those properties, we can not automatically deduce OP-COMPAT, since it is the property of the machines, and thus we must require it as a part of the definition of the `repr`-refinement.

$$\forall \iota \in \mathcal{M}_A . T_{\mathtt{a}}(\mathcal{M}_A(\iota)) \supseteq \mathcal{M}_C(\iota)$$

The other properties that we need to show only depend on the definition of $T_{\mathtt{a}}$, and we will show that these properties are indeed true. Most of these properties are simple to show, but unfortunately we will run into trouble when trying to prove the preservation of weaker-than relations for branching and choice operations. We will not be able to prove either of the following properties for arbitrary actions:

$$\frac{}{T_{\mathtt{a}}(\mathtt{a}_A \oplus \mathtt{a'}_A) \supseteq (T_{\mathtt{a}}(\mathtt{a}_A) \oplus T_{\mathtt{a}}(\mathtt{a'}_A))} \qquad \frac{}{T_{\mathtt{a}}((\mathcal{M}_A(b)?\ \mathtt{a}_A \oplus \mathtt{a'}_A)) \supseteq (\mathcal{M}_C(b)?\ T_{\mathtt{a}}(\mathtt{a}_A) \oplus T_{\mathtt{a}}(\mathtt{a'}_A))}$$

To understand the problem, we must first consider the cases of multiple abstract states that are related to the particular concrete state. We call these states `repr`-related states, which are defined formally as follows:

$$\mathbb{S}_A \approx \mathbb{S'}_A \triangleq \exists \mathbb{S}_C.\ \mathtt{repr}(\mathbb{S}_A, \mathbb{S}_C) \wedge \mathtt{repr}(\mathbb{S'}_A, \mathbb{S}_C)$$

Now consider a program that contains a branch operation in both abstract and concrete machines (since the program is the same for both). For a particular concrete state $\mathbb{S}_C$, either the left or the right side of the branch will be chosen. However, there is no guarantee that the same side will be chosen for the two related abstract states $\mathbb{S}_A$ and $\mathbb{S'}_A$. But if the sides are different, then we must make sure that both sides are proper refinements for the original single branch. This is generally not the case for the branching, and since the order-preserving refinement reconstructs branches the same way they were in the original code, it creates a problem for the proof of the weaker-than preservation.

The simple way to fix this is by ensuring that all `repr`-related states must always select the same side of the branching operation. For the cases where we have a branching condition, we can simply make sure that in both machines the branching condition will select the same side. Such a condition can be defined as follows:

$$\mathit{BranchPreserve}(b) := \forall \mathbb{S}_A, \mathbb{S}_C.\, \texttt{repr}(\mathbb{S}_A, \mathbb{S}_C) \to (\mathcal{M}_A(b)\ \mathbb{S}_A \leftrightarrow \mathcal{M}_C(b)\ \mathbb{S}_C)$$

By removing the possibility of having to deal with branching going into different directions, it becomes possible to prove the property over the translation of actions as long as we know the conditional obeys the restriction.

$$\mathit{BranchPreserve}(b) \to T_{\mathtt{a}}(\mathcal{M}_A(b)?\ \mathtt{a}_A \oplus \mathtt{a'}_A) \supseteq (\mathcal{M}_C(b)?\ (T_{\mathtt{a}}(\mathtt{a}_A)) \oplus (T_{\mathtt{a}}(\mathtt{a'}_A)))$$

The problem with the proof of the choice operation exists for a similar reason, even through there is no branch selector. There is additional difficulty that comes from the fact that both sides may be valid at the same time. Thus, we handle this problem in the most general way by a predicate that allows the `repr`-related states to choose different sides only in those cases where both sides are valid for the states. We can define a predicate that restricts such choice operation by the following predicate:

$$\mathit{Excl}(\mathtt{a}, \mathtt{a'}) := \begin{array}{l} \forall \mathbb{S}_A, \mathbb{S'}_A, \mathbb{S''}_A.\, \mathbb{S'}_A \approx \mathbb{S''}_A \to \\ \quad \mathbb{S'}_A \in \mathtt{a}\ \mathbb{S}_A \to \mathbb{S''}_A \in \mathtt{a'}\ \mathbb{S}_A \to \\ \qquad (\mathbb{S''}_A \in \mathtt{a}\ \mathbb{S}_A \land \mathbb{S'}_A \in \mathtt{a'}\ \mathbb{S}_A) \end{array}$$

This condition is a bit more complex than the previous one. The new version states that if the two abstract states get translated to the same concrete state, then, even if they take two separate sides of the choice operation, the resulting set of both operations are the same. In other words, the predicate guarantees that for the `repr`-related states, the side chosen by the choice operation does not matter, as the resulting set of operations of any `repr`-related states gives a complete set of possibilities. With this bit of information, it becomes possible to prove the preservation of branching under translation.

$$\mathit{Excl}(\mathtt{a}, \mathtt{a'}) \to T_{\mathtt{a}}(\mathtt{a}_A \oplus \mathtt{a'}_A) \supseteq (T_{\mathtt{a}}(\mathtt{a}_A) \oplus T_{\mathtt{a}}(\mathtt{a'}_A))$$

However, to integrate this rule into the code-preserving refinement, we need to express these

$$\frac{}{Acc_{\mathbb{I}}(\mathcal{M}_A,\Psi'_A \vdash \mathtt{nil} : id)} \qquad \frac{}{Acc_{\mathbb{I}}(\mathcal{M}_A,\Psi'_A \vdash \iota : \mathcal{M}_A(\iota))}$$

$$\frac{}{Acc_{\mathbb{I}}(\mathcal{M}_A,\Psi'_A \vdash [\mathtt{l}] : \Psi'_A(\mathtt{l}))} \qquad \frac{Acc_{\mathbb{I}}(\mathcal{M}_A,\Psi'_A \vdash \mathbb{I}_1 : \mathtt{a}_1) \qquad Acc_{\mathbb{I}}(\mathcal{M}_A,\Psi'_A \vdash \mathbb{I}_2 : \mathtt{a}_2)}{Acc_{\mathbb{I}}(\mathcal{M}_A,\Psi'_A \vdash \mathbb{I}_1;\mathbb{I}_2 : \mathtt{a}_1 \circ \mathtt{a}_2)}$$

$$\frac{Acc_{\mathbb{I}}(\mathcal{M}_A,\Psi'_A \vdash \mathbb{I}_1 : \mathtt{a}_1) \qquad Acc_{\mathbb{I}}(\mathcal{M}_A,\Psi'_A \vdash \mathbb{I}_2 : \mathtt{a}_2) \qquad BranchPreserve(\mathtt{p})}{Acc_{\mathbb{I}}\left(\mathcal{M}_A,\Psi'_A \vdash \mathbb{I}_1 \underset{\mathtt{p}}{+} \mathbb{I}_2 : \mathtt{a}_1 \oplus \mathtt{a}_2\right)}$$

$$\frac{Acc_{\mathbb{I}}(\mathcal{M}_A,\Psi'_A \vdash \mathbb{I}_1 : \mathtt{a}_1) \qquad Acc_{\mathbb{I}}(\mathcal{M}_A,\Psi'_A \vdash \mathbb{I}_2 : \mathtt{a}_2) \qquad Excl(\mathtt{a}_1,\mathtt{a}_2)}{Acc_{\mathbb{I}}(\mathcal{M}_A,\Psi'_A \vdash \mathbb{I}_1 + \mathbb{I}_2 : \mathtt{a}_1 \oplus \mathtt{a}_2)}$$

Figure 4.4: $Acc_{\mathbb{I}}$ for Order-Preserving Refinement with Branch Control

restrictions as the restriction predicate $Acc_{\mathbb{I}}$, which we define in Figure 4.4. The definition simply incorporates the *BranchPreserve* and *Excl* predicates into the certification of modules so that only the code containing branches valid for the particular `repr` can be refined.

The presence of the $Acc_{\mathbb{I}}$ predicate defined in this way allows us to know additional information that can be used to show that the $T_{\mathtt{a}}$ respects the properties of the order-preserving refinement. And indeed we have shown that the restrictions provided by the *BranchPreserve* and *Excl* are adequate for the proof. Thus, we can finally state the lemma that formally defines the refinement.

Lemma 4.3.2 (`repr`-refinement Valid)

For machines $\mathcal{M}_A$ and $\mathcal{M}_C$ that are related using `repr`, and the operations of two machines are related by $\forall \iota \in \mathcal{M}_A. T_{\mathtt{a}}(\mathcal{M}_A(\iota)) \supseteq \mathcal{M}_C(\iota)$, given a certified module $\mathcal{M}_A, \mathcal{L}_A \vdash \mathbb{C} : \Psi$, such that this module satisfies *Acc*, then we know that

$$\mathcal{M}_C, T_{\Psi}(\mathcal{L}_A) \vdash \mathbb{C} : T_{\Psi}(\Psi)$$

where $T_{\Psi}(\Psi) := \lambda \mathtt{l}. T_{\mathtt{a}}(\Psi(\mathtt{l}))$ if $\mathtt{l} \in \mathtt{dom}(\Psi)$.

Pf. The proof involves showing the properties of $T_{\mathtt{a}}$, which makes the `repr`-refinement a special case of the code-preserving refinement. For details, please see the Coq proof.

□

So, the result of this section is that we can produce a refinement from a single relation `repr` between abstract and concrete states, and all we have to show is that the specifications of the machine operations are related by the automatically produced action relation, $T_{\mathtt{a}}$.

Invariant Refinement

One of the issues of the `repr`-refinement is that the refined actions lose all information that was not contained in the abstract machine. This happens because $T_{\mathtt{a}}$ allows for all possible transitions that relate to the abstract information, but it does not maintain any information based on the starting concrete state. This loss of information makes `repr`-refinement not useful for information hiding.

We fix this by mixing the representation relation with the concrete invariant relation. This invariant relation will be incorporated into the translation function $T_{\mathtt{a}}$ to ensure that certain information in the concrete state is always preserved in the translated specification. This invariant relation, *Inv*, can be arbitrarily selected for a specific refinement and machines, as long as the invariant is Kleene-closed, namely, $Inv \supseteq Inv \circ Inv$ and $Inv \supseteq id$.

Given the `repr` relation and the *Inv*, we can now define an action refinement function that can be used to create a code-preserving refinement. We define it using the original action refinement function from the `repr`-refinement.

$$T_{\mathtt{a}}(\mathtt{a}) \triangleq \uparrow_{\mathtt{repr}} (\mathtt{a}) \wedge Inv$$

To generate this refinement, we will need to show that the $T_{\mathtt{a}}$ satisfies the requirements of the code-preserving refinement. As with `repr`-refinement, we will need the same $Acc_{\mathbb{I}}$, as we will still need to control the branching. The proofs that the $T_{\mathtt{a}}$ satisfies the requirements are similar to those for the `repr`-refinement, except with the invariant component added in. However, the *Inv* introduces no additional difficulties.

Thus, we can construct the theorem which allows us to use the representation and the invariant to construct refinements based on the relation between the states of the abstract and the concrete machines that, when used, generate specifications that guarantee preservation of the invariant.

Lemma 4.3.3 (Invariant Refinement Valid)
Given two machines $\mathcal{M}_A$ and $\mathcal{M}_C$, a state relation $\mathtt{repr} : \mathcal{M}_A.\Sigma \rightarrow \mathcal{M}_C.\Sigma \rightarrow Prop$, and a Kleene-closed invariant relation $Inv : \mathcal{M}_C.\Sigma \rightarrow \mathcal{M}_C.\Sigma \rightarrow Prop$, if we define $T_{\mathtt{a}}$ from `repr` and *Inv* as given

above, then if $\forall\iota \in \mathcal{M}_A . T_{\mathtt{a}}(\mathcal{M}_{\mathtt{a}}(\iota)) \supseteq \mathcal{M}_C(\iota)$, then

$$\frac{\mathcal{M}_A, \mathcal{L}_A \vdash \mathbb{C} : \Psi \qquad Acc(\mathcal{M}_A, \mathcal{L}_A \vdash \mathbb{C} : \Psi)}{\mathcal{M}_C, T_\Psi(\mathcal{L}_A) \vdash \mathbb{C} : T_\Psi(\Psi)}$$

where $T_\Psi(\Psi) := \lambda \mathtt{l}. T_{\mathtt{a}}(\Psi(\mathtt{l}))$ if $\mathtt{l} \in \mathtt{dom}(\Psi)$.

Pf. Similar to the proof of the `repr`-refinement. First we show that the $T_{\mathtt{a}}$ satisfies all the requirements for the code-preserving refinement. Then we use the validity of the code-preserving refinement to guarantee that the rule holds.

□

It is our belief that this refinement is extremely powerful in how it can translate certified modules from one machine to another. Although, we try to use the slightly simpler `repr`-refinement for most of our work, the invariant refinement is more general, and, unlike the `repr`-refinement, can be used for information hiding, making it a very powerful tool in our arsenal.

4.4 Other Refinements

The refinements we have presented in this chapter are aimed at creating well protected abstract data structures that modify the operational semantics of the system. These refinements will be extremely useful for our task of verification of the virtual memory manager, and thus these are the only ones on which we have focused our efforts. However, the same infrastructure can be used to define refinements that work differently from the ones we have presented. For example, compilation is a form of refinement, which is definitely not code preserving, but can be expressed within our general framework (and specialized refinements specifically for compilation can be developed). Similarly, refinements may be developed for time-sharing - where several separate linearly analyzed modules can be threaded together into a thread-switching semantics.

However, we leave the development of these other refinements for another time and another project.

Chapter 5

Verification of the Virtual Memory Manager

To demonstrate that our system is extremely useful for simplifying certification of large systems, we have done a complete mechanized certification of a small virtual memory manager that runs on simplified hardware.

The purpose of doing this was to show the effectiveness of the multi-machine verification framework without getting bogged down in the extreme complexities of the realistic hardware. Aiming for the less-featured version of the memory manager reduces the amount of code that has to be implemented to get a complete certification, and using the simplified hardware allows for smaller, more readable specifications. In fact, our first attempt at verification aimed for realistic hardware, but doing so resulted in us debugging the framework, the definition of the machine, and the complex specifications and code of the manager. Our failure to accomplish this task, resulted in this more accessible attempt to produce a complete and mechanized verification.

Thus in the rest of the chapter, we will discuss the verification of the verified virtual memory manager, and in Chapter 6 we will show how to extend this work to produce the certification of a real virtual memory manager.

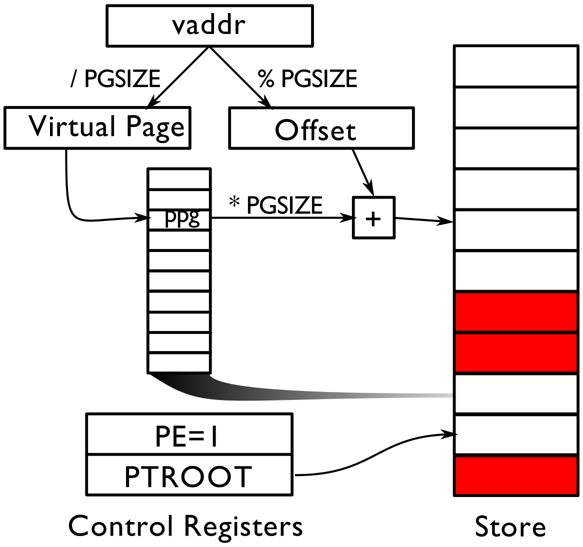


Figure 5.1: Model of Address Translation

5.1 The Purpose of the Virtual Memory Manager

Before we dive into verification, we will first describe what is a virtual memory manager, and why our multi-machine verification system is so useful for this purpose. To start this discussion, we must first look at the hardware that the modern computers use.

All general purpose computers now feature a mechanism called the address translation. This mechanism is extremely useful in that it allows the kernels of operating systems to offer protection and abstraction mechanisms expected by the programs. Using address translation, OS kernels protect themselves from harm by malicious programs, protect programs from each other, and allow for user-level memory models such as growing stacks, fixed-location code, memory address sizes greater than actual memory, and other features.

Figure 5.1 shows the example of a simplified address-translated memory access. When address translation is turned on, every single read or write from memory goes through the address translation table, and gets converted into a physical address, where the actual read or write is being made. On real machines, since the virtual address space is large and sparse, the table is actually a multi-level (2-4 levels are common) tree structure.

The most common use of the address translation is to set up a virtual address space, also called virtual memory. The idea behind virtual memory is that address translation can be utilized to define a

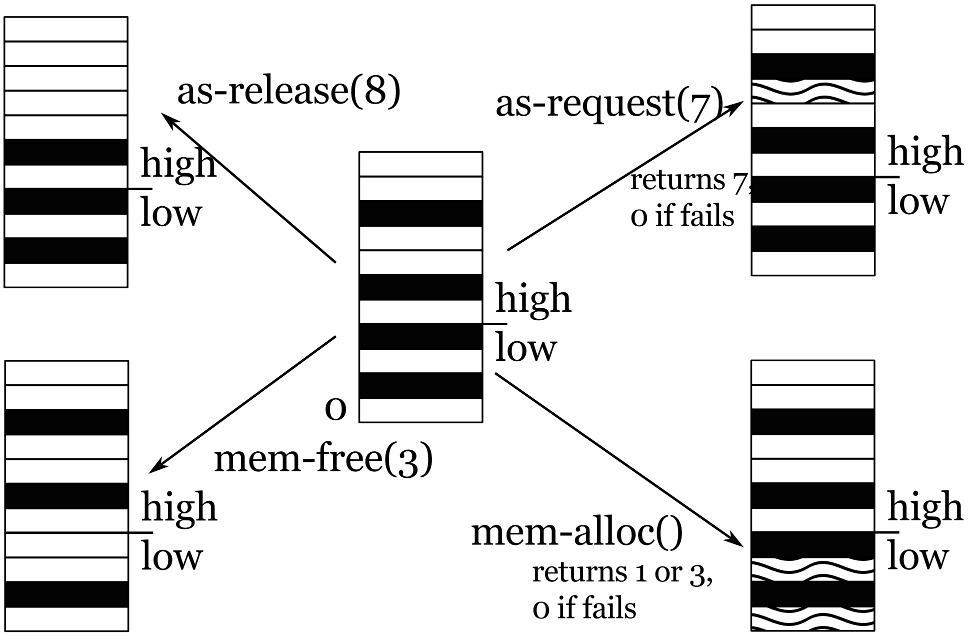


Figure 5.2: Model of Virtual Address Space

virtual store that works somewhat like a larger physical memory, except memory must be requested before it can be used. There is no one single definition of an address space model: some may be very simple, others may come with advanced features, i.e. multiple switchable spaces, page remapping, copy-on-write, etc. We have aimed for a basic model, which is shown in Figure 5.2. Our model features a single address space, divided into low and high regions. Loads and stores can access any address which is located in a allocated page (marked in black). There are 4 special commands that allow the programmer to alter allocation, `as-request`, `as-release`, `mem-alloc`, and `mem-free`, which allocate and free pages in the virtual address space. The difference between the high and low regions is that in high regions, the programmer using this model can request any page specifically using as-request, whereas in the low region the programmer must use mem-alloc, and can not request a specific page. Such a virtual memory system is quite common in kernel development where a particular area of memory is used as a “window” into the physical memory, which is what the low region is in our model.

Virtual address spaces have a benefit in that it is simpler to reason about the virtual address space memory than about the address translated memory. Memory loads and stores work in a simple way, and the only additional information that needs to be kept is whether memory is allocated or not.

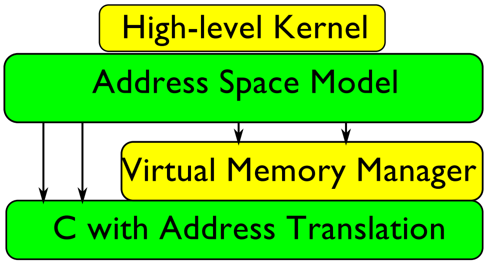


Figure 5.3: Diagram of the Kernel and Virtual Memory Design

However, virtual address space model is not automatically created by hardware. Instead, it is the virtual memory manager that uses the address translation hardware to create the virtual address space model of memory.

Because the virtual address space model is easier to reason about, the high-level portions of OS kernels (process management, etc.) are written with the assumption of virtual memory. At no point does reasoning about the kernels requires reasoning about the underlying code of the virtual memory implementation, but only its abstract model. This causes issues when we try to certify the kernel using single model certification frameworks. If we try to certify the kernel using a machine model that supports the address translation, then the address translation information has to be included in all the reasoning of all the software components, which brings additional complexities to the verification of the entire system. But, if the machine model uses the abstract memory, then the virtual memory manager can not be verified in the same system, and thus must be trusted. With our framework, we can verify the virtual memory manager, and still use the address space model for the rest of the system.

The diagram of this design is in Figure 5.3. The diagram shows that the kernel is implemented using the machine that has an address space model. Most of the operations of that machine are just passed through to the underlying hardware machine, but some other operations that the kernel may use are implemented by the virtual memory manager.

5.2 The Code of a Virtual Memory Manager

The first step to the verification of the virtual memory manager is to write it. We have written our small virtual memory manager for our simplified hardware that features one-level address

```
#define PGSIZE  0x1000
#define NPAGES  0x1000
#define VPAGES  0x2000

extern void setPE(uint64_t);
extern void setPTROOT(uint64_t);
```

Figure 5.4: Hardware-Specified Constants and Functions (hw.h)

```
#define PMM 0x150000

void mem_init()
{
        uint64_t i=1;
        *PMM = 1;   // special page - keep it reserved
        while(i < 0xA0) // free pages between page 0 and 640KB
        {
                *(PMM+i*8) = 0;
                i = i + 1;
        }
        while(i < 0x200) // memory hole, code, static data (1MB - 2MB)
        {
                *(PMM+i*8) = 1;
                i = i + 1;
        }
        while(i < NPAGES)
        {
                *(PMM+i*8) = 0;
                i = i + 1;
        }
}

uint64_t mem_alloc()
{
  uint64_t curpage=1;
  uint64_t found=0;
  while(found == 0 && curpage < NPAGES)
  {
        if (PMM[curpage] == 0)
                found=1;
        else
                curpage=curpage+1;
  }
  if (found == 1) {
     PMM[curpage] = 1;
     return curpage;
  }
  else return 0;
}

void mem_free(uint64_t page)
{
        PMM[page] = 0;
}
```

Figure 5.5: Code of the Memory Allocator (mem.c)

```
#define PT   0x160000

void pt_set (uint64_t vaddr, uint64_t pg)
{
        *(PT + vaddr / PGSISE * 8) = pg;
}

uint64_t pt_lookup (uint64_t vaddr)
{
        return *(PT + vaddr / PGSIZE * 8);
}

void pt_init () {
  int i = 0;
  while(i<NPAGES) {
    *(PT + i * 8) = i;
        // page 0 is effectively unavailable
    i++;
  }
  while(i<VPAGES) {
    *(PT + i * 8) = 0;
    i++;
  }
}
```

Figure 5.6: Code of the Page Table System

```
uint64_t as_request(uint64_t page)
{
        uint64_t ppage;
        ppage = mem_alloc();
        if (ppage == 0) return 0;
        pt_set(page, ppage);
        return page;
}

void as_release(uint64_t vpage)
{
        uint64_t ppage;
        ppage = pt_lookup(vpage); //look up page
        mem_free(ppage); // free page
        pt_set(vpage,0); // unlink page
}
```

Figure 5.7: Code of the Address Spaces (as.c)

```
void init()
{
        mem_init();
        pt_init();
        setPTROOT(PT);
        setPE(1);
        kernel_init(); //never returns
}
```

Figure 5.8: Code of the Initialization (init.c)

translation mechanism. The code of the virtual memory is separated into several files, which are as follows:

- hw.h - headers defining the hardware of the machine
- mem.c - implementation of physical page allocator
- pt.c - functions that operate on page table data structures.
- as.c - functions that initialize and control address spaces.
- init.c - initialization and the rest of the high-level kernel.

The listings of these modules are given in Figures 5.4, 5.5, 5.6, 5.7, and 5.8.

Although the code is very simple, it is scattered around, and thus a quick explanation is needed. The virtual memory manager consists of a memory allocator, the page table manager, and an address space API implementation.

The hw.h header file contains several definitions that capture important information about the hardware, namely the number of pages of memory that the hardware can address, and the size of each individual page. It also contains the headers of two functions, whose code is not given, but they are effectively just wrappers on top of assembly code to set the control registers.

The memory allocator (mem.c) first defines a constant (`PMM`), which points to the area of memory where the allocation table is to be located. This is followed by three functions - the initialization, which creates an allocation table (a simple array that marks a page as allocated or free) at the location specified by the constant. The memory initialization marks several pages as in-use before it returns. These pages contain kernel code, and other data structures that the kernel requires (including the allocation table itself). This prevents these pages from being handed out for other purposes. The other functions are the `mem_alloc()` and `mem_free()` functions, which find and reserve a free page, or free an already reserved page, respectively. It must be noted that the allocator is simply an accountant - it does not, and can not prevent misuse. It simply guarantees that calling alloc will return a number of the page that was marked as free, and will mark it as used, while free will simply mark a page free.

The page table manager (pt.c) is similar to the memory allocator in that it also manages a table, whose location is also specified by a constant defined at the top of the module, in this case `PT`.

However, the page table that is being managed has to be exactly what is expected by hardware. The pt_init() functions sets up such a table. The table starts out mapping all pages from 0 to NPAGES to themselves, thus creating a direct mapping of all addresses (page 0 is not accessible in the virtual address space in our model). All the addresses that use pages greater than NPAGES are marked as 0, and thus the accessing these areas will cause a fault. The page table manager provides a function pt_set(), which allows one to set any value for any particular virtual page. There are no bounds checking in this version, as we will ensure them through certification. Setting a particular virtual page to 0 clears the mapping. The pt_lookup() function looks up the current value in the table for a particular virtual address. Once again, there are no bounds checks.

The address space API is simply the functions that make use of the memory allocator and the page table manager to provide the functionality that we expect our address space to have. It defines two functions: as_request() and as_release(). The as_request() does two things - it tries to allocate a fresh page, and then map it to the address requested by the caller. It may also fail, and return a 0, indicating that it failed. The as_release() function does the opposite - given a virtual page number, it finds out which actual page it maps to, and then frees the page and removes the mapping. Thus these two functions, together with mem_alloc() and mem_free() form the address space API that the higher-level components of the system will be using.

To activate all of these systems, we have the init() function. It initializes the memory allocator, and then initializes the page tables. Once these two systems are operational, it uses the special hardware calls (inline assembly wrapped in functions) to point the address translation to the initialized page table, and to activate address translation. Once this is done, it calls kenel_init() function, which represents the entry point into the rest of the system written (and certified) using the virtual address model of memory.

We should also note that we assume that the compiled code of the kernel and virtual memory manager is to be located at the address 0x100000-0x150000, and thus neither the code, nor the page allocation table, nor the page tables will ever overlap. The complete memory map that is defined by our virtual memory manager is given in Figure 5.9.

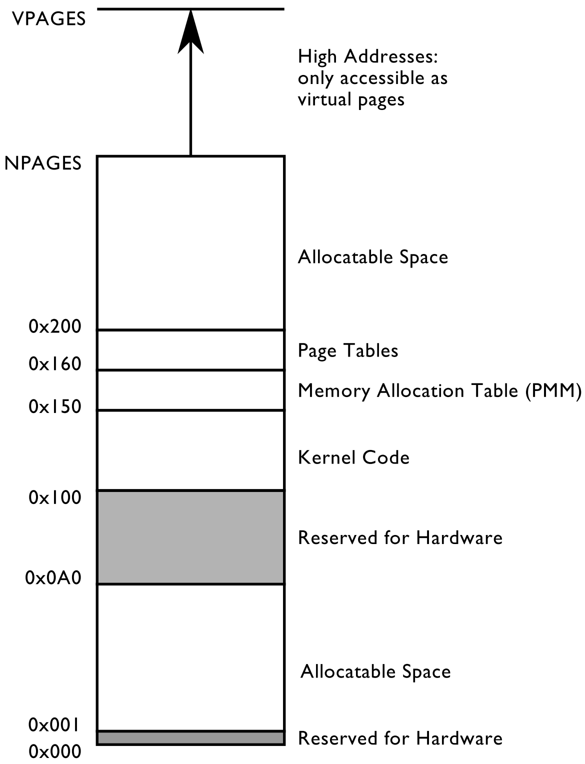


Figure 5.9: Memory Map Defined by Virtual Memory Manager

5.3 Formalized C language

To perform a verification of the virtual memory manager, we have chosen to do our work in a C-like language. However, this C language must expose much of the semantics of the actual hardware. A simple example of this is a memory-mapped I/O device that can be a part of the hardware. To provide access to the device, the C machine must connect the I/O range of memory addresses to the models of these devices. Similarly, to support address translation, the memory model of C must incorporate all of the details of the memory translation in the C memory model.

Before getting into the machines used for verification, we will present our simplified syntax of C given in Figure 5.10. Because C is such a complex language, we have decided that it is easier to start with a somewhat reduced form of this language, which, in spirit, is similar to C- -[22] or the intermediate languages used within the CompCert certified C compiler[26]. The language has been simplified in the following ways:

- No types. All values used in this language are 64-bit integers.

$$\begin{array}{rll}
(\textit{Physical Code Heap}) & \mathbb{Z} & ::= \textit{fname} \rightarrow \textit{fun} \\
(\textit{Function}) & \textit{fun} & ::= \textit{fname}(\texttt{list v})\{\mathbb{B}\} \\
(\textit{Statement}) & \texttt{s} & ::= \texttt{v} := \texttt{e} \mid *(\texttt{e}_{loc}) := \texttt{e} \mid \text{if}(\texttt{e})\{\mathbb{B}\} \mid \text{if}(\texttt{e})\{\mathbb{B}\}\text{ else }\{\mathbb{B}\} \\
 & & \quad \mid \text{while}(\texttt{e})\{\mathbb{B}\} \mid \texttt{v} := \textit{fname}(\text{list }\texttt{e}) \mid \text{ret}(\texttt{e}) \\
(\textit{Block}) & \mathbb{B} & ::= \texttt{list s} \\
(\textit{Expression}) & \texttt{e} & ::= w \mid \texttt{v} \mid *\texttt{e} \mid \textit{unop}(u, \texttt{e}) \mid \textit{binop}(b, \texttt{e}, \texttt{e}) \\
(\textit{Binary Operator}) & b & ::= + \mid - \mid \times \mid \textit{div} \mid \textit{mod} \mid \&\& \mid \| \mid = \mid < \mid <= \mid > \mid >= \\
(\textit{Unary Operator}) & u & ::= \,! \mid -(\textit{unary}) \\
(\textit{Labels}) & \texttt{f} & ::= i \text{ (natural numbers)} \\
(\textit{Names}) & \textit{fname}, \texttt{v} & ::= \textit{(a decidable set of names)} \\
(\textit{Words}) & w & ::= \text{(64-bit integers)}
\end{array}$$

Figure 5.10: Syntax of the C-like language

- No global variables. This can be replaced by using predetermined memory locations.

- No structures, arrays, etc. All data has to be accessed using pointer arithmetic. This is not a big change, since most complex accesses into data structures is just syntactic sugar for pointer arithmetic.

- All functions have to have a return value - e.g. use a return operation before completing the function. Each function call is required to assign the result into a variable. Our syntax, however, does not enforce this, and it also permits code that has the return operation in the middle of functions. However, it does not make sense to return without a proper return frame, nor does it make sense to set up a return frame and continue executing the same function. Even though our syntax permits such functions, they will likely not be certifiable.

- Variables are created dynamically on assignment.

- Function calls are statements and not expressions.

- No assignment expressions, e.g. (x++), or other esoteric features

The above syntax can account for most common C programs after some desugaring and small amount of linearization. However, it is not our goal to use this as a language in our certification. Since our certification system works over the meta-language, we will actually redefine C as a list of operations combined with the meta-language. The above syntax will serve as a starting point of

$$\begin{array}{rl}
(\mathit{Stack}) & S ::= \mathit{nil} \mid F :: S \\
(\mathit{Frame}) & F ::= \mathit{Call}(\mathit{list}\ w) \mid \mathit{Data}(\{\mathtt{v} \rightsquigarrow w\}) \mid \mathit{Ret}(w) \\
(\mathit{Variables}) & \mathtt{v} ::= \textit{(a decidable set of names)} \\
(\mathit{Words}) & w ::= n \text{ (integers)}
\end{array}$$

Notation:

$$\begin{array}{lll}
\mathtt{v} & \triangleq f(v) & \mathtt{if}\ \ \mathit{top}(S) = \mathit{Data}(f) \\
\mathtt{v} := z & \triangleq S := (\mathit{Data}(f\{\mathtt{v} \rightsquigarrow z\}) :: S') & \mathtt{if}\ \ S = \mathit{Data}(f) :: S' \\
\mathtt{call}(zl) & \triangleq S := (\mathit{Call}(zl) :: S) & \\
\mathtt{callfin}(vl : \mathtt{list\ v}) & \triangleq S := (\mathit{Data}(\mathit{mkFrame}(vl, al)) :: S' & \mathtt{if}\ \ S = \mathit{Call}(al) :: S') \\
\mathtt{return}(z) & \triangleq S := (\mathit{Ret}(z) :: S') & \mathtt{if}\ \ S = \mathit{Data}(f) :: S' \\
\mathtt{retfin}(\mathtt{v}) & \triangleq S := (\mathit{Data}(f\{\mathtt{v} \rightsquigarrow w\}) :: S') & \mathtt{if}\ \ S = \mathit{Ret}(w) :: \mathit{Data}(f) :: S'
\end{array}$$

where

$$\mathit{mkFrame}(vl, al) \triangleq \begin{cases} \emptyset & \mathtt{if}\ \ vl = al = \mathit{nil} \\ \{\mathtt{v} \rightsquigarrow w\} \cup \mathit{mkFrame}(vl', al') & \mathtt{if}\ \ vl = \mathtt{v} :: vl'\ \ \mathtt{and}\ \ al = w :: al' \end{cases}$$

Figure 5.11: Basic C Machine Stack

such automated conversion. But before we can get to the mapping of C into our meta-C, we must give a complete account of the meta-C language itself.

The meta-C language is just the semantics of C computation, expressed in terms of our meta-framework. Thus we must define the state of the C machine, and the set of operations that can be performed over this machine.

However, instead of defining the C machine in one shot, we will instead build it up from two modules - a module defining the notion of a variable stack, and a module that defines the memory system.

5.3.1 The C-Machine Stack Definition

Our definition of the type of the state of the C machine must include the definition of stack. This stack is modelled by the definitions in Figure 5.11. The stack is a list of frames, which can be of three types: call frames used to pass arguments into a called function, data frames that keep track of local variables, and return frames used to pass return values to the callee.

To help manipulate the stack data structure, we have defined a few notational predicates. For example, the notation that consists of just a name of the variable will return the value contained

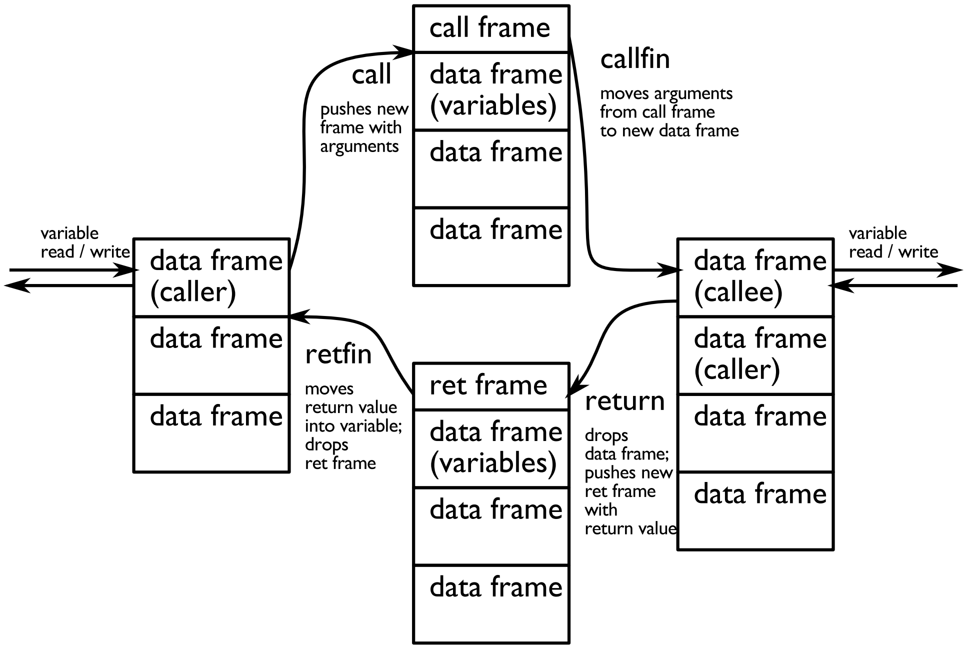


Figure 5.12: Diagram of Stack Behavior

in top frame of the stack for a variable with that name. If the top frame is not a data frame, then the lookup fails. The notation that shows the assignment to a variable (`v` := z) will update the top frame of the stack so that variable `v` will now contain the new value z. The `call` notation updates the stack to push on a new call frame with the arguments placed onto it. This call frame can then be processed using the `callfin` predicate, which will take the top call frame off the stack, place a fresh data frame, and populate the variables with the argument values. The `return` and `retfin` do similar updates to the stack, except with return frames.

These notations help us to quickly define common stack behavior that occurs during execution, namely variable access and function call and return. This behavior is digrammed in Figure 5.12.

It is important to note that the stack is not a machine by itself, but rather a data structure and some notation that we will later use to define the semantics of the C machine.

5.3.2 The Memory Model

As we have explained earlier, the semantics of the C language only require that we are able to read and write from an address in memory. How that memory behaves is not a part of the C specification. Thus, the semantics of the C language are parametric over some definition of memory. Any

(*Memory System*) $M ::= \ldots$

Operation	Meaning
$M(vaddr)$	Get the value at address *vaddr*
$M(vaddr) := w$	Store *w* at address *vaddr*

Figure 5.13: Signature of the Memory Interface for the C Semantics

such definition of memory, however, needs to implement a specific interface that is used by the C semantics. This interface has a specific signature, which is shown in Figure 5.13.

The signature is extremely minimal. A memory is just any state whatsoever. The only thing that the memory must define is a load ($M(addr)$) and store ($M(addr) := w$) operations. The state and the operations, however, are completely arbitrary. There is no requirement that store will actually record a value that can be later read by a load. There is not even the requirement that these operations succeed for any particular address. In a sense, the semantics of memory are completely arbitrary.

Thus, to make sense of any program that uses memory, it will be important to know which precise memory model is in use. For example, both the address translated memory, and virtual address space can be made to fit this interface, but the certification will be different depending on the model. We will define several memory models, after we have completed our definition of the C machine.

5.3.3 The Semantics of Meta-C

Using these definitions of stack and memory interface, we can define the semantics of the meta-C language. These semantics are given in Figure 5.14. The meta-C differs from the original C in the following ways:

- Function call is split into two operations - the fcall and readret. The former begins the function call, the latter cleans up the function call, and assigns the return value to a variable.

- An operation readargs is present in the new meta language. It is usually the first operation within a function. Its purpose is to set up the new stack frame for a function, as well as to assign the arguments coming from the function call into variables.

- There are no control flow operations. Instead, all control flow operations are handled by our

$$
\begin{aligned}
(\mathit{State}) \quad & \mathbb{S} ::= (M, S) \\
(\mathit{Operation}) \quad & \iota ::= \mathtt{v} := \mathtt{e} \mid *(\mathtt{e}_{loc}) := \mathtt{e} \mid \mathrm{fcall}(\mathit{fname}, \mathrm{list}\ \mathtt{e}) \mid \mathrm{ret}(\mathtt{e}) \\
& \quad \mid \mathrm{readargs}(\mathrm{list}\ \mathtt{v}) \mid \mathrm{readret}(\mathtt{v})
\end{aligned}
$$

Operation if ι =	Action then $\mathcal{M}(\iota)$ =
$\mathtt{v} := \mathtt{e}$	$\mathtt{v} := \mathit{eval}_{\mathbb{S}}(\mathtt{e})$
$*(\mathtt{e}_{loc}) := \mathtt{e}$	$M(\mathit{eval}_{\mathbb{S}}(\mathtt{e}_{loc})) := \mathit{eval}_{\mathbb{S}}(\mathtt{e})$
$\mathrm{fcall}(\mathtt{f}, \mathit{el})$	$\mathtt{call}(\mathit{map\ eval\ el})$
$\mathrm{readargs}(\mathit{vl})$	$\mathtt{callfin}(\mathit{vl})$
$\mathrm{readret}(\mathtt{v})$	$\mathtt{retfin}(\mathtt{v})$
$\mathrm{ret}(\mathtt{e})$	$\mathtt{return(eval(e))}$

where

$$
\mathit{eval}_{\mathbb{S}}(\mathtt{e}) ::= \begin{cases}
w & \text{if } \mathtt{e} = w \\
\mathbb{S}.S(\mathtt{v}) & \text{if } \mathtt{e} = \mathtt{v} \\
M(\mathit{eval}_{\mathbb{S}}(\mathtt{e}_1)) & \text{if } \mathtt{e} = (*\mathtt{e}_1) \\
\mathit{bop}(\mathit{eval}_{\mathbb{S}}(\mathtt{e}_1), \mathit{eval}_{\mathbb{S}}(\mathtt{e}_2)) & \text{if } \mathtt{e} = \mathit{binop}(\mathit{bop}, \mathtt{e}_1, \mathtt{e}_2) \\
u(\mathit{eval}_{\mathbb{S}}(\mathtt{e}_1)) & \text{if } \mathtt{e} = \mathit{unop}(u, \mathtt{e}_1)
\end{cases}
$$

A bit of notation:

$$
\left([\mathit{arglist}] \mapsto \begin{cases} \mathtt{a}_1 \\ \mathtt{a}_2 \end{cases} \right) \triangleq \mathtt{readargs}(\mathit{arglist}) \circ (\mathtt{a}_1 \vee \mathtt{a}_2)
$$

$$
\left([\mathit{arglist}] \mapsto \begin{cases} \mathtt{a}_1 \mapsto n_1 \\ \exists x.\, \mathtt{a}_2 \mapsto n_2 \end{cases} \right) \triangleq \mathtt{callfin}(\mathit{arglist}) \circ (\mathtt{a}_1 \circ \mathit{return}(n_1)) \vee (\bigvee_x \mathtt{a}_2 \circ \mathtt{return}(n_2))
$$

Figure 5.14: Primitive C-like machine

meta-language control flow.

The state of our meta-C consists of a tuple containing the state of the stack, and the state of the memory. The operations of the machine closely mimic the original C-like language whose syntax we have defined earlier. The semantics of these operations usually depends on the operations provided by the model of the stack and the model of the memory. For example, the semantics of the assignment operation (`v := e`), uses the definition of the variable assignment operation defined by the stack model. Similarly the dereference expression (`*e`) makes use of the memory load defined by the memory model ($M(w)$). Thus the semantics of the C language are just a proper combination of the operations over the stack and memory as needed by the individual operations within the C language.

These combinations of operations always remain the same for any C-language definitions. However, whether the operation is successful, and the operations actual effects are dependent on the exact semantics of the models of memory and stack. We have already defined the structure of the stack, and we must now focus on particular implementations of memory to get a complete and precise picture of how the C language actually works.

5.3.4 The Hardware Memory Model

In this section, we will define a memory model that will include the notion of address translation that is defined by hardware, and is the part of the hardware for which we are creating an abstraction. First we will informally discuss the memory, and then we will provide a formal definition.

The hardware in our system deals with memory in groups of addresses that we call pages. A page of memory is 4 KB in size, and thus has 512 64-bit words. To deal with these pages, we create several constants and predicates that allow us to deal with addresses and pages. Also since the memory is finite, we also list the number of pages and addresses that the memory has. These definitions are given in Figure 5.15. The figure lists several constants, which are also equivalent to the constants defined within our actual code.

Figure 5.16 gives a formal model of the address translated memory system. This model is a formal definition of the actual address translation implemented by our simplified hardware. This model includes the load and store operations from addresses, and thus satisfies the signature that is

Definition	Value	Description
PGSIZE	4096	Number of bytes per page
NPAGES	unspecified	Number of phys. pages in memory
MEMSIZE	NPAGES*PGSIZE	Total bytes of memory
VPAGES	unspecified	Maximum page number of a virtual address
Pg($addr$)	$addr/\mathtt{PGSIZE}$	gets page of address
Off($addr$)	$addr\%\mathtt{PGSIZE}$	offset into page of address
LowPg(pg)	$0 \le pg < \mathtt{NPAGES}$	valid physical page number
HighPg(pg)	$\mathtt{NPAGES} \le pg < \mathtt{VPAGES}$	valid physical page number
Low(addr)	LowPg(Pg(addr)) $\wedge\ addr\%8 = 0$	valid physical page address
High(addr)	HighPg(Pg(addr)) $\wedge\ addr\%8 = 0$	valid virtual page address
PMMdom($addr$)	$\mathtt{PMM} \le addr < (\mathtt{PMM} + \mathtt{NPAGES})$	allocation table address
PTdom($addr$)	$\mathtt{PT} \le addr < (\mathtt{PT} + \mathtt{VPAGES})$	pagetable address

Figure 5.15: Page Definitions

$$\begin{array}{rll}
(\textit{Memory System}) & M & ::= (D, \mathtt{PE}, \mathtt{PTROOT}) \\
(\textit{Data}) & D & ::= \{addr \rightsquigarrow w \mid \text{Low}(addr)\}^* \\
(\textit{Address Translation Enabled}) & \mathtt{PE} & ::= b \text{ (bool)} \\
(\textit{Pointer to AT Table}) & \mathtt{PTROOT} & ::= w \text{ (address)}
\end{array}$$

Notation	Definition
$M(vaddr)$	$D(trans_M(vaddr))$
$M(vaddr) := w$	$D(trans_M(vaddr)) := w$

where

$$trans_M(va) \quad := \begin{cases} M(M.\mathtt{PTROOT} + \text{Pg}(va) * 8) * \mathtt{PGSIZE} + \text{Off}(va) & \text{if } M.\mathtt{PE} = \mathit{true} \\ va & \text{otherwise} \end{cases}$$

Function	Specification
hw-setPE	$[] \mapsto (\mathit{ValidPT}(\mathtt{PTROOT})?, \mathtt{PE} := \mathtt{true}, \mathtt{ret}(0))$
hw-setPTROOT	$[newroot] \mapsto (\mathit{ValidPt}(newroot)?, \mathtt{PTROOT} := newroot, \mathtt{ret}(0))$

Figure 5.16: Address Translated Memory Interface (M_{HW}) and Stub Library ($\mathcal{L}_{HW}$)

required of the memory models to be used with our C machine.

This definition of memory uses the state that consists of the data store (D), which represents the contents of the memory, and two control registers - a boolean register PE, which controls whether address translation is active or not, and PTROOT register, which points to the location of a single level pagetable within the memory. The data store is restricted to only valid addresses.

The fact that this model defines a single level pagetable can be seen in the load and store operations defined by this machine. Instead of directly accessing the data, addresses go through a translation function defined by $trans_M$. That function shows the effect of having PE set or not, as well as the purpose of the PTROOT register, which is used as a starting address of a single level pagetable.

We refer to this model of memory as M_{HW}, which we have called this way to indicate that this model is supposed to faithfully simulate the memory provided by our simplified hardware. Since this memory satisfies the interface required by our C machine, we can instantiate the C machine with this memory model. We will refer to this C machine as $\mathcal{M}_{HW}$ to indicate that this is both a complete machine, and that it uses the HW memory model.

Using this machine, we would now be able to prove C programs written for the address translated memory.

5.3.5 Dealing with Special Memory Features

A careful reader might note that there is what seems to be a significant problem with our machine model, namely that the memory model exposes several features that the C language is not capable of using. For example, our C-language lacks a way to update the PE or PTROOT registers of the memory that enable the address translation and modify its page table pointer.

The way that we handle this issue is by defining a stub library that includes primitive stubs for functions that set the appropriate registers. To show concretely what we mean, the functions for the $\mathcal{M}_{HW}$ machine are given by $\mathcal{L}_{HW}$. This library is a specification heap that contains the two functions: hw-setPE and hw-setPTROOT. The first one is used to switch the address translation on, and the second one to update the PTROOT register.

The specifications of these primitive hypothetical functions makes them appear to be functions. That means that they expect an argument frame, and result in a return frame, preserving the rest of

$$\begin{array}{rll}
(\mathit{Memory\ System}) & M & ::= (D, A) \\
(\mathit{Data\ Store}) & D & ::= \{addr \rightsquigarrow w \mid \mathrm{Low}(addr) \vee \mathrm{High}(addr)\}^* \\
(\mathit{Page\ Allocation}) & A & ::= \{pg \rightsquigarrow \mathtt{b} \mid \mathrm{LowPg}(pg) \vee \mathrm{HighPg}(pg)\}^* \\
(\mathit{Words}) & addr, pg, w & ::= \mathit{(64\text{-}bit\ values)}
\end{array}$$

Notation	`Definition`
$M(va)$	$(M.A(Pg(addr))?,\ M.D(addr))$
$M(va) := w$	$(M.A(Pg(addr))?,\ M.D(addr) := w)$

Label	Specification
`mem-alloc`	$[] \mapsto \begin{cases} \mathtt{ret}(0) \\ \bigvee_{pg} (\mathrm{LowPg}(pg)?, (pg \neq 0)?, A(pg) = \mathrm{false}?, A(pg) := \mathrm{true}, \mathtt{ret}(pg)) \end{cases}$
`mem-free`	$[pg] \mapsto (\mathrm{LowPg}(pg)?,\ A(pg) = \mathit{true}?,\ A(pg) := \mathrm{false}, \mathtt{ret}(0))$
`as-reserve`	$[vpg] \mapsto \begin{cases} (\mathrm{HighPg}(vpg)?, \mathtt{ret}(0)) \\ (\mathrm{HighPg}(vpg)?, A(vpg) = \mathrm{false}?, A(vpg) := \mathrm{true}, \mathtt{ret}(vpg)) \end{cases}$
`as-release`	$[vpg] \mapsto (\mathrm{HighPg}(vpg)?,\ A(vpg) = \mathrm{true}?,\ A(vpg) := \mathrm{false}, \mathtt{ret}(0))$

Figure 5.17: Address Space Memory Interface (M_{AS}) and Library ($\mathcal{L}_{AS}$)

the stack. This mechanism enables us to simply use the function call mechanism as a way to extend the operations of the machine beyond those provided by the C language directly.

There remains a concern about how to implement these functions if they are not definable in the machine. The answer to this question is that that these functions are a part of the trusted base of this particular C machine with this particular memory. Since the C machine is not equivalent to hardware, these functions can be implemented in code, but not in pure C code with HW memory model that we have laid out. Thus, in this machine, we do not define the actual code of these functions, and we trust the fact that their effect is completely described by the specifications that are given.

If we were to define a machine which corresponds to the assembly code of the processor, we would be able to write and certify these functions. We, however, leave this out as future work.

5.3.6 Address Space Memory Model

To show that verification can be made simpler using a different memory model, in this section we will show a full formal definition of M_{AS}, the virtual address space memory model. This model will be used to verify all the code that runs on top of our virtual memory manager.

Figure 5.17 gives the formal definition of a memory system with a single virtual address space. The model defines the memory as a pair of data store and page-based allocation table. The data store is similar to that of the M_{HW} model, except its domain has grown from all valid physical (low) addresses, to all virtual addresses (both low and high). The loads and stores in this memory have completely lost the translation function - there is no translation of addresses in this model. However, they now have an additional requirement that any address being accessed must be marked as valid in the allocation table (A). Other than this check and the low-high separation of memory, this memory model is about as simple definition of virtual memory as one can imagine.

When we combine this memory model with our C-machine, we define $\mathcal{M}_{AS}$, a machine with an address space memory model over the entire virtual address space. The fact that this machine allows simpler use of virtual addresses than $\mathcal{M}_{HW}$, makes it very useful for verification of all the code that sits on top of our virtual memory manager.

However, just like we have seen in the M_{HW}, not all features of the memory are accessible directly from the language, e.g. there is no way to adjust the allocation table. In fact all the operations that we have describe in section 5.1, such as page requests and releases, can not be controlled by the C language directly. Thus we define a primitive library ($\mathcal{L}_{AS}$) to be used with the M_{AS} memory model. The library features the four function stubs defined by the model to be used by programs:

- **mem-alloc.** This function allocates a page in the physical (low) memory area. Any page within the physical memory may be allocated. There is no way to request a specific page. Also, the function may fail to reserve a page and return a 0 for any reason.

- **mem-free.** This function frees a page in the physical memory area. The page must be allocated before it can be freed. The function does not fail as long as preconditions are satisfied.

- **as-reserve.** This function is designed to allocate a specific virtual page. The requirement is that this virtual page is not previously allocated (and is in the high memory range). If the function succeeds, it returns the page number allocated, which necessarily matches the parameter passed to the function. The function may also fail to allocate a page, and return a 0 for any reason. Although it is not visible from this specification, the function will actually use up a physical page of memory, and thus some page of memory in the low memory space will not be available for physical allocation. This fact, however, is completely abstracted by

$$\mathit{convert}(\mathbb{Z}) \quad := \bigcup_{\mathit{fun}\,\in\,\mathbb{Z}} \mathit{convert}(\mathit{fun},\emptyset)$$

$$\mathit{convert}(\mathit{fun},\mathbb{C}) \quad := \begin{cases} \texttt{let }(\mathbb{I}_1,\mathbb{C}_1) := \mathit{convert}(\mathbb{B},\mathbb{C})\texttt{ in} & \\ \mathbb{C}_1 \cup \{\mathit{fname} \rightsquigarrow (\texttt{readargs}(\mathit{vl});\mathbb{I}_1)\} & \text{if } \mathit{fun} = \mathit{fname}(\mathit{vl})\{\mathbb{B}\} \end{cases}$$

$$\mathit{convert}(\mathbb{B},\mathbb{C}) \quad := \begin{cases} (\mathit{nil},\mathbb{C}) & \text{if } \mathbb{B} = \mathit{nil} \\ \texttt{let }(\mathbb{I}_1,\mathbb{C}_1) := \mathit{convert}(\mathtt{s},\mathbb{C})\texttt{ in} & \\ \texttt{let }(\mathbb{I}_2,\mathbb{C}_2) := \mathit{convert}(\mathbb{B}',\mathbb{C}_1)\texttt{ in} & \text{if } \mathbb{B} = \mathtt{s} :: \mathbb{B}' \\ (\mathbb{I}_1;\mathbb{I}_2,\mathbb{C}_2) & \end{cases}$$

if $\mathtt{s}$ =	then $\mathit{convert}(\mathtt{s},\mathbb{C})$ =
$\mathtt{v} := \mathtt{e}$	$(\mathtt{v} := \mathtt{e},\mathbb{C})$
$*(\mathtt{e}_{loc}) := \mathtt{e}$	$(*(\mathtt{e}_{loc}) := \mathtt{e},\mathbb{C})$
if(e)$\{\mathbb{B}_1\}$	$\texttt{let }(\mathbb{I}_1,\mathbb{C}_1) := \mathit{convert}(\mathbb{B}_1,\mathbb{C})\texttt{ in}$ $((\mathtt{e} \neq 0?\ \mathbb{I}_1 + \texttt{nil}),\mathbb{C}_1)$
if(e)$\{\mathbb{B}_1\}$ else $\{\mathbb{B}_2\}$	$\texttt{let }(\mathbb{I}_1,\mathbb{C}_1) := \mathit{convert}(\mathbb{B}_1,\mathbb{C})\texttt{ in}$ $\texttt{let }(\mathbb{I}_2,\mathbb{C}_2) := \mathit{convert}(\mathbb{B}_2,\mathbb{C}_1)\texttt{ in}$ $((\mathtt{e} \neq 0?\ \mathbb{I}_1 + \mathbb{I}_2),\mathbb{C}_2)$
while(e)$\{\mathbb{B}_1\}$	$\texttt{let }(\mathbb{I}_1,\mathbb{C}_1) := \mathit{convert}(\mathbb{B}_1,\mathbb{C})\texttt{ in}$ $([\mathit{new}],\mathbb{C}_1 \cup \{\mathit{new} \rightsquigarrow ((\mathtt{e} \neq 0?\ (\mathbb{I}_1;[\mathit{new}]) + \texttt{nil}))\})$
$\mathtt{v} := \mathit{fname}(\mathit{el})$	(fcall($\mathit{fname},\mathit{el}$); $[\mathit{fname}]$; readret(v)$,\mathbb{C}$)
ret(e)	(ret(e)$,\mathbb{C}$)

Figure 5.18: Conversion of C into Meta-C

our semantics.

- **as-release.** This function will release a virtual page in the high memory area back into the free pages list. The requirement to run this function is that the virtual page is already allocated. The function will then remove the page from the domain of the memory. As long as the page was allocated before calling the as-release, it will not fail to release the page. The function always returns 0.

With these primitive functions our abstract memory model is complete.

5.3.7 Converting from C to Meta-C

Now that we have discussed our C machine that will be used for verification, we can now talk about how the actual C code and the C machine are related. Figure 5.18 shows the conversion process from the functions of the C code into a meta-machine codeheap that works with our C machine described above. The conversion process converts all functions into procedures labelled with the same name. Loops are also converted into procedures, as is required by our meta-machine.

Of specific interest is the conversion of the function call, which gets converted into three operations: the setup of the call (using fcall), the call to the procedure labelled with the same name (the conversion assumes that all functions get converted into the same names, and thus the label will exist in the code heap), and the readret post-processing to remove the return frame, and place the return value into a variable.

A while loop is converted into a procedure that makes use of the branch operation of the meta-machine. From there the branch operation handles all the conditional statements, without the need for additional operations to handle control flow.

Although we will not prove this fact (as we would need complete semantics of the C language), the conversion process effectively defines a meta-program that is equivalent in execution to the original C program. This is important for several reasons:

1. Any verification over the converted program is a verification over the C program.

2. We do not need to use state to keep track of where we are in the code through the use of instruction pointers or program rests. The procedures uniquely determine our execution and thus our place in the code, making that information redundant.

3. The fact that the converted program has a specific pattern allows us to map back the specifications to the original program. This can be important if this work is to be extended with certified compilation (and corresponding specification translation) of the program into assembly code.

Thus at this point, we have defined an abstract C machine that can be useful for certifying our virtual memory manager.

5.4 Verification Plan

At this point, we have defined the two machines that define the essence of our multi-machine verification. We have the $\mathcal{M}_{HW}$ machine, which defines the semantics that correspond to the C code running on the machine with address translation, and the $\mathcal{M}_{AS}$ machine that corresponds to the C code that assumes a virtual address space.

Since, we are focusing on the certification of the virtual memory manager, we will make the assumption that the rest of the kernel is already certified. Since the kernel assumes the virtual address space semantics, it would be certified in the machine $\mathcal{M}_{AS}$. More formally, this means that there is some code, which we can call $\mathbb{C}^{kernel}$, and there is a specification Ψ_{AS}^{kernel}. We do not know what the specifications are, but we know that the kernel is certified. Moreover, we make the assumption about the entry point of the kernel - namely that it is a function that expect zero arguments, never returns, and does not expect anything in particular about the contents of the virtual address space.

In other words, we assume the following

Definition 5.4.1 (Kernel Soundness)

$$\mathcal{M}_{AS}, \mathcal{L}_{AS} \vdash \mathbb{C}^{kernel} : \Psi_{AS}^{kernel}$$

where

$$\Psi_{AS}^{kernel}(\texttt{kernel-init}) \subseteq [] \mapsto loop$$

We have already defined the code of our virtual memory manager. To proceed with its certification, we will take the actual text of the C code, and convert it to the procedures or our meta-C language using the conversion we have laid out. We will also group the individual procedures into modules by placing the related modules together into procedure heaps. The result would be the procedure heaps that are given in the following definition.

Definition 5.4.2 (Code of the Virtual Memory Manager)

$\mathbb{C}^{mem}$	- `mem-alloc` and `mem-free` functions
$\mathbb{C}^{meminit}$	- `mem-init` function
$\mathbb{C}^{pt}$	- `pt-set` and `pt-lookup` functions
$\mathbb{C}^{ptinit}$	- `pt-init` function
$\mathbb{C}^{as}$	- `as-request` and `as-release` functions
$\mathbb{C}^{init}$	- `init` function that serves as entry point

Our goal in this verification is to get a complete certification of the kernel running on the address translated memory ($\mathcal{M}_{HW}$ machine). To do this, we would have to somehow create the $\mathcal{M}_{HW}$-based specifications for all our modules: Ψ^{mem}_{HW}, $\Psi^{meminit}_{HW}$,Ψ^{pt}_{HW}, Ψ^{ptinit}_{HW}, Ψ^{as}_{HW}, Ψ^{init}_{HW}. Moreover we would have to somehow come up with the $\mathcal{M}_{HW}$ specification for the kernel (Ψ^{kernel}_{HW}), but we currently know only its $\mathcal{M}_{AS}$ specification. Then with these specifications, we would need to produce the following result in order to complete the certification:

$$\mathcal{M}_{HW}, \mathcal{L}_{HW} \vdash \mathbb{C}^{mem} \cup \mathbb{C}^{pt} \cup \mathbb{C}^{as} \cup \mathbb{C}^{meminit} \cup \mathbb{C}^{ptinit} \cup \mathbb{C}^{init} \cup \mathbb{C}^{kernel} :$$
$$\Psi^{mem}_{HW} \cup \Psi^{pt}_{HW} \cup \Psi^{as}_{HW} \cup \Psi^{meminit}_{HW} \cup \Psi^{ptinit}_{HW} \cup \Psi^{init}_{HW} \cup \Psi^{kernel}_{HW}$$

5.4.1 The Two-Machine Approach

This verification problem is exactly what our framework was designed to solve. The kernel is already verified using the $\mathcal{M}_{AS}$ machine, and we certainly do not want to have to re-certify it. Instead, we refine the original specification, and then link it with all the other pieces.

So, suppose we are successful at defining a refinement from $\mathcal{M}_{AS}$ to $\mathcal{M}_{HW}$. Such a refinement would define a specification translation function, T_{AS-HW}, which will also guarantee the following property:

$$\frac{\mathcal{M}_{AS}, \mathcal{L}_{AS} \vdash \mathbb{C}^{kernel} : \Psi^{kernel}_{AS}}{\mathcal{M}_{HW}, T_{AS-HW}(\mathcal{L}_{AS}) \vdash \mathbb{C}^{kernel} : T_{AS-HW}(\Psi^{kernel}_{AS})}$$

This means that we can use $T_{AS-HW}(\Psi^{kernel}_{AS})$ as Ψ^{kernel}_{HW}, and the kernel is valid under this specification, but is dependent on the specification of the $\mathcal{M}_{AS}$'s primitive library, $T_{AS-HW}(\mathcal{L}_{AS})$. However, what we will show is that this library signature is exactly implemented by our memory manager modules. In other words we would need to prove the following:

$$T_{AS-HW}(\mathcal{L}_A S) \supseteq \mathcal{L}_{HW} \cup \Psi^{mem}_{HW} \cup \Psi^{pt}_{HW} \cup \Psi^{as}_{HW}$$

And from the above definitions, as well as the assumption of kernel's soundness (Definition 5.4.1), we would get the following result:

$$\mathcal{M}_{HW}, \mathcal{L}_{HW} \cup \Psi^{mem}_{HW} \cup \Psi^{pt}_{HW} \cup \Psi^{as}_{HW} \vdash \mathbb{C}^{kernel} : T_{AS-HW}(\Psi^{kernel}_{AS})$$

The next step would be to show that the virtual memory functions are valid. We would have to show this by single machine verification:

$$\mathcal{M}_{HW}, \mathcal{L}_{HW} \vdash \mathbb{C}^{mem} \cup \mathbb{C}^{pt} \cup \mathbb{C}^{as} : \Psi_{HW}^{mem} \cup \Psi_{HW}^{pt} \cup \Psi_{HW}^{as}$$

Then we would have to show the initialization pieces, which are just verifications of the init codes, except that they have to call the kernel. In order to prove those, we would have to make sure that we can call the `kernel-init` function of the kernel, and for that - we would need to have its specification translated to the $\mathcal{M}_{HW}$ machine. More precisely we would have to get the following verification:

$$\mathcal{M}_{HW}, \mathcal{L}_{HW} \cup T_{AS-HW}(\Psi_{AS}^{kernel}(\text{kernel-init})) \vdash \mathbb{C}^{meminit} \cup \mathbb{C}^{ptinit} \cup \mathbb{C}^{init} : \Psi_{HW}^{meminit} \cup \Psi_{HW}^{ptinit} \cup \Psi_{HW}^{init}$$

If we take these three pieces and link them together, we will achieve our goal.

The approach we have defined here is a completely valid one. However, it is not necessarily simple. There are several large steps that have to taken:

- The refinement from $\mathcal{M}_{AS}$ to $\mathcal{M}_{HW}$ is very complicated to define. There are several data structures that cooperate to define this relation, and the amount of data makes proofs complicated. Defining this relation in stages can make these definitions simpler.

- The address space code, which does not depend on the precise details of the page table structures or memory allocation structures would be easier to certify on a more abstract memory model, that is in between address translation model and virtual address space model.

In a sense, what we are trying to argue is that the virtual memory manager makes a very large abstraction leap, and it might be better to define it as several small jumps of abstraction that are chained together. This would be even more important if we move to more realistic and more complicated models of memory - involving more features and larger and much more complicated code base. Thus to both plan for the future, and to show that our framework is capable of handling larger problems by splitting them into series of small problems, we try to verify the virtual memory manager using several (as in more than two) layers.

5.4.2 The 7-layer Certification Cake

Our plan for certification involves creating several abstract machines in-between the hardware machine and the address space machine. This will allow us to construct smaller refinements and then stack them together. The result can be seen in Figure 5.19.

At the bottom of the diagram, we have the C machine with the HW memory model. This machine is our definition of the hardware. On the right side of the machine definition, we can see that in addition to the standard C semantics, it exposes two primitives of the HW model. On the left half of the diagram, we can see that the HW machine is being abstracted into the PE machine, a small simplification that restricts the address translation. In this PE machine, we will implement our memory allocator, which will then play a role in the ALE machine - the next abstraction. The ALE machine has memory allocation as part of the operational semantics of memory. In this abstraction, we will implement our page table library, which will, together with the memory abstraction from another layer which will hold an abstract page map as a part of the memory system. Then on top of this machine, we can certify the address space library, which will serve as a basis for the AS memory model used to certify the high-level kernel.

The right half of the diagram is a parallel of the left, except the assumption in those machines is that the paging is turned off entirely, making certification and analysis even easier.

In the next several sections we will describe each of these layers in detail, followed by the verification of the code modules using the appropriate memory models shown in the diagram, and then we will show that each of the abstraction links is valid, allowing us to put the entire certification together.

5.4.3 Address Translation Restriction

There are many design choice that the kernel must make, and stick to them. One of these choices is how the virtual address space is set up. In our case, we have decided that the kernel will use the low area of memory (those addresses that correspond to physical memory address) for directly referencing the physical memory, while the high-address areas (above the physical memory addresses) will be virtual. This means that the kernel always maintains an invariant that guarantees that the address translation will be an identity map for all pages numbered 0 to `NPAGES`, whenever the address

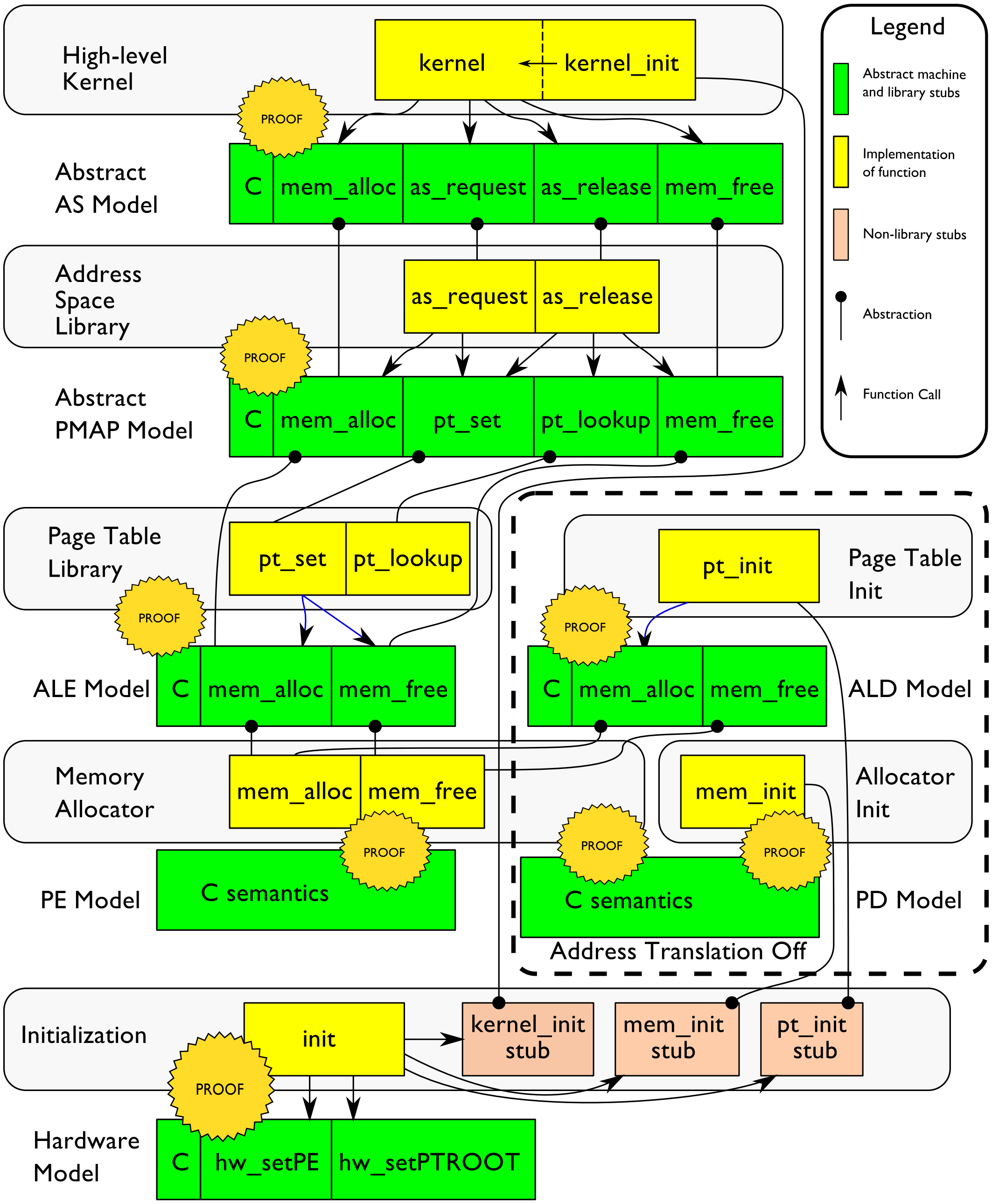


Figure 5.19: Kernel Abstraction Diagram

$$
\begin{array}{rl}
(\textit{Memory System}) & M ::= \{addr \rightsquigarrow w \mid \mathrm{Low}(addr)\}^* \\
(\textit{Page Table Location}) & \mathtt{PT} ::= w\ \textit{(CONSTANT)}
\end{array}
$$

Notation	Definition
$M(va)$	$\mathrm{Low}(trans_M(va))?,\ dm?,\ M(trans_M(va))$
$M(va) := w$	$\mathrm{Low}(trans_M(va))?,\ dm?,\ M(trans_M(va)) := w$

where

$$
\begin{array}{lll}
trans_M(va) & := & \begin{cases} va & \text{if } \mathrm{Low}(va) \\ M(\mathtt{PT} + \mathrm{Pg}(va) * 8) * \mathtt{NPAGES} + \mathrm{Off}(va) & \text{otherwise} \end{cases} \\
dm & := & \forall vp.\, \mathrm{LowPg}(vp) \rightarrow \mathrm{Low}(\mathtt{PT} + vp * 8) \wedge M(\mathtt{PT} + vp * 8) = vp
\end{array}
$$

Figure 5.20: C with Paging ($\mathcal{M}_{PE}$) (PE always, fixed PTROOT, no alloc)

translation is enabled.

This fact we can encode directly into the operational semantics of a new, more abstract memory model, which we call M_{PE}, which is described in Figure 5.20. The new memory model differs from M_{HW}, in the following ways:

- The PE register is no longer there, as this machine always uses address translation.

- The PTROOT register is gone, the machine has a constant that corresponds to the active page table root address. Since this address can not be changed, the register has been removed.

- The address translation function has been modified to automatically return identity for the low addresses.

- The specs of loads and stores of memory ensure that the page table contains mappings that define the address translations for the low addresses to be identity.

- This machine has no hypothetical library, as there are no features that are not directly accessible through load and store.

Using this machine does not simplify the verification as compared to verification over $\mathcal{M}_{HW}$ by a lot, but it, nonetheless, helps us set up for the more abstract machine by incorporating the address translation invariants into the operational semantics before we define the more abstract machines.

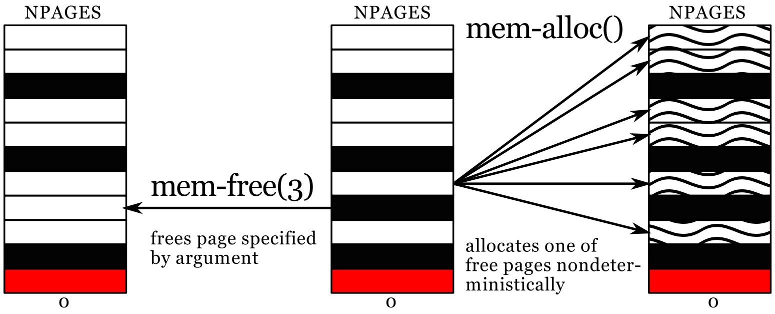

NPAGES
NPAGES
NPAGES
mem-alloc()
mem-free(3)
frees page specified by argument
allocates one of free pages nondeterministically
0
0
0

Figure 5.21: Informal Diagram of $\mathcal{M}_{ALE}$

5.4.4 The Allocated Memory Model (with Address Translation)

Going up the stack of the machine models on the left side of the diagram, the next abstraction of memory is the Allocated Memory Model, marked by $\mathcal{M}_{ALE}$, which is informally shown in Figure 5.21.

In the lower models, we envisioned the memory as a large chunk, all of which is available for use, although using certain areas of memory could be dangerous. In this model of memory, we try to take control of the memory that we have. We do this by marking the pages, which ensures that we only read and write from those pages that are safe to use.

What makes this model a bit more complicated is the address translation, which we are still required to follow. There are several important things to consider about the address translation. First, the identity mapping for the low addresses must still be enforced. Second, the entire page table area must be marked as allocated, otherwise we will not be able to perform the address translation. Both of these quirks must be controlled.

The memory model defines two operations that work over this memory model: `mem-alloc` and `mem-free`. These operations work in the same way they do in our M_{AS}. The allocation returns a page of its choosing, and marks it available for use. The freeing of the page unmarks that page, and we can no longer access it, and we also lose the contents of that page.

The formal model of the allocated memory model is given in Figure 5.22. The model implements the markings of the pages using an allocation table, A. The actual contents of the memory are in D. The load and store operations include checks against A to make sure that the translated addresses which they access are allocated. To make sure that the allocation map remains sane, both

$$\begin{aligned}
&(\textit{Memory System}) && M ::= (D, A)\\
&(\textit{Data Store}) && D ::= \{addr \rightsquigarrow w \mid \text{Low}(addr)\}^*\\
&(\textit{Page Allocation Table}) && A ::= \{pg \rightsquigarrow b \mid \text{LowPg}(pg)\}^*
\end{aligned}$$

Notation	Definition
$M(va)$	$PTalloc?,\ dm?,\ M.A(Pg(trans_M(va)))?, M.D(trans_M(va))$
$M(va) := w$	$PTalloc?,\ dm?,\ M.A(Pg(trans_M(va)))?, M.D(trans_M(va)) := w$

where

$$trans_M(va) := \begin{cases} va & \text{if Low}(va) \\ M.D(\texttt{PT} + \text{Pg}(va) * 8) * \texttt{NPAGES} + \text{Off}(va) & \text{otherwise} \end{cases}$$

$$PTalloc := \forall pg.\, (\text{HighPg}(pg) \vee \text{LowPg}(pg)) \rightarrow M.A(Pg(\texttt{PT} + pg * 8)) = true$$

$$dm := \forall vp.\, \text{LowPg}(vp) \rightarrow \text{Low}(\texttt{PT} + vp * 8) \wedge M.D(\texttt{PT} + vp * 8) = vp$$

Function	Specification
mem-alloc	$[] \mapsto \begin{cases} PTalloc?,\ dm?,\ \texttt{ret}(0) \\ \bigvee_{pg} \left(\begin{array}{l} PTalloc?,\ dm?,\ \text{LowPg}(pg)?,\ A(pg) = \text{false}?, \\ A(pg) := \text{true}, \texttt{ret}(pg) \end{array} \right) \end{cases}$
mem-free	$[pg] \mapsto \left(\begin{array}{l} PTalloc?,\ dm?,\ A(pg) = \text{true}?, PTdom(pg * \texttt{PGSIZE}) = \text{false}?, \\ A(pg) := \text{false}, \cancel{\{D(l) \mid \text{Pg}(l) = pg\}}, \texttt{ret}(0) \end{array} \right)$

Figure 5.22: Allocated Memory Model Semantics (M_{ALE}) and Library($\mathcal{L}_{ALE}$)

load and store instructions make sure that $PTalloc$ and dm predicates hold, which check that the pagetables are marked as usable, and that the address translation for lower addresses will always be the identity map, respectively.

The model fits all the requirements to be used as a parameter to the C language definition, and thus we can define a complete C machine with allocated memory and translation, which we will refer to as $\mathcal{M}_{ALE}$.

Unlike M_{PE}, the allocated memory model now has features that can no longer be directly controlled by the C language, namely the values in the allocation table. The two such actions are the allocation and freeing of pages. Thus, we must supply a stub library in order to verify programs created for the $\mathcal{M}_{ALE}$ machine. This library, which we will refer to as $\mathcal{L}_{ALE}$, defines the `mem-alloc` and `mem-free` primitives which are used to control the allocation table.

The `mem-alloc` primitive's specification resembles a spec of a function, meaning that it expects an argument frame and when it completes, it places a return frame. `Mem-alloc`, according to its spec, can either return a zero value and do nothing, or it may return any non-zero value that corresponds to a number of the physical page which was not allocated before `mem-alloc` and is allocated after. In simpler terms, `mem-alloc` may allocate a page and return that page's number, or it may fail and return 0, and it is allowed to fail, even if there are pages available for allocation. The `mem-free` is simpler in that it can never fail, and its spec ensures that it can only be called on an allocated page. These two primitives define the complete interface that programs over the ALE model will have to the memory system.

5.4.5 The Page Map Model

The final intermediate machine that we have yet to define on the left side of our plan is the page map machine, $\mathcal{M}_{PMAP}$. The purpose of this machine is to separate out the page tables from the actual memory, and define them as an auxiliary data structure. The model still maintains the markings of whether the page is allocated or not, but the entire address translation is moved out of the data storage, and into a separate table. The pictorial view of this memory model is given in Figure 5.23.

Such a change brings drastic modifications to the address translation mechanism of the memory. The address translation in more concrete machines always performed a lookup by addressing certain areas of memory. In this machine, the translation goes through a separate table, which can not be

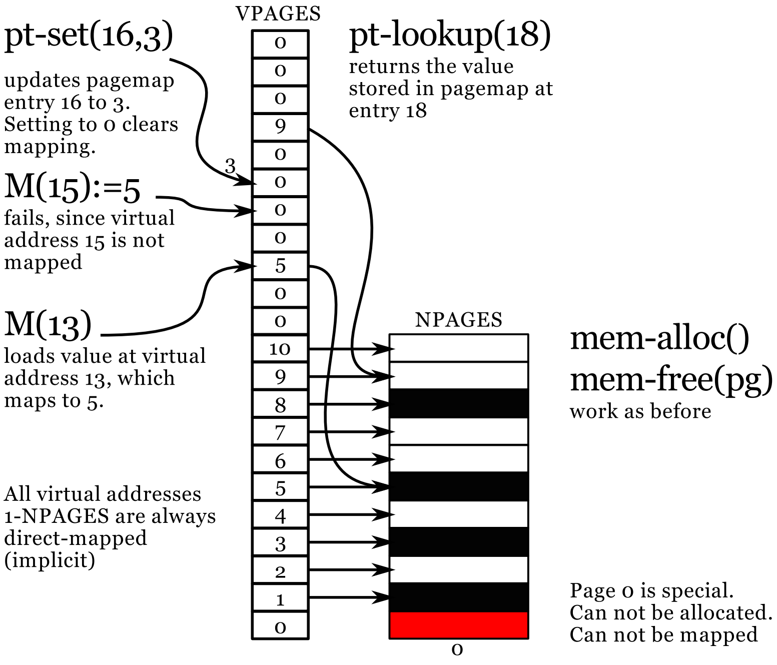

VPAGES
pt-set(16,3)
updates pagemap entry 16 to 3. Setting to 0 clears mapping.
pt-lookup(18)
returns the value stored in pagemap at entry 18
0
0
0
9
0
0
0
0
5
0
0
10
9
8
7
6
5
4
3
2
1
0
3
M(15):=5
fails, since virtual address 15 is not mapped
M(13)
loads value at virtual address 13, which maps to 5.
All virtual addresses 1-NPAGES are always direct-mapped (implicit)
NPAGES
mem-alloc()
mem-free(pg)
work as before
Page 0 is special. Can not be allocated. Can not be mapped
0

Figure 5.23: Page Map Memory Model

updated by load and store operations of memory. This yields several benefits for verification. First, it is no longer possible to update address translation by using stores - meaning that the addresses are persistent, unless modified in the page map. Second, the computation of the address translation is simpler, as it bypasses memory. This simplification would be extremely valuable if we were working with the multi-level page tables, as instead of traversing trees of tables, we would only have a flat lookup.

Figure 5.24 gives a definition of the memory system that uses the page table system to translate the virtual memory addresses into physical memory area. This version of memory uses three data structures to accomplish this goal. The first two are the actual storage, defined by D, and the allocation table, still defined by A. These two data structures are equivalent to their ALE model counterparts. The model then adds the page map structure, defined by PM, which is a table containing entries indexed from $\mathtt{NPAGES}$ to $\mathtt{VPAGES}-1$, as we do not need entries from 0 to $\mathtt{NPAGES}-1$ since they are constant. Each entry contains a valid page number (with 0 indicating unmapped).

Memory loads and stores are similar to the loads and stores we have seen in the *ALE* model. However, the translation function, $trans(va)$, that they rely on, has changed drastically, as it is now uses PM and not a particular area of D. If the address given to trans is already a physical address,

$$\begin{array}{rll}
(\textit{Global Storage System}) & M & ::= (D, A, PM) \\
(\textit{Allocatable Memory}) & D & ::= \{addr \rightsquigarrow w \mid \mathrm{Low}(addr)\}^* \\
(\textit{Page Allocation Table}) & A & ::= \{pg \rightsquigarrow b \mid \mathrm{LowPg}(pg)\}^* \\
(\textit{Page Map}) & PM & ::= \{pg \rightsquigarrow pg' \mid \mathrm{HighPg}(pg)\}^*
\end{array}$$

Notation	`Definition`
$M(va)$	$\texttt{let}\ \ pa := trans(va)\ \ \texttt{in}\ \ (M.A(Pg(pa))?.\ D(pa))$
$M(va) := w$	$\texttt{let}\ \ pa := trans(va)\ \ \texttt{in}\ \ (M.A(Pg(pa))?,\ D(pa) := w)$

where

$$trans(va) \quad := \begin{cases} PM(Pg(va)) * \texttt{PGSIZE} + \mathrm{Off}(va) & \text{if } \mathrm{High}(va) \\ va & \text{otherwise} \end{cases}$$

Label	Specification
mem-alloc	$[] \mapsto \begin{cases} \texttt{ret}(0) \\ \Big(\bigvee_{pg} \mathrm{LowPg}(pg)?,\ A(pg) = \text{false}?, A(pg) := \text{true}, \cancel{\{D(l) \mid A(\mathrm{Pg}(l)) = \text{false}\}}, \texttt{ret}(pg)\Big) \end{cases}$
mem-free	$[pg] \mapsto A(pg) = \text{true}?, A(pg) := \text{false}, \cancel{\{D(l) \mid A(\mathrm{Pg}(l)) = \text{false}\}}, \texttt{ret}(0)$
pt-set	$[vp, pp] \mapsto \mathrm{HighPg}(vp)?, \mathrm{LowPg}(pp), PM(vp) := pp, \cancel{\{D(\texttt{l}) \mid A(Pg(l)) = \text{false}\}}, \texttt{ret}(0)$
pt-lookup	$[vp] \mapsto (\mathrm{HighPg}(vp)?, \texttt{ret}(PM(vp)))$

Figure 5.24: The Pagemap Memory Model (M_{PMAP}) and Library ($\mathcal{L}_{PMAP}$)

it works as an identity function, ignoring the PM table. If it is a virtual address, then it is separated into virtual page and offset components, then the virtual page is looked up in the page map, and the corresponding physical page retrieved, then the offset is added back to the physical page to produce the physical address corresponding to the virtual address in question. The result is then used by the load and store operation.

Now that we have isolated the address translation mechanism into a separate structure, we have lost the ability to update the pages through the use of normal memory stores. Thus, to perform these operations in this memory model, we must define a primitive library, ($\mathcal{L}_{PMAP}$), that features four functions. The `mem-alloc` and `mem-free` functions that we have seen earlier are still present, and still perform the same task: finding safe and unused pages to use, and marking them. The new model features two new library stubs:

- **pt-set** This function sets a value in the PM. It takes two parameters, a virtual page for which an entry is to be set, and the entry itself, which must be either 0 or a valid non-zero physical page number. If these requirements are satisfied, this function does not fail.

- **pt-lookup** This function looks up a specific virtual page in the PM. It just returns a value stored in the page tables, and is not strictly necessary for the operation of virtual memory.

This new machine, $\mathcal{M}_{PMAP}$, allows for a much simpler certification of the code that relies on the page table. It abstracts the details about the translation mechanism, and completely hides the hardware details of the address translation mechanism, substituting it with a much simpler, machine-independent version. Thus, it is a perfect machine on which to implement the address space API, which is exactly the purpose for which we will use this machine in our certification.

5.4.6 Hardware with AT off

The intermediate machines we have described so far only define what happens once the address translation is correctly configured and activated. This, however, completely disregards our need for initialization procedures that record the starting information into the tables. These intermediate machines used for initialization are shown on the right side of our main design diagram.

The machines on the initialization side parallel those that are on the runtime side of the diagram, but with one important difference. Where the run-time side assumes that the address translation is

$$(\textit{Memory System}) \quad M ::= \{addr \rightsquigarrow w \mid \mathrm{Low}(addr)\}^*$$

Notation	Definition
$M(pa)$	$(\mathrm{Low}(pa)?,\ M(pa))$
$M(pa) := w$	$(\mathrm{Low}(pa)?,\ M(pa) := w)$

Figure 5.25: Hardware Machine with AT disabled ($\mathcal{M}_{PD}$)

$$\begin{aligned} (\textit{Memory System}) &\quad M ::= (D, A) \\ (\textit{Memory Data}) &\quad D\ ::= \{addr \rightsquigarrow w \mid \mathrm{Low}(addr)\}^* \\ (\textit{Allocation Table}) &\quad A\ ::= \{pg \rightsquigarrow b \mid \mathrm{LowPg}(b)\}^* \end{aligned}$$

Notation	Definition
$M(pa)$	$(M.A(Pg(pa))?,\ M.D(pa))$
$M(pa) := w$	$(M.A(Pg(pa))?,\ M.D(pa) := w)$

Function	Specification
mem-alloc	$[] \mapsto \begin{cases} \texttt{ret}(0) \\ \left(\bigvee_{pg} \mathrm{LowPg}(pg)?,\ A(pg) = \mathrm{false}?, A(pg) := \mathrm{true}, \{\cancel{D(l) \mid A(\mathrm{Pg}(l)) = \mathrm{false}}\}, \texttt{ret}(pg)\right) \end{cases}$
mem-free	$[pg] \mapsto A(pg) = \mathrm{true}?, A(pg) := \mathrm{false}, \{\cancel{D(l) \mid A(\mathrm{Pg}(l)) = \mathrm{false}}\}, \texttt{ret}(0)$

Figure 5.26: Allocatable Memory with AT Disabled Model ($\mathcal{M}_{ALD}$) and Its Library ($\mathcal{L}_{ALD}$)

on, and configured correctly, the initialization makes the assumption that the memory translation is completely off. This defines the restriction of the address translated machine $\mathcal{M}_{PD}$, that we see near the bottom. It is just $\mathcal{M}_{HW}$, with no address translation whatsoever, nor a way to turn it on. As one can imagine, without the address translation, the semantics of memory accesses are greatly reduced, simplifying the verification of code. The formal specifications of such a machine are given in Figure 5.25.

The state of the *PD* memory model is similar to that of the *HW* memory model, except we are guaranteed that the PE is set to off. Under these assumptions the registers that form a part of the state become unnecessary and can be removed. This also means that all checks associated with them are removed as well, and the translation function as well. Thus, the memory model is just the trivial fixed-size memory model.

5.4.7 The Non-Address Translated Allocated Memory

The next abstract memory model that we define, parallels our allocated memory model, *ALE*, but with the address translation turned off. This memory model, we call *ALD*, and, thus $\mathcal{M}_{ALD}$ is the C-machine that uses it. The reason for the existence of this machine is that we may want to use the memory allocator before we have turned on the address translation, e.g. we may want to rely on the `mem-alloc` and `mem-free` functions within some initialization code, for example `pt-init`. Although our version of `pt-init` does not rely on `mem-alloc` and `mem-free`, a future enhancement of the the function may do so, and it is important that we demonstrate the ability to handle this.

The formal specification of this machine is given in Figure 5.26, and as we can see it copies all the data structures of the *ALE* memory model, except that since the address translation is off, there is no need for the *trans* predicate. Similar in what happens in *ALE*, we must also define a primitive library for this machine, $\mathcal{L}_{ALD}$, which features the `mem-alloc` and `mem-free` functions. However, the specifications of these functions no longer take into account the memory translation mechanism, and thus their specifications are slightly simpler.

5.5 Verification of Code

At this point, we have defined all the memory models and the abstract machines that we will use to verify our virtual memory manager, which we will now verify module by module. In section 5.6, we will show how to link these modules, but for now, our goal is to give the specifications for these blocks of code, and to make sure that the code does indeed follow the specification.

5.5.1 Verification of the Memory Allocator

To verify the memory allocator, we will use the $\mathcal{M}_{PE}$ machine that we have defined. The first step in verification is to convert the code of the functions into procedures that are compatible with the certification in our C-machine. The result of this conversion is given in Figure 5.27.

To certify them, we need to give the specifications for these two functions. The specifications that we have defined are given by the specification heap Ψ_{PE}^{mem} given in Figure 5.28.

mem-alloc	mem-alloc-loop	mem-free

$readargs([]);$
$curpage := 1;$
$found := 0;$
[mem-alloc-loop];
if $found = 1$ then
  $*(\mathtt{PMM} + curpage * 8) = 1;$
  $return(curpage)$
else
  $return(0)$
end if
	if $(found = 0 \wedge curpage < \mathtt{NPAGES})$
	  if $(*(\mathtt{PMM} + curpage * 8) = 0)$
	    $found := 1$
	  else
	    $curpage := curpage + 1$
	  end if;
	  [mem-alloc-loop]
	end if
		$readargs[page];$
		$*(\mathtt{PMM} + page * 8) := 0;$
		$return(0)$

Figure 5.27: The C-machine procedure of the memory allocator ($\mathbb{C}^{mem}$)

Label	Specification
mem-alloc	$[] \mapsto \begin{cases} dm?, \mathtt{ret}(0) \\ \bigvee_{pg} \begin{pmatrix} dm?, \mathrm{LowPg}(pg)?, pg \neq 0?, \\ M(\mathtt{PMM} + pg * 8) = 0?, M(\mathtt{PMM} + pg * 8) := 1, \mathtt{ret}(pg) \end{pmatrix} \end{cases}$
mem-alloc-loop	$(\mathrm{LowPg}(S(curpage))?,\ S(found) \neq 0?) \vee$ $(\mathrm{LowPg}(S(curpage))?,\ S(found) = 0?,\ S(curpage) := \mathtt{NPAGES}) \vee$ $\bigvee_{pg'} \begin{pmatrix} \mathrm{LowPg}(S(curpage))?,\ S(found) = 0?,\ S(curpage) \leq pg'?, \\ \mathrm{LowPg}(pg')?,\ M(\mathtt{PMM} + pg' * 8) = 0?, \\ S(found) := 1,\ S(curpage) := pg' \end{pmatrix}$
mem-free	$[pg] \mapsto (\mathrm{LowPg}(pg)?,\ M(\mathtt{PMM} + pg * 8) = 1?,\ M(\mathtt{PMM} + pg * 8) := 0,\ \mathtt{ret}(0))$

Figure 5.28: The Specification of Memory Allocator in the $\mathcal{M}_{PE}$ machine (Ψ_{PE}^{mem})

Lemma 5.5.1 (Certified Memory Allocator)

We show that the following are true:

$$\mathcal{M}_{PE}, \mathcal{L}_{PE} \cup \Psi^{mem}_{PE} \vdash \mathbb{C}^{mem}(\texttt{mem-alloc-loop}) : \Psi^{mem}_{PE}(\texttt{mem-alloc-loop})$$
$$\mathcal{M}_{PE}, \mathcal{L}_{PE} \cup \Psi^{mem}_{PE} \vdash \mathbb{C}^{mem}(\texttt{mem-alloc}) : \Psi^{mem}_{PE}(\texttt{mem-alloc})$$
$$\mathcal{M}_{PE}, \mathcal{L}_{PE} \cup \Psi^{mem}_{PE} \vdash \mathbb{C}^{mem}(\texttt{mem-free}) : \Psi^{mem}_{PE}(\texttt{mem-free})$$

And therefore, by WF-CODE following holds:

$$\mathcal{M}_{PE}, \mathcal{L}_{PE} \vdash \mathbb{C}^{mem} : \Psi^{mem}_{PE}$$

Pf. Please see Coq proof. □

5.5.2 Verification of the Page Table Driver

The next component we will certify are the functions of the page table driver, namely `pt-set` and `pt-lookup`. Although these functions do not rely on memory allocation directly, e.g. they never call `mem-alloc` or `mem-free`, we will certify them on the machine that is designed to support memory allocation, $\mathcal{M}_{ALE}$. The reason for doing so is that when we will extend our code to include multi-level page tables, the page table driver will need to perform allocation, and thus we are making sure that our approach will work there as well.

First, we must convert the C code of these functions into the meta-C procedures. These functions are straight-line code, and the conversion is trivial, and the result can be seen in Figure 5.29. Since these procedures are straight-line code, the only challenge is to give these functions proper specifications, which are given in the same figure.

Lemma 5.5.2 (Certified Page Table Driver) The functions of the page table driver are correct with respect to their specifications, e.g.

$$\mathcal{M}_{ALE}, \mathcal{L}_{ALE} \cup \Psi^{pt}_{ALE} \vdash \mathbb{C}^{pt}(\texttt{pt-set}) : \Psi^{pt}_{ALE}(\texttt{pt-set})$$
$$\mathcal{M}_{ALE}, \mathcal{L}_{ALE} \cup \Psi^{pt}_{ALE} \vdash \mathbb{C}^{pt}(\texttt{pt-lookup}) : \Psi^{pt}_{ALE}(\texttt{pt-lookup})$$

pt-set:	pt-lookup:

readargs[vpage,ppage];
*(PTROOT + vpage * 8) := ppage;
return(0);
	readargs[vpage];
	return (*(PTROOT + vpage * 8));

Label	Specification
pt-set	$[vp, pp] \mapsto \left(\begin{array}{l} dm?,\ \text{PT-ALLOC}(M)?,\ \text{HighPg}(vp)?,\ \text{LowPg}(pp)?, \\ D(\mathtt{PT} + pg * 8) := pp,\ \mathtt{ret}(0) \end{array} \right)$
pt-lookup	$[vp] \mapsto (dm?,\ \text{PT-ALLOC}(M)?,\ \text{HighPg}(vp)?,\ \mathtt{ret}(D(\mathtt{PT} + vp * 8)))$

where

$$\text{PT-ALLOC}(M) := \forall pg.\, \text{ValidVPg(pg)} \rightarrow M.A(\mathtt{PMM} + Pg(\mathtt{PT} + pg * 8) * 8) = true$$

Figure 5.29: Code of the Page Table Driver ($\mathbb{C}^{pt}$) and Spec (Ψ^{pt}_{ALE})

And therefore, the entire module containing the page table driver is certified:

$$\mathcal{M}_{ALE}, \mathcal{L}_{ALE} \vdash \mathbb{C}^{pt} : \Psi^{pt}_{ALE}$$

Pf. Please see the Coq proof. □

5.5.3 Verification of the Address Space API

The address space API consists of four functions: two of which are just calls to the memory allocation functions (mem-alloc and mem-free) that we have already verified in the PE model. Thus we will not need to verify them again, but only to link them, which we will show later in the chapter.

However, we do need to verify the two functions, as-request and as-release, defined in the as.c file in our virtual memory manager. These functions make use of the page table driver, and thus we must certify them using the $\mathcal{M}_{PMAP}$ machine and rely on the $\mathcal{L}_{PMAP}$ library for the abstract specifications of the actions performed by that page table driver.

The first step in verification is to convert the code from pure C into the meta-C procedures. These functions are also straight-line code, and thus the conversion is quite trivial. The result of this conversion can be seen in Figure 5.30. The figure also shows the specifications of these procedures.

Then we show the usual lemma that the code is certified.

Lemma 5.5.3 (Address Space Library Certified) The code of the procedures of Address Space

as-request	as-release
$readargs([vpage]);$ $fcall(\text{mem-alloc}, []);$ $[\text{mem-alloc}];$ $fcallret(pg);$ if $(pg = 0)$ then $return(0);$ else $fcall(\text{pt-set}, [vpage, pg]);$ $[\text{pt-set}];$ $fcallret(junk);$ $return(vpage);$ end if	$readargs([vpage]);$ $fcall(\text{pt-lookup}, [vpage]);$ $[\text{pt-lookup}];$ $fcallret(page);$ $fcall(\text{mem-free}, [page]);$ $[\text{mem-free}];$ $fcallret(junk);$ $fcall(\text{pt-set}, [vpage, 0]);$ $[\text{pt-set}];$ $fcallret(junk);$ $return(0);$

Label	Specification
as-request	$[vp] \mapsto \begin{cases} \text{HighPg}(vp)?,\ \cancel{\{D(\mathtt{1}) \mid A(\text{Pg}(\mathtt{1})) = \text{false}\}},\ \mathtt{ret}(0) \\ \bigvee_{pg} \begin{pmatrix} \text{HighPg}(vp)?, A(pg) = \text{false}?, \\ A(pg) := \text{true}, \cancel{\{D(\mathtt{1}) \mid A(\text{Pg}(\mathtt{1})) = \text{false}\}}, PM(vp) := pg, \mathtt{ret}(vp) \end{pmatrix} \end{cases}$
as-release	$[vp] \mapsto \begin{pmatrix} \text{HighPg}(vp)?,\ \text{LowPg}(PM(vp))?,\ PM(vp) \neq 0?,\ A(PM(vp)) = \text{true}?, \\ PM(vp) := 0,\ A(PM(vp)) := \text{false},\ \cancel{\{D(\mathtt{1}) \mid A(\text{Pg}(\mathtt{1})) = \text{false}\}},\ \mathtt{ret}(0) \end{pmatrix}$

Figure 5.30: The Procedures of the Address Space API Implementation ($\mathbb{C}^{as}$) and Specs (Ψ^{as}_{PMAP})

Library is valid:

$$\mathcal{M}_{PMAP}, \mathcal{L}_{PMAP} \cup \Psi^{as}_{PMAP} \vdash \mathbb{C}^{as}(\texttt{as-request}) : \Psi^{as}_{PMAP}(\texttt{as-request})$$

$$\mathcal{M}_{PMAP}, \mathcal{L}_{PMAP} \cup \Psi^{as}_{PMAP} \vdash \mathbb{C}^{as}(\texttt{as-release}) : \Psi^{as}_{PMAP}(\texttt{as-release})$$

Therefore, the address space library is valid using the WF-CODE rule.

$$\mathcal{M}_{PMAP}, \mathcal{L}_{PMAP} \vdash \mathbb{C}^{as} : \Psi^{as}_{PMAP}$$

Pf. Please see Coq proof. □

5.5.4 Verification of Allocator Initialization

Now we will verify the initialization of the allocator, which is just a single function: `mem-init`. Surprisingly, the initialization function is actually one of the more complex functions in our current system, as it contains several loops, thus requiring us to specify loop invariants in order to complete

mem-init	mem-init-loop1	mem-init-loop2	mem-init-loop3

$readargs([])$;
$i := 1$;
$*(\texttt{PMM}) := 1$;
[mem-init-loop1];
[mem-init-loop2];
[mem-init-loop3];
$ret(0)$
	if $(i < 0xA0)$ then
	$*(\texttt{PMM} + i * 8) := 0$;
	$i := i + 1$;
	[mem-init-loop1];
	end if
		if $(i < 0x200)$ then
		$*(\texttt{PMM} + i * 8) := 1$;
		$i := i + 1$;
		[mem-init-loop2];
		end if
			if $(i < \texttt{NPAGES})$ then
			$*(\texttt{PMM} + i * 8) := 0$;
			$i := i + 1$;
			[mem-init-loop3];
			end if

Procedure	Specification
mem-init-loop1	$\left(\begin{array}{l}(0 \le S(i) \le 0\text{xA0})?,\ S(i) := 0\text{xA0}, \\ M(\texttt{PMM} + S(i) * 8) := 0,\ \ldots,\ M(\texttt{PMM} + 0\text{x9F} * 8) := 0\end{array}\right)$
mem-init-loop2	$\left(\begin{array}{l}(0 \le S(i) \le 0\text{x200})?,\ S(i) := 0\text{x200}, \\ M(\texttt{PMM} + S(i) * 8) := 0,\ \ldots,\ M(\texttt{PMM} + 0\text{x1FF} * 8) := 0\end{array}\right)$
mem-init-loop3	$\left(\begin{array}{l}(0 \le S(i) \le \texttt{NPAGES})?,\ S(i) := \texttt{NPAGES}, \\ M(\texttt{PMM} + S(i) * 8) := 0,\ \ldots,\ M(\texttt{PMM} + (\texttt{NPAGES} - 1) * 8) := 0\end{array}\right)$
mem-init	$[] \mapsto \left(\begin{array}{l}(M(\texttt{PMM} + Pg(l) * 8) := 1 \mid \text{PMMdom}(l) \vee \text{PTdom}(l), \\ (M(\texttt{PMM} + pg * 8) := \{0, 1\} \mid \text{LowPg}(pg)),\ \ \texttt{ret}(0)\end{array}\right)$

Figure 5.31: Procedures of Memory Initialization ($\mathbb{C}^{meminit}_{PD}$) and Their Specs ($\Psi^{meminit}_{PD}$)

the verification.

Proceeding methodically, the first step is to convert this function into procedures. Since the function contains loops, the conversion will require replacing these loops with recursive procedures. Since there are three loops, three recursive procedures are created, which we will call `mem-init-loop`n (with n is the number of the loop). The converted procedures that correspond to the original functions make up a procedure heap $\mathbb{C}^{meminit}$, which is listed in Figure 5.31.

As we have outlined in our verification plan, we will certify the memory initialization in the machine $\mathcal{M}_{PD}$, a machine which restricts our base machine to having the address translation turned off, a small simplification. The specification for these procedures are then defined by the specification heap $\Psi^{meminit}_{PD}$, listed in the same figure.

Lemma 5.5.4 (Memory Allocator Initialization Certified)

The following procedures are certified:

$$\begin{aligned}
&\mathcal{M}_{PD}, \Psi_{PD}^{meminit} \vdash \mathbb{C}^{meminit}(\texttt{mem-init-loop1}) : \Psi_{PD}^{meminit}(\texttt{mem-init-loop1})\\
&\mathcal{M}_{PD}, \Psi_{PD}^{meminit} \vdash \mathbb{C}^{meminit}(\texttt{mem-init-loop2}) : \Psi_{PD}^{meminit}(\texttt{mem-init-loop2})\\
&\mathcal{M}_{PD}, \Psi_{PD}^{meminit} \vdash \mathbb{C}^{meminit}(\texttt{mem-init-loop3}) : \Psi_{PD}^{meminit}(\texttt{mem-init-loop3})\\
&\mathcal{M}_{PD}, \Psi_{PD}^{meminit} \vdash \mathbb{C}^{meminit}(\texttt{mem-init}) : \Psi_{PD}^{meminit}(\texttt{mem-init})
\end{aligned}$$

These procedures can then be put together in a certified module using WF-CODE rule

$$\mathcal{M}_{PD}, \emptyset \vdash \mathbb{C}^{meminit} : \Psi_{PD}^{meminit}$$

Pf. Please see Coq proof. □

5.5.5 Verification of Page Table Initialization

The last bit of code that we have to verify is the initialization of the page tables, which is done by the function `pt-init`. In accordance with our plan, we will verify this code on the $\mathcal{M}_{ALD}$ machine, the C-machine which has address translation disabled, but it supports the memory allocation primitives.

First, we make the conversion of the C code of `pt-init` into procedures of the meta C language. Since the `pt-init` function contains loops, our conversion will create separate recursive procedures that will work as loops. The result of our conversion is given in Figure 5.32. The same figure also shows the specifications that we have defined for these procedures.

Lemma 5.5.5 (Page Table Driver Initialization Certified)

The procedures of the page table driver are certified with the following signatures:

$$\begin{aligned}
&\mathcal{M}_{ALD}, \Psi_{ALD}^{ptinit} \vdash \mathbb{C}^{ptinit}(\text{pt-init-loop1}) : \Psi_{ALD}^{ptinit}(\text{pt-init-loop1})\\
&\mathcal{M}_{ALD}, \Psi_{ALD}^{ptinit} \vdash \mathbb{C}^{ptinit}(\text{pt-init-loop2}) : \Psi_{ALD}^{ptinit}(\text{pt-init-loop2})\\
&\mathcal{M}_{ALD}, \emptyset \vdash \mathbb{C}^{ptinit} : \Psi_{ALD}^{ptinit}
\end{aligned}$$

These procedure are combined into a certified module with the following definition:

$$\mathcal{M}_{ALD}, \emptyset \vdash \mathbb{C}^{ptinit} : \Psi_{ALD}^{ptinit}$$

pt-init	pt-init-loop1	pt-init-loop2

$readargs([])$;
$i := 0$;
[pt-init-loop1];
[pt-init-loop2];
$ret(0)$
	if ($i < \mathtt{NPAGES}$) then
	    $*(\mathtt{PT} + i * 8) := i$;
	    $i := i + 1$;
	    [pt-init-loop1];
	end if
		if ($i < \mathtt{VPAGES}$) then
		    $*(\mathtt{PT} + i * 8) := 0$;
		    $i := i + 1$;
		    [pt-init-loop2];
		end if

Procedure	Specification
pt-init-loop1	LowPg($S(i)$)?, PT-ALLOC(M)?, $S(i) := \mathtt{NPAGES}$, $D(\mathtt{PT} + S(i) * 8) := S(i),\ \ldots,\ D(\mathtt{PT} + (\mathtt{NPAGES} - 1) * 8) := (\mathtt{NPAGES} - 1)$
pt-init-loop2	HighPg($S(i)$)?, PT-ALLOC(M)?, $S(i) := \mathtt{VPAGES}$, $D(\mathtt{PT} + S(i) * 8) := 0,\ \ldots,\ D(\mathtt{PT} + (\mathtt{VPAGES} - 1) * 8) := 0$
pt-init	$[] \mapsto \left(\begin{array}{l} \text{PT-ALLOC}(M)?, \\ (M(\mathtt{PT} + pg * 8) := pg \mid \text{LowPg}(pg)), \\ (M(\mathtt{PT} + pg * 8) := 0 \mid \text{HighPg}(pg)), \\ \mathtt{ret}(0) \end{array}\right)$

Figure 5.32: Procedures of Page Table Driver Initialization ($\mathbb{C}^{ptinit}$) and Their Specs (Ψ^{ptinit}_{ALD})

Pf. Please see Coq proof. □

5.5.6 Recap of Verification

At this point, we have verified all the components of the virtual memory manager using C-machines instantiated with several different memory models. The following is the list of all certified modules that we have so far.

$$\mathcal{M}_{PE}, \mathcal{L}_{PE} \vdash \mathbb{C}^{mem} : \Psi^{mem}_{PE}$$
$$\mathcal{M}_{ALE}, \mathcal{L}_{ALE} \vdash \mathbb{C}^{pt} : \Psi^{pt}_{ALE}$$
$$\mathcal{M}_{PMAP}, \mathcal{L}_{PMAP} \vdash \mathbb{C}^{as} : \Psi^{as}_{PMAP}$$
$$\mathcal{M}_{PM}, \emptyset \vdash \mathbb{C}^{meminit} : \Psi^{meminit}_{PM}$$
$$\mathcal{M}_{ALD}, \emptyset \vdash \mathbb{C}^{ptinit} : \Psi^{ptinit}_{ALD}$$

We are also operating under the assumption that the high-level kernel, e.g. the code that makes use of our virtual memory manager, is completely verified, with the following signature:

$$\mathcal{M}_{AS}, \mathcal{L}_{AS} \vdash \mathbb{C}^{kernel} : \Psi^{kernel}_{AS}$$

Now that we have performed the actual code verification, simplified by the use of the abstract machines, we now have to make an effort to define the refinements that will allow us to link all these pieces together.

5.6 Refinements

Now that we have actually used our multiple machine models to certify the programs, we now have to concentrate on putting them together. The way that we will accomplish this is by defining representation refinements between all the adjacent machines shown in our plan: AS-PMAP, PMAP-ALE, ALE-PE, PE-HW, ALD-PD, and PD-HW.

To show this we would have to demonstrate the representation between the C-machines given each model. However, there are two optimizations that we use to make defining these refinements a lot more painless. These optimizations are based on two facts about our machines:

- The operations of the machines never change - the relation between operations and labels is an identity map.

- The machines have an identical stack. This means several things. First, we can define a `repr` between the machines automatically from the relation between just the memory components. Second, is that the operational semantics of all C languages are identical, except for the memory read and store operations that the semantics rely on.

This means that we can create refinements between the machines with two different definition of memory only from the information about the relation between memory models. More precisely, to define a $\texttt{repr}_{\mathcal{M}_{M1}-\mathcal{M}_{M2}}$, a representation relation between two meta-C machine that feature two different memory models ($M1$ and $M2$), we will require that the memory models are related by the relation $M1 \leq M2$ with the following definition:

Definition 5.6.1 (Memory Relation)

A memory relation $m1 \leq m2$ is any relation between two memory states $m1$ and $m2$ such that the following is true:

$$\forall l, v.\ (m1(l) = v) \to (m2(l) = v)$$
$$\forall l, v, m1'.\ (m1' = (m1(l) := v)) \to (m1' \leq (m2(l) := v))$$

This definition is not a relation in itself, but rather a template for some such relation. For any precise relation, the actual definition of $m1 \leq m2$ might be different, but the two properties described must hold. The first of these properties states that if we can read a value from some location l from $m1$, then we must be able to read the same value from location in $m2$. The second one states that if in $m1$, we can update location l with value v successfully, we must be able to successfully update the same location with the same value in $m2$, and, moreover, the resulting states of $m1$ and $m2$ must also be related. Essentially, any relation between memory models that fits this template will guarantee equivalence on load operation, and the relation will be preserved on store operations.

Now we will show how to define a complete `repr` between two meta-C machines that use memory models related by some relation that fits the pattern above. The first thing that we will do is to define a `repr` relation between the machine states. This relation is very simple - it is just an extension of the relation between the memory models, since the stack is always the same:

Definition 5.6.2 (`repr`-relation between two Meta-C Machines)

$$\texttt{repr}_{\mathcal{M}_{M1}-\mathcal{M}_{M2}}(\mathbb{S}_1, \mathbb{S}_2) := \mathbb{S}_1.S = \mathbb{S}_2.S \wedge \mathbb{S}_1.M \leq \mathbb{S}_2.M$$

Following the meaning of `repr`-refinement, we can use the relation to define a specification translation functions, to which we will refer as $\uparrow_{\texttt{repr}_{\mathcal{M}_{M1}-\mathcal{M}_{M2}}}(\texttt{a})$ or more simply as $T_{M1-M2}(\texttt{a})$. This function is the `repr`-refinements automatic conversion of abstract specification to concrete specifications using the relation between states.

Now that we have defined the `repr` and, subsequently, T_{M1-M2}, we have to show that this representation relation satisfies the requirements for being a valid `repr`-refinement. This means that we have to show that the OP-COMPAT rule, reproduced below, holds.

$$\forall \iota \in \mathcal{M}_{M1}.\ T_{M1-M2}(\mathcal{M}_{M1}(\iota)) \supseteq \mathcal{M}_{M2}(\iota)$$

Intuitively, since both $\mathcal{M}_{M1}$ and $\mathcal{M}_{M2}$ are really the same machines, only with different behavior for loads and stores, lifting the operations that do not involve memory should result in actions that are exactly the same. For those that do involve loads and stores, we can show that this property holds from the properties of the relation between memory models.

To begin to show that the above property holds, we first show any expression will evaluate to the same value in machine $\mathcal{M}_{M1}$ and $\mathcal{M}_{M2}$, if the machines are related by a $\texttt{repr}_{\mathcal{M}_{M1}-\mathcal{M}_{M2}}$.

Lemma 5.6.3 (*eval* equivalence under `repr`)

For all expressions $\texttt{e}$, for any state $\mathbb{S}_1$ of machine $\mathcal{M}_{M1}$, and any state $\mathbb{S}_2$ of machine $\mathcal{M}_{M2}$, such that $\texttt{repr}_{\mathcal{M}_{M1},\mathcal{M}_{M2}}(\mathbb{S}_1 - \mathbb{S}_2)$, if there is a v, such that $eval(\texttt{e})\mathbb{S}_1 = v$, then $eval(\texttt{e})\mathbb{S}_2 = v$.

Pf.
By def. of $\texttt{repr}_{\mathcal{M}_{M1},\mathcal{M}_{M2}}$, we know that $\mathbb{S}_1.S = \mathbb{S}_2.S$ and $\mathbb{S}_1.M \leq \mathbb{S}_2.M$.
Assume $eval(\texttt{e})\mathbb{S}_1 = v$.
Proceed by induction on structure of $\texttt{e}$.
The only interesting case is $\texttt{e} = *(\texttt{e}')$.
By definition of *eval*, $eval(*(\texttt{e}'))\mathbb{S}_1 = \mathbb{S}_1.M(eval(\texttt{e}')\ \mathbb{S}_1) = v$
Similarly, $eval(*(\texttt{e}'))\mathbb{S}_2 = \mathbb{S}_2.M(eval(\texttt{e}')\ \mathbb{S}_2)$
By IH, there is a x, such that $eval(\texttt{e}')\ \mathbb{S}_1 = x$, and thus $eval(\texttt{e}')\ \mathbb{S}_2 = x$.
Since, $\mathbb{S}_1.M \leq \mathbb{S}_2.M$, by def 5.6.1 if $\mathbb{S}_1.M(x) = v$ then $\mathbb{S}_2.M(x) = v$.
Thus, if $eval(*(\texttt{e}'))\mathbb{S}_1 = v$, then $eval(*(\texttt{e}'))\mathbb{S}_2 = v$, as required.
All other cases are trivial, since $\mathbb{S}_1.S = \mathbb{S}_2.S$, and from IH.

□

Now we can focus on the operations themselves, by showing the following lemma:

Lemma 5.6.4 (meta-C Valid Refinement)

$\texttt{repr}_{\mathcal{M}_{M1}-\mathcal{M}_{M2}}$ satisfies the OP-COMPAT property needed to define a valid `repr`-refinement.

$$\forall \iota \in \mathcal{M}_{M1}.\, T_{M1-M2}(\mathcal{M}_{M1}(\iota)) \supseteq \mathcal{M}_{M2}(\iota)$$

Pf.
Proceed by cases of ι.
The interesting case is $*(\texttt{e}_{loc}) := \texttt{e}$.
Need to show that $T_{M1-M2}(\mathcal{M}_{M1}(*(\texttt{e}_{loc}) := \texttt{e})) \supseteq \mathcal{M}_{M2}(*(\texttt{e}_{loc}) := \texttt{e})$.
This is established by showing two things:

1. If $\mathbb{S}_{M1} \in \texttt{dom}(\mathcal{M}_{M1}(*(\texttt{e}_{loc}) := \texttt{e}))$ then the related $\mathbb{S}_{M2} \in \texttt{dom}(\mathcal{M}_{M2}(*(\texttt{e}_{loc}) := \texttt{e}))$.
   If $\mathbb{S}_{M1} \in \texttt{dom}(\mathcal{M}_{M1}(*(\texttt{e}_{loc}) := \texttt{e}))$, then we know that there is l and v such that $eval(\texttt{e}_{loc})\ \mathbb{S}_{M1} = l$ and $eval(\texttt{e})\ \mathbb{S}_{M1} = v$.
   By lemma 5.6.3, we know that $eval(\texttt{e}_{loc})\ \mathbb{S}_{M2} = l$ and $eval(\texttt{e})\ \mathbb{S}_{M2} = v$.
   Thus we need to show that if exists $\mathbb{S}_{M1}.M(*l := v)$, then there exists $\mathbb{S}_{M2}.M(*l := v)$.

This is true by definition 5.6.1.

2. For any related $\mathbb{S}_{M1}$ and $\mathbb{S}_{M2}$, $\mathcal{M}_{M1}(*(\mathtt{e}_{loc}) := \mathtt{e})\mathbb{S}_{M1}$ is related to $\mathcal{M}_{M2}(*(\mathtt{e}_{loc}) := \mathtt{e})\mathbb{S}_{M2}$.
Since we know that $\mathbb{S}_{M1} \in \mathtt{dom}(\mathcal{M}_{M1}(*(\mathtt{e}_{loc}) := \mathtt{e}))$, then we know that there is l and v such that $eval(\mathtt{e}_{loc})\ \mathbb{S}_{M1} = l$ and $eval(\mathtt{e})\ \mathbb{S}_{M1} = v$.
By lemma 5.6.3, we know that $eval(\mathtt{e}_{loc})\ \mathbb{S}_{M2} = l$ and $eval(\mathtt{e})\ \mathbb{S}_{M2} = v$.
Thus we need to show that $\mathtt{repr}_{\mathcal{M}_{M1}-\mathcal{M}_{M2}}(\mathbb{S}_{M1}.M(*l := v), \mathbb{S}_{M2}.M(*l := v)$.
Since $\mathbb{S}_{M1}.S = \mathbb{S}_{M2}.S$, $(\mathbb{S}_{M1}.M(*l := v)).S = (\mathbb{S}_{M2}.M(*l := v)).S$.
Since $\mathbb{S}_{M1}.M \leq \mathbb{S}_{M2}.M$, by def 5.6.1, $(\mathbb{S}_{M1}.M(*l := v)).M \leq (\mathbb{S}_{M2}.M(*l := v)).M$.
By def. of $\mathtt{repr}_{\mathcal{M}_{M1}-\mathcal{M}_{M2}}$, $\mathtt{repr}_{\mathcal{M}_{M1}-\mathcal{M}_{M2}}(\mathbb{S}_{M1}.M(*l := v), \mathbb{S}_{M2}.M(*l := v))$.

Other operations are simpler, since they only update the stack with exactly the same values. For complete proof, see Coq implementation.

□

At this point we have shown that if we have two machines with memory models $M1$ and $M2$, such that these memory models are related, we can automatically generate the representation relation for the two machines. We have shown that this representation relation satisfies all the conditions necessary to form a `repr`-refinement. Now we can show how a `repr`-refinement can be used in our verification. To do this we expand the meaning of refinement to show the rule for linking the certified modules between related meta-C machines.

Corollary 5.6.5 (meta-C Refinement)

Given any two machines $\mathcal{M}_{M1}$ and $\mathcal{M}_{M2}$ for which a $\mathtt{repr}_{M1-M2}$ is defined, the following holds:

$$\frac{\mathcal{M}_{M1}, \mathcal{L} \vdash \mathbb{C} : \Psi}{\mathcal{M}_{M2}, T_{M1-M2}(\mathcal{L}) \vdash \mathbb{C} : T_{M1-M2}(\Psi)}\ M1 - M2$$

Pf. By lemma 5.6.4 our `repr` defines a valid `repr`-refinement.
By definition of valid `repr`-refinement (Lemma 4.3.2), we get above translation.

□

Thus we know that if we have two C-machines that have related memory models, then we have a working refinement between the two machines. Our next step is the to show the relations between all the memory models shown in our plan.

5.6.1 Intuition About Refinements

Our verification plan features seven models of verification. However, PE and PD are just restrictions of HW. And ALD is similar to ALE. Thus we can give the intuition to all the relations among

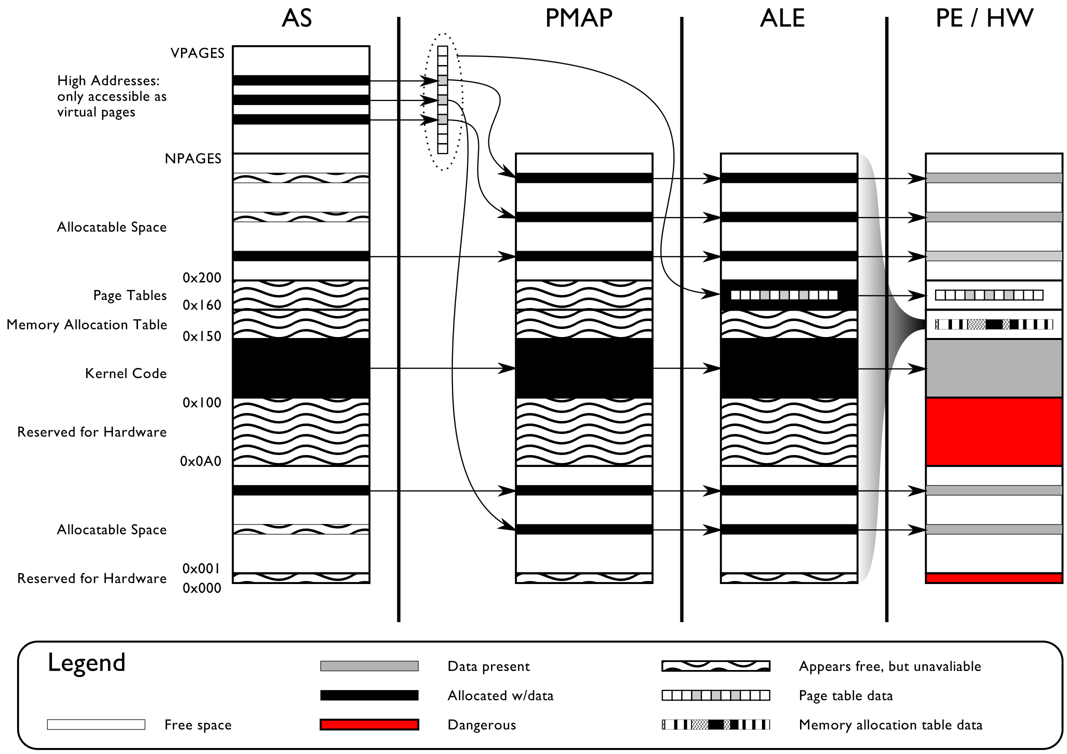


Figure 5.33: Diagram of the Relation between Memory Models

memory models by presenting the relations AS-PMAP-ALE-HW. We will explain these relations going from concrete to abstract (right to left), and we will allude to the graphical representations of these relations, which is given in Figure 5.33.

On the right is a state of the hardware memory, whose operational semantics gives little protection from accessing data. Some areas of memory are dangerous, some are empty, others contain data, including the allocation tables and page tables. This memory relates to the ALE memory model by abstracting out the memory allocation table. This allocation table now offers protection for accessing both the unallocated space, and the space that seems unallocated, but is actually dangerous to use (marked by wavy lines). An example of such area is the allocation table itself - the ALE model hides the table, making it look free in the ALE model. However, the mem-alloc of the ALE model will never allocate pages from these wavy areas, protecting these areas without complicating the memory model.

The relation between the PMAP and ALE models shows that the abstract pagemap of PMAP model is actually contained within the specific area of the ALE model. The relation makes sure that the mappings contained in the PMAP's pagemap are the same as the translation results of the

$$\begin{aligned}
&\forall l.\,\mathrm{Low}(l) \to M_{AS}.A(Pg(l)) = true \to M_{AS}.D(l) = M_{PMAP}.D(l)\\
&\forall l.\,\mathrm{High}(l) \to M_{AS}.A(Pg(l)) = true \to M_{AS}.D(l) = M_{PMAP}.D(trans(M_{PMAP}, l))\\
&\forall pg.\,\mathrm{LowPg}(pg) \to M_{AS}.A(pg) = true \to M_{PMAP}.A(pg) = true\\
&\forall pg.\,\mathrm{HighPg}(pg) \to M_{AS}.A(pg) = true \to\\
&\qquad (\exists ppg.\, M_{PMAP}.PM(pg) = ppg \land \texttt{LowPg}(ppg) \land ppg \neq 0 \land M_{PMAP}.A(ppg) = true)\\
&\forall pg.\,\mathrm{HighPg}(pg) \to M_{AS}.A(pg) = false \to M_{PMAP}.PM(pg) = 0\\
&\forall l, l'.\, l \neq l' \to (\mathrm{High}(l) \lor \mathrm{Low}(l)) \to (\mathrm{High}(l') \lor \mathrm{Low}(l')) \to M_{AS}.A(l) = true \to M_{AS}.A(l') = true \to\\
&\qquad trans(M_{PMAP}, l) \neq trans(M_{PMAP}, l')
\end{aligned}$$

where

$$trans(M_{PMAP}, l) = ppg + \mathrm{Off}(l) \text{ if} M_{PMAP}.PM(Pg(l)) = ppg \land ppg \neq 0$$

Figure 5.34: The `repr`-relation between AS and PMAP Models ($\texttt{repr}_{AS-PMAP}$)

ALE's page table structures. To protect the in-memory page tables, the relation hides the page table memory area from the PMAP model, using the same trick as the one used to protect the allocation tables in the ALE model.

The relation between the AS and PMAP models collapses PMAP's memory and the page maps into a single memory like structure in the AS model. This is mostly accomplished by chaining the translation mechanism with the storage mechanism. However, to make this work, it is imperative that the relation ensures that no two pages of the AS model ever map to the same physical page in the PMAP model. This means that all physical pages that are mapped from the high-addresses become hidden in the AS model.

We will now show the refinements more formally, as needed by our refinement system.

5.6.2 AS-PMAP

The first refinement we will define is the AS to PMAP memory model refinement. The refinement will allow us to conclude that any program certified in the AS model is also a valid program in the PMAP model.

To create this refinement we begin by showing a memory relation between the AS and the PMAP memory models. The exact mathematical relation is the conjunction of all relations described in Figure 5.34.

This relation is quite large as it describes the connections between address spaces and page maps. The intuitive description can be given as follows:

- The content of all pages in the low address area marked as valid in the AS model is the same as the content of the same pages in the PMAP model.
- The content of all pages in the high address area marked as valid in the AS model is the same as the content of low pages which are related to the original high page through the page map.
- All low pages that are allocated in that AS model are also allocated in the PMAP model.
- All high pages that are allocated in the AS model have a corresponding low page (referenced by the page map, and is non-zero), and that page is allocated.
- If a high page is not allocated in the AS model, the corresponding entry in the page map is zero.
- No two pages marked allocated in the AS model can ever map to the same page in the PMAP model.

Although such a relation is an explanation of how an address space memory model can map to a memory model with a page map based address translation, we must still show that the relation between these memory models preserves the load and stores, so that we would be able to define a refinement from it. Thus we must prove the following lemma:

Lemma 5.6.6 (AS-PMAP Memory Models Related)

$M_{AS} \preceq M_{PMAP}$

Pf. To show this, we need to demonstrate that if the load is successful in the AS state, then it must be successful and retrieve the same value in the related PMAP state. Second, we need to show that a store in the AS state will result in a state that is related to the state produced by the store applied to a related starting state. We will give the intuition here, but the details of the proof are left to the Coq implementation.

Suppose that we do a load in the AS. Either that load came from a low address or a high address, and the address must have been allocated. If the address is low, then the PMAP state must have that address allocated, and the value must be the same. If the address is high and allocated, then by our relation there must be a non-zero entry in the page map which point to some physical page, which contains the exact same values. Hence the load of the high address in the related PMAP state will generate a translated address that points to the same page, and thereby retrieves the same value

Suppose that we do a store on the same address in the relates AS and PMAP states. For store to be successful the address must be allocated in the AS model. If the address is low, then the stores write directly to the same location in D. The property that keeps pages separated guarantees that we do not update more than a single location, and hence the new states continue to be related. Similarly, if a high address is stored, then in PMAP, the mapped address is updated, but since the mapping do not change, the relation will still hold.

□

The above lemma shows that the relation that we have defined does indeed satisfy all the conditions necessary to generate a refinement between $\mathcal{M}_{AS}$ and $\mathcal{M}_{PMAP}$ meta-C machines. Thus we get the following rule:

Corollary 5.6.7 (AS-PMAP Refinement)

$$\frac{\mathcal{M}_{AS}, \mathcal{L}_{AS} \vdash \mathbb{C}^{module} : \Psi_{AS}^{module}}{\mathcal{M}_{PMAP}, T_{AS-PMAP}(\mathcal{L}_{AS}) \vdash \mathbb{C}^{module} : T_{AS-PMAP}(\Psi_{AS}^{module})}$$

Pf. Direct result of lemma 5.6.5 and lemma 5.6.6. □

The above rule is a valid refinement, but it is not quite usable yet. The problem lies in the $T_{AS-PMAP}(\mathcal{L}_{AS})$ in the result. Some of the $\mathcal{L}_{AS}$ is actually implemented in the $\mathcal{M}_{PMAP}$ machine, namely the as-request and as-release functions, which have the specification Ψ_{PMAP}^{AS}. The mem-alloc and mem-free primitives are actually defined by the $\mathcal{L}_{PMAP}$, which are not the same as the translated specifications of the $\mathcal{L}_{AS}$. The result we are looking for is this:

$$\mathcal{M}_{PMAP}, \mathcal{L}_{PMAP} \vdash \mathbb{C}^{module} \cup \mathbb{C}^{as} : T_{AS-PMAP}(\Psi_{AS}^{module}) \cup \Psi_{PMAP}^{as}$$

To achieve this result, we will need to show the following lemma:

Lemma 5.6.8 (AS-PMAP Library Weakening)

$$T_{AS-PMAP}(\mathcal{L}_{AS}(\text{mem-alloc})) \supseteq \mathcal{L}_{PMAP}(\text{mem-alloc})$$
$$T_{AS-PMAP}(\mathcal{L}_{AS}(\text{mem-free})) \supseteq \mathcal{L}_{PMAP}(\text{mem-free})$$
$$T_{AS-PMAP}(\mathcal{L}_{AS}(\text{as-request})) \supseteq \Psi_{PMAP}^{as}(\text{as-request})$$
$$T_{AS-PMAP}(\mathcal{L}_{AS}(\text{as-release})) \supseteq \Psi_{PMAP}^{as}(\text{as-release})$$

and therefore

$$T_{AS-PMAP}(\mathcal{L}_{AS}) \supseteq (\mathcal{L}_{PMAP} \cup \Psi_{PMAP}^{as})$$

Pf. Please see Coq proof. □

$$\begin{array}{l}
\forall l.\,\mathrm{Low}(l) \rightarrow M_{PMAP}.A(Pg(l)) = true \rightarrow M_{PMAP}.D(l) = M_{ALE}.D(l) \\
\forall l.\,\mathrm{Low}(l) \rightarrow \neg\mathrm{PTdom}(l) \rightarrow M_{PMAP}.A(Pg(l)) = M_{ALE}.A(Pg(l)) \\
\forall l.\,\mathrm{Low}(l) \rightarrow \mathrm{PTdom}(l) \rightarrow M_{PMAP}.A(Pg(l)) = false \wedge M_{ALE}.A(Pg(l)) = true \\
\forall vpg.\,\mathrm{HighPg}(vpg) \vee \mathrm{LowPg}(vpg) \rightarrow M_{PMAP}.PM(vpg) = M_{ALE}.D(\mathtt{PT} + vpg * 8) \\
dm(M_{ALE})
\end{array}$$

Figure 5.35: The `repr`-relation between PMAP and ALE Models ($\mathtt{repr}_{PMAP-ALE}$)

Now we can use the refinement together with our weakenings to generate the translation rule that is suitable for quickly refining code from the AS machine to the PMAP machine.

Theorem 5.6.9 (AS-PMAP Translation)

$$\frac{\mathcal{M}_{AS}, \mathcal{L}_{AS} \vdash \mathbb{C}^{module} : \Psi_{AS}^{module}}{\mathcal{M}_{PMAP}, \mathcal{L}_{PMAP} \vdash \mathbb{C}^{module} \cup \mathbb{C}^{as} : T_{AS-PMAP}(\Psi_{AS}^{module}) \cup \Psi_{PMAP}^{as}}$$

Pf.

Assume $\mathcal{M}_{AS}, \mathcal{L}_{AS} \vdash \mathbb{C}^{module} : \Psi_{AS}^{module}$.
By Lemma 5.6.7, $\mathcal{M}_{PMAP}, T_{AS-PMAP}(\mathcal{L}_{AS}) \vdash \mathbb{C}^{module} : T_{AS-PMAP}(\Psi_{AS}^{module})$.
By Library Streng. (Lemma 3.7.4), and Lemma 5.6.8, $\mathcal{M}_{PMAP}, \mathcal{L}_{PMAP} \cup \Psi_{PMAP}^{as} \vdash \mathbb{C}^{module} : T_{AS-PMAP}(\Psi_{AS}^{module})$.
By as certification (Lemma 5.5.3), $\mathcal{M}_{PMAP}, \mathcal{L}_{PMAP} \vdash \mathbb{C}^{as} : \Psi_{PMAP}^{as}$
By Linking Lemma, $\mathcal{M}_{PMAP}, \mathcal{L}_{PMAP} \vdash \mathbb{C}^{module} : T_{AS-PMAP}(\Psi_{AS}^{module}) \cup \Psi_{PMAP}^{as}$.

□

This is the result that are looking for. We can take any module written in the AS machine, and automatically convert it to a certified module in the PMAP machine by linking it with our tiny AS library.

The relations between the other memory models, and the refinements that are created by these relations tend to follow the same pattern as the AS-PMAP model, and as such, we will be a bit more brief in describing them.

5.6.3 PMAP-ALE

The PMAP memory model uses an additional data structure called the page map to relate the virtual addresses to the physical ones, whereas the ALE model uses normal address translation. Thus the relation between the PMAP and ALE models needs to show exactly how the pagemap structure is defined within the ALE memory model.

The full mathematical definition of the relation between the PMAP and the ALE memory models is given in Figure 5.35. It consists of a conjunction of five clauses which establish the following:

1. The contents of all pages that are marked allocated in the PMAP model are the same as the contents of the same pages in the ALE model.

2. The allocation status of all pages that are not used for page tables must be the same in both models.

3. Pages that contain page tables must be marked as unallocated in the PMAP model and allocated in the ALE model.

4. Every entry in the page map must be equal to the entry in the page table.

5. The page tables must contain identity entry for all pages below `NPAGES`. (The page map does not have these entries, as all low pages are implicit in the pagemap.)

This relation is represented graphically in Figure 5.33 we have seen earlier. The diagram shows that the pagemap corresponds to the contents of the page tables, located at addresses that are not accessible from within PMAP model.

The next step is to show that the PMAP-ALE relation has satisfies all the requirements for creating a `repr`-relation.

Lemma 5.6.10 (PMAP-ALE Memory Models Related)

$M_{PMAP} \preceq M_{ALE}$

Pf. Suppose we load a value from an address from the PMAP state. If the address is low, the page containing that address must be allocated. By relation, it must be allocated in the related ALE state. The relation guarantees that the AT will translate the low address into themselves, and also the relation guarantees us that both of these addresses will contain the same value in both states. Thus the loads will retrieve the same value. If the address is high, then the relation guarantees that the translation function in both memory models, will result in the same physical address, and thus will also result in the same value being loaded.

If we start from related states in PMAP and ALE models, and we store the same value into the same address. If the address is low, then the translations in both models are identity, and thus are the same, and the same part of the store gets updated. Since the page tables are marked unallocated in the PMAP model, the store can not alter the translation, and thus all the mapping continue to be related. Hence the resulting states are still related. If the address is high, the relation guarantees that the translated addresses in both models will be equal, and thus the store operation will result in related states, the same way it did for a low address.

For more specific details, please see Coq proof.

□

With the above lemma, we can define the refinement between meta-C machine with PMAP and ALE memory models.

Corollary 5.6.11 (PMAP-ALE refinement)

$$\frac{\mathcal{M}_{PMAP}, \mathcal{L}_{PMAP} \vdash \mathbb{C}^{module} : \Psi^{module}_{PMAP}}{\mathcal{M}_{ALE}, T_{PMAP-ALE}(\mathcal{L}_{PMAP}) \vdash \mathbb{C}^{module} : T_{PMAP-ALE}(\Psi^{module}_{PMAP})}$$

Pf. Direct result of Lemma 5.6.5 and Lemma 5.6.10. □

Similar to what we have encountered in the AS-PMAP refinement, we have the $T_{PMAP-ALE}(\mathcal{L}_{PMAP})$ in our result, which needs to be replaced with the appropriate definitions from the ALE library and the pt module.

Lemma 5.6.12 (PMAP-ALE Library Weakening)

$$T_{PMAP-ALE}(\mathcal{L}_{PMAP}(\text{mem-alloc})) \supseteq \mathcal{L}_{ALE}(\text{mem-alloc})$$
$$T_{PMAP-ALE}(\mathcal{L}_{PMAP}(\text{mem-free})) \supseteq \mathcal{L}_{ALE}(\text{mem-free})$$
$$T_{PMAP-ALE}(\mathcal{L}_{PMAP}(\text{pmap-set})) \supseteq \Psi^{pt}_{ALE}(\text{pmap-set})$$
$$T_{PMAP-ALE}(\mathcal{L}_{PMAP}(\text{pmap-lookup})) \supseteq \Psi^{pt}_{ALE}(\text{pmap-lookup})$$

and therefore

$$T_{PMAP-ALE}(\mathcal{L}_{PMAP}) \supseteq \mathcal{L}_{ALE} \cup \Psi^{pt}_{ALE}$$

Pf. Please see Coq proof. □

With this weakening, we can get our desired PMAP-ALE refinement result:

Theorem 5.6.13 (PMAP-ALE refinement)

$$\frac{\mathcal{M}_{PMAP}, \mathcal{L}_{PMAP} \vdash \mathbb{C}^{module} : \Psi^{module}_{PMAP}}{\mathcal{M}_{ALE}, \mathcal{L}_{ALE} \vdash \mathbb{C}^{module} \cup \mathbb{C}^{pt} : T_{PMAP-ALE}(\Psi^{module}_{PMAP}) \cup \Psi^{pt}_{ALE}}$$

Pf.

Assume $\mathcal{M}_{PMAP}, \mathcal{L}_{PMAP} \vdash \mathbb{C}^{module} : \Psi^{module}_{PMAP}$.

By Lemma 5.6.11, $\mathcal{M}_{ALE}, T_{PMAP-ALE}(\mathcal{L}_{PMAP}) \vdash \mathbb{C}^{module} : T_{PMAP-ALE}(\Psi^{module}_{PMAP})$.

$$\begin{array}{l}
\forall l.\,\mathrm{Low}(l) \rightarrow M_{ALE}.A(Pg(l)) = true \rightarrow M_{ALE}.D(l) = M_{PE}.D(l) \\
\forall pg.\,\mathrm{LowPg}(pg) \rightarrow M_{ALE}.A(pg) = true \rightarrow M_{PE}.D(\mathtt{PMM} + pg * 8) = 1 \\
\forall l.\,\mathrm{Low}(l) \rightarrow \mathrm{PMMdom}(l) \rightarrow M_{ALE}.A(Pg(l)) = false \wedge M_{PE}.D(\mathtt{PMM} + Pg(l) * 8) = 1 \\
\forall l.\,\mathrm{Low}(l) \rightarrow \mathrm{PTdom}(l) \rightarrow M_{ALE}.A(Pg(l)) = true
\end{array}$$

Figure 5.36: The `repr`-relation between ALE and PE Models ($\mathtt{repr}_{ALE-PE}$)

By Lib Strength. (Lemma 3.7.4) and Lemma 5.6.12, $\mathcal{M}_{ALE}, \mathcal{L}_{ALE} \cup \Psi^{pt}_{ALE} \vdash \mathbb{C}^{module} : T_{PMAP-ALE}(\Psi^{module}_{PMAP})$.
By pt cert. (Lemma 5.5.2), $\mathcal{M}_{ALE}, \mathcal{L}_{ALE} \vdash \mathbb{C}^{pt} : \Psi^{pt}_{ALE}$.
By Linking Lemma, $\mathcal{M}_{ALE}, \mathcal{L}_{ALE} \vdash \mathbb{C}^{module} \cup \mathbb{C}^{pt} : T_{PMAP-ALE}(\Psi^{module}_{PMAP}) \cup \Psi^{pt}_{ALE}$.

□

This theorem completes the useful rule of refinement from PMAP to ALE.

5.6.4 ALE-PE

The ALE memory model is an abstraction of the PE memory model in that it provides memory management on top of the basic memory interface. To establish the relation between these models, we must show how the allocation in the ALE model maps to the data in the PE model. This relation is described in Figure 5.36. The relation consists of the conjunction of four clauses, which can be explained as follows:

1. For all allocated pages, the contents of the pages must be the same in both models.

2. If a page *pg* is allocated, then the corresponding entry in the PMM must be marked as 1. Note that pages that are unallocated in the ALE model are not necessarily marked 0.

3. The pages containing the PMM are marked unallocated in the ALE model, but are marked with 1 in the PMM table. This means that more abstract ALE model will not be able to access or allocate these pages.

4. The page tables are allocated (and thus marked with 1 in the PMM).

To show that this representation forms a refinement between PMAP and PE, we must show the usual lemmas showing that the loads and stores of both machine models are related by the representation.

Lemma 5.6.14 (ALE-PE Memory Models Related)

$M_{ALE} \preceq M_{PE}$

Pf. Given two related states in ALE and PE models, suppose we do a load from an address in the ALE model. Perform a translation on this address to get a corresponding physical address. By definition of load, the page containing the translated address must be allocated. By relation, that means that the PE model must contain the same value at the same translated address, and since translation functions are the same for both models, a load of the same address in the PE model will result in same value as a load in the ALE model.

Given two related states in ALE and PE models, suppose we store some value at some address in both models and that the store in ALE model succeeds. Then we know that the translated address must be allocated, and hence can not be the allocation table itself (which is always unallocated in ALE). Since we did not change allocations and updated the same location, then the data store continues to be related, and the relation that maps allocation tables did not change, thus keeping the allocation table hidden and page table allocated in the ALE model. Thus the relation still holds on the updated state.

Please see Coq proof for a full and precise proof.

□

Corollary 5.6.15

$$\frac{\mathcal{M}_{ALE}, \mathcal{L}_{ALE} \vdash \mathbb{C}^{module} : \Psi^{module}_{ALE}}{\mathcal{M}_{PE}, T_{ALE-PE}(\mathcal{L}_{ALE}) \vdash \mathbb{C}^{module} : T_{ALE-PE}(\Psi^{module}_{ALE})}$$

Pf. By Lemma 5.6.5 and Lemma 5.6.14. □

Now that we have defined the refinement from $\mathcal{M}_{ALE}$ to $\mathcal{M}_{PE}$, we have to take care of the primitive library. The primitive library for the ALE model includes the `mem-alloc` and `mem-free` functions, which are implemented by the *mem* module certified in the PE machine. Thus we would need to show that the implemented functions have a stronger specification than the refined specifications of the primitive functions of the $\mathcal{L}_{ALE}$. We proceed by showing the following lemmas.

Lemma 5.6.16 (ALE-PE Library Weakening)

$$T_{ALE-PE}(\mathcal{L}_{ALE}(\text{mem-alloc})) \supseteq \Psi^{mem}_{PE}(\text{mem-alloc})$$
$$T_{ALE-PE}(\mathcal{L}_{ALE}(\text{mem-free})) \supseteq \Psi^{mem}_{PE}(\text{mem-free})$$

and therefore

$$T_{ALE-PE}(\mathcal{L}_{ALE}) \supseteq \left(\mathcal{L}_{PE} \cup \Psi^{mem}_{PE}\right)$$

Pf. Please see Coq proof. □

$$\begin{array}{l}\forall l.\, M_{PE}(l) = M_{HW}.D(l)\\ M_{HW}.\mathtt{PTROOT} = \mathtt{PT}\\ M_{HW}.\mathrm{PE} = \mathit{true}\end{array}$$

Figure 5.37: The `repr`-relation between PE and HW Models ($\mathtt{repr}_{PE-HW}$)

Using this lemma, we can show a refinement rule, which is the simplest and the most useful for certifying PMAP code.

Theorem 5.6.17 (ALE-PE refinement)

$$\frac{\mathcal{M}_{ALE}, \mathcal{L}_{ALE} \vdash \mathbb{C}^{module} : \Psi^{module}_{ALE}}{\mathcal{M}_{PE}, \mathcal{L}_{PE} \vdash \mathbb{C}^{module} \cup \mathbb{C}^{mem} : T_{ALE-PE}\left(\Psi^{module}_{ALE}\right) \cup \Psi^{mem}_{PE}}$$

Pf. Assume $\mathcal{M}_{ALE}, \mathcal{L}_{ALE} \vdash \mathbb{C}^{module} : \Psi^{module}_{ALE}$.
By Lemma 5.6.15, $\mathcal{M}_{PE}, T_{ALE-PE}(\mathcal{L}_{ALE}) \vdash \mathbb{C}^{module} : T_{ALE-PE}\left(\Psi^{module}_{ALE}\right)$.
By Lib Strength. (Lemma 3.7.4) with Corollary 5.6.16, $\mathcal{M}_{PE}, \mathcal{L}_{PE} \cup \Psi^{mem}_{PE} \vdash \mathbb{C}^{module} : T_{ALE-PE}\left(\Psi^{module}_{ALE}\right)$
By Linking Lemma, $\mathcal{M}_{PE}, \mathcal{L}_{PE} \vdash \mathbb{C}^{module} \cup \mathbb{C}^{mem} : T_{ALE-PE}\left(\Psi^{module}_{ALE}\right) \cup \Psi^{mem}_{PE}$.

□

The above lemma completes the refinement. Any certified module in the PMAP machine is also certified in the PE machine when combined with the memory allocator library. This completes the refinement to the PE machine.

5.6.5 PE-HW

Since the PE machine is just a restriction of the HW machine, it is pretty clear that the code that works on the $\mathcal{M}_{PE}$ machine also works on the $\mathcal{M}_{HW}$ machine. The relation between the memory models is also simple - it just maintains the restriction. The details are given in Figure 5.37.

The relation consists of several components. First it connects the `PT` constant of M_{PE} to the `PTROOT` register of M_{HW}. It guarantees that all related states of machine HW have paging enabled. It also shows that the contents of memory must be the same. In addition, it makes sure that the page tables include the direct map for all physical addresses - thereby guaranteeing the invariant provided by the PE machine.

We now show that the refinement generated by our `repr` is correct and valid. To show this, we show that the relation between the PE and HW memory models satisfies our requirements.

Lemma 5.6.18 (PE-HW Memory Models Related)

$M_{PE} \preceq M_{HW}$

Pf. The PE model is a restriction of an HW model. Hence if a load works in a PE model, it will have the same effect in the HW model. Similarly if a store is successful in a PE model, it will update the HW state in the exact same way, and performing a store valid in the PE model can not break the restrictions placed upon the model.

□

As usual, the fact that $M_{PE} \preceq M_{HW}$ means that we can generate the complete and valid `repr`-refinement:

Corollary 5.6.19

$$\frac{\mathcal{M}_{PE}, \mathcal{L}_{PE} \vdash \mathbb{C}^{module} : \Psi_{PE}^{module}}{\mathcal{M}_{HW}, T_{PE-HW}(\mathcal{L}_{PE}) \vdash \mathbb{C}^{module} : T_{PE-HW}(\Psi_{PE}^{module})}$$

Pf. By Lemma 5.6.5 using Lemma 5.6.18. □

As usual, we want to convert the $T_{PE-HW}(\mathcal{L}_{PE})$ into the primitives and functions of the $\mathcal{M}_{HW}$ machine. Turns out that this is entirely trivial, since $\mathcal{L}_{PE}$ is empty.

Corollary 5.6.20 (PE-HW hypo refinement)

$$T_{PE-HW}(\mathcal{L}_{PE}) \supseteq \mathcal{L}_{HW}$$

Pf. Trivial since $\mathcal{L}_{PE}$ is empty. □

Thus we can simply our the refinement so that we no longer have to deal with translations of the (non-existent) stubs.

Theorem 5.6.21 (PE-HW refinement)

$$\frac{\mathcal{M}_{PE}, \mathcal{L}_{PE} \vdash \mathbb{C}^{module} : \Psi_{PE}^{module}}{\mathcal{M}_{HW}, \mathcal{L}_{HW} \vdash \mathbb{C}^{module} : T_{PE-HW}(\Psi_{PE}^{module})}$$

Pf. By Lemma 5.6.19, $\mathcal{M}_{HW}, T_{PE-HW}(\mathcal{L}_{PE}) \vdash \mathbb{C}^{module} : T_{PE-HW}(\Psi_{PE}^{module})$

By Lib Strengthening (Lemma 3.7.4) and Lemma 5.6.20, $\mathcal{M}_{HW}, \mathcal{L}_{HW} \vdash \mathbb{C}^{module} : T_{PE-HW}(\Psi_{PE}^{module})$.

$$\begin{array}{l}
\forall l.\,\mathrm{LowPg}(Pg(l)) \to M_{ALD}.A(Pg(l)) = true \to M_{ALD}.D(l) = M_{PD}(l) \\
\forall l.\,\mathrm{Low}(l) \to \neg\mathrm{PMMdom}(l) \to M_{ALD}.A(Pg(l)) = true \to M_{PD}(\mathtt{PMM} + Pg(l) * 8) = 1 \\
\forall l.\,\mathrm{Low}(l) \to \neg\mathrm{PMMdom}(l) \to M_{ALD}.A(Pg(l)) = false \to M_{PD}(\mathtt{PMM} + Pg(l) * 8) = 0 \\
\forall l.\,\mathrm{Low}(l) \to \mathrm{PMMdom}(l) \to M_{ALD}.A(Pg(l)) = false \wedge M_{PD}(\mathtt{PMM} + Pg(l) * 8) = 1
\end{array}$$

Figure 5.38: The relation between ALD and PD Memory Models ($M_{ALD} \preceq M_{PD}$)

□

The above theorem shows that any certified module defined in the PE machine that uses PE primitive library is also a certified module in the HW machine when the specifications are translated using T_{PE-HW}, with no additional code added.

5.6.6 ALD-PD

At this point we have shown the certification of the machines that are used for the verification of the functions that run when the address translation is enabled. We now turn our attention to the machines that support initialization functions that run when the address translation is off.

The first such refinement we will show is from the $\mathcal{M}_{ALD}$ machine to the $\mathcal{M}_{PD}$ machine. As usual, we give the representation relation, in Figure 5.38. The relation is very similar to the relation that is between the M_{ALE} and M_{PE}, except we never have to worry about the locations of page tables or that they provide an identity map all pages 0-`NPAGES`.

As usual, we must show that the relation between the memory models satisfies the requirements of load and store preservations. Thus we define the following lemma.

Lemma 5.6.22 (ALD-PD Memory Models Related)

$M_{ALD} \preceq M_{PD}$

Pf. The proof is similar to the proof of ALE-PE relation, except we no longer need to worry about high addresses. □

Now that we have established the fact that the relation between two memory models satisfies the requirements, we go through the usual motions to construct the refinement based on the `repr`-relation.

Corollary 5.6.23

$$\frac{\mathcal{M}_{ALD}, \mathcal{L}_{ALD} \vdash \mathbb{C}^{module} : \Psi^{module}_{ALD}}{\mathcal{M}_{PD}, T_{ALD-PD}(\mathcal{L}_{ALD}) \vdash \mathbb{C}^{module} : T_{ALD-PD}(\Psi^{module}_{ALD})}$$

Pf. Use Lemma 5.6.5 with Lemma 5.6.22. □

The usual next step is to replace the translated primitive library into the specifications that are actually defined within the more concrete machine. Thus we prove the usual weakening lemma.

Lemma 5.6.24 (ALD-PD Library Weakening)

$$T_{ALD-PD}(\mathcal{L}_{ALD}(\text{mem-alloc}) \supseteq \Psi^{mem}_{PD}(\text{mem-alloc})$$
$$T_{ALD-PD}(\mathcal{L}_{ALD}(\text{mem-free}) \supseteq \Psi^{mem}_{PD}(\text{mem-free})$$

and therefore

$$T_{ALD-PD}(\mathcal{L}_{ALD}) \supseteq (\mathcal{L}_{PD} \cup \Psi^{mem}_{PD})$$

Pf. Please see Coq proof. □

And now, we improve our refinement lemma into a nicer form.

Theorem 5.6.25 (ALD-PD refinement)

$$\frac{\mathcal{M}_{ALD}, \mathcal{L}_{ALD} \vdash \mathbb{C}^{module} : \Psi^{module}_{ALD}}{\mathcal{M}_{PD}, \mathcal{L}_{PD} \vdash \mathbb{C}^{module} \cup \mathbb{C}^{mem} : T_{ALD-PD}(\Psi^{module}_{ALD}) \cup \Psi^{mem}_{ALD}}$$

Pf. Assume $\mathcal{M}_{ALD}, \mathcal{L}_{ALD} \vdash \mathbb{C}^{module} : \Psi^{module}_{ALD}$.

By Lemma 5.6.23, $\mathcal{M}_{PD}, T_{ALD-PD}(\mathcal{L}_{ALD}) \vdash \mathbb{C}^{module} : T_{ALD-PD}(\Psi^{module}_{ALD})$.

By Lib Strength. (Lemma 3.7.4) and Lemma 5.6.24, $\mathcal{M}_{PD}, (\mathcal{L}_{PD} \cup \Psi^{mem}_{PD}) \vdash \mathbb{C}^{module} : T_{ALD-PD}(\Psi^{module}_{ALD})$.

By Lemma 5.5.1, $\mathcal{M}_{PD}, \mathcal{L}_{PD} \vdash \mathbb{C}^{mem} : \Psi^{mem}_{PD}$.

By Linking Lemma, $\mathcal{M}_{PD}, \mathcal{L}_{PD} \vdash \mathbb{C}^{module} \cup \mathbb{C}^{mem} : T_{ALD-PD}(\Psi^{module}_{ALD}) \cup \Psi^{mem}_{PD}$.

□

5.6.7 PD-HW

The last relation between memory models that we will show is the M_{PD} to M_{HW} memory relation. This relation is similar to the $PE-HW$ relation, except the PD model uses a different restriction on

$$\begin{array}{l}\forall l.\, M_{PD}(l) = M_{HW}.D(l) \\ M_{HW}.PE = false\end{array}$$

Figure 5.39: The `repr`-relation between PD and HW Modules ($\texttt{repr}_{PD-HW}$)

memory that is even simpler. The relation is given in Figure 5.39. The relation simply states that the contents of memory are identical in both models, and that to be related the address translation has to be off.

Under such a simple relation, it is quite trivial to show that $M_{PD} \leq M_{HW}$.

Lemma 5.6.26 (PD-HW Memory Models Related)

$M_{PD} \leq M_{HW}$

Pf. Similar to the proof of PE-HW relation, except we no longer worry about high addresses. □

Once again we apply our usual steps. First we define a refinement.

Corollary 5.6.27

$$\frac{\mathcal{M}_{PD}, \mathcal{L}_{PD} \vdash \mathbb{C}^{module} : \Psi_{PD}^{module}}{\mathcal{M}_{HW}, T_{PD-HW}(\mathcal{L}_{PD}) \vdash \mathbb{C}^{module} : T_{PD-HW}(\Psi_{PD}^{module})}$$

Pf. Direct result of Lemma 5.6.5 with Lemma 5.6.26. □

Then we try to replace the $T_{PD-HW}(\mathcal{L}_{PD})$ with the appropriate equivalents in the $\mathcal{M}_{HW}$ machine, which in this case is a trivial problem since $\mathcal{L}_{PD}$ is empty.

Lemma 5.6.28 (PD-HW Library)

$$T_{PD-HW}(\mathcal{L}_{PD}) \supseteq \mathcal{L}_{HW}$$

Pf. Trivial since $\mathcal{L}_{PD}$ is empty. □

And then we can convert the refinement into an easy to use form.

Theorem 5.6.29 (PD-HW Refinement)

$$\frac{\mathcal{M}_{PD},\mathcal{L}_{PD} \vdash \mathbb{C}^{module} : \Psi_{PD}^{module}}{\mathcal{M}_{HW},\mathcal{L}_{HW} \vdash \mathbb{C}^{module} : T_{PD-HW}(\Psi_{PD}^{module})}$$

Pf. By Lemma 5.6.27, $\mathcal{M}_{HW}, T_{PD-HW}(\mathcal{L}_{PD}) \vdash \mathbb{C}^{module} : T_{PD-HW}(\Psi_{PD}^{module})$

By Lib Strengthening (Lemma 3.7.4) and Lemma 5.6.28, $\mathcal{M}_{HW},\mathcal{L}_{HW} \vdash \mathbb{C}^{module} : T_{PD-HW}(\Psi_{PD}^{module})$

□

5.7 Initialization

At this point we have given specification to all the functions of the VMM, and we have all the refinements necessary to show that the kernel can execute in the HW model of memory. However, we still must show that we can actually activate the kernel, as its initialization function is defined over the AS model of memory, and it is not clear that we can actually start it.

What starts the kernel is the `init()` function, which we will show is safe to run in the HW machine. As its last action it calls the `kernel-init()` function. To show that this is safe, we must somehow guarantee that it is safe to jump into the kernel. This is done by setting up all the data structures necessary for the HW model to be related to some state of the AS memory model that satisfies the specification of `kernel-init`. Setting up this state is exactly the task that the `init` performs.

To show that `init` is certified, we must first convert it to our meta-C notation. Since there are no loops, the result is achieved trivially, and is given in Figure 5.40.

This procedure is written with HW memory model in mind, as it needs to be able to set the PTROOT and PE registers, as well as to be able to start with proper assumptions about the state of the hardware. However, as we can see it also calls `mem-init` and `pt-init` to initialize memory, whose specifications are given for a different memory models. For example, the specification of `mem-init()` is given in $\Psi_{PD}^{meminit}$.

To be able to call `mem-init()` from `init()`, we must translate the specifications from the $\mathcal{M}_{PD}$ machine to the $\mathcal{M}_{HW}$ machine. We have already shown this to be a refinement, and thus the specification of `mem-init()` is $T_{PD-HW}(\Psi_{PD}^{meminit})$(mem-init).

However, it is very messy, although not impossible, to call the translated specification directly,

```
init
$fcall$(mem-init, []);
[mem-init];
$readret(junk)$;
$fcall$(pt-init, []);
[pt-init];
$readret(junk)$;
$fcall$(set-PTROOT, [PT]);
[set-PTROOT];
$readret(junk)$;
$fcall$(set-PE, []);
[set-PE];
$readret(junk)$;
$fcall$(kernel-init, []);
[kernel-init];
//never returns
```

Figure 5.40: `init` Procedure

$$[] \mapsto \begin{pmatrix} \texttt{PE} = \text{false?},\ (D(\texttt{PMM} + \text{Pg}(l) * 8) = 1 \mid \text{PtDom}(l) \vee \text{PMMdom}(l)), \\ (D(\texttt{PMM} + l * 8) = \{0,1\} \mid \text{LowPg}(l)),\ \texttt{ret}(0) \end{pmatrix}$$

Figure 5.41: Specification of mem-init for HW Model ($a_{HW}^{mem-init}$)

as it has several quantifiers and relations introduced by the automatic `repr`-based spec translation. Another option is to give the `mem-init()` function a specification for the HW model of memory, and to show that this new specification is weaker than the translated one. Such specification is given in Figure 5.41, and the weakening is shown by the following lemma:

Lemma 5.7.1 (PD-HW mem-init)

$$\mathsf{a}_{HW}^{mem-init} \supseteq T_{PD-HW}(\Psi_{PD}^{mem-init})$$

Pf. The only difference between $\mathsf{a}_{HW}^{mem-init}$ and $\Psi_{PD}^{mem-init}$ is that the HW version requires that `PE` is set to false. This is the only requirement that the relation PD-HW imposes on the code, and this additional precondition ensures that the HW specification of mem-init can relate to its PD spec. Intuitively, the weakening is obvious, since both specifications are the same. For details of this proof, we refer the reader to the Coq implementation.

□

Thus we can now use $\mathsf{a}_{HW}^{mem-init}$ as a specification of mem-init that is available for the use in the $\mathcal{M}_{HW}$ machine. Thus we will be able to use this specification when we call `mem-init` from the init

$$[] \mapsto \left(\begin{array}{l} (M(\texttt{PMM} + \text{Pg}(l) * 8) \in \{0,1\} \mid \text{Low}(l))?, \\ (M(\texttt{PMM} + \text{Pg}(l) * 8) = 1 \mid \text{PTdom}(l) \vee \text{PMMdom}(l))?, \\ (M(\texttt{PT} + pg * 8) := pg \mid \text{LowPg}(pg)), \\ (M(\texttt{PT} + pg * 8) := 0 \mid \text{HighPg}(pg)), \\ \texttt{ret}(0) \end{array}\right)$$

Figure 5.42: Specification of pt-init for PD Model ($a_{PD}^{pt-init}$)

function.

Now, we must do the same trick with pt-init. pt-init has its specification defined for the ALD memory model, so its HW specification undergoes two refinements:

$$T_{PD-HW}(T_{ALD-PD}(\Psi_{ALD}^{ptinit}))(\text{pt-init})$$

Needless to say that trying to verify the init using this specification of pt-init would be even more annoying, as the specification of `pt-init` for the HW machine will have two translations. Thus we try to find a nicer specification of `pt-init` for the HW model ($\mathsf{a}_{HW}^{pt-init}$), which will be weaker than the translated version, so that we could verify `init` with the weaker spec, and then link it with the stronger translated specification of the actual `pt-init`.

But even this approach is already a bit annoying, as the spec undergoing two translations becomes messy quick. To try to limit the mess, we chain the process by defining yet another intermediate spec for `pt-init` in the *PD* model of memory. The specification that we have come up with is given in Figure 5.42. Using this spec, we can now show that the new specification we have designed is weaker than the original specification of `pt-init` refined to the PD model. This is shown by the following lemma:

Lemma 5.7.2 (ALD-PD pt-init)

$$\mathsf{a}_{PD}^{pt-init} \supseteq T_{ALD-PD}\left(\Psi_{ALD}^{ptinit}(\text{pt-init})\right)$$

Pf. Please refer to the Coq proof. □

Now we have a specification of `pt-init` for the PD machine. We can now show its specification for the HW model, which we show in Figure 5.43. And once again we show that the new specification is weaker than the translated specification of `pt-init` for the PD model. The following lemma

$$[] \mapsto \left(\begin{array}{l} \mathtt{PE} = \text{false}?, \\ (M(\mathtt{PMM} + \text{Pg}(l) * 8) \in \{0, 1\} \mid \text{Low}(l))?, \\ (M(\mathtt{PMM} + \text{Pg}(l) * 8) = 1 \mid \text{PTdom}(l) \vee \text{PMMdom}(l))?, \\ (M(\mathtt{PT} + pg * 8) := pg \mid \text{LowPg}(pg)), \\ (M(\mathtt{PT} + pg * 8) := 0 \mid \text{HighPg}(pg)), \\ \cancel{\{M(l) \mid M(\mathtt{PMM} + \text{Pg}(l) * 8) = 0\},} \\ \mathtt{ret}(0) \end{array} \right)$$

Figure 5.43: Specification of pt-init for HW Model ($a_{HW}^{pt-init}$)

shows this:

Lemma 5.7.3 (PD-HW pt-init)

$$\mathsf{a}_{HW}^{pt-init} \supseteq T_{PD-HW}(\mathsf{a}_{PD}^{pt-init})$$

Pf. Please see Coq proof. □

Now we just have to put them together. To do so, we rely on the fact of our framework that shows that the refinements of specifications for any order-preserving refinements must preserve the weaker-than relation. Thus we can conclude the following

Corollary 5.7.4 (ALD-PD pt-init corollary)

$$T_{PD-HW}(\mathsf{a}_{PD}^{pt-init}) \supseteq T_{PD-HW}(T_{ALD-PD}(\Psi_{ALD}^{ptinit}))$$

Pf. Direct consequence of Lemma 5.7.2 and that T_{PD-HW} preserves weaker-than relations (one of properties we prove for all `repr`-refinements). □

Corollary 5.7.5 (ALD-HW pt-init)

$$\mathsf{a}_{HW}^{pt-init} \supseteq T_{PD-HW}(T_{ALD-PD}(\Psi_{ALD}^{ptinit}))(\text{pt-init})$$

Pf. By applying transitivity of weaker-than relation to Corollary 5.7.4 and Lemma 5.7.3. □

Thus, when the `init` function calls `pt-init`, we will use the $\mathsf{a}_{HW}^{pt-init}$ as the specification of the callee, and we will be able to link it with the actual specification given in Ψ_{ALD}^{ptinit} later.

Name	Specification
Ψ^{kernel}_{AS}(kernel-init)	$[] \mapsto \mathit{loop}$
$\mathsf{a}^{\text{kernel-init}}_{PMAP}$	$[] \mapsto ((PM(pg) = 0 \mid \text{HighPg}(pg))? \circ \mathit{loop})$
$\mathsf{a}^{\text{kernel-init}}_{ALE}$	$[] \mapsto \left(\begin{array}{l} (A(\text{Pg}(l)) = \text{true} \mid \text{PTdom}(l))?, \\ (D(\texttt{PT} + pg * 8) = pg \mid \text{LowPg}(pg))?, \\ (D(\texttt{PT} + pg * 8) = 0 \mid \text{HighPg}(pg))? \end{array} \right) \circ \mathit{loop}$
$\mathsf{a}^{\text{kernel-init}}_{PE}$	$[] \mapsto \left(\begin{array}{l} (M(\texttt{PMM} + \text{Pg}(l) * 8) = 1 \mid \text{PTdom}(l) \vee \text{PMMdom}(l))?, \\ (M(\texttt{PTROOT} + pg * 8) = pg \mid \text{LowPg}(pg))?, \\ (M(\texttt{PTROOT} + pg * 8) = 0 \mid \text{HighPg}(pg))? \end{array} \right) \circ \mathit{loop}$
$\mathsf{a}^{\text{kernel-init}}_{HW}$	$[] \mapsto \left(\begin{array}{l} \texttt{PE} = \mathit{true}?,\ \texttt{PTROOT} = \texttt{PT}?, \\ (D(\texttt{PMM} + \text{Pg}(l) * 8) = 1 \mid \text{PTdom}(l) \vee \text{PMMdom}(l))?, \\ (D(\texttt{PTROOT} + pg * 8) = pg \mid \text{LowPg}(pg))?, \\ (D(\texttt{PTROOT} + pg * 8) = 0 \mid \text{HighPg}(pg))? \end{array} \right) \circ \mathit{loop}$

Figure 5.44: Intermediate Specifications of kernel-init

5.7.1 Calling kernel-init

The last such complicated call that we have to worry about is the final call to the `kernel-init()` function. The specification of the kernel init is given in Ψ^{kernel}_{AS}(kernel-init), and is defined for the AS model. To be able to call this function, we will have to translate its specification to the HW model as well. This is actually a bit annoying, since we have to go through a long series of translations. What is nice, however, is that `kernel-init` never returns, and thus its specification consists almost completely of the precondition, which is easier to refine.

Figure 5.44 list the specifications that we have defined for each level, including the original specification that we expect the kernel to follow. Each of these specifications is a refinement of the previous level, and thus if we call $\mathsf{a}^{\text{kernel-init}}_{HW}$, we would show all the requirements to actually satisfy the call the actual kernel-init function that would be defined by those who supply the high-level kernel.

However, we still need to show that this is indeed true. The procedure is the same as the one for the pt-init, except with more steps. This time, we will not show the individual steps, but demonstrate them all in one shot.

Lemma 5.7.6 (AS-HW kernel init)

$$
\begin{aligned}
&\mathsf{a}_{HW}^{\text{kernel-init}} \supseteq \\
&\quad T_{PE-HW}(\mathsf{a}_{PE}^{\text{kernel-init}}) \supseteq \\
&\quad T_{PE-HW}(T_{ALE-PE}(\mathsf{a}_{ALE}^{\text{kernel-init}})) \supseteq \\
&\quad T_{PE-HW}(T_{ALE-PE}(T_{PMAP-ALE}(\mathsf{a}_{PMAP}^{\text{kernel-init}}))) \supseteq \\
&\quad T_{PE-HW}(T_{ALE-PE}(T_{PMAP-ALE}(T_{AS-PMAP}(\Psi_{AS}^{kernel}))))(\text{kernel-init})
\end{aligned}
$$

Pf. We show weaker-than relations for each of the translations (see Coq proofs). These weakening can be chained. The proof makes use of the fact that `repr`-refinement's action translation function (T_{a}) preserves weaker-than relation ($\mathsf{a} \supseteq \mathsf{a}' \to T_{\mathsf{a}}(\mathsf{a}) \supseteq T_{\mathsf{a}}(\mathsf{a}')$). This is in fact a property that `repr`-refinement had to satisfy to be a special case of order-preserving refinement.

□

5.7.2 Certifying `init`

We have finally come to the point where we could certify the `init` function. First, we need to give it a specification, which is quite simple, as the function starts in a starting state of the HW model, and never returns, as it calls `kernel-init`. Thus the specification of the `init` is the following:

$$\Psi_{HW}^{init}(init) \quad \triangleq \quad [\,] \mapsto (\mathtt{PE} = \text{false?}) \circ loop$$

We use our new specifications of `mem-init`, `pt-init`, and `kernel-init` redesigned for the HW model to perform the certification. The exact certification that we are performing is defined by the following lemma:

Lemma 5.7.7 (`init` Correct)

$$
\begin{aligned}
&\mathcal{M}_{HW}, \\
&\quad \mathcal{L}_{HW} \cup \{\text{kernel-init} \rightsquigarrow \mathsf{a}_{HW}^{\text{kernel-init}}, \text{mem-init} \rightsquigarrow \mathsf{a}_{HW}^{\text{mem-init}}, \text{pt-init} \rightsquigarrow \mathsf{a}_{HW}^{pt-init}\} \vdash \\
&\qquad \mathbb{C}^{init}(\text{init}) : \Psi_{HW}^{\text{init}}(\text{init})
\end{aligned}
$$

Pf. By the spec, we start AT off, and an empty data stack. Then proceed to call mem-init, pt-init, setPTROOT, and setPE in that order, each one setting up a part of the state that will satisfy the kernel-init, which will never return. For precise proof, see Coq proof.

□

The certification of init relies on our newly defined HW-model specifications of all the callees. We must now strengthen the specification to use the translated specifications of the actual functions that we have verified. This is shown in the next lemma.

Theorem 5.7.8 ($\mathbb{C}^{init}$ certified)

$$\begin{aligned}&\mathcal{M}_{HW}, \mathcal{L}_{HW} \cup \\ &\quad T_{PD-HW}(\Psi^{meminit}_{PD}) \cup \\ &\quad T_{PD-HW}(T_{ALD-PD}(\Psi^{ptinit}_{ALD})) \cup \\ &\quad T_{PE-HW}(T_{ALE-PE}(T_{PMAP-ALE}(T_{AS-PMAP}(\Psi^{kernel}_{AS})))) \\ &\qquad \vdash \mathbb{C}^{init} : \Psi^{init}_{HW}\end{aligned}$$

Pf.
Take the result of lemma 5.7.7.
Then apply Library Strengthening (Lemma 3.7.4) and lemmas 5.7.6, 5.7.5, and 5.7.1.
Since `init` is the only label in $\mathbb{C}^{init}$, we get the final result.

□

5.8 Putting Everything Together

At this point we have proven all the modules in their appropriate memory models, and we have shown the refinements that are present in our model diagram. Now we can start putting them together towards the final result.

We will start with our kernel assumption:

$$\mathcal{M}_{AS}, \mathcal{L}_{AS} \vdash \mathbb{C}^{module} : \Psi^{kernel}_{AS}$$

We will use the AS-PMAP refinement (Lemma 5.6.7) to produce the following result:

$$\mathcal{M}_{PMAP}, \mathcal{L}_{PMAP} \vdash \mathbb{C}^{kernel} \cup \mathbb{C}^{as} : T_{AS-PMAP}(\Psi^{kernel}_{AS}) \cup \Psi^{as}_{PMAP}$$

Then we refine it using our PMAP-ALE theorem (Lemma 5.6.13) to produce the following:

$$\mathcal{M}_{ALE}, \mathcal{L}_{ALE} \vdash \mathbb{C}^{kernel} \cup \mathbb{C}^{as} \cup \mathbb{C}^{pt} : T_{PMAP-ALE}(T_{AS-PMAP}(\Psi^{kernel}_{AS})) \cup T_{PMAP-ALE}(\Psi^{as}_{PMAP}) \cup \Psi^{pt}_{ALE}$$

Progressing with ALE-PE refinement (Lemma 5.6.17) to get the following:

$$\begin{aligned}
&\mathcal{M}_{PE}, \mathcal{L}_{PE} \vdash \mathbb{C}^{kernel} \cup \mathbb{C}^{as} \cup \mathbb{C}^{pt} \cup \mathbb{C}^{mem} : \\
&\quad T_{ALE-PE}(T_{PMAP-ALE}(T_{AS-PMAP}(\Psi_{AS}^{kernel}))) \cup \\
&\quad T_{ALE-PE}(T_{PMAP-ALE}(\Psi_{PMAP}^{as})) \cup \\
&\quad T_{ALE-PE}(\Psi_{ALE}^{pt}) \cup \Psi_{PE}^{mem}
\end{aligned}$$

And finally we refine the kernel down to the HW machine by using the PE-HW refinement (Lemma 5.6.21):

$$\begin{aligned}
&\mathcal{M}_{HW}, \mathcal{L}_{HW} \vdash \mathbb{C}^{kernel} \cup \mathbb{C}^{as} \cup \mathbb{C}^{pt} \cup \mathbb{C}^{mem} : \\
&\quad T_{PE-HW}(T_{ALE-PE}(T_{PMAP-ALE}(T_{AS-PMAP}(\Psi_{AS}^{kernel})))) \cup \\
&\quad T_{PE-HW}(T_{ALE-PE}(T_{PMAP-ALE}(\Psi_{PMAP}^{as}))) \cup \\
&\quad T_{PE-HW}(T_{ALE-PE}(\Psi_{ALE}^{pt})) \cup \\
&\quad T_{PE-HW}(\Psi_{PE}^{mem})
\end{aligned}$$

Now that we have refined the kernel certification down to the HW machine, we can now refine the initialization functions. By using PD-HW refinement (Lemma 5.6.29), we can take the certification of mem-init (Lemma 5.5.4), and refine it to the HW machine as well.

$$\mathcal{M}_{HW}, \mathcal{L}_{HW} \vdash \mathbb{C}^{meminit} : T_{PD-HW}(\Psi_{PD}^{meminit})$$

We do the same with the pt-init, except we have to take it through ALD-PD refinement (Lemma 5.6.25) and then through PD-HW refinement (Lemma 5.6.29) to get the following:

$$\mathcal{M}_{HW}, \mathcal{L}_{HW} \vdash \mathbb{C}^{ptinit} : T_{PD-HW}(T_{ALD-PD}(\Psi_{ALD}^{ptinit}))$$

The refined certifications of the meminit and ptinit can be now linked to the initialization function (Lemma 5.7.8), by using the linking lemma.

$$
\begin{aligned}
&\mathcal{M}_{HW}, \mathcal{L}_{HW} \cup \\
&\quad T_{PE-HW}(T_{ALE-PE}(T_{PMAP-ALE}(T_{AS-PMAP}(\Psi_{AS}^{kernel})))) \\
&\qquad \vdash \mathbb{C}^{init} \cup \mathbb{C}^{meminit} \cup \mathbb{C}^{ptinit} : \Psi_{HW}^{init} \cup T_{PD-HW}(\Psi_{PD}^{meminit}) \cup T_{PD-HW}(T_{ALD-PD}(\Psi_{ALD}^{ptinit}))
\end{aligned}
$$

And now we link the piece containing the kernel and its supporting modules together with the initialization code by using the linking lemma.

$$
\begin{aligned}
&\mathcal{M}_{HW}, \mathcal{L}_{HW} \vdash \\
&\quad \mathbb{C}^{kernel} \cup \mathbb{C}^{as} \cup \mathbb{C}^{pt} \cup \mathbb{C}^{mem} \cup \mathbb{C}^{init} \cup \mathbb{C}^{meminit} \cup \mathbb{C}^{ptinit} : \\
&\quad T_{PE-HW}(T_{ALE-PE}(T_{PMAP-ALE}(T_{AS-PMAP}(\Psi_{AS}^{kernel})))) \cup \\
&\quad T_{PE-HW}(T_{ALE-PE}(T_{PMAP-ALE}(\Psi_{PMAP}^{as}))) \cup \\
&\quad T_{PE-HW}(T_{ALE-PE}(\Psi_{ALE}^{pt})) \cup \\
&\quad T_{PE-HW}(\Psi_{PE}^{mem}) \cup \\
&\quad \Psi_{HW}^{init} \cup \\
&\quad T_{PD-HW}(\Psi_{PD}^{meminit}) \cup \\
&\quad T_{PD-HW}(T_{ALD-PD}(\Psi_{ALD}^{ptinit}))
\end{aligned}
$$

The above conclusion shows us that given any high-level kernel certified using the virtual address space model ($\mathcal{M}_{AS}$), linked together with the pieces of our virtual memory manager, is certified in the C-machine with address translation model ($\mathcal{M}_{HW}$). Thus this means that we can execute the `init()` function and guarantee safety of the entire kernel.

Chapter 6

Towards Realism

At this point, we have done a complete verification of the small virtual memory manager on simplified hardware. We think that our approach will scale to real virtual memory managers, and real hardware. To show how this can be accomplished, we will present several updates to the memory manager, making it more realistic. In this chapter, we will present the following improvements:

- Including the Translation Look-aside Buffer (TLB), which adds a small degree of complexity to the address translation mechanism.
- Allowing for the page tables to be allocated, and thus not fixed to a particular location in memory.
- Updating the virtual memory manager to handle multiple address spaces. The new model of memory will include a way to quickly switch between the address spaces.
- Updating the page table driver to handle the more realistic multi-level page tables.

For each of these, we will present the updates to the memory models and to the relations between the memory models to get each of these features to work. However, we may not have a complete and formal proof for all of these additional features.

6.1 Translation Look-aside Buffer

The TLB works as a caching system for the translation mechanism. The semantics of this cache are non-trivial, and what is worse, they can vary by model and make of the versions of hardware. The

$$\begin{array}{rll}
(\textit{Memory System}) & M & ::= (D, \mathtt{PE}, \mathtt{PTROOT}, \mathit{TLB}) \\
(\textit{Data}) & D & ::= \{\mathit{addr} \rightsquigarrow w \mid \mathrm{Low}(\mathit{addr})\}^* \\
(\textit{Address Translation Enabled}) & \mathtt{PE} & ::= (\mathrm{bool}) \\
(\textit{Pointer to AT Table}) & \mathtt{PTROOT} & ::= w\ (\mathrm{address}) \\
(\textit{TLB Synchronized}) & \mathit{TLB} & ::= (\mathrm{bool})
\end{array}$$

Notation	`Definition`
$M(\mathit{vaddr})$	$D(\mathit{trans}_M(\mathit{vaddr}))$
$M(\mathit{vaddr}) := w$	$(D(\mathit{trans}_M(\mathit{vaddr})) := w, \mathit{TLB} := (\mathit{TLB} \wedge \neg(\mathrm{PTaffected}_M(\mathit{vaddr}))))$

where

$$\begin{array}{ll}
\mathit{trans}_M(\mathit{va}) & := \begin{cases} D(\mathtt{PTROOT} + \mathrm{Pg}(\mathit{va}) * 8) * \mathtt{PGSIZE} + \mathrm{Off}(\mathit{va}) & \text{if } \mathtt{PE} = \mathit{true} \wedge \mathit{TLB} = \text{true} \\ \mathit{va} & \text{otherwise} \end{cases} \\
\mathrm{PTaffected}_M(\mathit{va}) & := \mathtt{PTROOT} < \mathit{trans}_M(\mathit{va}) \leq \mathtt{PTROOT} + \mathtt{VPAGES} * 8
\end{array}$$

Function	Specification
`hw-setPE`	$[] \mapsto (\mathit{ValidPT}(\mathtt{PTROOT})?, \mathtt{PE} := \mathtt{true}, \mathit{TLB} := \text{true}, \mathtt{ret}(0))$
`hw-setPTROOT`	$[\mathit{newroot}] \mapsto (\mathit{ValidPt}(\mathit{newroot})?, \mathtt{PTROOT} := \mathit{newroot}, \mathit{TLB} := \text{true}, \mathtt{ret}(0))$

Figure 6.1: Hardware Memory Model with TLB and Stub Library

system designers take a conservative approach to dealing with the TLB (except on hardware where TLB is programmer managed, such as ARM[43]) - they do not assume that they know exactly how it works, or when it will be used. They simply keep track of when the TLB may result in something different from what is defined by the page tables.

This is the approach that we would try to follow. Instead of defining TLB as a cache, we instead define it as a flag that indicates whether the TLB is synchronized with the information contained in the page tables or not. By following the conservative approach, we define all memory accesses when the TLB is not synchronized as causing a crash.

There are several things that we would need to do to incorporate the TLB in our verification. First and most obvious, is the need to add the semantics of the TLB to the HW model. The new HW model is given in Figure 6.1. The main modification to the state of the machine is the addition of the TLB flag. This flag is now checked by the *trans* function, and if AT is enabled, but the TLB flag is off, the translation will fail. The other big change is to the store operation, which will now set the TLB flag to off if the updated address points to an area marked for page table access. Lastly, setting `PE` and `PTROOT` registers will reset the TLB, and thus mark it true. This completely defines the semantics of the hardware.

As the PD model of memory marks address translation to be off, the model itself is not modified. However, the relation $PD \leq HW$ must now include the TLB. However, we purposefully ignore the TLB, meaning that any code running in the PD model will have specifications that leave TLB unknown, which is perfectly acceptable for code that runs with AT off. However, because the HW model is different, we will need to produce new stubs for `mem-init` and `pt-init` functions in the HW model, which we labelled $\mathtt{a}_{HW}^{mem-init}$ and $\mathtt{a}_{HW}^{pt-init}$. These new stubs update the specifications to include the following: $\cancel{TLB}$. This addition is necessary in order to satisfy the relation, as all functions refined from PD may desync the TLB. However, as AT is off, this change does not matter, and the verification of the `init` is practically unchanged.

The PE model is an "AT always on" restriction of the HW model, and thus it includes the TLB as a part of the definition, with a similar update to the store functions. However, in the original plan we have no stubs, as updates to the `PTROOT` and `PE` registers are not necessary. In this case we will need to be able to reset the TLB, and thus we will define the new `resetTLB` stub that resets the TLB. This stub will, in reality, be an alias to the `setPE` stub of the HW model, but since the AT is always on the only effect of `setPE` will be TLB reset. Thus the new $\mathcal{L}_{PE}$ will be the following:

$$\mathcal{L}_{PE} := \{\mathtt{resetTLB} \rightsquigarrow ([] \mapsto (TLB := \text{true}, \mathtt{ret}(0)))\}$$

The relation between PE and HW models will now have to include an additional clause, which states that $\mathbb{S}_{PE}.TLB = \mathbb{S}_{HW}.TLB$. This, in addition to the new behavior of store would require redoing the proofs that the relation preserves loads and stores, which would not be too complicated. We will have to also show the following,

$$T_{PE-HW}(\mathcal{L}_{PE}(\text{resetTLB})) \supseteq \mathcal{L}_{HW}(\mathtt{hw\text{-}setPE})$$

which is a simple proof.

The `mem-alloc` and `mem-free` functions are no longer certified, since the model has been modified. Thus we must re-certify them, which we do by adding a $(TLB = \mathtt{true})$? precondition to guarantee that the functions are executed in a synchronized state, and although we keep the same notation for the spec, the actual spec now guarantees us that the memory functions keep the TLB synchronized. These changes require us to reprove the allocation functions. The updated proof is similar to the original one, but during every store we have to show that the address is not in the protected area, which is only a slight complication, since there is only one store in each procedure.

The trend of pushing the TLB up the abstraction continues, as we also have to include it into the ALE model. Just like before, we add the *TLB* to the state, add the check to the translation function, and alter the store to unset the flag when an address within the page table area is affected. We also have to add the `resetTLB` stub to the machine so that the programs have a way to reset the TLB when needed, and the stub will have the specification that appears the same as the one in TLB model. However, it is not exactly that same, as the state is different, and thus the same notation defines slightly different specifications.

$$\mathcal{L}_{ALE} := \{\texttt{resetTLB} \rightsquigarrow ([] \mapsto (\mathit{TLB} := \text{true}, \texttt{ret}(0)))\}$$

The relation between the ALE and PE models will also add the clause about the TLB equality: $\mathbb{S}_{ALE}.\mathit{TLB} = \mathbb{S}_{HW}.\mathit{TLB}$, and since the semantics of loads and stores are different, we also have to redo the proofs that show that the memory relation is valid for our C refinement. This change also necessitates reproving all the refinements for the stubs, since the specifications of the stubs, and the specifications of the actual implementations have been modified. Thus we reprove all of them:

$$\begin{aligned}
&T_{ALE-PE}(\mathcal{L}_{ALE}(\text{mem-alloc})) \supseteq \Psi^{mem}_{PE}(\text{mem-alloc})\\
&T_{ALE-PE}(\mathcal{L}_{ALE}(\text{mem-free})) \supseteq \Psi^{mem}_{PE}(\text{mem-free})\\
&T_{ALE-PE}(\mathcal{L}_{ALE}(\text{resetTLB})) \supseteq \Psi^{mem}_{PE}(\text{resetTLB})
\end{aligned}$$

This task is not that complex, since the only thing that changes in the specifications is that the the TLB needs to be preserved, which is quite simple. Thus this completes the update of the ALE-PE refinement with the TLB included.

Finally we get to the point where we can actually handle and abstract the TLB. The idea is that the `pt-set` function, which updates the page tables will be the only function which can possibly cause the TLB to become desynchronized. Thus, if we add the TLB reset command to the function, we no longer have to worry about the TLB in any higher models of abstraction. The new code of

```
void pt_set (uint64_t vaddr, uint64_t pg)
{
        *(PT + vaddr / PGSIZE * 8) = pg;
        resetTLB();
}
```

Figure 6.2: The Code of `pt-set` for Hardware with TLB

Label	Specification
pt-set	$[vp, pp] \mapsto \left(\begin{array}{l} TLB = \text{true}?,\ dm?,\ \text{PT-ALLOC}(M)?,\ \text{HighPg}(vp)?,\ \text{LowPg}(pp)?, \\ D(\mathtt{PT} + pg * 8) := pp,\ \mathtt{ret}(0) \end{array} \right)$
pt-lookup	$[vp] \mapsto (TLB = \text{true}?,\ dm?,\ \text{PT-ALLOC}(M)?,\ \text{HighPg}(vp)?,\ \mathtt{ret}(D(\mathtt{PT} + vp * 8)))$

Figure 6.3: Specs of the Page Table Driver Ψ^{pt}_{ALE} with TLB modification

`pt-set` is shown in Figure 6.2.

With this addition, both functions of the page table driver code can be certified using the original specification with an addition that the function starts with TLB synchronized, and will end with the TLB synchronized as well. The updated specifications of the functions of the page table driver are given in Figure 6.3. The proofs of these functions do not change much. However, the store in the `pt-set` will trigger the TLB to go off, and then the following call to the `resetTLB` will reset the flag back, which will complete the proof.

Next we show that the original PMAP memory model can be properly refined to the updated ALE memory model. The new relation adds a guarantee that any PMAP state relates only to the ALE state which has a synchronized TLB. Thus $M_{PMAP} \preceq M_{ALE}$ will have an additional clause that states $M_{ALE}.TLB = \text{true}$. To prove that this relation preserves loads and stores requires showing that the TLB can never go out of sync. This relies on the fact that the page tables are marked unallocated in the PMAP model, and thus the store in the PMAP model can never write to any address that will trigger TLB desync. Then, we need to show that the refined PMAP primitives are implemented by the primitives and specs in the ALE model. This is only a minor alteration of proof, since the relation just ensures that the TLB is always synced.

The certification of the address space library, the relation between the AS and PMAP models, and the certification of the kernel remain completely unchanged. However, we need to make some updates to allow the `init` to call the kernel. Since the ALE, PE, and HW models are different, we have to produce new $\mathsf{a}^{kernel-init}_{HW}$, $\mathsf{a}^{kernel-init}_{PE}$, and $\mathsf{a}^{kernel-init}_{ALE}$, which are the same as before, except they

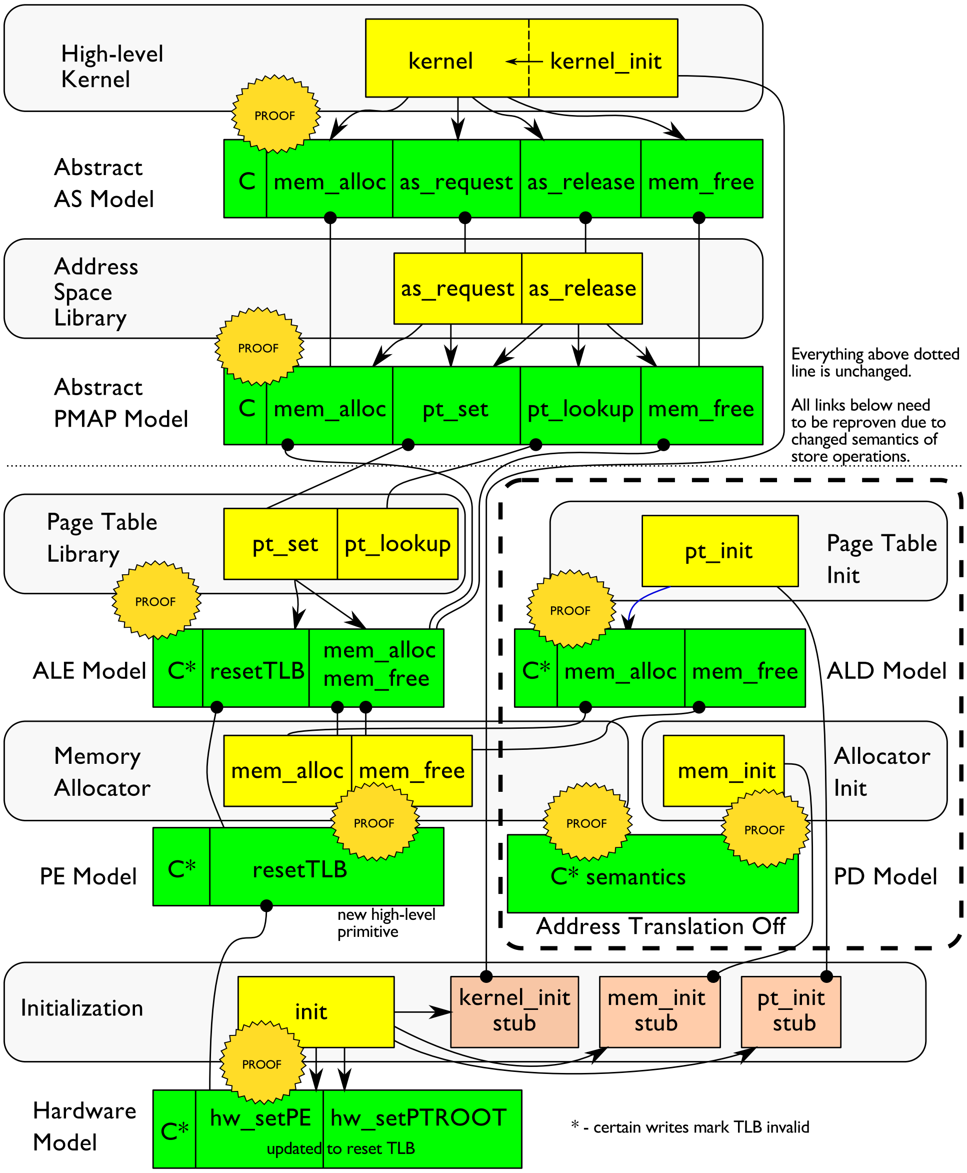


Figure 6.4: Plan of Verification on Hardware that Features TLB

include the precondition that *TLB* = true?. Then, we need to reprove the following weakenings.

$$\mathsf{a}_{ALE}^{kernel-init} \supseteq T_{PMAP-ALE}(\mathsf{a}_{PMAP}^{kernel-init})$$
$$\mathsf{a}_{PE}^{kernel-init} \supseteq T_{ALE-PE}(\mathsf{a}_{ALE}^{kernel-init})$$
$$\mathsf{a}_{HW}^{kernel-init} \supseteq T_{PE-HW}(\mathsf{a}_{PE}^{kernel-init})$$

These can be chained the same way, and thus the `init` function can once again make the upcall to the `kernel-init`.

Thus adding a TLB to the system makes quite a few changes in the verification, as all the low level models are updated. However, all these changes are quite minor: a single line of code, and throwing in a TLB check into the preconditions of many procedures, and updating all weakenings to incorporate the TLB. Pictorially, the updated plan for verification of the virtual memory manager with TLB is shown in Figure 6.4.

However, as the TLB is not going to be needed for the other features we present as possible extensions of our VMM verification, these updates will not be used in the remainder of this chapter.

6.2 Allocatable Page Tables

In the version of the virtual memory manager we have certified, the manager has used fixed locations for the core data structures. The physical memory allocation table was fixed at constant location`PMM`, and the single page table was fixed at location `PT`. While it is common for the memory allocation table to be defined at a specific location, the real VMM implementations always dymaically allocate the page tables. This is necessary to support many of the features that the real virtual memory managers provide. We will show how to extend our VMM implementation to create these tables, which we will do this in three steps. First we will update most of the common layers to use the `PTROOT` register instead of relying on a constant `PT`. Then, we will add a new memory allocator function necessary for dynamic allocation of page tables. Then we will patch up our initialization code to allow for different locations of the page tables.

6.2.1 Pushing PTROOT Register Higher

Our first task is to add the PTROOT register to the PE and ALE models, and update all verifications to use these registers. However, to make this work, we will need to be able to figure out where the page table is currently located. For that, we will require a new hardware primitive that allows us to read the PTROOT register to figure out where the page tables are located. This new primitive is added to the HW model, and has the following signature:

$$\mathcal{L}_{HW}(\texttt{hw-getPTROOT}) := [] \mapsto \texttt{ret}(\texttt{PTROOT})$$

Now we propagate the PTROOT register to the PE and ALE memory models. Along with the register, we propagate the getPTROOT and setPTROOT primitives to the PE and ALE libraries as well.

The updated models of PE and ALE machines are shown in Figure 6.5. The changes to those machines are the addition of PTROOT register (which implies that the specifications of loads, stores, and all previous library functions now preserve this register). Because the location is now flexible, the *trans* AT function is updated to use the dynamic location of the pagetables, rather than relying on the constant. Similarly, the predicates that ensure that the page tables are allocated (*PTalloc*) and that the address translation maps all low addresses to themselves (*dm*) are also updated to use the dynamic location of the page tables. Finally, we have added the primitives for reading and setting the PTROOT register. However, because we have altered the memory models, we must re-certify the memory allocator and the page table driver using new specifications. Because the changes are small, these certifications are similar to the original proofs.

The next step is to reestablish the relations between the memory models. First, we will reestablish the relation between the PE and the HW memory models. The original relation between the models includes the clause that

$$M_{HW}.\texttt{PTROOT} = \texttt{PT}$$

This clause is replaced with one that does not fix the value of PTROOT, but instead makes sure that

PE:

$$\begin{array}{rll} (\textit{Memory System}) & M & ::= (D, \texttt{PTROOT}) \\ (\textit{Memory Data}) & D & ::= \{addr \rightsquigarrow w \mid \mathrm{Low}(addr)\}^* \\ (\textit{Page Table Register}) & \texttt{PTROOT} & ::= w \end{array}$$

Notation	Definition
$M(va)$	$\mathrm{Low}(trans_M(va))?,\ dm?,\ M(trans_M(va))$
$M(va) := w$	$\mathrm{Low}(trans_M(va))?,\ dm?,\ M(trans_M(va)) := w$

where

$$\begin{array}{ll} trans(va) & := \begin{cases} va & \text{if } \mathrm{Low}(va) \\ M(\texttt{PTROOT} + \mathrm{Pg}(va) * 8) * \texttt{NPAGES} + \mathrm{Off}(va) & \text{otherwise} \end{cases} \\ dm & := \forall vp.\, \mathrm{LowPg}(vp) \to \mathrm{Low}(\texttt{PTROOT} + vp * 8) \wedge M(\texttt{PTROOT} + vp * 8) = vp \end{array}$$

ALE:

$$\begin{array}{rl} (\textit{Memory System}) & M ::= (D, A, \texttt{PTROOT}) \\ (\textit{Data Store}) & D\ ::= \{addr \rightsquigarrow w \mid \mathrm{Low}(addr)\}^* \\ (\textit{Page Allocation Table}) & A\ ::= \{pg \rightsquigarrow b \mid \mathrm{LowPg}(pg)\}^* \end{array}$$

Notation	Definition
$M(va)$	$PTalloc?, dm?, M.A(Pg(trans_M(va)))?, M.D(trans_M(va))$
$M(va) := w$	$PTalloc?, dm?, M.A(Pg(trans_M(va)))?, M.D(trans_M(va)) := w$

where

$$\begin{array}{ll} trans_M(va) & := \begin{cases} va & \text{if } \mathrm{Low}(va) \\ M.D(\texttt{PTROOT} + \mathrm{Pg}(va) * 8) * \texttt{NPAGES} + \mathrm{Off}(va) & \text{otherwise} \end{cases} \\ PTalloc & := \forall pg.\, (\mathrm{HighPg}(pg) \vee \mathrm{LowPg}(pg)) \to M.A(Pg(\texttt{PTROOT} + pg * 8)) = true \\ dm & := \forall vp.\, \mathrm{LowPg}(vp) \to \mathrm{Low}(\texttt{PTROOT} + vp * 8) \wedge M.D(\texttt{PTROOT} + vp * 8) = vp \end{array}$$

Figure 6.5: ALE and PE Memory Models with PTROOT register

Label	Specification
mem-alloc	$[] \mapsto \begin{cases} dm?, \text{PMMnotPT}?,\ \texttt{ret}(0) \\ \bigvee_{pg} \begin{pmatrix} dm?,\ \text{PMMnotPT}?,\ \text{LowPg}(pg)?,\ pg \neq 0?, \\ M(\texttt{PMM} + pg * 8) = 0?, M(\texttt{PMM} + pg * 8) := 1, \texttt{ret}(pg) \end{pmatrix} \end{cases}$
mem-free	$[pg] \mapsto \begin{pmatrix} dm?,\ \text{PMMnotPT}?,\ \text{LowPg}(pg)?, \\ M(\texttt{PMM} + pg * 8) = 1?,\ M(\texttt{PMM} + pg * 8) := 0,\ \texttt{ret}(0) \end{pmatrix}$

$$\text{PMMnotPT} \quad := (\texttt{PMM} + \texttt{NPAGES} * 8 < \texttt{PTROOT}) \vee (\texttt{PTROOT} + \texttt{VPAGES} * 8 < \texttt{PMM})$$

Figure 6.6: Updated Specifications of the Memory Manager (Ψ_{PE}^{mem})

the more abstract model has the same value in the register, e.g.

$$M_{PE}.\texttt{PTROOT} = M_{HW}.\texttt{PTROOT}$$

At this point, we can reestablish the proofs for loads and stores between the PE and HW machines, as now this relation once again guarantees that the *trans* works the same way in both models.

Now that we have reestablished the relation between the PE and HW model, we can continue up the abstraction chain. The specifications of the memory allocator require several changes (shown in Figure 6.6) to work on the new PE model. First, the specifications of `mem-alloc` and `mem-free` now guarantee that the PTROOT register is preserved (since we do not update it, our notation guarantees it does not change). Second, the *dm* predicate has been modified to ensure that the area of memory pointed to by `PTROOT` has a direct mapping for low addresses at all times. Third, there is an additional precondition (PMMnotPT) that requires that the area pointed to by `PTROOT` does not overlap with the allocation table (which is still at a specific area in memory). The code of the memory allocation functions themselves do not change. They are just re-proven under new specifications on an updated PE memory model.

Since the ALE and PE memory models have added the `PTROOT` register, the relation between them must also incorporate it. This is done by adding the $M_{ALE}.\texttt{PTROOT} = M_{PE}.\texttt{PTROOT}$ clause to the relation. With this clause we have to reestablish the properties over the loads and stores that the relation $M_{ALE} \preceq M_{PE}$ requires. This required redoing the proof, replacing the references to `PT` constants with reads of `PTROOT` register, and relying on the relation property to guarantee equality.

We must also update the specifications of the stubs of the memory allocation primitives in the

Label	Specification
mem-alloc	$[] \mapsto \begin{cases} PTalloc?,\ dm?,\ \texttt{ret}(0) \\ \bigvee_{pg} \begin{pmatrix} PTalloc?,\ dm?,\ \mathrm{LowPg}(pg)?,\ A(pg) = \mathrm{false}?, \\ A(pg) := \mathrm{true}, \texttt{ret}(pg) \end{pmatrix} \end{cases}$
mem-free	$[pg] \mapsto \begin{pmatrix} PTalloc?,\ dm?,\ A(pg) = \mathrm{true}?, PTdom(pg * \texttt{PGSIZE}) = \mathrm{false}?, \\ A(pg) := \mathrm{false}, \cancel{\{D(l) \mid \mathrm{Pg}(l) = pg\}}, \texttt{ret}(0) \end{pmatrix}$

Figure 6.7: Updated Specs of the Memory Manager Stubs ($\mathcal{L}_{ALE}$)

ALE model. These are given in Figure 6.7, and although the specifications look the same as before, the changes to the meanings of the *dm* and *PTalloc* predicate in the ALE model imply that the new specification now handles the dynamic location of the page tables. Also the new specification preserves the PTROOT register as it has been added by the model, and the new specification does not update it.

Changes in the specifications require us to show that the update ALE stubs of the memory allocator are correctly implemented by the memory allocator code re-certified in the PE model. To do this, we will need to re-establish the following weakenings:

$$T_{ALE-PE}(\mathcal{L}_{ALE}(\text{mem-alloc}) \supseteq \Psi_{PE}^{mem}(\text{mem-alloc})$$
$$T_{ALE-PE}(\mathcal{L}_{ALE}(\text{mem-free}) \supseteq \Psi_{PE}^{mem}(\text{mem-free})$$

The modified relation between the ALE and PE memory models that guarantees that the PTROOT in related states of both models must be equal is adequate to re-work the original proof of these weakenings.

Now that we have the ALE model modified, we now have to deal with the code of the page table driver. The page table driver must now update the page tables that are indicated by the PTROOT register. To do so, it must first read the value of the register to discover the location of the page tables. The updated code of the page table driver is given in Figure 6.8.

These new functions require similar minor modifications as before, namely that PT is replaced with PTROOT. With this modification, we can reprove the page table driver over the ALE memory model: $\mathcal{M}_{ALE}, \mathcal{L}_{ALE} \vdash \mathbb{C}^{pt} : \Psi_{ALE}^{pt}$.

The very last step is to show that the original, unchanged PMAP model can properly link up

```
void pt_set (uint64_t vaddr, uint64_t pg)
{
    int pt = getPTROOT();
    *(pt + vaddr / PGSIZE * 8) = pg;
}

uint64_t pt_lookup (uint64_t vaddr)
{
    int pt = getPTROOT();
    return *(pt + vaddr / PGSIZE * 8);
}
```

Figure 6.8: Updated Code of Page Table Driver

with the modified ALE model and the modified page table driver. For this to happen, we modify the relation $M_{PMAP} \leq M_{ALE}$ and replace all references to `PT` with `PTROOT`, meaning that the abstract page map of the PMAP machine is now related to the dynamic location of the page tables. It also means that the relation model must guarantee that the area specified by the PTROOT register in the ALE model is not marked allocated in the PMAP model. Once this relation is established, we are required to prove load and store consistency once again, as well as to establish that the `pt-set` and `pt-lookup` stubs of the PMAP model are correctly implemented by our updated functions. All these proofs undergo minor changes as the constant is replaced with register lookup, but the proof is similar to the original.

The last piece of our proof broken by this update is the call from init into the kernel. As we have modified the memory models, the stubs of the `kernel-init` function are no longer correct for the altered specifications, and thus must be updated. The changes affect the stubs $\mathsf{a}_{ALE}^{\text{kernel-init}}, \mathsf{a}_{PE}^{\text{kernel-init}}, \mathsf{a}_{HW}^{\text{kernel-init}}$, with the constant `PT` being replaced with the `PTROOT` register lookup. These stubs are then shown to be the weakening of each other under the new memory model relations. Because, the register is set correctly to the values initialized by the `mem-init` and `pt-init` functions, the proof of `init` has to rely on the fact that `PTROOT` is set to `PT` during initialization, but the rest of the proof does not really change.

6.2.2 Bulk Allocation

The next thing we do is to add a new memory allocator that can allocate several consecutive pages. This is needed by our implementation, since we use a consecutive page table that spans multiple pages. We will call this memory allocator `mem-bulkalloc`(n). The code of the bulk allocator

```
uint64_t mem_bulkalloc(uint64_t n)
{
  uint64_t curpage=1;
  uint64_t found=0;
  uint64_t run;
  while(found == 0 && curpage < NPAGES)
  {
        run=0;
        while (run < n && PMM[curpage+run] == 0) {
                run++;
        }
        if (run == n)
                found=1;
        else
                curpage=curpage+1;
  }
  if (found == 1) {
     run=0;
     while (run < n) {
         PMM[curpage+run] = 1;
         run=run+1;
     }
     return curpage;
  }
  else return 0;
}
```

Figure 6.9: Code and Specs of the Multi-Page Allocator

is given in Figure 6.9, and is slightly more complex than the code of the single page allocator. We do not need to create any special deallocation function - the programmer can call the standard `mem-free` to free up one by one all the pages that were allocated.

Just like `mem-alloc`, `mem-bulkalloc` has to be certified twice - once on the PE model and once on the PD model, with both verifications nearly the same. The abstract version of the `mem-bulkalloc` is also pushed in to the ALE and ALD models, so that the page table initialization and page table driver can make use of the allocation. The specifications of the bulk allocator over the PE machine, as well as abstract specifications of the ALE model's bulk allocator primitive are given in Figure 6.10.

To allow the abstract layers to call the bulk allocator, we will need to prove that the abstract specifications are correctly implemented by the concrete ones. For this we need to show the following weakening:

$$T_{ALE-PE}(\mathcal{L}_{ALE}(\text{mem-bulkalloc})) \supseteq \Psi^{mem}_{PE}(\text{mem-bulkalloc})$$

$$T_{ALD-PD}(\mathcal{L}_{ALD}(\text{mem-bulkalloc})) \supseteq \Psi^{mem}_{PD}(\text{mem-bulkalloc})$$

As we will not need to have the bulk allocator at the PMAP and AS memory models, we do not have to define the stubs at those levels.

$$\Psi_{PE}^{mem}(\text{mem-bulkalloc}) := [n] \mapsto \begin{cases} dm?, \texttt{ret}(0) \\ \bigvee_{pg} \begin{pmatrix} dm?, \mathrm{LowPg}(pg)?, \ldots, \mathrm{LowPg}(pg+n-1)?, pg \neq 0?, \\ M(\texttt{PMM} + pg * 8) = 0?, \ldots, M(\texttt{PMM} + (pg+n-1) * 8) = 0?, \\ M(\texttt{PMM} + pg * 8) := 1, \ldots, M(\texttt{PMM} + (pg+n-1) * 8 := 1, \\ \texttt{ret}(pg) \end{pmatrix} \end{cases}$$

$$\mathcal{L}_{ALE}(\text{mem-bulkalloc}) := [n] \mapsto \begin{cases} \texttt{ret}(0) \\ \left(\bigvee_{pg} \begin{pmatrix} \mathrm{LowPg}(pg)?, \ldots, \mathrm{LowPg}(pg+n-1)?, pg \neq 0? \\ A(pg) = \mathrm{false}?, \ldots, A(pg+n-1) = \mathrm{false}?, \\ A(pg) := \mathrm{true}, \ldots, A(pg+n-1) := \mathrm{true}, \\ \cancel{\{D(l) | A(\mathrm{Pg}(l)) = \mathrm{false}\}}, \texttt{ret}(pg) \end{pmatrix} \right. \end{cases}$$

The version for PD and ALD memory models are similar.

Figure 6.10: Specs for `mem-bulkalloc`

6.2.3 Dynamic Initialization of Page Tables

At this point, we have updated our page table driver functions to be able to work with any values written to the PTROOT register. However, our initialization currently uses a fixed location for these tables. In this section, we will update the initialization function to allocate the page table area rather than rely on a predetermined, preallocated location.

To accomplish this, we need to update the code of the page table initialization. The new code of these functions is given in Figure 6.11, which has the several changes from the original code. The functionality of the `pt-init` function is moved into the new `pt-new` functions. This function still initializes a page table to be identity mapped for low addresses, and completely unmapped for high addresses. However, it does so not on a fixed location pointed to by the PT, but uses `mem-bulkalloc` to allocate a new chunk of space for the page table. Unlike the original function which returned zero, this function returns the location of the page table, and since this function may not succeed, it may return 0 on failure.

Since the function is completely different, we have to re-certify it. Since its action is different, it is given a new specification (listed in Figure 6.12). And although the proof is a bit more complex due to allocation of memory, and possible failures, it is not incredibly difficult to re-certify it in the ALD model. What is a bit more annoying is that we must also define a_{HW}^{pt-new} stub, and show that it is a weakened version of the refined specification of `pt-new`, e.g.

```
uint64_t pt_new() {
  uint64_t newpt;
  int i=0;
  // we assume VPAGES is a multiple of 512
  newpt = mem-bulkalloc(VPAGES/(PGSIZE/8))*PGSIZE;
  if (!newpt) return(0); //in case of failure return 0

  // initialize the page tables
  while(i<NPAGES) {
    *(newpt + i * 8) = i;
        // page 0 is effectively unavailable
    i++;
  }
  while(i<VPAGES) {
    *(newpt + i * 8) = 0;
    i++;
  }
  return(newpt);
}

void init()
{
        mem_init();
        uint64_t pt = pt_new();
        if (!pt) {while(1)};
        setPTROOT(pt);
        setPE(1);
        kernel_init(); //never returns
}
```

Figure 6.11: Code of Page Table Driver and Initialization

$$[] \mapsto \begin{cases} \left(\; \texttt{ret}(0) \; \right) \\ \bigvee_{ptpg} \left(\begin{array}{l} (M.A(ptpg) = \text{false})?, \ldots, (M.A(ptpg + \texttt{VPAGES}/(\texttt{PGSIZE}/8)) = \text{false})?, \\ (M.A(ptpg) := \text{true})?, \ldots, (M.A(ptpg + \texttt{VPAGES}/(\texttt{PGSIZE}/8)) := \text{true})?, \\ (M.D(ptpg * \texttt{PGSIZE} + pg * 8) := pg \mid \text{LowPg}(pg)), \\ (M.D(ptpg * \texttt{PGSIZE} + pg * 8) := 0 \mid \text{HighPg}(pg)), \\ \texttt{ret}(ptpg * \texttt{PGSIZE}) \end{array} \right) \end{cases}$$

Figure 6.12: Specification of Page Table Init ($\mathcal{L}_{PD}$(pt-new))

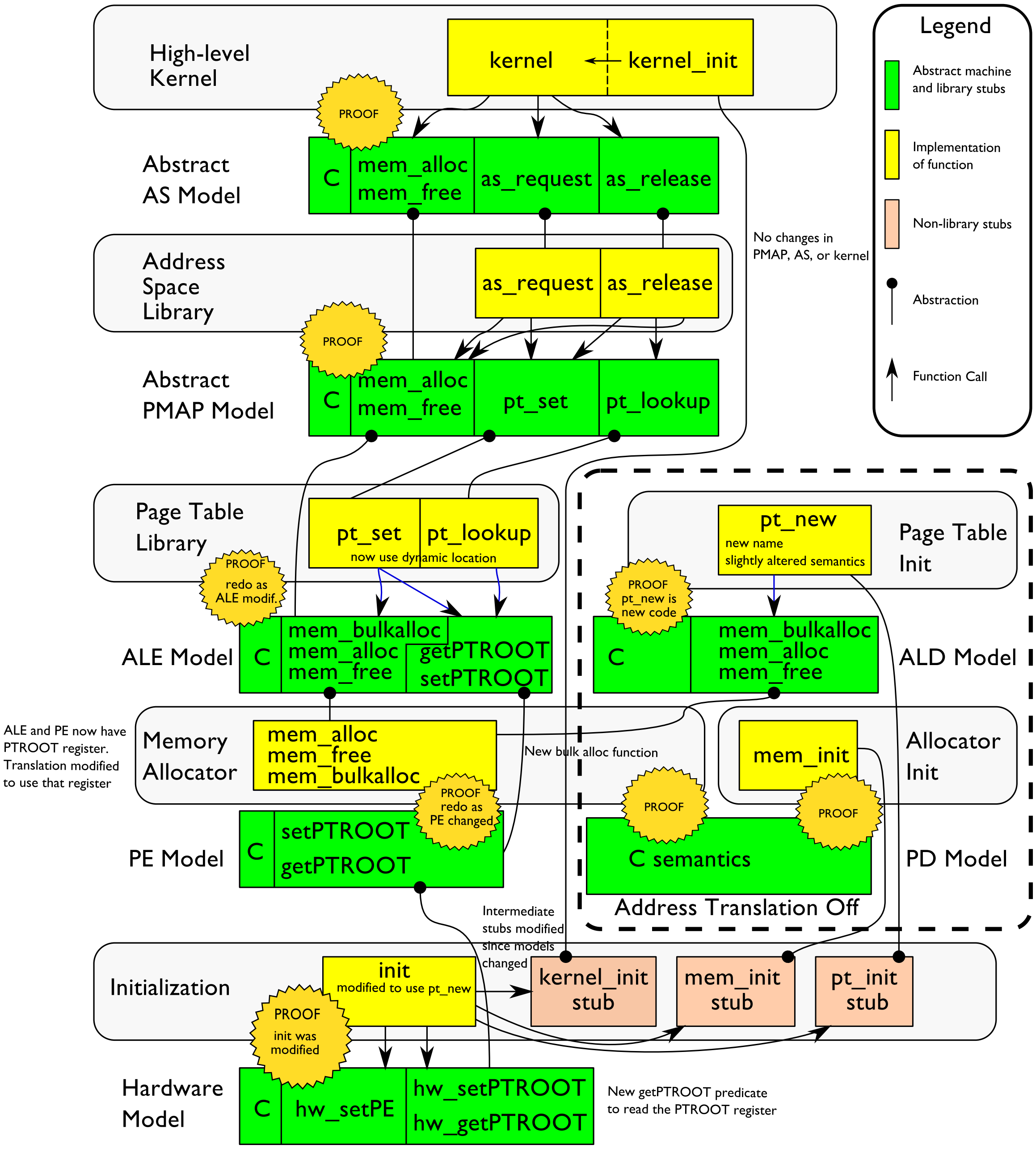


Figure 6.13: Certification Plan for VMM with Dynamically Allocated Pagetables

$\mathsf{a}_{HW}^{pt-new} \supseteq T_{PD-HW}(T_{ALD-PD}(\Psi_{ALD}^{ptinit}(pt-new)))$, so that the initialization can safely call `pt-new`.

The initialization function, `init`, is updated to work with the updated page table initialization function. The new code saves the return value of `pt-new`, and checks it for failure. Since failures are not supposed to happen on initialization, we safely handle errors by an infinite loop. However, if the allocation succeeds, then we set the value of the `PTROOT` register to the returned location, which allows us to proceed with the fully initialized page table in a dynamic location.

Because of the changes, the init function has to be completely re-certified, although its specification does not change. Our new `kernel-init` stub is already set up to handle dynamic locations of the page tables, and thus the proof encounters no problems.

6.2.4 Recapping Dynamically Allocated Page Tables

All of the changes are summarized in Figure 6.13. The set of updates needed to support the dynamic allocation of page tables is quite extensive, but the changes are not very complex, and many of the proofs remain quite similar. The most difficult parts are the fact that we must dynamically separate the page tables from the allocation tables (which was trivial when they were in specific locations), and that we had to make a fairly significant change in the initialization functions, which required us to redo all the weakening and certification proofs about initialization. However, these updates make our system much more realistic, as it no longer requires a fixed position of the page tables.

6.3 Multiple Address Spaces

Now that we have support for the dynamic allocation of page tables, we can add one of the most important features of virtual address spaces - the ability to have multiple address spaces, and to switch between them trivially. We will proceed to define this system in stages. First, we will add functions to create, delete, and switch page tables in the ALE model. Then we will update the PMAP model to include multiple pagemaps, creating primitives that are analogous to the functions for the page tables. Then, we will move the concept of multiple pagemaps into the AS model, where they will become multiple address spaces. Finally, we will have to rework the refined specifications of `kernel-init` to work with the updated AS and PMAP memory models.

```
pt_delete(uint64_t pt) {
        int i=0;
        while(i<VPAGES/PGSIZE*8) {
                mem-free(pt/PGSIZE + i);
                i++;
        }
        return(0);
}
```

Figure 6.14: Code of the `pt-delete` function

6.3.1 New Page Table Functions

The first step in creating multiple address spaces is the ability to create and delete page tables. In Section 6.2.3, we have already created a `pt-new` function that allocates page tables on demand. Here, we will make use of this function to create page tables, not just during initialization, but at any point during execution of the kernel. This means that we will be able to have several page tables to exist at the same time. To do this, we will take the pt-new function from the ALD machine, and also verify it on the ALE machine - meaning that it becomes usable even when AT is on.

Then, we create a new `pt-delete` function, whose code is listed in Figure 6.14. The function is a simple deallocator of the exact number of pages that a page table requires. However, the simple function hides a very important fact that our specification will ensure that the deallocator can deallocate only page tables, and only those not currently active, as deallocation of an active page table will cause all memory operations to fail.

Now that we have the code of the two new page table functions, we can design the specifications for them, which are listed in Figure 6.15. The specifications of `pt-new` are very similar to its specifications over the ALD machine that we have encountered in the previous section. The only difference between the specification of `pt-new` in the ALE model and the ALD model is the addition of the direct map precondition, which makes sure that the active translation ensures that all low addresses use identity mapping. This allows the same code to function even when AT is on. The specification of the `pt-delete` function is similar to the specification of the `mem-free` primitive, except it works for several pages (the exact number of which are determined by the size of the page table defined by hardware).

To switch between the active pagetables, we already have a `setPTROOT` primitive, to which we can pass the returned address of the page table as a parameter.

$$\mathcal{L}_{ALE}(\text{pt-new}) := [\,] \mapsto \left\{ \begin{array}{l} \left(\; dm?, \texttt{ret}(0) \; \right) \\ \bigvee_{ptpg} \left(\begin{array}{l} dm?, \\ (M.A(ptpg) = \text{false})?, \ldots, (M.A(ptpg + \texttt{VPAGES}/(\texttt{PGSIZE}/8)) = \text{false})?, \\ (M.A(ptpg) := \text{true})?, \ldots, (M.A(ptpg + \texttt{VPAGES}/(\texttt{PGSIZE}/8)) := \text{true})?, \\ (M.D(ptpg * \texttt{PGSIZE} + pg * 8) := pg \mid \text{LowPg}(pg)), \\ (M.D(ptpg * \texttt{PGSIZE} + pg * 8) := 0 \mid \text{HighPg}(pg)), \\ \texttt{ret}(ptpg * \texttt{PGSIZE}) \end{array} \right) \end{array} \right.$$

$$\mathcal{L}_{ALE}(\text{pt-delete}) := [ptpg] \mapsto \left(\begin{array}{l} dm?, \\ M.A(ptpg) = \text{true}?, \ldots, (M.A(ptpg + \texttt{VPAGES}/(\texttt{PGSIZE}/8) - 1) = \text{true}?), \\ M.A(ptpg) := \text{false}?, \ldots, (M.A(ptpg + \texttt{VPAGES}/(\texttt{PGSIZE}/8) - 1) = \text{false}?), \\ \texttt{ret}(0), \end{array} \right)$$

where

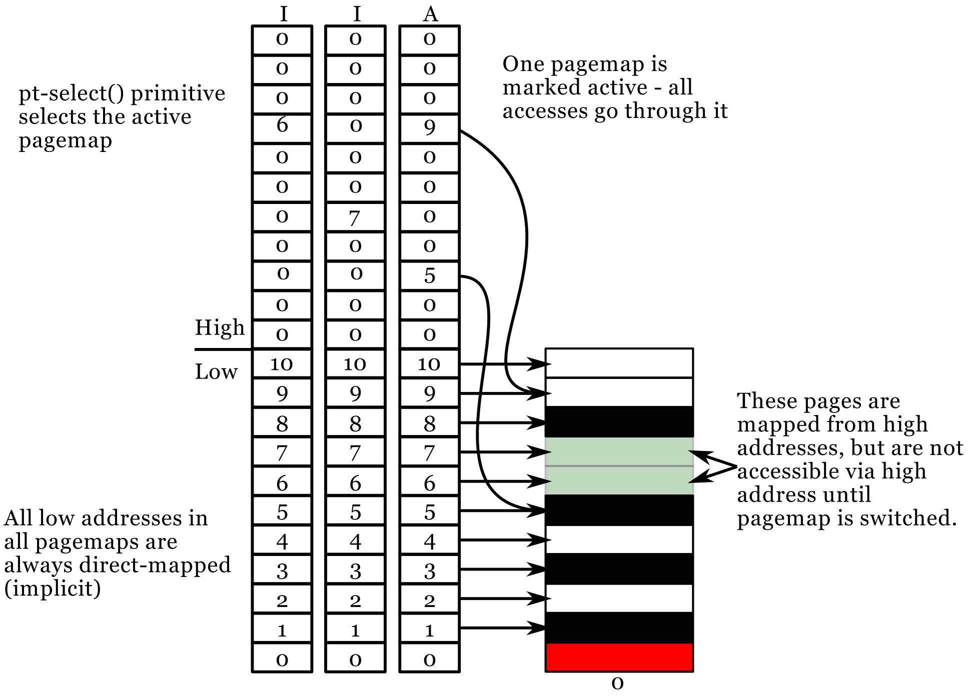
$$dm := \forall pg.\, \text{LowPg}(pg) M.D(\texttt{PTROOT} + pg * 8) = pg$$

Figure 6.15: Specifications of New Page Table Functions (Ψ^{pt}_{ALE})

6.3.2 Multiple Page Map Memory Model

The most important change we make to support this system is the new PMAP memory model that is capable of keeping track of multiple page maps, which we will call MPMAP. This is accomplished by modifying the original PMAP machine to have multiple page maps, one of which is marked active. The active page map is used for translations, and is also the target of the `pt-set` and `pt-lookup` stubs. The model features new stubs, `pt-new`, `pt-delete`, `pt-select`, `pt-current`, that create and delete page maps, select the active page map, and return the index of the active page map, respectively. The visual diagram of the MPMAP model is in Figure 6.16, and its formal definition is in Figure 6.17.

The challenge of this memory model is to reconnect it with the original ALE model. The relation between the MPMAP model and the ALE model (Figure 6.18) now has to properly account for several page maps at once. The relation consists of six clauses. The first clause ensures that the data regions of memory marked allocated in the MPMAP model is equal to that in the ALE model. The second clause ensures that the allocation information for pages that are not used for page tables is equal in both tables. The third clause forces all pages used for page tables to be allocated in the ALE model, and unallocated in the MPMAP model, thus hiding them from the accesses by software written for MPMAP model. The fourth and fifth clauses make sure that the mappings of all page


pt-select() primitive selects the active pagemap
One pagemap is marked active - all accesses go through it
High
Low
These pages are mapped from high addresses, but are not accessible via high address until pagemap is switched.
All low addresses in all pagemaps are always direct-mapped (implicit)
0

I	I	A
0	0	0
0	0	0
0	0	0
6	0	9
0	0	0
0	0	0
0	7	0
0	0	0
0	0	5
0	0	0
0	0	0
10	10	10
9	9	9
8	8	8
7	7	7
6	6	6
5	5	5
4	4	4
3	3	3
2	2	2
1	1	1
0	0	0

Figure 6.16: Multiple Page Map Memory Model Diagram

maps correspond to the infomation in the page tables and also check that the low addresses are direct mapped. Finally, the sixth clause ensures that the active page map corresponds to the page table pointed to by the PTROOT register. Thus, the relation between MPMAP and ALE models is similar to the relation between PMAP and ALE models, except generalized to support multiple page maps at the same time.

Using this relation, we have to show that the primitives of the MPMAP model match up correctly with the ALE model. This is also the case for the previously shown `pt-set` and `pt-lookup` functions, which now function differently due to the modified structure of the PMAP memory model. We also have to reprove the weakening for `mem-alloc` and `mem-free` even though they do affect the page tables. Since the translation has been modified, the proofs have to be reworked, and this time they have to additionally guarantee that they maintain all the pagemaps, and rely on a different predicate to ensure that low addresses are direct mapped. All the weakening that we have to

$$\begin{array}{rll}
(\textit{Global Storage System}) & M & ::= (D, A, PMS, pmapid) \\
(\textit{Allocatable Memory}) & D & ::= \{addr \rightsquigarrow w \mid \mathrm{Low}(addr)\}^* \\
(\textit{Page Allocation Table}) & A & ::= \{pg \rightsquigarrow b \mid \mathrm{LowPg}(pg)\}^* \\
(\textit{Page Maps}) & PMS & ::= \{pmapid \rightsquigarrow PM\}^* \\
(\textit{Page Map}) & PM & ::= \{pg \rightsquigarrow pg' \mid \mathrm{HighPg}(pg)\}^* \\
(\textit{Page Map ID}) & pmapid & ::= w
\end{array}$$

Notation	`Definition`
$M(va)$	$\texttt{let}\ \ pa := trans(va)\ \ \texttt{in}\ \ (M.A(Pg(pa))?,\ D(pa))$
$M(va) := w$	$\texttt{let}\ \ pa := trans(va)\ \ \texttt{in}\ \ (M.A(Pg(pa))?,\ D(pa) := w)$

where

$$trans(va) \quad := \begin{cases} PMS(pmapid)(Pg(va)) * \texttt{PGSIZE} + \mathrm{Off}(va) & \text{if High}(va) \\ va & \text{otherwise} \end{cases}$$

Label	Specification
mem-alloc	$[] \mapsto \begin{cases} \texttt{ret}(0) \\ \left(\bigvee_{pg} \begin{pmatrix} \mathrm{LowPg}(pg)?,\ A(pg) = \text{false}?, \\ A(pg) := \text{true}, \cancel{\{D(l) \mid A(\mathrm{Pg}(l)) = \text{false}\}}, \texttt{ret}(pg) \end{pmatrix} \right) \end{cases}$
mem-free	$[pg] \mapsto A(pg) = \text{true}?, A(pg) := \text{false}, \cancel{\{D(l) \mid A(\mathrm{Pg}(l)) = \text{false}\}}, \texttt{ret}(0)$
pt-set	$[vp, pp] \mapsto \begin{pmatrix} \mathrm{HighPg}(vp)?, \mathrm{LowPg}(pp), PMS(pmapid)(vp) := pp, \\ \cancel{\{D(\texttt{l}) \mid A(Pg(l)) = \text{false}\}}, \texttt{ret}(0) \end{pmatrix}$
pt-lookup	$[vp] \mapsto (\mathrm{HighPg}(vp)?, \texttt{ret}(PMS(pmapid)(vp)))$
pt-new	$[] \mapsto \begin{cases} \texttt{ret}(0) \\ \bigvee_{pt} \begin{pmatrix} pt \notin \texttt{dom}(PMS)?, pt \neq 0?, \\ PMS(pt) := \{pg \mapsto 0 \mid \mathrm{HighPg}(pg)\}^*, \texttt{ret}(pt) \end{pmatrix} \end{cases}$
pt-delete	$[pt] \mapsto (pt \in \texttt{dom}(PMS)?,\ PMS \setminus pt,\ \texttt{ret}(0))$
pt-select	$[pt] \mapsto (pt \in \texttt{dom}(PMS),\ pmapid := pt)$
pt-current	$[] \mapsto (\texttt{ret}(pmapid))$

Figure 6.17: Multiple Page Map Memory Model

$$
\begin{aligned}
&\forall l.\, \mathrm{Low}(l) \rightarrow M_{MPMAP}.A(Pg(l)) = true \rightarrow M_{MPMAP}.D(l) = M_{ALE}.D(l) \\
&\forall pmapid, l.\, pmapid \in \mathtt{dom}(M_{MPMAP}.PMS) \rightarrow \mathrm{Low}(l) \rightarrow \neg\mathrm{PTdom}(pmapid, l) \rightarrow \\
&\qquad M_{MPMAP}.A(Pg(l)) = M_{ALE}.A(Pg(l)) \\
&\forall pmapid, l.\, pmapid \in \mathtt{dom}(M_{MPMAP}.PMS) \rightarrow \mathrm{Low}(l) \rightarrow \mathrm{PTdom}(pmapid, l) \rightarrow \\
&\qquad M_{MPMAP}.A(Pg(l)) = false \wedge M_{ALE}.A(Pg(l)) = true \\
&\forall pmapid, vpg.\, pmapid \in \mathtt{dom}(M_{MPMAP}.PMS) \rightarrow \mathrm{HighPg}(vpg) \rightarrow \\
&\qquad M_{MPMAP}.PMS(pmapid)(vpg) = M_{ALE}.D(pmapid + vpg * 8) \\
&\forall pmapid, vpg.\, pmapid \in \mathtt{dom}(M_{MPMAP}.PMS) \rightarrow \mathrm{LowPg}(vpg) \rightarrow \\
&\qquad M_{ALE}.D(pmapid + vpg * 8) = vpg \\
&M_{MPMAP}.pmapid = M_{ALE}.\mathtt{PTROOT} \wedge M_{PMAP}.pmapid \in \mathtt{dom}(M_{MPMAP}.PMS)
\end{aligned}
$$

where

$$\mathrm{PTdom}(pmapid, l) := pmapid \leq l < pmapid + \mathtt{VPAGES}$$

Figure 6.18: Relation Between MPMAP and ALE Models

re-prove are as follows:

$$
\begin{aligned}
&T_{MPMAP-ALE}(\mathcal{L}_{MPMAP}(\mathtt{mem\text{-}alloc})) \supseteq \mathcal{L}_{ALE}(\mathtt{mem\text{-}alloc}) \\
&T_{MPMAP-ALE}(\mathcal{L}_{MPMAP}(\mathtt{mem\text{-}free})) \supseteq \mathcal{L}_{ALE}(\mathtt{mem\text{-}free}) \\
&T_{MPMAP-ALE}(\mathcal{L}_{MPMAP}(\mathtt{pt\text{-}set})) \supseteq \Psi^{pt}_{ALE}(\mathtt{pt\text{-}set}) \\
&T_{MPMAP-ALE}(\mathcal{L}_{MPMAP}(\mathtt{pt\text{-}lookup})) \supseteq \Psi^{pt}_{ALE}(\mathtt{pt\text{-}lookup}) \\
&T_{MPMAP-ALE}(\mathcal{L}_{MPMAP}(\mathtt{pt\text{-}select})) \supseteq \mathcal{L}_{ALE}(\mathtt{hw\text{-}setPTROOT}) \\
&T_{MPMAP-ALE}(\mathcal{L}_{MPMAP}(\mathtt{pt\text{-}current})) \supseteq \mathcal{L}_{ALE}(\mathtt{hw\text{-}getPTROOT})
\end{aligned}
$$

The last two weakening shows that the `pt-select` function is completely implemented by the `setPTROOT` primitive, and that calling `pt-current` function is actually calling `hw-getPTROOT` primitive. To handle this in our framework, we simply show that `pt-select` label is defined to be `hw-setPTROOT` label.

These weakenings show that there is a valid refinement from the MPMAP model into the ALE model.

6.3.3 Multiple Address Space Model

The final target of this improvement to our VMM system is the multiple address space model of memory (MAS). The MAS model differs from the AS model in that there are now several high

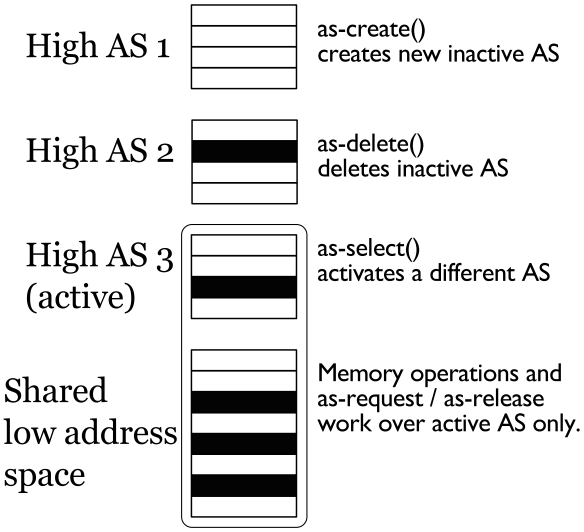


Figure 6.19: Diagram of the MAS Memory Model

areas of memory, as can be seen in Figure 6.19. One of these address spaces is marked as active, and all of the original AS model functionality, such as loads and stores, as well as allocations work over the active address space. To deal with several address spaces, the MAS model includes several new primitives. First, the `as-create` and `as-delete` will create and delete the address spaces. The deletion can not work on the active address space. To change the active address space, there is a new `as-select` primitive. These adjustments result in the mathematical definition of address spaces that can be seen in Figure 6.20.

The formal definition of the MAS model clearly shows the several address spaces present as a part of the state. Each of these spaces is marked with an address space id (*asid*), and the state of the memory model includes and id (*Hid*) as a part of its state to mark the currently active space. All the original operations and primitives operate over the current high address space, and the new operations are designed to manipulate the set of high address spaces. The low address space is always present and does not change, no matter which of the high address spaces is selected.

From this definition is is clear that MAS is a more general case of the AS model, and thus any code that works over the AS memory model will also work over the MAS model as well. This means that our AS kernel can be refined to work over the MAS machine, however, we will not show this fact, and instead focus on connecting the MAS model to the MPMAP model to verify our new more powerful VMM.

To relate the MPMAP and MAS, we connect each address space to its unique page table. This

$$\begin{array}{rll}
(\textit{Memory System}) & M & ::= (LM, HM, Hid) \\
(\textit{Low Memory Area}) & LM & ::= (LD, LA) \\
(\textit{Low Address Data Store}) & LD & ::= \{addr \leadsto w \mid \text{Low}(addr)\}^{*} \\
(\textit{Low Page Allocation}) & LA & ::= \{pg \leadsto \texttt{b} \mid \text{LowPg}(pg)\}^{*} \\
(\textit{High Memory Area}) & HM & ::= \{asid \leadsto (HD, HA)\}^{*} \\
(\textit{High Address Data Store}) & HD & ::= \{addr \leadsto w \mid \text{High}(addr)\}^{*} \\
(\textit{Page Allocation}) & HA & ::= \{pg \leadsto \texttt{b} \mid \text{HighPg}(pg)\}^{*} \\
(\textit{Words}) & addr, pg, w & ::= \textit{(64-bit values)}
\end{array}$$

Notation	`Definition`
$M(va)$	$\begin{cases} (M.LM.LA(Pg(va))?,\ M.LM.LD(va)) & \text{if Low}(va) \\ (M.HM(Hid).HA(Pg(va))?,\ M.HM(Hid).HD(va)) & \text{if High}(va) \end{cases}$
$M(va) := w$	$\begin{cases} (M.LM.LA(Pg(va))?,\ M.LM.LD(va) := w) & \text{if Low}(va) \\ (M.HM(Hid).HA(Pg(va))?,\ M.HM(Hid).HD(va) := w) & \text{if High}(va) \end{cases}$

Label	Specification
`mem-alloc`	$[] \mapsto \begin{cases} \texttt{ret}(0) \\ \bigvee_{pg} (\text{LowPg}(pg)?, (pg \neq 0)?, LA(pg) = \text{false}?, LA(pg) := \text{true}, \texttt{ret}(pg)) \end{cases}$
`mem-free`	$[pg] \mapsto (\text{LowPg}(pg)?,\ LA(pg) = \textit{true}?,\ LA(pg) := \text{false}, \texttt{ret}(0))$
`as-reserve`	$[vpg] \mapsto \begin{cases} (\text{HighPg}(vpg)?, \texttt{ret}(0)) \\ \begin{pmatrix} \text{HighPg}(vpg)?, HM(Hid).HA(vpg) = \text{false}?, \\ HM(Hid).HA(vpg) := \text{true}, \texttt{ret}(vpg) \end{pmatrix} \end{cases}$
`as-release`	$[vpg] \mapsto \begin{pmatrix} \text{HighPg}(vpg)?,\ HM(Hid).HA(vpg) = \text{true}?, \\ HM(Hid).HA(vpg) := \text{false}, \texttt{ret}(0) \end{pmatrix}$
`as-create`	$[] \leadsto \begin{cases} \texttt{ret}(0) \\ \bigvee_{asid} \begin{pmatrix} asid \notin \texttt{dom}(HM)?, \\ M.HM(asid) := (\{addr \leadsto ?\}, \{pg \leadsto \text{false}\}), \\ \texttt{ret}(asid) \end{pmatrix} \end{cases}$
`as-delete`	$[asid]] \mapsto ((asid \in \texttt{dom}(HM))?,\ asid \neq Hid?,\ HM \setminus asid)$
`as-select`	$[asid] \mapsto ((asid \in \texttt{dom}(HM))?,\ Hid := asid,\ \texttt{ret}(asid))$
`as-current`	$[asid] \mapsto (\texttt{ret}(Hid))$

Figure 6.20: Multiple Address Space (MAS) Model and Its Library

$$
\begin{aligned}
&\forall l.\, \mathrm{Low}(l) \to M_{AS}.LM.LA(Pg(l)) = \mathrm{true} \to M_{AS}.LM.LD(l) = M_{PMAP}.D(l)\\
&\forall pg.\, \mathrm{LowPg}(pg) \to M_{AS}.LM.LA(pg) = \mathrm{true} \to M_{PMAP}.A(pg) = \mathrm{true}\\
&\forall asid, l.\, \mathrm{High}(l) \to M_{AS}.HM(asid).HA(Pg(l)) = \mathrm{true} \to\\
&\qquad M_{AS}.HM(asid).HD(l) = M_{PMAP}.D(trans(M_{PMAP}, asid, l))\\
&\forall asid, pg.\, \mathrm{HighPg}(pg) \to M_{AS}.HM(asid).HA(pg) = \mathrm{true} \to\\
&\qquad (\exists ppg.\, M_{PMAP}.PMS(asid)(pg) = ppg \land \mathtt{LowPg}(ppg) \land ppg \neq 0 \land M_{PMAP}.A(ppg) = \mathrm{true})\\
&\forall asid, pg.\, \mathrm{HighPg}(pg) \to M_{AS}.HM(asid).HA(pg) = false \to M_{PMAP}.PMS(asid)(pg) = 0\\
&\forall asid, l, l'.\, \mathrm{High}(l) \to \mathrm{Low}(l') \to M_{AS}.HM(asid).HA(l) = \mathrm{true} \to M_{AS}.LM.LA(l') = \mathrm{true} \to\\
&\qquad trans(M_{PMAP}, asid, l) \neq l'\\
&\forall asid, l, asid', l'.\, l \neq l' \to \mathrm{High}(l) \to \mathrm{High}(l') \to M_{AS}.HM(asid).HA(l) = \mathrm{true} \to\\
&\qquad M_{AS}.HM(asid').HA(l') = \mathrm{true} \to trans(M_{PMAP}, asid, l) \neq trans(M_{PMAP}, asid', l')
\end{aligned}
$$

where

$$trans(M_{PMAP}, asid, l) = ppg + \mathrm{Off}(l) \text{ if} M_{PMAP}.PM(asid)(Pg(l)) = ppg \land ppg \neq 0$$

Figure 6.21: Relation between MAS and MPMAP models of memory

is done by ensuring that the address space id is exactly the same as the page table id. The complete relation is given in Figure 6.21. The relation differs from the original AS-PMAP relation by the fact that it has to relate all address spaces to the page tables. This guarantees that all address spaces have a corresponding valid page table, although it does not guarantee that all page tables define an address space. The last two clauses of the relation are the updated non-interference guarantees. They make sure that no two pages in any address space can ever map to the same page.

Using this new relation, we can now show the fact that the MAS stubs are correctly implemented by the implementations and stubs of the MPMAP model. For this, we have to show that

$$
\begin{aligned}
&T_{MAS-MPMAP}(\mathcal{L}_{MAS}(\text{mem-alloc})) \supseteq \mathcal{L}_{MPMAP}(\text{mem-alloc})\\
&T_{MAS-MPMAP}(\mathcal{L}_{MAS}(\text{mem-free})) \supseteq \mathcal{L}_{MPMAP}(\text{mem-free})\\
&T_{MAS-MPMAP}(\mathcal{L}_{MAS}(\text{as-request})) \supseteq \Psi^{as}_{MPMAP}(\text{as-request})\\
&T_{MAS-MPMAP}(\mathcal{L}_{MAS}(\text{as-release})) \supseteq \Psi^{as}_{MPMAP}(\text{as-release})\\
&T_{MAS-MPMAP}(\mathcal{L}_{MAS}(\text{as-create})) \supseteq \mathcal{L}_{MPMAP}(\text{pt-new})\\
&T_{MAS-MPMAP}(\mathcal{L}_{MAS}(\text{as-delete})) \supseteq \mathcal{L}_{MPMAP}(\text{pt-delete})\\
&T_{MAS-MPMAP}(\mathcal{L}_{MAS}(\text{as-select})) \supseteq \mathcal{L}_{MPMAP}(\text{pt-select})\\
&T_{MAS-MPMAP}(\mathcal{L}_{MAS}(\text{as-current})) \supseteq \mathcal{L}_{MPMAP}(\text{pt-current})
\end{aligned}
$$

Given that the relation between the MAS and MPMAP memory models is quite lengthy, even if simple, proving the weakenings above is tedious. It may be possible to lessen the work by defining lemmas that show that specific relations are always preserved unless certain operations are performed by the code. Such lemmas would then be reused in each of the weakenings, reducing the proof size. However, such techniques are not required to prove the above. Once the weakening are proven, it would guarantee that any modules certified in the MAS memory model can be refined to the MPMAP model, and therefore refined to the original ALE model.

6.3.4 Updated Kernel and Initialization

Since we have replaced PMAP and AS models with MPMAP and MAS models, we now have to rework the upcall to `kernel-init` from the initialization code. Because we have replaced AS model with the more powerful MAS model, we will assume a new kernel that is certified over the MAS model, which can make use of the features present. This new kernel will have the following certification: $\mathcal{M}_{MAS}, \mathcal{L}_{MAS} \vdash \mathbb{C}^{kernel} : \Psi^{kernel}_{MAS}$. The specification of `kernel-init` will remain the same ($[] \mapsto loop$), although it is now defined over the MAS memory model.

To certify the upcall, we need to show that the translation of the specification of the `kernel-init` is something that `init` can call. For this we define an intermediate specification in the MPMAP model.

$$\mathsf{a}^{\text{kernel-init}}_{MPMAP} := [] \mapsto ((\forall ptid, pg.\, ptid \in \mathtt{dom}(PMS) \to \text{HighPg}(pg) \to PMS(ptid)pg = 0)? \circ loop)$$

What is interesting is that we do not have to modify any of the specification of the `kernel-init` in that ALE, PE, and HW memory models. We just need to make sure that they are compatible with the new specifications that we have just defined. For this, we just need to prove the following properties:

$$\mathsf{a}^{\text{kernel-init}}_{MPMAP} \supseteq T_{AS-PMAP}(\Psi^{kernel}_{MAS}(\text{kernel-init}))$$
$$\mathsf{a}^{\text{kernel-init}}_{ALE} \supseteq T_{MPMAP-ALE}(\mathsf{a}^{\text{kernel-init}}_{MPMAP})$$

These two properties ensure that when we follow the final linking procedure, the new MAS kernel can properly link with the certified `init` function, thereby completing the certification of our much more powerful VMM.

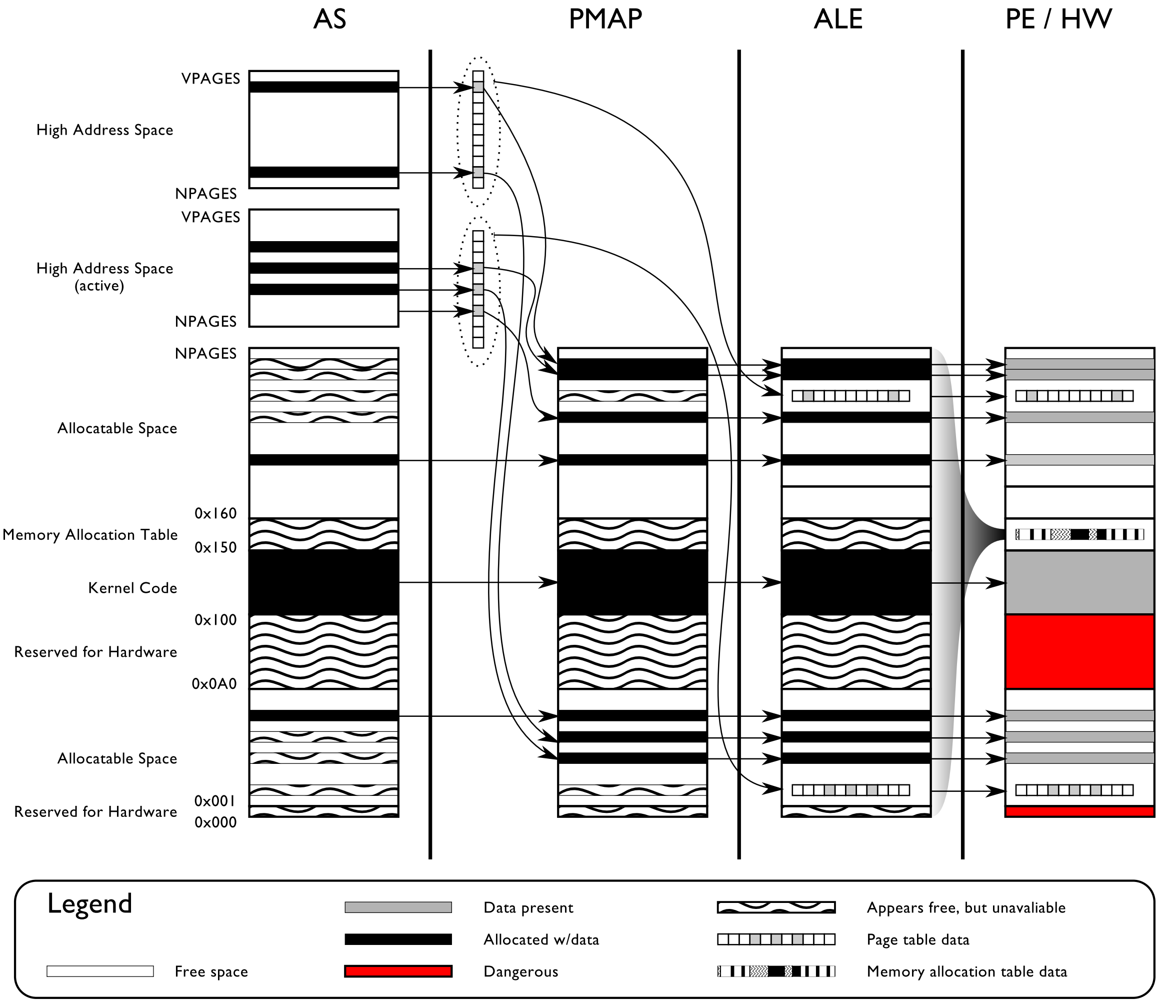


Figure 6.22: Relation between Memory Models of Multi-AS VMM

6.3.5 Recap of the Relation between Memory Models of Multi-AS VMM

Instead of showing the new plan of the certification of the memory, which is now significantly bigger, but not more descriptive, we instead wanted to show the new diagram of the relation between the memory models used for certification of the new virtual memory manager that can now handle multiple memory models. The diagram is given in Figure 6.22.

The diagram looks similar to the diagram of relations between memory models for our simple VMM. However, there are several interesting and important differences that make this a much more powerful representation. First, the AS model now has several address spaces shown. The diagram shows how each address space in the MAS model corresponds to the pagemap in the MPMAP model. In turn, each pagemap corresponds to a page table located in some location of the ALE

$$
\begin{aligned}
&\mathit{index}(i, va) &&:= \begin{cases} va\&\mathtt{0xFFF} & \text{if } i = 0 \\ va > (12 + (i-1) * 10)\&\mathtt{0x1FF} & \text{otherwise} \end{cases} \\
&\mathit{lookup}(tbl, level, va) &&:= \begin{cases} 0 & \text{if } tbl = 0 \\ tbl + \mathit{index}(0, va) & \text{if } level = 0 \\ \mathit{lookup}(M.D(tbl + \mathit{index}(level, va) * 8), level - 1, va) & \text{if } level > 0 \end{cases} \\
&\mathit{trans}(va) &&:= \mathit{lookup}(\mathtt{PTROOT}, 4, va)
\end{aligned}
$$

Figure 6.23: Predicates for Realistic Address Translation

model, which is a change from before, as in our simpler VMM, there was a specific location for the page table. This lack of reserved area for the page table is the only change in the ALE and HW models for this diagram.

It is worth noting that the same system that we have had in place for hiding the system tables (page tables and allocation table) from the abstract address models is still in place. Moreover, it now has to hide the page tables even though their location is not pre-determined, which is one of the reasons for the growth in complexity of verification.

6.4 Multi-level Page Tables

Yet another improvement to our certified VMM is the handling of the multi-level page tables. Real computer hardware does not use a single flat page table to produce the translation. Much of the virtual address space is not allocated, resulting in a sparse page table. For large address spaces, flat page tables would waste a lot of memory, and thus, hardware defines the page tables as a fixed size pruned tree structure rather than a table.

Our goal, that we will not achieve fully for reasons to be shown later, is to have our VMM work with such a page table structure, and thus support real hardware, rather than the simplified hardware that we have worked with thus far. The first step in the verification would be to update the HW memory model to include such an AT system. The change requires modification of the translation predicate, which must now be defined recursively.

The full definition of such a translation for the HW memory model is given in Figure 6.23. The translation is designed for a 64-bit machine with a 4k page size, and 64-bit entries. This fact can be seen in the definition of the *index* predicate which is designed to pull out specific bits of the virtual

$$PTdom(tbl, level)(l) := \begin{cases} \text{false} & \text{if } level = 0 \lor tbl = 0 \\ \left(\begin{array}{l} (tbl \le l < tbl + \texttt{PGSIZE}) \lor \\ \exists 0 \le i < 512. PTdom(M.D(tbl + i * 8), level - 1,)(l) \end{array} \right) & \text{if } level > 0 \end{cases}$$

Figure 6.24: Multi-Level Page Table Domain Predicate

address, which correspond to the indexes of the address at different levels of the page tables. The levels of the page tables can be observed by the definition of the *lookup* predicate, which looks up the virtual address in a table given by *tbl*, assuming that *level* is the current level of lookup. If the level is 0, then the *tbl* is assumed to be the translated base address, in which case the offset of the address is added to get the final translated address. If the level is greater than zero, then a lookup in the table is performed with the next pointer retrieved from the table, and a recursive call to lookup is made (with a smaller level). The complete translation is then defined by calling lookup with the starting table location read from the PTROOT register, and using the standard 64-bit 4-level page tables, making the starting value of level to be 4. If any of the values are 0, then that means that the translation is not valid, and thus the translation shortcuts to the value 0, which indicates an invalid mapping.

To update our VMM to use the new AT algorithm, we have to switch to the new *trans* function in the HW, PE, and ALE machine models. However, not everything is this simple. Because of the new translation function, we also have to modify several predicates that are used within the machine. First, we would need to update the *dm* predicate, which has to be more complex, as ensuring that all low addresses are mapped, is now easier to define using *trans* itself: $dm := \forall l. \text{Low}(l) \to trans(l) = l$ We could have used this definition earlier, however, such a definition incorporates *trans* into the formulas, which makes the proofs more complicated.

Another complication is the requirement that the domain of the page tables is separate from the domain of the allocation tables, the precondition necessary to certify the memory allocator. Because the page tables are now tree shaped, we would need to update another recursive predicate that will generate the domain of a particular page table (see Figure 6.24).

These predicates allow us to alter the HW, PE, and ALE machines to work with the new translation system. Moreover, as the code of our memory allocator accessed only direct mapped addresses, neither the code, nor specifications change. Although the machine semantics have been modified, the *dm* property of the translation would allow us to access memory without additional difficulty. The new complexity arises from the fact that the precondition that guarantees that the page tables are separate from memory is now a lot more complex. And though the proof may now be a lot more complicated for that reason, we should still be able to prove that the page tables are not affected by the memory allocator.

The complexity of the page tables would also affect the code and the specifications of `pt-init`, which must now create a much more complex page table. The modifications to `pt-init` will also affect `init` as well, as the more complex predicates are now required to safely activate address translation. However, we will ignore the problems of initialization, and focus on the page table driver and the translation between the ALE and MPMAP models of memory.

Figures 6.25 and 6.26 show the code of the multi-level page table driver. The driver consists of mostly the same interface as the simple drivers of for our earlier hardware, although the complexity of the code has increased dramatically. There are several new functions that are needed due to the more complicated data structure used. The first such function is `index` which returns the page table index of a virtual address for a given level. In other words, the return value of such function is equivalent to the *index* predicate.

The other new function, `pt-traverse`, is the function for the traversal of the data structure. In its most basic usage, it returns the address of leaf entry of the page table tree structure, thereby allowing both reads and updates of the entry. The function is recursive in that it will follow the tables to the depth specified by the `level` argument. This function is also designed to handle the fact that the page table tree is pruned. In a traversal, if the function encounters a pruned node, then it would return a zero, indicating that the branch is pruned, and no access is possible. A special flag argument named `create` tells the traversal to un-prune the tree for the specific address requested. If the argument is specified, then if a pruned branch is encountered, the function will allocate a new node with all pruned entries, and the continue the same action recursively until the final entry is created. Because this process requires allocation, it may fail, and in that situation, the traversal will return a zero to indicate a failure.

The other functions are the actual PMAP style interface functions. `pt-new` creates a new page table, which it does by creating a completely pruned tree, and then issuing calls to insert a direct

```
uint64_t index(uint64_t va, int level) {
        if (level == 0) return (va & 0xfff);
        return ((va >> (12 + 9 * (level-1))) & 0x1ff);
}

uint64_t pt_traverse(uint64_t va, uint64_t tbl, int level, int create) {
        // level > 0
        uint64_t i = index(va, level);
        uint64_t entry_addr = tbl + index * 8;
        if(level == 1) return entry_addr;
        uint64_t entry = *(entry_addr);
        if (entry == 0 && create == 0) return 0;
        if (entry == 0) {
                entry = mem_alloc() * PGSIZE;
                if (entry == 0) return 0;
                bzero(entry,PGSIZE);
                *(entry_addr) = entry;
        }
        pt_traverse(va,entry,level-1, create);
}

void pt_delete_tbl(uint64_t tbl, int level) {
        uint64_t i, entry;
        if (level > 1) {
                // delete subtables
                for(i=0;i<512;i++) {
                        entry = *(tbl+i*8);
                        if (entry > 0) pt_delete_tbl(entry,level-1);
                }
        }
        mem_free(tbl/PGSIZE);
}

void bzeropg(uint64_t addr) {
        size_t final = addr+PGSIZE;
        while(addr < final) {
                *addr =0;
                addr++;
        }
}
```

Figure 6.25: Multi-Level Page Table Driver Code (Helper Functions)

```
uint64_t pt_new() {
        uint64_t pg;
        uint64_t pt = mem_alloc() * PGSIZE;
        if (pt == 0) return 0;
        bzeropg(pt);
        // slow, but works (no need to set 0 - invalid)
        for (pg=1;pg<NPAGES;pg++) pt_set(pg,pg);
        return pt;
}


void pt_delete(uint64_t pt) {
        pt_delete_tbl(pt,4);
}

void pt_set (uint64_t vp, uint64_t pp) {
        uint64_t entry_addr = pt_traverse(vp*PGSIZE, hw_getPTROOT(), 4, 1);
        if (entry_addr == 0) return 0;
        *entry_addr = pp*PGSIZE;
        return vp;
}

uint64_t pt_lookup(uint64_t vp) {
        uint64_t entry_addr = pt_traverse(vp*PGSIZE, hw_getPTROOT(), 4, 0);
        if (entry_addr == 0) return 0;
        return (*entry_addr / PGSIZE);
}
```

Figure 6.26: Multi-Level Page Table Driver Code

mapping. `pt-delete` deletes the entire page table, which now has to happen recursively, as all nodes of the table need to be deleted as well. For this, it used the `pt-delete-tbl` helper function, which keeps track of the levels of the tree. The pair of `pt-set` and `pt-lookup` set and lookup the values of the page table, respectively. For this they use the traversal function to get to the location of the entry (un-pruning if necessary), and then performing the appropriate action with that entry. The only minor difference is that now `pt-set` may fail, and thus it returns zero on error, and the virtual address updated on success.

6.4.1 Verification of Multi-Level Page Table Driver

We will ignore the verification of the memory allocator and initialization in the low-level machine models, and instead focus on the verification of the page table driver code as defined over the ALE machine model.

The way that we believe such specification should be written is by using a set of abstract predicates defining the correctness of the data structure. Thus the page tables can be expressed as an abstract data tree, which we define in Figure 6.27. The tree is simply a multilevel tree, where each

$$\begin{aligned}
&index && := \{0\ldots512\}\\
&entry && := word\\
&Tree0 && : \{index \rightsquigarrow entry\}^{*}\\
&Treen && : \{index \rightsquigarrow option(Tree0 + Treen)\}^{*}
\end{aligned}$$

$$PT_M(tbl,l,T) := \begin{cases} \text{true} & \text{if } tbl = 0 \land T = None \\ \left(\begin{array}{l} M.A(tbl/\texttt{PGSIZE}) = \text{true}\land \\ \forall(0 \le i < 512).\, M.D(tbl + i * 8) = d(i).v * \texttt{PGSIZE} \end{array} \right) & \text{if } level = 1 \land T = Some(d) \\ \begin{array}{l} M.A(tbl/\texttt{PGSIZE}) = \text{true}\land \\ \forall(0 \le i < 512).\, PT_M(M.D(tbl + i * 8), l-1, d(i)) \end{array} & \text{if } level > 1 \land T = Some(d) \end{cases}$$

$$PTlook(T,l,va) := \begin{cases} T(index(va,l)) & \text{if } l = 1 \\ PTlook(T(index(va,l)), l-1, va) & \text{if } l > 1 \end{cases}$$

$$PTupd(T,l,va,value) := \begin{cases} T\{index(va,l) \rightsquigarrow value\} & \text{if } l = 1 \\ T\{index(va,l) \rightsquigarrow PTupd(T(index(va,l)), l-1, va, value)\} & \text{if } l > 1 \end{cases}$$

$$unp(T,l,va) := \begin{cases} T & \text{if } T = Some(d) \land l = 1 \\ \{0 \rightsquigarrow 0, \ldots, 511 \rightsquigarrow 0\} & \text{if } T = None \land l = 1 \\ \{0 \rightsquigarrow 0, \ldots, index(va,l) \rightsquigarrow unp(None, l-1, va), \ldots 511 \rightsquigarrow 0\} & \text{if } T = None \land l > 1 \\ T\{index(va,l) \rightsquigarrow unp(d(index(va,l)), l-1, va)\} & \text{if } l > 1 \land T = Some(d) \end{cases}$$

Figure 6.27: Abstract Page Table Definition

Function	Specification
index	$[va, level] \mapsto \texttt{ret}(index(va, level))$
bzeropg	$[start] \mapsto \Big(\text{ LowPg}(start/\texttt{PGSIZE})? \circ zeropg(start/\texttt{PGSIZE} \circ \texttt{ret}(0) \Big)$
pt-traverse	$[va, tbl, l, c] \mapsto \begin{cases} (c = 0)? \circ \text{traverse}(tbl, va, l)(\lambda entry.\,\texttt{ret}(entry)) \\ (c = 1)? \circ \texttt{ret}(0) \\ (c = 1)? \circ \bigvee_T \left(\begin{array}{l} \text{upd-pt}(l, T, unp(T, l, va)) \circ \\ \text{traverse}(tbl, va, l)(\lambda entry.\,\texttt{ret}(entry)) \end{array} \right) \end{cases}$
pt-delete-tbl	$[tbl, l] \mapsto \text{delete-pt}(tbl, l) \circ \texttt{ret}(0)$
pt-new	$[] \mapsto alloc(\lambda pg.\,\texttt{ret}(pg * \texttt{PGSIZE}))$
pt-delete	$[tbl] \mapsto \text{delete-pt}(tbl, 4) \circ \texttt{ret}(0)$
pt-set	$[vp, pp] \mapsto \bigvee_T \left(\begin{array}{l} \text{upd-pt}(\texttt{PTROOT}, T, unp(\texttt{PTROOT}, vp * \texttt{PGSIZE}, 4)) \circ \\ \text{upd-pt}(\texttt{PTROOT}, T, PTupd(T, 4, vp * \texttt{PGSIZE}, pp * \texttt{PGSIZE})) \end{array} \right)$
pt-lookup	$[vp] \mapsto \bigvee_T (PT_M(\texttt{PTROOT}, 4, T)?, \texttt{ret}(PTlook(T, 4, vp * \texttt{PGSIZE})))$

where

$$\text{zeropg}(pg) \quad := (M.D(pg * \texttt{PGSIZE}) := 0, \ldots, M.D(pg * \texttt{PGSIZE} + \texttt{PGSIZE} - 8) := 0)$$

$$\text{alloc}(cont) \quad := \begin{cases} cont(0) \\ \bigvee_{pg} ((M.A(pg) = \text{false}, M.A(pg) := \text{true}) \circ \text{zeropg}(pg) \circ (cont(pg))) \end{cases}$$

$$\text{traverse}(tlb, l, va) \quad := \begin{cases} tbl + index(va, l) * 8 & \text{if } l = 1 \\ traverse(M.D(tbl + index(va, l) * 8), l - 1, va) & \text{if } l > 1 \end{cases}$$

$$\text{preserve}(tbl, l) \quad := \lambda(\mathbb{S}, \mathbb{S}').\left(\begin{array}{l} \mathbb{S}'.S = \mathbb{S}.S \wedge \\ \forall pa.\, \neg PTdom_{\mathbb{S}.M}(pa, tbl, l) \wedge \\ \qquad \mathbb{S}.M.D(\text{Pg}(pa)) = \text{true} \rightarrow \\ \qquad \mathbb{S}'.M.D(pa) = \mathbb{S}.M.D(pa) \\ \forall pa.\, \neg PTdom_{\mathbb{S}.M}(pa, tbl, l) \wedge \mathbb{S}'.M.A(\text{Pg}(pa)) = \text{true} \rightarrow \\ \qquad \mathbb{S}'.M.A(Pg(pa)) = \text{true} \end{array} \right)$$

$$\text{delete-pt}(tbl, l) \quad := \lambda \mathbb{S}. \left\{ (S', M') \;\middle|\; \begin{array}{l} \exists T.\, (PT_M(tbl, l, T)) \wedge \text{preserve}(tbl, l)(\mathbb{S}, \mathbb{S}') \wedge \\ \forall pa.\, PTdom_{\mathbb{S}.M}(pa, loc, 4) \wedge \mathbb{S}'.M.A(\text{Pg}(pa)) = \text{false} \end{array} \right\}$$

$$\text{upd-pt}(loc, T, T') \quad := \lambda \mathbb{S}. \left\{ (S', M') \;\middle|\; \begin{array}{l} (PT_M(loc, 4, T)) \wedge \text{preserve}(loc, 4)(\mathbb{S}, \mathbb{S}') \wedge \\ \qquad PT_{M'}(loc, 4, T') \end{array} \right\}$$

$$\text{new-pt}(cont) \quad := \bigvee_{loc} \lambda \mathbb{S}. \left\{ (S', M') \;\middle|\; \begin{array}{l} \text{preserve}(0, 0)(\mathbb{S}, \mathbb{S}') \wedge \\ \qquad PT_{M'}(loc, 4, \{0 \rightsquigarrow 0, \ldots, 512 \rightsquigarrow 0\}) \end{array} \right\}$$

Figure 6.28: Multi-level Page Table Driver Specification

node contains 512 entries. The leaf node entries are values, while the non-leaf nodes contain entries which point to the subtrees, or indicate that this branch is pruned. Not all trees are valid page tables, and thus we define the $PT_M(tbl, l, T)$ predicate, which checks whether the tree is valid and corresponds to the data structure in memory. The *tbl* argument points to the location of the page table in memory. The l argument indicates which level we are currently analyzing, and the T argument is the actual abstract tree. The individual cases of the predicate are designed to ensure correspondence with actual in-memory page tables, e.g. location 0 indicates that the table is pruned, level 1 trees correspond to entries, and level > 1 trees must be checked for subtrees.

The other predicates defined in the figure are simple lookups and updates over the abstract tree structures. $PTlook$ looks up the translation, $PTupd$ updates the mapping, and un-prune defines a new tree that is equivalent in translation to the tree given to it as an argument, but expands the tree so that a particular address is not pruned.

At this point we have additional machinery which will allow us to give specifications to the functions of the page table driver, which are given in Figure 6.28. However, to define these specification concisely, we have created predicates given at the bottom of the figure. These predicates are as follows:

- zeropg(*pg*) - An action that zeroes out the page *pg*.
- alloc(*cont*) - Similar to the specification of `mem-alloc`, except it zeroes out the page, and chains the action constructor specified by *cont* instead of returning.
- traverse(*tbl*, *l*, *va*) - calculates the address of the entry specified by *va* in the pagetable *tbl* with level *l*.
- preserve(*tbl*, *l*) - a state relation that indicates that all allocated memory information, *except* the page table located at *tbl* with level *l* is preserved. This predicate is useful in defining actions that update the page table.
- delete-pt(*tbl*, *l*) - an action that represents the deletion of the *l*-level pagetable at location *tbl*.
- upd-pt(loc, T, T') - an action that represents the update of the 4-level pagetable at location *loc* from being equivalent to abstract tree T to being equivalent to abstract tree T'.

- new-pt(*cont*) - an action that represents the creation of a new page table, and keeping all other allocated memory data equivalent. The location of the new page table is then passed to the action constructor *cont*, which completes the action.

Using these predicates, the specifications of most of the functions of the page table driver are self-explanatory. However, some are still fairly complex. For example, the spec of the `pt-traverse` function still requires several cases: one case for the non-updating traversal, one case for un-pruning, which generates an error, and another case for a successful un-pruning, followed by a traversal. pt-set function is also fairly complex, since the action is a disjunction over all possible trees that can be both successfully un-pruned and then updated with a new entry.

Since we have not actually proved these specifications with the code, it is highly likely that these specifications (and possibly code) may contain errors. However, we believe that the approach is sound, and once the certification process is underway, the specifications and the code can be debugged, yet remain similar in spirit to the specifications we have given in this section.

6.4.2 Abstracting the Multi-level Page Table driver

One of the very nice results of the our framework is that changes are mostly contained within a single level. Although, the hardware has been severely modified from the original specification, the MPMAP model is still a sound abstraction for the modified page tables. All we need to do to update the higher-level code to work with the new page tables is to show the relation between the MPMAP machine and the updated ALE machine with the new page table driver library. To do this, we will need to update the relation between the MPMAP and ALE memory models ($M_{MPMAP} \leq M_{ALE}$), and to reprove the compatibility proofs between abstract primitives of MPMAP and the specification of the page table library.

First, we will focus on the relation, which we think will be quite similar to the one in Figure 6.29. The relation is actually much more complex than before, though this additional complexities are hidden within the *trans* operation. This additional complexity would make proving the load and store preservation much more difficult.

The compatibility proofs between MPMAP's abstractions and the new page table library will also be a bit more difficult to prove due to the more complex translation function. However, we

$$
\begin{array}{l}
\forall l.\,\mathrm{Low}(l) \rightarrow M_{MPMAP}.A(Pg(l)) = true \rightarrow M_{MPMAP}.D(l) = M_{ALE}.D(l) \\
\forall l.\,\mathrm{Low}(l) \rightarrow \neg\mathrm{PTdom}(l) \rightarrow M_{MPMAP}.A(Pg(l)) = M_{ALE}.A(Pg(l)) \\
\forall l.\,\mathrm{Low}(l) \rightarrow \mathrm{PTdom}(l) \rightarrow M_{MPMAP}.A(Pg(l)) = false \wedge M_{ALE}.A(Pg(l)) = true \\
\forall pmapid, va.\, pmapid \in \mathtt{dom}(PMS) \rightarrow \mathrm{HighPg}(vpg) \rightarrow \\
\qquad M_{MPMAP}.PMS(pmapid)(\mathrm{Pg}(va)) = \mathrm{Pg}(trans_{M_{ALE}}(pmapid, va))) \\
\forall pmapid, va.\, pmapid \in \mathtt{dom}(PMS) \rightarrow \mathrm{LowPg}(\mathrm{Pg}(va)) \rightarrow trans_{M_{ALE}}(va) = va \\
M_{MPMAP}.pmapid = M_{ALE}.\mathtt{PTROOT}
\end{array}
$$

Figure 6.29: Relation Between MPMAP and ALE with Multi-level Pagetable Models

believe that some of these difficulties can be abstracted since the *trans* and *PTlook* predicates can be related via a lemma that expresses the translation in terms of the results of *PTlook*, which will reduce the amount of recursion and pointer chasing. We will not go into more details about how we believe these proofs will go, as until we complete them, all of this is conjecture.

6.4.3 Lessons about Updating the Semantics of Low-Level Models

Many of the updates we have discussed above have updated the definitions of the translation predicate *trans*. Because address translation is such an important part of the semantics, its definition is not only important in the semantics of the load and store memory operations, but also in the relations between the memory models. These relations tend to include facts about the translation system for several reasons: to relate the virtual addresses, to ensure that the mapping for low addresses is identity, and other similar facts. Thus when the *trans* predicate is updates, many of the definitions in the relation have changed as well, requiring us to update the proofs of memory properties and the weakening proofs between the abstract and concrete specifications. Unfortunately, these proofs are not trivial, resulting in significant time spent in updating proofs for the code where neither the code, nor the high-level meaning of the code has changed.

This problem is easiest to see in the fact that the memory allocator had to be reproven each time the memory model has changed. The memory allocator itself does not really depend on the effects of address translation, just on the fact that the low addresses are direct-mapped. Neither do the primitives it supplies introduce anything to the address translation, but becuase the underlying machine has changed we are obligated to reprove the weaknings.

The most obvious lesson here is that good proof engineering is still helpful. If we were to abstract the translation with a trans primitive, prove properties about the translation such as for any related states in ALE and PE, any virtual address would translate to same physical address. Using these properties, we would be able to define both verifications of code and weakning properties. Then, when the translation is updated, we would need to reprove the properties of translation, but the verifications and weakening proofs would remain the same.

The second lesson here is that our machine models are not perfect. Ideally, we should be able to create an ALE machine that is parametric on the underlying system of translation. In that case, the proofs that connect ALE to PE would be independent of a particular translation formula. In essence, we should be able to use our framework to force us to obey the first lesson. When developing our framework, we have toyed with the idea of machines depending on other machines, or machines being constructed with other machines as base, and when we were updating the virtual memory manager, we could clearly see why such work is needed.

One of the goals of our framework has been code reuse and modularity. And although we have seen that we did not achieve as much modularity and re-usability as we have hoped, we can clearly see where the framework has done us a great favor. Our separation into machines has contained the modifications in those machine models where the changes have occured. Neither the code, nor the weakening proofs had to be altered in those machines where the changes did not take place. Had we not used our framework, and have tried to have a single proof of virtual memory manager, we might have made a mistake in our proof engineering, and, perhaps, made the address space library verification depend on the address translation, and, thereby, ending up updating those proofs when changing the address translation formula.

The fact that our hard abstractions are helpful in controlling the damage caused by updates is the main lesson of our work. Our goals were to develop a framework that makes verification more organized, more modular, and more reusable. The fact that we were able to update our verification with relative ease, even though our proofs and specifications were not well-designed, lets us know that we have achieved our goal.

Chapter 7

Coq Implementation

One of the goals of this thesis is to have the modular code certified not only on paper, but formally and mechanically verified using a proof assistant. There are several reasons why we wanted the operating system formally verified.

1. Having our code formally certified gives us a formal proof object that corresponds to the certification of the system. Using this proof object it is possible to check that it corresponds to the operating system, and it is possible to check that the proof is valid using a very small checker, based on typechecking. Thus, if we attach this proof object to our operating system, we would allow every user of our operating system to easily check the validity of our proof, without meticulously following every written step.

2. The verification community research tends to focus on very specific and challenging problems. To do so, they narrow down the examples to be small and specific to the problem they are solving. The machine models of such work tend to be theoretical - simplified to exhibit the core issues. Our work, on the other hand, aims to work on real machines. The models of these machines tend to have a lot of tedious details, and, therefore, the proofs have to deal with them. When there are so many details, it is simply impossible to deal with them on paper.

For these reasons, we have tried to include as much of the work presented in this thesis in our formal verification. This section tries to give an overview of how this work is structured, as well as some of the interesting details and the challenges that we have had along the way.

7.1 Overview of the Implementation

Our Coq implementation that is included in this thesis can be separated into two halves: the verification and refinement system, and the actual code (virtual memory manager) that we verify. There are several directories that are part of the implementation:

`util` Contains various mathematical lemmas, general tactics, and other simplifications that are useful in our proofs, but are not dependent on the particular definitions of our framework. We will not discuss this portion of the implementation in detail.

`framework` Contains the definitions of our framework which includes the definition of the meta-language as well as the static semantics and safety proofs.

`refinement` Contains the definitions of refinements and all the definitions necessary to use them.

`refinement_samecode` Contains the same definitions as the `refinement` directory, but all definitions assume that the refinements do not modify the code. Because this reduces constructors, this is the version that is used for verification of the VMM.

`machines` Contains the definitions of the machines used in the VMM certification. The directory also contains refinement definitions and weakening lemmas between the library specifications.

`vmm` Contains the virtual memory manager code and its verification. It also contains the linking lemmas that connect all the code and a hypothetical kernel together into one certified whole.

We will now explain the purposes and the particular definitions that make up our framework.

7.1.1 Implementing the Certification System

We will first focus on our verification system, which is located in the `framework` directory. This directory contains all the files necessary for creation of abstract machines and their verification systems as well as for creation of refinements between these machines.

The directory contains several files, the purpose of which we will cover in detail, and also present the key definitions that are important for this work.

spec.v

Defines the type of actions that are used in our system. The Coq definition uses the (p, r) style actions, which are easier to work with within Coq as compared with sets of states in our paper version. The module also defines action combinators and certain lemmas about these combinators as needed by the verification system.

```
Record Action := mkAct { pre: State -> Prop ; rel : State -> State -> Prop }.
```

We have already discussed this approach to actions and how it is equivalent to the one we use in the paper. The definitions used in our implementation mirror the ones in our paper. For example, here is the definition of the weaker-than relation ($a \supseteq a'$) :

```
Definition ActionGTE (g g' : Action) : Prop :=
  (forall s, g.(pre) s -> g'.(pre) s) /\
  (forall s s', g.(pre) s -> g'.(rel) s s' -> g.(rel) s s').
```

Many other operations on actions are defined within this file.

label.v

Defines the type of labels that are encountered within the heaps. Normally this could be defined by the Z (or any other decidable type), but for the sake of simplifying proofs we have defined a label as an inductive set of actual label names. Although this means that our system is not truly general, we have made this sacrifice in order to gain the ability to invert (rather than induct) on labels, which greatly reduces proof size. Ideally, a better approach should be used.

specheap.v and codeheap.v

The `specheap.v` file defines the notion of the specification heap used by the modules specifications Ψ and libraries $\mathcal{L}$ within our system. The file also includes operations for joining the specification heaps, and deciding whether they are jisjoint. The `codeheap.v` file is similar to the specification heap definitions, but it is tailored to contain procedures instead of actions.

proc.v

The `proc.v` file defines the concept of procedures, and the notion of well-formedness of procedures. The definitions in the file mirror the ones we presented in the thesis. For example, the procedure is defined by the following:

```
Inductive Proc : Type :=
| Inil : Proc
| Iperf : Op -> Proc
| Icall : Label -> Proc
| Iseq : Process -> Proc -> Proc
| Ibranch : Decider -> Proc -> Proc -> Proc.
```

This definition is parametric over the definitions of the operations, labels and the branch condition (named decider in the implementation). This makes the Proc type parametric over the definition of the machine.

With the definition of procedure in place, the same module defines the well-formedness rules for the procedures, e.g. the definition of $\mathcal{M}, \Psi \vdash \mathbb{I} : \mathsf{a}$:

```
Inductive WFProc :  SpecHeap MachineState -> Action MachineState ->
      (Proc MachineOp Decider) -> Prop :=
| WFProc_nil : forall Psi, WFProc Psi (ActionID MachineState) (Inil _ _)
| WFProc_perf: forall Psi i g, getOpAction _ _ Mach i g -> WFProc Psi g (Iperf _ _ i)
| WFProc_call: forall Psi l (g : Action MachineState),
                 getLabelAction _ Psi l g -> WFProc Psi g (Icall _ _ l)
| WFProc_weaken: forall Psi g g’ I, ActionGTE g g’ -> WFProc Psi g’ I -> WFProc Psi g I
| WFProc_seq : forall Psi g’ g’’ I’ I’’,
  WFProc Psi g’ I’ -> WFProc Psi g’’ I’’ -> WFProc Psi (ActionChain g’ g’’) (Iseq _ _ I’ I’’)
| WFProc_branch : forall Psi b g’ g’’ I’ I’’,
  WFProc Psi g’ I’ -> WFProc Psi g’’ I’’ ->
  WFProc Psi (ActionPlus (dp b) g’ g’’) (Ibranch _ _ b I’ I’’).
```

The above definition is also parametric over the definition of the machine. Not only does it require knowing the types of operations and the branch conditionals, but also it needs to know the definition of the machine’s branch conditional decoder (`dp`), and the operational semantics (defined by `Mach`). The type of `Mach` is given by the following definition:

```
Definition Machine := Op -> option (Action MachineState.t).
```

The name of type `Machine` is a bit confusing, but it made sense a bit earlier, before branch conditionals were added to the system, when the operational semantics was an entire definition of the machine.

Other than the definitions of procedures and well-formedness of the procedures, `proc.v` includes lemmas about strengthening the stub library, as well as linking lemmas for well-formed procedures. These are self-explanatory.

scap.v

Defines the concept of certified modules, as well as the proofs of safety.

`scap.v` is the main component of our framework. It defines the notions of the well-formed code (certified module), as well as give the soundness lemmas for our framework.

The certified module is given by the following definition, which is just the encoding of the WF-CODE rule of our framework.

```
Inductive WFCode : SpecHeap -> CodeHeap -> SpecHeap -> Prop :=
  | WFCode_1 : forall Hypo C Psi (disj : SpecHeap_disjoint _ Hypo Psi),
      (forall f g, getLabelAction _ Psi f g ->
      (exists I, C f = Some I /\ WFProc (SpecHeap_join _ Hypo Psi disj) g I)) ->
    WFCode Hypo C Psi.
```

The next part of `scap.v` defines linking theorems for certified code modules (e.g. linking of certified modules which use the same machine), as well as theorems for strengthening the library, and other auxiliary theorems useful for actual verification.

Lastly, the module defines the safety theorems for our meta-language. These theorems follow exactly the safety proof described in our paper.

sact.v and actsmp.v

The first of these defines a simplified version of actions that can be written as deterministic functions. This way of defining actions is completely embeddable into the standard form, and is used to simplify proofs where we can make use of this determinism. The second file contains several lemmas that are used to simplify proofs over actions, such as automatic proof cleaners and structured unfolders. Some lemmas are remnants of an attempt to automate verification, but are still used in the manual version.

7.1.2 Implementation of the Refinement System

There are two directories in our code system that are used to define refinements. The main directory is `refinement`, with the `refinement_samecode` being a special restriction on refinements that at all stages ensures that the code of the program being refined does not change. We will discuss the

```
Variable MOp MDecider : Type.
Variable AState : Type.
Variable CState : Type.

Definition ASpecHeap := SpecHeap AState.
Definition CSpecHeap := SpecHeap CState.

Definition Proc := Proc MOp MDecider.
Definition CodeHeap := CodeHeap MOp MDecider.

Variable AMach : Machine AState MOp.
Variable CMach : Machine CState MOp.

Variable Adp : Decider_to_Prop AState MDecider.
Variable Cdp : Decider_to_Prop CState MDecider.

Definition A_WFCode := WFCode AState MOp MDecider AMach Adp.
Definition C_WFCode := WFCode CState MOp MDecider CMach Cdp.

Definition Refinement
   (SpecRel : ASpecHeap -> CSpecHeap -> Prop) :=
   forall Code APsi AHypo CPsi CHypo,
   SpecRel AHypo CHypo ->
   SpecRel APsi CPsi ->
   A_WFCode AHypo Code APsi ->
   C_WFCode CHypo Code CPsi.
```

Figure 7.1: Listing of refinement.v

only the restricted version, as it is used for VMM certification, but the more general framework is very similar.

refinement.v

This file (the main portion of which we show in Figure 7.1) contains the definition of refinement. The code is incredibly important as it shows how we have set up refinement templates in Coq. The listing shows that the refinement takes several types as parameters, namely the types of operations, deciders, and the states of the abstract and concrete machines. The next parameters, `AMach` together with `Adp` and `CMach` with `Cdp` are the definitions of the operational semantics and branch decoders. Notice both the abstract machine and the concrete machines uses the same type of operations and branch deciders but different state type. This allows the type of Proc to be shared by both the abstract and the concrete machine, meaning that the same exact procedure can be certified in both AMach and CMach, thereby simplifying the definitions.

The actual definition of the refinement is exactly what we would expect given our definition of the REFINE rule: if we are given something of the refinement type we can use it to convert an abstract certified module (`A_WFCode AHypo Code APsi`) into a concrete certified module (`C_WFCode CHypo Code CPsi`).

The next modules are all about produce objects of `Refinement` type from as little information as possible.

perproc.v

The first such reduction in complexity is the per-procedure refinement the we define in the `perproc.v` file. This refinement is defined by the following:

```
Definition PerProcRefinement
   (ActionRel : Action AState -> Action CState -> Prop) :=
   forall APsi Ag CPsi Cg I,
   specrel ActionRel APsi CPsi ->
   ActionRel Ag Cg ->
   A_WFProc APsi Ag I ->
   C_WFProc CPsi Cg I.
```

Comparing this definition to the definition of `Refinement` in the previous section, there are several differences. Instead of a SpecRel predicate, this refinement relies on the ActionRel predicate that is defined over individual actions, and not whole specification heaps. The conversion is different as well - instead of converting WFCode into another WFCode, this refinement converts WFProc into a WFProc. In other words, an object of type PerProcRefinement can be used to refine a single procedure.

However, the definition of the refinement relies on the specific relation between the specification heaps defined by specrel:

```
Definition specrel
     (ActionRel : Action AState -> Action CState -> Prop) :
                   ASpecHeap -> CSpecHeap -> Prop :=
    fun APsi CPsi =>
   (forall l, match APsi l with
      | Some Ag => match CPsi l with
                   | Some Cg => ActionRel Ag Cg
                   | None => False
                   end
      | None => match CPsi l with
                   | Some _ => False
                   | None => True
                   end
      end
   ).
```

This definition of specrel matches the way we construct $T_{\mathbb{C}}$ from $T_{\mathbb{I}}$ (named `ActionRel` in our implementation) in the per-procedure refinements. Using this definition, we can construct the relation over specification heaps from a procedure relation, and then show that the PerProcRefinement can be used to construct a complete refinement. We do this using the following lemma:

```
Lemma CodeProcRefinementValid :
  forall ActionRel,
   PerProcRefinement ActionRel->
   Refinement MOp MDecider AState CState
                     AMach CMach Adp Cdp
                     (specrel ActionRel).
```

This lemma shows us that the per-proc refinement is a refinement in the general sense, where the relation between the specification heaps and libraries is defined by (`specrel ActionRel`). Thus any module can be refined into a more specific module by finding a concrete specification that satisfies such a relation between the specification heaps.

order.v

This file contains the definition of order-preserving refinement. The definition is actually one of the more complex ones in our work.

```
Definition OrderRefinement
   (ActionRel : Action AState -> Action CState -> Prop) :=
   forall APsi CPsi,
   specrel _ _ ActionRel APsi CPsi ->
   (forall Ag, exists Cg, ActionRel Ag Cg) /\
   (forall Ai, match AMach Ai with
              | Some _ => exists Cig, CMach Ai = Some Cig
              | None => True
              end) /\
   (forall i Aig Cig Ag Cg,
       AMach i = Some Aig -> CMach i = Some Cig ->
       ActionRel Ag Cg -> ActionGTE Ag Aig -> ActionGTE Cg Cig) /\
   (forall Ag Cg Ag’ Cg’,
       ActionRel Ag Cg -> ActionRel Ag’ Cg’ -> ActionGTE Ag Ag’ -> ActionGTE Cg Cg’) /\
   (forall Ag Cg,
       ActionRel Ag Cg -> ActionGTE Ag (ActionID AState) -> ActionGTE Cg (ActionID CState)) /\
   (forall d Ag Ag’ Cg Cg’ Cg’’,
       ActionRel Ag Cg -> ActionRel Ag’ Cg’ -> ActionRel (ActionPlus (Adp d) Ag Ag’) Cg’’ ->
       ActionGTE Cg’’ (ActionPlus (Cdp d) Cg Cg’)) /\
   (forall Ag Ag’ Cg Cg’ Cg’’,
       ActionRel Ag Cg -> ActionRel Ag’ Cg’ -> ActionRel (ActionChain Ag Ag’) Cg’’ ->
       ActionGTE Cg’’ (ActionChain Cg Cg’)).
```

This definition differs from the PerProcRefinement in that the PerProcRefinement required us to supply a proof of WFProc → WFProc for a given ActionRel. This definition does not have such a burden. However, it requires us to show that the ActionRel obeys the specific properties, which can be used to automatically show that a converted procedure will remain well-formed. These rules are Coq encodings of the rules given in Figure 4.2.

As with the per-procedure refinement, we can show that any term of the OrderRefinement type can be converted into a term of general refinement. The `order.v` code proves the following lemma:

```
Lemma OrderRefinementValid :
  forall ActionRel,
   OrderRefinement ActionRel ->
    Refinement _ _ _ _ AMach CMach Adp Cdp
       (specrel _ _ ActionRel).
```

Thus by selecting an ActionRel, and proving the properties specified by OrderRefinement, we get the object of type Refinement, which means that we can convert WFCode terms over abstract machines AMach into WFCode terms over the concrete machines CMach.

repr.v

The main refinement of our thesis is implemented in the `repr.v` file in our code base. The refinement is defined by the following:

```
Variable repr : AState -> CState -> Prop.
Variable Decider_repr : forall d sa sc, repr sa sc -> Adp d sa -> Cdp d sc.
Variable Decider_repr’ : forall d sa sc, repr sa sc -> Cdp d sc -> Adp d sa.

Definition reprAction :
     Action AState -> Action CState := fun g =>
mkAct (fun s => exists bs, repr bs s /\ g.(pre) bs)
      (fun s s’ => forall bs, repr bs s -> g.(pre) bs ->
                (exists bs’, repr bs’ s’ /\ g.(rel) bs bs’)).

Definition reprrel : Action AState -> Action CState -> Prop :=
  fun (Ag : Action AState) (Cg : Action CState) =>
       ActionGTE Cg (reprAction Ag) /\  ActionGTE (reprAction Ag) Cg.

Definition ReprRefinement :=
   (forall Ai Aig,
          AMach Ai = Some Aig ->
          exists Cig, CMach Ai = Some Cig /\ ActionGTE (reprAction Aig) Cig).
```

The refinement is parameterized using three parameters, which are given as variables, so that they can be used without repetition. The first of these parameters is `repr`, which is defined as a simple relation between two states. The other two parameters, `Decider_repr` and `Decider_repr’` are the proofs that the `repr` relation preserves branching conditions, as this fact will be crucial for our proof.

The next definition, `reprAction` creates an action refinement from `repr`. The `reprrel` is the relational version of this conversion, which states that any action is related to the original action if it is equivalent to the `repr`-refined action. These definitions implement the $T_{\mathtt{a}}$ of our refinement framework in our code base.

Finally, we get to the actual definition of the repr-refinement, which encodes the single condition needed for ensuring that the repr-refinement is valid: that for every operation in the machine the repr-refined abstract action is weaker than the related concrete action.

Thus, we have made it quite simple to generate terms of the ReprRefinement type. To make use of them, we must convert them into the Refinement terms which can actually be used to refine certified modules. This is done by proving the following lemma:

```
Lemma ReprRefinementValid :
   ReprRefinement ->
    Refinement _ _ _ _ AMach CMach Adp Cdp
              (specrel _ _ reprrel).
```

Although we do not need to do anything further to make use of this module, we have defined a macro that is useful for making use of repr-refinements. This macro is defined by the following definitions:

```
Definition specmake
      (APsi : ASpecHeap) : CSpecHeap :=
      fun l =>
      match APsi l with
      | Some Ag => Some (reprAction Ag)
      | None => None
      end.

Lemma EqAutoReprRefinement :
   ReprRefinement ->
   forall AHypo AC APsi,
   WFCode AState MOp MDecider AMach Adp AHypo AC APsi ->
   WFCode CState MOp MDecider CMach Cdp (specmake AHypo) AC (specmake APsi).
```

This definition expands out the meaning of the Refinement type and makes the conversion between the well-formed modules more explicit. However, it does not provide anything over the original definition of Refinement, except it saves a few lines when we make use of the refinement.

Other Refinements

The other files contained in the directory present other refinements defined in our thesis. They are similar in spirit and in structure to the repr-refinement, and thus we will not present them in detail.

7.1.3 Implementation of the Virtual Memory Manager

The code of the virtual memory manager is distributed over two directories: `machines` and `vmm`. The first directory contains the definition of the C machine complete with definitions of the template of the memory system. It also contains the definitions of all the memory systems used by the vmm as well as the specification libraries that are used by the memory systems. This directory also contains the refinements between the C machines instantiated with all the memory systems and the refinements between the specification libraries.

The `vmm` directory contains the actual code of the virtual memory manager, as well as the certification of the said code. This directory also contains the final linking of all the modules into a completely certified system.

We will now go over the structure of the implementation, highlighting the most important details that make the implementation function.

Common Definitions

The `mach/common.v` file contains constants and mathematical lemmas used by the rest of the virtual memory manager. These constants include vmm specification definitions, such as the size of the memory, as well as the location of the memory allocation tables, etc.

Memory Template and Definitions

The first crucial file in our system is `mach/mem.v`. This file contains the template for defining the memory system. The full listing of this template is informative, and thus we reproduce it here:

```
Module Type Mem_T <: Typ.
Parameter t : Type.
Parameter getMemCheck : Z -> SCheck (State := t).
Parameter getMem : Z -> SGet (State := t) Z.
Parameter setMemCheck : Z -> SCheck (State := t).
Parameter setMem : Z -> Z -> SAct (State := t).
End Mem_T.
```

As you can see from this definition, a memory is just an arbitrary type that provides four predicates for accessing it. These four predicates implement the loads and stores from the memory, each separated into two predicates - one for performing the actions (getting of the value from the memory or updating the memory with a new value), and one for checking that the action was successful.

In fact, this separation of every action into two predicates permeates our state updates. The reason for doing so is that it eliminates option types on the actions, which, if present, require case analysis. The two predicate approach tends to eliminate this case analysis and allows the terms to simplify quicker.

The `mem.v` file only defines the template of memory, but does not define any particular memory systems. The particular memory systems are defined one per file, in the files corresponding to the particular names of the memory models, namely `hw.v`, `pd.v`, `ald.v`, `pe.v`, `ale.v`, `pmap.v`, and `as.v`.

All of these files share a common pattern, and thus we will describe only one of these: `pmap.v`, which defines the PMAP memory model. The key definition in that file is the definition of the state of memory, defined by the following

```
Record State := { data : Z -> Z; pagemap : Z -> option Z; allocdata : Z -> bool}.
Definition t := State.
```

This definition shows us exactly how the abstract PMAP state is implemented. The memory is defined as a triple containing the data defined by a mapping from locations into values, a pagemap, which defines the translations from virtual pages into physical pages, as well as the allocdata mapping that defines which physical pages are present in memory. The state of the memory itself is, however, not precise without the definitions of access. Without such definitions, we would not know whether allocdata was per location or per page. Nor do we know how the translation mechanism works. Thus we must give these predicates to complete the definition.

```
Definition transCheck (l : Z) : SCheck (State := State) :=
fun s => ValidVA l &&
              match s.(pagemap) (Pg l) with
              | Some ppg => ValidPA (ppg * PGSIZE + Off l)
              | None => false
              end.

Definition trans (l : Z) : SGet Z (State := State) :=
fun s => match s.(pagemap) (Pg l) with
              | Some ppg => (ppg * PGSIZE + Off l)%Z
              | None => (-1)%Z
              end.

Definition paCheck (l : Z) : SCheck (State := State) :=
fun s => ValidPA l && s.(allocdata) (Pg l).

Definition vaCheck (l :Z) : SCheck (State := State) :=
fun s =>
      if isPhys l then paCheck l s
      else transCheck l s && paCheck (trans l s) s.

Definition getMemCheck (l:Z) : SCheck (State := State) :=
fun s => vaCheck l s.

Definition getMem (l:Z) : SGet (State := State) Z :=
fun s => if isPhys l  then s.(data) l else s.(data) (trans l s)
```

The above definitions clearly spell out the translation function, given by the `trans` and `transCheck` predicates. The `vaCheck` and `paCheck` predicates ensure that the memory access is only valid when the relevant page is allocated. The `getMemCheck` and `getMem` predicates make use of the checks and the translation to return the correct results.

The store is very similar to the load of memory, and reuses both the memory checks as well as the translation. It is slightly more complex than the load due to the need to return the updated memory state.

```
Definition setMemCheck (l : Z) : SCheck (State := State) :=
fun s => vaCheck l s.

Definition updateMem (l z : Z) : SAct (State := State) :=
fun s => {| data := fun l' => if Zeq_bool l l' then z else s.(data) l';
                                  allocdata := s.(allocdata); pagemap := s.(pagemap) |}.

Definition setMem (l z : Z) :  SAct (State := State) :=
fun s => if isPhys l then updateMem l z s
            else updateMem (trans l s) z s.
```

The interesting portion of the listing is the `updateMem` predicate which shows how a state of the memory is changed when physical location `l` is updated with value `z`.

The rest of the `pmap.v` is devoted to lemmas that simplify certification that are specific to the PMAP memory model. An example of such lemma is the following:

```
Lemma trans_update_unch : forall l l' z s, trans l (updateMem l' z s) = trans l s.
```

The lemma proves that an update of the memory does not change the value of the translation. This in turn allows us to figure out the result of the address translations in the presence of memory stores. Other lemmas define similar simplification properties.

The implementation of the other memory models are similar in structure to the `pmap.v`, similarly defining the memory loads and stores as required by the memory template as well as the lemmas that allow simplification of certain operations over the memory model.

These memory models are then plugged in to the C language, which we will now define.

The Implementation of C

The memory models do not define valid C language machines by themselves. For this we need to provide the syntax and the semantics of the C language. These definitions are located in the `mach/c.v` file in our implementation.

The definition of C is a fairly lengthy piece as the C language itself, even in the extremely simplified form we use, is not tiny. The first part we define are the variables and the stack

```
Definition Var := Z.

Definition dataframe := Var -> option Z.

Inductive Frame : Type :=
  | DataFrame : dataframe -> Frame
  | CallFrame : list Z -> Frame
  | RetFrame : Z -> Frame.

Definition t := list Frame.
```

The stack is defined as a list of frames, each being one of the three types - the data frame used for storing local variables, the call frame for passing arguments to the functions, and the return frame for return values. This definition almost exactly parallels the definition used in the written definition of our thesis. What is a bit different are how we access these values, as we no longer can rely on the notation to access these values.

Similar to the memory loads, we define variable reads using two predicates - one for getting the values, and the other for making sure that the read was successful. The definition of these predicates are as follows:

```
Definition getVarCheck (v : Var) : SCheck (State := t) :=
fun s =>
  match s with
   | DataFrame vars :: _ => match vars v with | Some _ => true  | _ => false end
   | _ => false
  end.

Definition getVar (v : Var) : SGet (State :=t) Z :=
fun s =>
  match s with
  | DataFrame vars :: _ => match vars v with | Some z => z | None => 0%Z end
  | _ => 0%Z
  end.
```

From these definitions one can clearly see that the reading of the variables always takes place over the top frame, and only if that frame is a dataframe. The definition also shows that if a read is not correct, the `getVar` will return a 0 value in order to avoid the option type. Other stack accesses such as setting of variables, reading and setting of arguments and return values, predicates that ensure the presence of a particular frame, as well as predicates that push and pop the stack frames follow a similar pattern.

Because the stack is such a complex system, with lots of different operation that can be done to it, the definition of the stack features many lemmas that allow for simplification of the stack updates. A few of these are listed below:

```
Lemma getVar_setVar : forall v v' z' s, getVar v (setVar v' z' s) =
   if isDataFrame s then
       (if Var_beq v v' then z' else getVar v s)
   else 0%Z.

Lemma getVarCheck_setRet : forall v z s, getVarCheck v (setRet z s) = false.
Lemma getVar_setRet : forall v z s, getVar v (setRet z s) = 0%Z.
Lemma getRet_setRet : forall z s, getRet (setRet z s) = z.
```

The most useful lemma is the getVar_setVar which simplifies the loading of variables after a variable update. For example, if the variable read is the same one that was updated, then the value is given in the update, and if the variable read is different than the one updated, then the update does not affect the read, and hence the read can be simplified. The other lemmas in the listing show the simplifications of updates after a setting of the return frame. In this case, attempts to read variables from the return frame automatically simplify to failures, while the read of the return value simplifies to the value itself. The actual file defines many more of these simplification lemmas.

After defining the stack, the c.v implementation proceeds to define the notion of expressions present in the C language, and the set of operators that are used to compute them. The expressions are defined by an inductive type that is fairly self-explanatory:

```
Inductive expr : Set :=
  | expr_z : Z -> expr
  | expr_var : Z -> expr
  | expr_mem : expr -> expr
  | expr_binop : BINOP -> expr -> expr -> expr
  | expr_unop : UNOP -> expr -> expr.
```

The implementation also defines a restricted form of expressions that does not allow memory access. This type is used as a branch conditional, which we have restricted from accessing memory in order to easily guarantee that a conditional will always select the same branch in all memory models, as needed for our linking.

The next part of the definition of the C language finally defines the complete C state by defining it as a tuple containing the stack and the memory state, which is defined parametrically.

```
Module C_State (Mem : Mem_T) <: Typ.
Record State := {stack : C_Stack.t;  memory : Mem.t}.
Definition t := State.

Definition StackCheck (f : SCheck (State := C_Stack.t)) : SCheck (State := t) :=
  fun s => (f (s.(stack))).
Definition StackGet {A} (f : SGet (State := C_Stack.t) A) : SGet (State := t) A :=
  fun s => f (s.(stack)).
Definition StackAct (f : SAct (State := C_Stack.t)) : SAct (State := t) :=
  fun s => {| stack := f (s.(stack)); memory := s.(memory) |}.
Definition MemCheck (f : SCheck (State := Mem.t)) : SCheck (State := t) :=
  fun s => (f (s.(memory))).
Definition MemGet {A} (f : SGet (State := Mem.t) A) : SGet (State := t) A :=
  fun s => f (s.(memory)).
Definition MemAct (f : SAct (State := Mem.t)) : SAct (State := t) :=
  fun s => {| stack := s.(stack); memory := f (s.(memory)) |}.
...
```

To handle allow our previous operations on tuples, the definition includes embeddings of memory and stack operation as operations on the state. This allows us to use `getVar var` on the state by lifting it to the `StackGet (getVar var)`, which is how we will see most of these state accesses in the specifications. In order to use the original simplifications, we have defined lemmas that allow us to group together stack and memory accesses, which allow the simplifications to work.

```
Lemma Stack_Mem_Act : forall st m s, MemAct m (StackAct st s) = StackAct st (MemAct m s).
Hint Rewrite Stack_Mem_Act : c.
Lemma StackGet_MemAct : forall A (get : SGet (State := C_Stack.t) A) m s,
                                   StackGet get (MemAct m s) = StackGet get s.
Lemma StackCheck_MemAct : forall (ch : SCheck (State := C_Stack.t)) m s,
                                   StackGet ch (MemAct m s) = StackCheck ch s.
Lemma StackGet_StackAct : forall A (get : SGet (State := C_Stack.t) A) m s,
                                   StackGet get (StackAct m s) = StackGet (fun s’ => get (m s’)) s.
Lemma StackCheck_StackAct : forall (ch : SCheck (State := C_Stack.t)) m s,
                                   StackGet ch (StackAct m s) = StackCheck (fun s’ => ch (m s’)) s.
Lemma MemGet_StackAct : forall A (get : SGet (State := Mem.t) A) m s,
                                   MemGet get (StackAct m s) = MemGet get s.
Lemma MemCheck_StackAct : forall (ch : SCheck (State := Mem.t)) m s,
                                   MemGet ch (StackAct m s) = MemCheck ch s.
Lemma MemGet_MemAct : forall A (get : SGet (State := Mem.t) A) m s,
                                   MemGet get (MemAct m s) = MemGet (fun mem => get (m mem)) s.
Lemma MemCheck_MemAct : forall (ch : SCheck (State := Mem.t)) m s,
                                   MemGet ch (MemAct m s) = MemCheck (fun mem => ch (m mem)) s.

Hint Rewrite StackGet_MemAct StackCheck_MemAct StackGet_StackAct StackCheck_StackAct : c.
Hint Rewrite MemGet_MemAct MemCheck_MemAct MemGet_StackAct MemCheck_StackAct : c.
```

These lemmas when used with autorewrite, automatically group together all stack updates and all memory updates, and automatically drop those portions of the state update that are not relevant. These together with the simplification lemmas within the definitions of stack and memory allow for a very powerful simplification of the state whenever such a simplification is possible.

With these predicates in place, we can define the evaluation function over the expressions, which also exists as two separate predicates:

```
Fixpoint eval(e : expr) : SGet Z := fun s =>
  match e with
  | expr_z z => z
  | expr_var v => StackGet (getVar v) s
  | expr_mem e => MemGet (getMem (eval e s)) s
  | expr_binop bop e1 e2 => (bop_op bop) (eval e1 s) (eval e2 s)
  | expr_unop uop e => (uop_op uop) (eval e s)
  end.


Fixpoint evalCheck (e : expr) : SCheck (State := t) :=
  match e with
  | expr_z z => (fun s => true)
  | expr_var v => StackCheck (getVarCheck v)
  | expr_mem e =>  SCheck_and (
                   fun s =>
                     MemCheck (getMemCheck (eval e s)) s)
                   (evalCheck e)
  | expr_binop bop e1 e2 => SCheck_and (evalCheck e1) (evalCheck e2)
  | expr_unop uop e => evalCheck e
  end.
```

The evaluation functions are fairly trivial, but are somewhat interesting in how they are defined via predicates such as `MemGet` and `getVar`.

Finally, we can get to the actual meat of the definition of the C machine, namely the operations and their actions. These are defined by the following:

```
Inductive perf : Set :=
  | perf_assign : Var -> expr -> perf
  | perf_fcall : FLABEL -> list expr -> perf
  | perf_ret : expr -> perf
  | perf_readargs : list Var -> perf
  | perf_readret : Var -> perf
  | perf_store : expr -> expr -> perf.

Definition Actions (i : perf) : Action State :=
  match i with
  | perf_assign v e => SAct_Action (SCheck_and (evalCheck e)
                                        (StackCheck (setVarCheck v)))
                                            (SAct_Let (eval e) (fun z => StackAct (setVar v z)))
  | perf_fcall f el => SAct_Action (evallistCheck el)
                                   (SAct_Let (evallist el) (fun z => StackAct (pushArgs z)))
  | perf_readargs vl => SAct_Action (StackCheck (loadArgsCheck vl)) (StackAct (loadArgs vl))
  | perf_readret v => SAct_Action (StackCheck getRetCheck)
                                  (SAct_Let (StackGet getRet)
                                      (fun z => StackAct (pop +++ (setVar v z))))
  | perf_store eloc e => SAct_Action
                         (SCheck_and (StackCheck isDataFrame)
                              (SCheck_and (evalCheck eloc)
                                (SCheck_and (evalCheck e)
                                  (SCheck_Let (eval eloc) (fun va => MemCheck (setMemCheck va))))))

                         (SAct_Let (eval eloc) (fun va =>
                         SAct_Let (eval e) (fun z =>
                         (MemAct (setMem va z))
                         )))
  | perf_ret e => SAct_Action (SCheck_and (StackCheck isFrame) (evalCheck e))
                           (SAct_Let (eval e) (fun z => StackAct (pop +++ (setRet z))))
  end.

Definition Mc_Machine := fun c => Some (Actions c).
```

In the listing above, we can immediately recognize all the operations that we needed to define in our C machine - the operations for assignment and store, as well as all the operations necessary for function call and return. The actions of these operations are given by the `Actions` definition. This definition may seem complex at the first glance, but is actually just a chaining of the state access predicates that we have defined up to this point. Once the meaning of the state access predicates and action combinators is understood, the definition is fairly natural to read and to follow. The `Mc_Machine` definition simply converts the operational semantics into the type that our framework expects.

This completes the definition of our C machine, and we can now move on to the refinements and the libraries.

REPR-Refinement between C machines

Our framework has already provided us with a way to create refinements by defining a representation relation between the C machines. However, in our thesis, we have reduced the proof burden even further by showing how to construct a representation between any two C machines by giving a representation relation between their memory models. This process is encoded in the `mach/mem_repr.v` file.

The first set of definition in that implementation is the template for defining relations between memory models, reproduced below:

```
Module Type Mem_Repr_T (Mem1 Mem2 : Mem_T).
Parameter Repr_Mem : Mem1.t ->Mem2.t -> Prop.
Parameter Repr_getMemCheck : forall (m1 : Mem1.t) (m2 : Mem2.t),
      Repr_Mem m1 m2 ->
      forall l, Mem1.getMemCheck l m1 = true -> Mem2.getMemCheck l m2 = true.
Parameter Repr_getMem : forall (m1 : Mem1.t) (m2 : Mem2.t),
      Repr_Mem m1 m2 ->
      forall l v, Mem1.getMemCheck l m1 = true -> Mem1.getMem l m1 = v -> Mem2.getMem l m2 = v.
Parameter Repr_setMemCheck : forall (m1 : Mem1.t) (m2 : Mem2.t),
      Repr_Mem m1 m2 ->
      forall l, Mem1.setMemCheck l m1 = true -> Mem2.setMemCheck l m2 = true.
Parameter Repr_setMem : forall (m1 : Mem1.t) (m2 : Mem2.t),
      Repr_Mem m1 m2 ->
      forall l v , Mem1.setMemCheck l m1 = true -> Repr_Mem (Mem1.setMem l v m1) (Mem2.setMem l v m2).
End Mem_Repr_T.
```

The core relation between the memory models is the relation between the memory state types given by `Repr_Mem`. However, the definition calls for ensuring that a load in the abstract memory model succeeds, then load of the same location in the concrete memory model must succeed as well. This is what the Repr_getMemCheck and Repr_getMem predicates require. Similarly, the other two predicates ensure a similar property about the stores in the two memory models. Thus a full module of type Mem_Repr_T M1 M2 will define a complete representation relation between the M1 and M2 memory models, e.g. ($M1 \le M2$). The rest of the mem_repr.v shows how to construct the refinement between the C machines instantiated with the related memory models.

The relation between the complete C states is reconstructed from the representation between the memory states is defined by the following predicate (C1 and C2 are defined as C machines instantiated with M1 and M2 memory models) :

```
Definition CRepr : C1.State.t -> C2.State.t -> Prop :=
  fun astate cstate =>
        (C1.State.stack astate = C2.State.stack cstate) /\
        Repr_Mem (C1.State.memory astate) (C2.State.memory cstate).
```

Then using this representation relation, we prove the following theorem.

```
Lemma refine : ReprRefinement _ _ _ (C1.Mc_Machine) (C2.Mc_Machine) CRepr.
```

The proof, which requires us to define several sub-lemmas, is a construction of the REPR-refinement between the C machines. Together with the proof that REPR-refinements are in fact real refinement, we can use this lemma to generate terms that allow us to convert a C1 certified module into a C2 certified module. Thus the code shows that by defining the memory relation, we automatically generate the refinement between the machines.

Example of Memory Relation

The next part of the code shows the definition of the relation between two particular memory models, in this case between PMAP and ALE models. This relation is defined in the mach/pmap_ale.v file. The first step in the file is to provide a repr relation between the memory models, which, as described in the previous section, is done by defining a module of type Mem_Repr_T. In the case of PMAP-ALE relation, the module definition is as follows:

```
Module PMAP_ALE <: Mem_Repr_T PMAP_State ALE_State.
```

The key definition here is the relation between states, which for PMAP-ALE is quite complex.

```
Definition Repr_Mem (pmap : PMAP_State.t) (ale : ALE_State.t) :=
   (forall l, ValidPA l = true ->
                PMAP_State.allocdata pmap (Pg l) = true ->
                   PMAP_State.data pmap l = ALE_State.data ale l) /\
   (forall l, ValidPA l = true ->
                ALE_State.PtDom l = false ->
                   PMAP_State.allocdata pmap (Pg l) = ALE_State.allocdata ale (Pg l)) /\
   (forall l, ValidPA l = true ->
                ALE_State.PtDom l = true -> PMAP_State.allocdata pmap (Pg l) = false) /\
   (forall l, ValidPA l = true ->
                ALE_State.PtDom l = true -> ALE_State.allocdata ale (Pg l) = true) /\
   (forall vpg, ValidVPg vpg = true ->
                     match PMAP_State.pagemap pmap vpg with
                     | Some ppg => ALE_State.data ale (ALE_State.PTROOT + vpg * 8)%Z = ppg
                     | None => ALE_State.data ale (ALE_State.PTROOT + vpg * 8)%Z = INVALID_ENTRY
                     end) /\
   ALE_State.DM ale.
```

This definition is exactly the memory-state relation that we have shown when defining the PMAP-ALE refinement in Section 5.6.3. This relation is followed by several lemmas that show that the loads and stores are preserved by the relation, as required by our memory relation template.

Once we define the relation between the memory models, the same file also includes weakening theorems between the specifications of high-level library and the underlying implementation. For example, the pmap_ale.v file includes the following lemma:

```
Lemma alloc_weak1 : ActionGTE (reprAction _ _ C_PMAP_ALE.CRepr PMAP_Lib.mem_alloc_spec)
                              ALE_Lib.mem_alloc_spec.
```

This lemma proves that under the PMAP-ALE relation the $T_{PMAP-ALE}(\mathcal{L}_{PMAP}(\text{mem-alloc})) \supseteq \mathcal{L}_{ALE}(\text{mem-alloc})$. The pmap_ale.v module also includes similar lemmas for weakenings of mem-free, pt-set and pt-lookup procedures. Thus the file includes all the lemmas necessary to construct a full refinement including the libraries, which culminates in the final definition the unfolds the refinement to produce a converter between the PMAP and ALE machines:

```
Lemma WF_kernel_pmap' : forall kernel_code kernel_spec pmap_lib_code
         (disj : SpecHeap_disjoint _ (C_PMAP_ALE.PsiMake kernel_spec) pmap_sig.pmap_spec)
         (disj' : SpecHeap_disjoint _ (C_PMAP_ALE.PsiMake kernel_spec) ALE_Lib.pmap_lib_spec),
         CodeHeap_disjoint _ pmap_lib_code kernel_code ->
         WFCode _ _ _ C_PMAP.Mc_Machine C_PMAP.State.expr_to_Prop
                       PMAP_Lib.pmap_lib_spec kernel_code kernel_spec ->
         WFCode _ _ _ C_ALE.Mc_Machine C_ALE.State.expr_to_Prop
                       ALE_Lib.pmap_lib_spec pmap_lib_code pmap_sig.pmap_spec ->

         WFCode _ _ _ C_ALE.Mc_Machine C_ALE.State.expr_to_Prop
                ALE_Lib.pmap_lib_spec
                (CodeHeap_join _ kernel_code pmap_lib_code)
                (SpecHeap_join _ (C_PMAP_ALE.PsiMake kernel_spec) pmap_sig.pmap_spec disj).
```

The lemma above states that if we take a module in the PMAP machine that relies on the PMAP library ($\mathcal{M}_{PMAP}, \mathcal{L}_{PMAP} \vdash \mathbb{C}^{module} : \Psi_{PMAP}^{module}$), and combine it with the certified implementation of the page table library on the ALE memory model ($\mathcal{M}_{ALE}, \mathcal{L}_{ALE} \vdash \mathbb{C}^{pt} : \Psi_{ALE}^{pt}$), e.g. implementation of the pt-set and pt-lookup functions, then the union of these modules is certified in the ALE machine $(\mathcal{M}_{ALE}, \mathcal{L}_{ALE} \vdash \mathbb{C}^{module} \cup \mathbb{C}^{pt} : T_{PMAP-ALE}(\Psi^{module}_{PMAP}) \cup \Psi^{pt}_{ALE})$. This lemma will become one of the key lemmas used to link the actual code.

The relations between other memory models is also located in the `mach` directory under the appropriate names. The specifications of the libraries and implementations are given in the files that have `sig` and `lib` as part of their names.

This actually completes the definitions of the memory models and the refinements between them.

Verification of Actual Code

The `vmm` directory of our implementation contains the actual code of the virtual memory manager, as well as its certification. Each function of the vmm has its own file in the directory, where the certification takes place. As a simple example, we will dissect the `mem_free.v` file, which verifies the mem-free procedure over the PE memory model, e.g. $(\mathcal{M}_{PE}, \mathcal{L}_{PE} \vdash \mathbb{C}^{mem}(\text{mem-free}) : \Psi^{mem}_{PE}(\text{mem-free}))$.

All certifications of procedures always contain very similar definitions. The first one is the code, defined as a procedure, which for mem-free takes the following form:

```
Definition mem_free : Proc (C_PE.Ops.t) expr_nomem :=
  Iperf _ _ (perf_readargs (page_var :: nil)) :::
  Iperf _ _ (perf_store (PMM_expr (expr_var page_var)) (expr_z 0%Z)) :::
  Iperf _ _ (perf_ret (expr_z 0%Z)).
```

The definition is just a straight line chaining of the 3 operations. The underlines are placeholders for the types of operations and the deciders, both of which are inferred from the signature. The specification of this code is already in the signature which was defined in the `mach/mem_sig.v` file where it was used as a part of the ALE-PE weakening for the memory library. Thus the proof of well-formedness reuses that definition.

```
Lemma WF_mem_free : WFProc _ _ _ Mc_Machine expr_to_Prop Psi mem_sig.mem_free_spec mem_free.
```

The proof of this lemma shows that the procedure of mem-free satisfies the specification of mem-free give in the `mem_sig.v` file. All the other lemmas and definitions in the `mem_free.v` are just supporting definitions and lemmas for the well-formedness proofs.

To actually use the definitions of well-formedness for the individual procedures, we must combine them into modules. In the case of mem-free, it is combined with mem-alloc to form a well-formed mem-module. The definition of this certified module is given in the `vmm/mem_lib_cert.v` file, which combines all the procedures and all the specifications using the WFCode rule. The result is the following:

```
Definition mem_code : CodeHeap _ _ := fun l =>
  match l with
  | mem_free_label => Some mem_free.mem_free
  | mem_alloc_label => Some mem_alloc.mem_alloc
  | mem_alloc_loop_label => Some mem_alloc.mem_alloc_loop
  | mem_alloc_body_label => Some mem_alloc.mem_alloc_body
  | _ => None
  end.


Definition mem_spec : SpecHeap _ := fun l =>
  match l with
  | mem_free_label => Some mem_sig.mem_free_spec
  | mem_alloc_label => Some mem_sig.mem_alloc_spec
  | mem_alloc_loop_label => Some mem_alloc.mem_alloc_loop_spec
  | mem_alloc_body_label => Some mem_alloc.mem_alloc_body_spec
  | _ => None
  end.

Lemma WF_mem_lib : WFCode _ _ _ C_PE.Mc_Machine C_PE.State.expr_to_Prop
      PE_Lib.pe_lib_spec mem_code mem_spec.
```

The `WF_mem_lib` lemma is the Coq term for the fact that the memory library is certified in the C machine with the PE memory model. The rest of the code is similarly verified and is placed into certified modules.

Final Linking

The entire implementation converges in the `vmm/main.v` file. This file performs the linking of all the modules into one final certified module containing everything, and certified over the C machine with the HW memory model.

The way that the linking works in our implementation is the following. First, we make the assumption of a completely certified high-level kernel. This is done by the following definitions:

```
Definition kernel_proc_spec : Action C_AS.State.t := {|
pre := (fun s => C_AS.State.StackCheck (C_AS.State.Stack.getArgsCheck 0) s = true);
rel := (fun s s' => False)
|}.

Parameter kernel : Proc C_AS.Ops.t expr_nomem.
Definition kernel_code : CodeHeap _ _ := fun l =>
  match l with
  | kernel_init_label => Some kernel
  | _ => None
  end.

Definition kernel_spec : SpecHeap _ := fun l =>
  match l with
  | kernel_init_label => Some kernel_proc_spec
  | _ => None
  end.
Parameter WF_kernel : WFCode _ _ _ C_AS.Mc_Machine C_AS.State.expr_to_Prop
                              AS_Lib.as_lib_spec kernel_code kernel_spec.
```

These definitions create a specification of the kernel-init function, and require that the kernel's code has this function, and the function is well-formed under the specification. We do not prove this fact, but instead assume them. Thus our certification can make use of any such kernel, but does not certify one. There is a small over-simplification that our code does: we assume that the kernel contains only the kernel-init function. Without this simplification, we would require a proof that the kernel does not contain any code with the same labels as our virtual memory manager. We have avoided this, as our attempt to write this down resulted in making the proofs quite a bit messier, as right now we get the disjointness of heaps automatically by simplification. Without it, we would have to make an assumption that the kernel and VMM do not conflict.

The next stage is the piece-wise linking of all the modules we have proven. For example, the linking that merges the kernel, the address space library, the page table library and the memory allocator is defined as follows:

```
Definition PSI2 : SpecHeap _ :=
    (C_PE_HW.PsiMake
          (SpecHeap_join _
          (C_ALE_PE.PsiMake
                 (SpecHeap_join _
                 (C_PMAP_ALE.PsiMake
                      (SpecHeap_join _
                                (C_AS_PMAP.PsiMake kernel_spec)
                                as_sig.as_spec
                                disj2
                      )
                 )
                 pmap_sig.pmap_spec
                 disj3
                 )
          )
          mem_spec
          disj4
          )
     ).

Definition C2 : CodeHeap _ _ :=
 CodeHeap_join _
       (CodeHeap_join _
             (CodeHeap_join _ kernel_code as_lib_code)
             pmap_lib_code
       )
       mem_code
```

.

```
Lemma WF2 :
      WFCode _ _ _ C_HW.Mc_Machine C_HW.State.expr_to_Prop HW_Lib.hw_lib_spec C2 PSI2.
```

The ugly definitions are mostly due to the fact that we have to attach proofs of disjointness of heaps (disj#) to union the specification heaps and the code heaps. Once they are unioned, a lemma (in this case WF2) is used to prove that the union of heaps forms a new certified module. The proof involves using the refinement lemmas for the particular pairs machines (for example, `WF_pmap_ale` from `mach/pmap_ale.v` and proofs of well-formedness of the certified modules that we have shown.

There are several other stages of linking in the `main.v` file, which follow the same pattern as WF2. The interesting one for us is the definition of C4, Psi4, and WF4, which link together all the code in the VMM, including the kernel in the C machine with HW memory model. Thus, when we prove the lemma named WF4, we now have proven a certified module containing everything, and by soundness, we know that this means that the kernel, linked with the virtual memory manager is now safe to execute on our simplified definition of the hardware, which is exactly the result that we were trying to accomplish.

7.2 Design Choices and Challenges

The goal of having a machine-checkable implementation of the framework is a critical component of this thesis. As the machines get more complex, it becomes more difficult to keep track of the details on paper. The proofs become more tedious, and the need for automation arises. Many proofs of this thesis, ones where the proof asks the reader to see Coq implementation, would be nearly impossible to lay out on paper in full detail. And if details are swept under the rug, then errors or problems from unexpected sources creep in. During the development of this implementation, errors were uncovered when we have made attempts to convert the paper proof to the mechanized proof. Hence, the fact that we have a fully mechanized implementation is a strong assurance to the reliability of our method and our proofs.

Although our implementation seems to be fairly simple and natural, it was not trivial to achieve. We have had many false starts that have almost derailed this work. Our terms were difficult to define, the specifications were impossible to write, and the proofs were too large to do by hand. It has taken us many simplifications and changes in design choices to get the implementation to work.

The main source of difficulties are verification lemmas where one action (usually a specification) has to be shown to be weaker than another actions (usually the specification of the program extracted from the code). When these actions are expanded, they tend to generate extensive terms with lots of chained state modifications and accesses to that state. A small example of this kind of term could be the following:

```
(getVarCheck entry_var (setVar entry_var (getRet (stack x0)) (pop (stack x0)))
```

This term describes a check that a variable named entry exists in the state which is popped then, from which also a return value is extracted and assigned to the entry variable.

In the current version, we already have a lemma that will quickly simplify this to true. Because the terms are fairly simple, we could also unfold all the terms and have the term above collapse to true automatically. However, in our previous implementation attempts, terms like above have been problematic, and even now in current design they still have a potential to cause trouble. Thus we will have a quick discussion about what makes terms such as above difficult, and how we are handling such difficulty.

7.2.1 The Simplification Problem

In the original implementation, we did not split a variable retrieval and other state accesses into a validity check and the value retrieval, e.g. `getVar` and `getVarCheck` were defined as a single `getVar` predicate which returned an option type. The result of this predicate was case analyzed before the value was used.

The problem originated from the case analysis. Because the sub-terms of case analysis depended on the results of the case, neither simplifications nor rewrites could be applied to the sub-terms. Thus every single case analysis has to be performed by a person. Because the terms frequently duplicated the state, the amount of human effort required for even simple terms was extensive.

Before we have split the terms into two parts, we have tried automation to make guesses for these terms, which was moderately successful, but ran into other issues.

7.2.2 The Automation Problem

One way we have tried to tackle the issues of simplification involving case analysis is by trying to automatically find common patterns in the terms and performing a "destruct" tactic so that both cases could be analyzed. The common occurrence was that one of the cases was a failure (None), and that this failure would propagate, and eventually generate a contradiction.

This plan was moderately successful, the destruct generally did simplify the code to the point where the simplified solution popped out. However, there were several issues that have killed this approach.

First, the automation tactic was searching for terms that could be destructed, and it was not entirely clever about it. Thus instead of destructing the choice such that one of the cases would be eliminated immediately, it would destruct something that could not be simplified until later. Thus automation had a tendency to take a very long road to the solution. By itself, this may not have been a disaster, except that it was exacerbated by the other problems.

The next problem was the fact that not all of the actions of the program were deterministic and defined by a function. Parts of actions commonly relied on relations, which could only be simplified by picking a state. We have tried solving this issue by using Coq's existential/logic variables (eexists tactic), but this ended up being a disaster, as Coq can make bad guesses in the unification of logic variables, eliminating the solution. As an alternative to using logic variables, we tried to have a person supply these intermediate states, but because our automation was not choosing the most concise paths, we frequently had to duplicate the work. Once the terms got large enough to have many choices of which information to case analyze, the amount of duplication became unreasonable.

The other problems we have encountered is that pattern matching became really slow on larger terms, and that when the automation asked a person to pick the next step, the terms presented were too large for a person to follow. We have tried countering that with autorewrites to simplify the terms. However, this approach did not work, since autorewrites do not work in the sub-terms of pattern matches. We have countered it by applying simplifying rewrites each time a pattern was eliminated, but that meant we had to perform autorewrites frequently. Unfortunately autorewrite tactic is a somewhat slow process, and its frequent use slowed down even the simpler automation.

When our tactics started taking 15 minutes before any result was returned, we started giving up on try-all-paths approach to automation. Combined with other problems, such as having to carefully specify which types to destruct and which ones to leave alone, we think that automation of verification needs to be designed a lot more carefully. Instead, we tried to focus on a manual proof, where primitive unfolding and beta-reduction can produce much of the simplification necessary to make the human effort reasonable. Splitting of the state updates into values and validity checks went a long way into allowing the beta-reduction to simplify the terms to a minimum.

There is still some automation that is left in our system, but it is mostly consists of automatic cleanup of junk terms and breaking up of conjunctions, autorewrites of some of the most common simplifications, e.g. $(P/ True = P)$, etc. These simple tools are actually quite useful in reducing the proof code that is spent on trivial tasks.

7.2.3 Simpler Actions

Another technique that we have used to simplify the proofs are simple actions (`sact.v`). These allow us to not define all actions as relations, but instead lift state functions into the action type. We can later detect these lifted simple actions, and automatically simplify them, since we know that the function tells us the final state that can match the relation, thus allowing us to correctly and automatically pick the existential variable.

These simple actions were crucial when we were using automation, as automation could not handle existentials well. Now that we no longer use the extensive automation , they are still helpful as they simplify the proofs.

Chapter 8

Related Work and Conclusion

One of the reasons why formal verification of software is a hard problem is that the natural reasoning that the programmer employs when thinking about the problem differs from the language model over which the program is specified. Moreover, a piece of complex software may involve multiple such models of reasoning.

In this thesis, we have created a new framework for program certification, which allows programmers to certify software using multiple abstraction levels. The framework that we have developed can be used to verify software written for any language or machine model whose operational semantics can be defined as a set of actions. The framework comes with proofs of safety and correctness that are machine independent, meaning that specializing the framework to any machine model can be done quickly and easily.

The framework also has a way to link verified code modules written in different machines in a general way. Our framework creates an explicit definition of refinement, and anything that fits that definition can be used to link the code. To simplify this task, we have defined several refinement constructors that can be used to generate refinements from information about the two machines, for example, from the representation relation in case of the `repr` and invariant refinements, or from projection functions in case of embedding refinements.

These refinement generators can be further specialized to particular machines and machine classes. In our verification of VMM, we have given an example how the C languages that use different memory models can be linked by relating only the memory models and not the whole language, thereby reducing the proof burden.

To show that this framework is sound, we have encoded it into Coq Proof Assistant, and have proved the soundness and correctness theorems presented in this thesis. Using this formalized framework, we have created several abstract machine models that were needed to verify the code of a small virtual memory manager. The code has been split up into modules, each verified over the appropriate abstract machine, and our framework was used to refine and link these modules. The final result was a complete verification of the entire system. By doing so, we believe that we have demonstrated the theoretical and practical value of our framework for verification of software that has code defined at multiple layers of abstraction.

8.1 Related Work

The work presented in this thesis is a continuation of the work on several Hoare-logic based verification frameworks such as SCAP[14] and GCAP[6]. The framework presented in the thesis generalizes and extends both of these frameworks to make them machine and language independent, as well as to simplify them. We have already discussed the improvements in Chapter 3, where we defined our framework.

The refinement system presented in this thesis is a generalization of the work by Andrew McCreight on the certified garbage collector[29]. The work on the garbage collector also uses a `repr`-like linking between two components which are verified on two separate abstract machines. However, that work was not general enough - the abstract machine could not have its own semantics - it was only a projection of the underlying assembly machine. The soundness proof of the refinement in that work is dependent on the individual representations and the definition of the concrete machine. Our work improves on the ideas found in the certified garbage collector by creating a system of refinements that work with any machines. The requirements needed to show that a refinement is valid are independent of any particulars of the machine or code.

Our abstraction approach competes with the methods that arise from the use of separation logic. For example, O'Hearn *et al.*[37, 38] have used separation logic to define frame rules and thereby make proofs more modular and abstract. Parkinson[39] has shown that it is possible to define abstract Java classes through the use of the star operator. Our work can accomplish many of the same goals without relying on separation logic.

Our meta-language approach to verification is similar to the language presented in Abstract Separation Logic by Calcagno *et al.*[7] In that paper, the language described by the authors is designed to represent any imperative language. However, we have taken this definition a step further, by explicitly making these definitions parametric. Thus our verification framework does not just define some ideal language, but is actually parametric on the definition of the abstract machine, allowing us to instantiate our framework for a specific target language. Another difference is that we replaced the pre- and post-condition based specification with action based specification, a move which makes it obvious that the behavior of code and behavior of primitive commands is not different from each other, which in turn makes it clear how we can connect code and primitive operations in our refinements.

The OCAP framework by Feng *et al.*[12] has some of the same goals as this thesis. Its aim was to certify different components under different logics. However, the OCAP approach is limited to changing the logic - it can not introduce such things as abstract state or new operations into the machine itself.

The work by the Verisoft group[16, 40] has many of the same goals and approaches as the work presented in our thesis. We both aim for pervasive verification of OS by doing foundational verification of all components. Both works utilize multiple machines [2, 21], and require linking. And as both projects aim for certification of a kernel, both have to handle VMM[1, 46]. We will highlight a few differences between the approaches.

The Verisoft project has verified a kernel that includes a virtual memory manager that runs on top of idealized and simplified hardware. Their verification works by defining two machines - one for the architecture, which they call VAMP, and one for the user-level code that includes concurrency and virtual memory, which they call CVM. The proof of correctness is a simulation argument that shows that the semantics of CVM are simulated by the microkernel running on top of the VAMP machine.

The authors mention that this simulation argument is one of the most complex proofs in Verisoft. The only modularization of the simulation that was mentioned in their papers is the break up of the relation between abstract states into sub-relations, such as relation between memory, relation between devices, etc. However, it seems that each relation tries to capture all of the abstraction that the kernel is defined in one shot. This has two consequences. First, this seems to imply that modifying one small invariant may potentially require reworking the entire proof. For example, a change in the allocation system will change how the relation between virtual and physical memory functions, and thus may require redoing certification of the code that includes updates of the page tables. The second consequence is that if we were to change something in the definition of one particular component, we have no apparent way reusing the simulation proof. There is no explicit guarantee of separation between different components of the kernel for us to reuse them.

Of course, a well-structured proof will be able to define some modularity. Specific lemmas can be created to define local effects, and these lemmas can be used to generalize parts of the proof. However, such structure is not made apparent. If the person verifying the code is not an expert at creating such proofs, there may no clear modularity, and there is no protection from leaky abstractions within such proofs. We think that using many more abstract machines with clearly defined specification is a better approach. It is not used because the typical ad-hoc definition of such machines can take a significant amount of time, and thus it seems unwise to define a new machine for every small abstraction.

This is where our works begins to show the benefits. Our framework allows us to define new abstract machines and semantics easier. The framework also simplifies the proofs between the machines, reducing the overhead of dealing with different semantics, etc. The idea is that once verification with multiple machines becomes easier to define, the advantages of having these additional abstraction, e.g. modularity and reusability, becomes to outweigh the disadvantages.

Marti *et al.*have published a work on the complete Coq verification of the heap allocator[28]. The work is interesting from the standpoint of a complete verification of a memory manager, however, it does not provide any abstract models of the heap, and thus relies on separation logic as a form of abstraction.

Another work on kernel verification is the L4.verified[23], which has verified a complete L4 microkernel. Other than the fact that they have verified the kernel, their work is not similar in the approaches used in this thesis. Their approach uses a high-level specification of the entire kernel (essentially a specification defined over a single abstract machine), which is then refined down to the system level. The proof that checks that the code correctly implements that abstract data in the specification completes the verification, although it relies on several trusted components such as memory allocator. Because of these differences in the verification strategy, it is not clear how to compare their work to ours. It is even more difficult to compare the handling of virtual memory in the two kernels. The L4 kernel passes through the memory for the user-level, completely ignoring address translation, but carefully ensuring that the page tables updates can never break the kernel. Meanwhile, our work takes the direct approach to creating a VM abstraction. The group is currently working on creating a proper abstraction of the virtual memory and using a complex separation logic[24] to describe it. So far, very little information has been published for us to say anything about it.

The CompCert project[25, 26] aims to build a certified compiler from C to assembly. We are excited to see this work, as we think that the CompCert compiler can be described as a valid refinement in our system. In that case, we could use the CompCert compiler to link C with the assembly code in our system, thereby having a complete certification of the kernel all the way to the hardware.

In order to speed up verifications further, we have attempted to define an extensive automation system for verification of our proofs. Although we were unsuccessful in our attempt, a more systematic and organized approach such as Chlipala's Bedrock[8] could prove useful. However, Bedrock in its current stage of development seems to be very specific to the assembly language, and it may run into difficulties dealing with abstract machines where actions may be non-deterministic.

The abstract state machine approach[18] to analyzing programs is not very common in current code verification frameworks, however, it is very common in model checking[9]. In particular, the B-method[42] includes a way to refine one abstract state machine into another abstract state machine. However, the model checking approach has not been used to certify actual programs, but to ensure that the specifications of algorithms are sound. Nonetheless, with the introduction of tools such as ASMl[19] and Spec#[5], the differences between the approaches are shrinking. We hope that this will further encourage the exchange of software verification techniques leading to future advances.

8.2 Future Work

There are many improvements that can be made to this work. These improvements can be grouped into three categories:

1. Improvements to the verification framework.

The simple improvement here would be to merge the operational semantics and library stubs together, as their types are already similar. Our initial attempt to do this, however, caused some confusion in how the system worked.

Another major improvement to the verification system would be to add lambda terms (and possibly fixpoints) to the meta-language. If these are incorporated, then the meta-language may be able to deal with first-class functions, and thus be able to describe control flow such as function pointer passing. Some of our group's research is moving in that direction.

2. Improvements to the refinements.

   Although many refinements were presented, we have really only used the `repr`-refinement for verification. We have aimed for completion of our goal, and using a single refinement everywhere allowed us to reuse quite a bit of boilerplate code. Once the framework matures further, it is likely that boilerplate code will decrease, as well as it will be more clear when one refinement is to be preferred over another. With this maturity, we would expect that creating specialized refinements will also become easier, thus allowing the programmers to spend less time proving that the linking between their machines are sound.

   We have also not linked the creation of new machines (via machine transformations) and `repr` relation. Right now the refinement and the machine creation are two separate parts of our framework. However, machines can generally reuse parts of each other, and thus it should be possible to use these similarities in the machines to define relations between them automatically. Doing so will allow the programmers to define machines, and the refinements between them in one step, speeding up the verification work.

3. Improvements to the practical aspects of this work.

   This thesis only presents a tiny amount of highly specialized linking and verification. However, the goal of this work is to create a way to reuse code, proof, and abstraction. The way to do this is to define libraries of pre-made refinements, coupled with code and verifications that can be used to quickly define the abstract machine on which to verify any software. Defining such libraries will allow the usefulness of this framework to improve.

Thus, this framework is only the first general step to create abstract and reusable verification methods. More work is needed for this framework to be useful to all the programmers. It is our hope that this framework, or another that allows multi-machine model verification, will eventually develop into something that has practical use, and not just a research project.

Appendix A

Proofs of Properties of Actions

Lemma A.0.1 (Reflexivity)
$\forall \mathtt{a}.\, \mathtt{a} \supseteq \mathtt{a}$

Pf. The domains of $\mathtt{a}$ are equivalent, and for any $\mathbb{S}$, the codomain of $\mathtt{a}\ \mathbb{S}$ is equal to itself.

□

Lemma A.0.2 (Transitivity)
$\forall \mathtt{a}, \mathtt{a}', \mathtt{a}''.\, \mathtt{a} \supseteq \mathtt{a}' \to \mathtt{a}' \supseteq \mathtt{a}'' \to \mathtt{a} \supseteq \mathtt{a}''$

Pf. Assume $\mathtt{a} \supseteq \mathtt{a}'$ and $\mathtt{a}' \supseteq \mathtt{a}''$.
By def. of $\supseteq$, $\mathtt{dom}(\mathtt{a}) \subseteq \mathtt{dom}(\mathtt{a}') \subseteq \mathtt{dom}(\mathtt{a}'')$.
Pick any $\mathbb{S} \in \mathtt{dom}(\mathtt{a}'')$.
By def. of $\supseteq$, $\mathtt{a}''\ \mathbb{S} \subseteq \mathtt{a}'\ \mathbb{S}$ or $\mathbb{S} \notin \mathtt{dom}(\mathtt{a}')$.
If $\mathbb{S} \notin \mathtt{dom}(\mathtt{a}')$, then $\mathbb{S} \notin \mathtt{dom}(\mathtt{a})$, and thus $\mathtt{a} \supseteq \mathtt{a}''$.
If $\mathbb{S} \in \mathtt{dom}(\mathtt{a}')$, then by def of $\supseteq$, either $\mathtt{a}'\ \mathbb{S} \subseteq \mathtt{a}\ \mathbb{S}$ or $\mathbb{S} \notin \mathtt{dom}(\mathtt{a})$.
In either case, $\mathtt{a} \supseteq \mathtt{a}''$.

□

Lemma A.0.3 (Preorder)
The weaker-than relation is a pre-order.

Pf. Direct from Lemma A.0.1 and Lemma A.0.2.

□

Lemma A.0.4 (Precondition Weakening)
$\forall \mathtt{p}, \mathtt{p}'.\, (\mathtt{p} \to \mathtt{p}') \to (\mathtt{p}? \supseteq \mathtt{p}'?)$

Pf. Assume $\mathtt{p}$ and $\mathtt{p}'$ such that $\mathtt{p} \to \mathtt{p}'$.
Arbitrarily choose state $\mathbb{S}$.
There are two cases:

- $\mathtt{p}'\ \mathbb{S}$ is false.
  By our assumptions $\mathtt{p}\ \mathbb{S}$ is also false.
  By definition of $\mathtt{p}?$, $\mathbb{S} \notin \mathtt{dom}(\mathtt{p}?)$ and $\mathbb{S} \notin \mathtt{dom}(\mathtt{p}'?)$.
  This is an acceptable condition for $\mathtt{p}? \supseteq \mathtt{p}'?$.

- $\mathtt{p}'\ \mathbb{S}$ is true.
  Then $\mathbb{S} \in \mathtt{dom}(\mathtt{p}'?)$, and $\mathtt{p}'?\ \mathbb{S} = \{\mathbb{S}\}$
  If $\mathtt{p}\ \mathbb{S}$ is false, then $\mathbb{S} \notin \mathtt{dom}(\mathtt{p}?)$. This is acceptable for $\mathtt{p}? \supseteq \mathtt{p}'?$.
  If $\mathtt{p}\ \mathbb{S}$ is true, then $\mathtt{p}?\ \mathbb{S} = \{\mathbb{S}\}$
  Since $\{\mathbb{S}\} \supseteq \{\mathbb{S}\}$, this condition is also acceptable for $\mathtt{p}? \supseteq \mathtt{p}'?$.

Since $\mathbb{S}$ is arbitrary, the required relation holds for all $\mathbb{S}$.
Thus $\mathtt{p}? \supseteq \mathtt{p}'?$.
□

Lemma A.0.5 (Precondition Weakening 2)

$\forall \mathtt{p}, \mathtt{a}. (\mathtt{p}? \circ \mathtt{a}) \supseteq \mathtt{a}$

Pf. Since $\mathtt{p}?$ is a restriction on domain, $\mathtt{dom}(()\mathtt{p}? \circ \mathtt{a}) \subseteq \mathtt{dom}(\mathtt{a})$.
Since $\mathtt{p}$ is only a restriction on domain, for any $\mathbb{S} \in \mathtt{dom}(()\mathtt{p}? \circ \mathtt{a})$, $(\mathtt{p}? \circ \mathtt{a})\ \mathbb{S} = \mathtt{a}\ \mathbb{S}$.
By def. of $\supseteq$, $(\mathtt{p}? \circ \mathtt{a}) \supseteq \mathtt{a}$.

□

Lemma A.0.6 (Strongest Action)

$\forall \mathtt{a}. (\mathtt{a} \supseteq \mathit{loop})$

Pf. Since any state $\mathbb{S}$ is in the domain of *loop*, we know $\mathtt{dom}(\mathtt{a}) \subseteq \mathtt{dom}(\mathit{loop})$.
Since for any $\mathbb{S}$, $\mathit{loop}\ \mathbb{S} = \emptyset$, then for any $\mathbb{S} \in \mathtt{dom}(\mathtt{a})$, $\mathtt{a}\ \mathbb{S} \supseteq \mathit{loop}\ \mathbb{S}$.
By def of $\supseteq$, $\mathtt{a} \supseteq \mathit{loop}$.

□

Lemma A.0.7 (Composition Associativity)

$\forall \mathtt{a}, \mathtt{a}', \mathtt{a}''. \mathtt{a} \circ (\mathtt{a}' \circ \mathtt{a}'') \cong (\mathtt{a} \circ \mathtt{a}') \circ \mathtt{a}''$

Pf. This proof is very tedious, so we will refer the reader to Coq implementation.

□

Lemma A.0.8 (Action Composition Weakening)

For all actions $\mathtt{a}_1$, $\mathtt{a}'_1$, $\mathtt{a}_2$, and $\mathtt{a}'_2$, if $\mathtt{a}_1 \supseteq \mathtt{a}'_1$ and $\mathtt{a}_2 \supseteq \mathtt{a}'_2$, then $(\mathtt{a}_1 \circ \mathtt{a}_2) \supseteq (\mathtt{a}'_1 \circ \mathtt{a}'_2)$.

Pf. Chose an arbitrary state $\mathbb{S}$.
If $\mathbb{S} \in \mathtt{dom}(\mathtt{a}_1 \circ \mathtt{a}_2)$, then we know $\mathbb{S} \in \mathtt{dom}(\mathtt{a}_1)$, and that for any $\mathbb{S}' \in (\mathtt{a}_1\ \mathbb{S})$, $\mathbb{S}' \in \mathtt{dom}(\mathtt{a}_2)$.
By $\mathtt{a}_1 \supseteq \mathtt{a}'_1$, we know that $\mathbb{S} \in \mathtt{dom}(\mathtt{a}'_1)$, and that $\mathbb{S}' \in (\mathtt{a}'_1\ \mathbb{S})$
By $\mathtt{a}_2 \supseteq \mathtt{a}'_2$, we know that $\mathbb{S}' \in \mathtt{dom}(\mathtt{a}'_2)$.
Thus, we know that $\mathbb{S} \in \mathtt{dom}(\mathtt{a}'_1 \circ \mathtt{a}'_2)$, and thus $\mathtt{dom}(\mathtt{a}_1 \circ \mathtt{a}_2) \subseteq \mathtt{dom}(\mathtt{a}'_1 \circ \mathtt{a}'_2)$.
Pick any $\mathbb{S}''$ such that $\mathbb{S}'' \in ((\mathtt{a}'_1 \circ \mathtt{a}'_2)\ \mathbb{S})$.
Then there exists $\mathbb{S}'$ such that $\mathbb{S}' \in (\mathtt{a}'_1\ \mathbb{S})$ and $\mathbb{S}'' \in (\mathtt{a}'_2\ \mathbb{S}')$.
By $\mathtt{a}_1 \supseteq \mathtt{a}'_1$ and $\mathtt{a}_2 \supseteq \mathtt{a}'_2$, we know that $\mathbb{S}' \in (\mathtt{a}_1\ \mathbb{S})$ and $\mathbb{S}'' \in (\mathtt{a}_2\ \mathbb{S}')$.
Thus $\mathbb{S}'' \in ((\mathtt{a}_1 \circ \mathtt{a}_2)\ \mathbb{S})$.
Hence for all $\mathbb{S}$, $(\mathtt{a}_1 \circ \mathtt{a}_2)\ \mathbb{S} \supseteq (\mathtt{a}'_1 \circ \mathtt{a}'_2)\ \mathbb{S}$.
Thus $(\mathtt{a}_1 \circ \mathtt{a}_2) \supseteq (\mathtt{a}'_1 \circ \mathtt{a}'_2)$.

□

Lemma A.0.9 (Action Choice Weakening)

For all actions $\mathtt{a}_1$, $\mathtt{a}'_1$, $\mathtt{a}_2$, and $\mathtt{a}'_2$, if $\mathtt{a}_1 \supseteq \mathtt{a}'_1$ and $\mathtt{a}_2 \supseteq \mathtt{a}'_2$, then $(\mathtt{a}_1 \oplus \mathtt{a}_2) \supseteq (\mathtt{a}'_1 \oplus \mathtt{a}'_2)$.

Pf. Pick arbitrary state $\mathbb{S}$.
If $\mathbb{S} \in \mathtt{dom}(\mathtt{a}_1 \oplus \mathtt{a}_2)$, then it must be either in domain of $\mathtt{a}_1$ or $\mathtt{a}_2$.
By def of $\supseteq$, $\mathbb{S} \in \mathtt{dom}(\mathtt{a}')_1$ or $\mathbb{S} \in \mathtt{dom}(\mathtt{a}')_2$.
Therefore $\mathbb{S} \in \mathtt{dom}(\mathtt{a}_1' \oplus \mathtt{a}_2')$.
Pick any $\mathbb{S}' \in (\mathtt{a}_1' \oplus \mathtt{a}_2')\ \mathbb{S}$.
Then if $\mathbb{S} \in \mathtt{dom}(\mathtt{a}_1')$ then $\mathbb{S}' \in \mathtt{a}_1'\ \mathbb{S}$ and if $\mathbb{S} \in \mathtt{dom}(\mathtt{a}_2')$ then $\mathbb{S}' \in \mathtt{a}_2'\ \mathbb{S}$.
Therefore, if $\mathbb{S} \in \mathtt{dom}(\mathtt{a}_1)$ then $\mathbb{S}' \in \mathtt{a}_1'\ \mathbb{S}$ and if $\mathbb{S} \in \mathtt{dom}(\mathtt{a}_2)$ then $\mathbb{S}' \in \mathtt{a}_2'\ \mathbb{S}$.
Therefore, if $\mathbb{S} \in \mathtt{dom}(\mathtt{a}_1)$ then $\mathbb{S}' \in \mathtt{a}_1\ \mathbb{S}$ and if $\mathbb{S} \in \mathtt{dom}(\mathtt{a}_2)$ then $\mathbb{S}' \in \mathtt{a}_2\ \mathbb{S}$.
Thus $\mathbb{S}' \in \mathtt{a}_1 \oplus \mathtt{a}_2\ \mathbb{S}$.
Thus $\mathtt{dom}(\mathtt{a}_1 \oplus \mathtt{a}_2) \subseteq \mathtt{dom}(\mathtt{a}_1' \oplus \mathtt{a}_2')$ and $(\mathtt{a}_1 \oplus \mathtt{a}_2)\ \mathbb{S} \supseteq (\mathtt{a}_1' \oplus \mathtt{a}_2')\ \mathbb{S}$.
Hence $(\mathtt{a}_1 \oplus \mathtt{a}_2) \supseteq (\mathtt{a}_1' \oplus \mathtt{a}_2')$.
□

Lemma A.0.10 (Branch Weakening)
For all actions $\mathtt{a}_1$, $\mathtt{a}_1'$, $\mathtt{a}_2$, and $\mathtt{a}_2'$, if $\mathtt{a}_1 \supseteq \mathtt{a}_1'$ and $\mathtt{a}_2 \supseteq \mathtt{a}_2'$, then $(\mathtt{p}?\ \mathtt{a}_1 \oplus \mathtt{a}_2) \supseteq \left(\mathtt{p}?\ \mathtt{a}_1' \oplus \mathtt{a}_2'\right)$.

Pf. Pick arbitrary state $\mathbb{S}$.
If $\mathtt{p}\ \mathbb{S}$, then the left branch is taken in both sides. But then we know that $\mathtt{a}_1 \supseteq \mathtt{a}_1'$.
Similarly, if it is not the case that $\mathtt{p}\ \mathbb{S}$, then the right branch is taken, and we use $\mathtt{a}_2 \supseteq \mathtt{a}_2'$ to get the result.

□

Lemma A.0.11 (Composition Equivalence)
$\forall \mathtt{a}_1, \mathtt{a}_1', \mathtt{a}_2, \mathtt{a}_2'.\ \mathtt{a}_1 \cong \mathtt{a}_1' \to \mathtt{a}_2 \cong \mathtt{a}_2' \to (\mathtt{a}_1 \circ \mathtt{a}_2) \cong (\mathtt{a}_1' \circ \mathtt{a}_2')$

Pf. Corollary of Composition Weakening (Lemma A.0.8).

□

Lemma A.0.12 (Choice Equivalence)
$\forall \mathtt{a}_1, \mathtt{a}_1', \mathtt{a}_2, \mathtt{a}_2'.\ \mathtt{a}_1 \cong \mathtt{a}_1' \to \mathtt{a}_2 \cong \mathtt{a}_2' \to (\mathtt{a}_1 \oplus \mathtt{a}_2) \cong (\mathtt{a}_1' \oplus \mathtt{a}_2')$

Pf. Corollary of Choice Weakening (Lemma A.0.9).

□

Lemma A.0.13 (Branch Equivalence)
$\forall \mathtt{a}_1, \mathtt{a}_1', \mathtt{a}_2, \mathtt{a}_2'.\ \mathtt{a}_1 \cong \mathtt{a}_1' \to \mathtt{a}_2 \cong \mathtt{a}_2' \to ((\mathtt{p}?\ \mathtt{a}_1 \oplus \mathtt{a}_2)) \cong (\left(\mathtt{p}?\ \mathtt{a}_1' \oplus \mathtt{a}_2'\right))$

Pf. Corollary of Branch Weakening (Lemma A.0.10).

□

Lemma A.0.14 (Conjunction Weakening)
$\forall \mathtt{a}, \mathtt{a}', \mathtt{a}''.\ \mathtt{a} \supseteq \mathtt{a}' \to (\mathtt{a} \wedge \mathtt{a}'') \supseteq (\mathtt{a}' \wedge \mathtt{a}'')$

Pf. Pick any $\mathbb{S}$.
If $\mathbb{S} \in \mathtt{dom}((\mathtt{a} \wedge \mathtt{a}''))$, then $\mathbb{S} \in \mathtt{dom}(\mathtt{a})$ and $\mathbb{S} \in \mathtt{dom}(\mathtt{a}'')$.
By $\mathtt{a} \supseteq \mathtt{a}'$, we know that $\mathbb{S} \in \mathtt{dom}(\mathtt{a}')$.
By def of $\wedge$, $\mathbb{S} \in \mathtt{dom}((\mathtt{a}' \wedge \mathtt{a}''))$.
Hence $\mathtt{dom}(\mathtt{a} \wedge \mathtt{a}'') \subseteq \mathtt{dom}(\mathtt{a}' \wedge \mathtt{a}'')$.
Pick any $\mathbb{S}' \in ((\mathtt{a}' \wedge \mathtt{a}'')\ \mathbb{S})$.
By def of $\wedge$, $\mathbb{S}' \in (\mathtt{a}'\ \mathbb{S})$ and $\mathbb{S}' \in (\mathtt{a}''\ \mathbb{S})$.
By $\mathtt{a} \supseteq \mathtt{a}'$, $\mathbb{S}' \in (\mathtt{a}\ \mathbb{S})$.
By def of $\wedge$, $\mathbb{S}' \in ((\mathtt{a} \wedge \mathtt{a}'')\ \mathbb{S}$.

Hence $((\mathtt{a} \wedge \mathtt{a}'') \, \mathbb{S}) \supseteq ((\mathtt{a}' \wedge \mathtt{a}'') \, \mathbb{S})$.
Thus $(\mathtt{a} \wedge \mathtt{a}'') \supseteq (\mathtt{a}' \wedge \mathtt{a}'')$. □

Bibliography

[1] E. Alkassar, N. Schirmer, and A. Starostin. Formal pervasive verification of a paging mechanism. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 109–123, 2008.

[2] Eyad Alkassar, Mark A. Hillebrand, Dirk C. Leinenbach, Norbert W. Schirmer, Artem Starostin, and Alexandra Tsyban. Balancing the load: Leveraging a semantics stack for systems verification. *Journal of Automated Reasoning: Operating System Verification*, 42(2–4):389–454, 2009.

[3] Andrew W. Appel. Foundational proof-carrying code. In *Proc. 16th IEEE Symposium on Logic in Computer Science*, pages 247–258. IEEE Computer Society, June 2001.

[4] M. Barnett, R. DeLine, M. Fähndrich, K.R.M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[5] M. Barnett, K.R.M. Leino, and W. Schulte. The spec# programming system: An overview. *Construction and analysis of safe, secure, and interoperable smart devices*, pages 49–69, 2005.

[6] Hongxu Cai, Zhong Shao, and Alexander Vaynberg. Certified self-modifying code. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 66–77, New York, NY, USA, 2007. ACM.

[7] C. Calcagno, P.W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *Proc. LICS'07*, pages 366–378, July 2007.

[8] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 234–245, New York, NY, USA, 2011. ACM.

[9] E. Clarke. Model checking. In *Foundations of software technology and theoretical computer science*, pages 54–56. Springer, 1997.

[10] Kevin Elphinstone, Gerwin Klein, Philip Derrin, Timothy Roscoe, and Gernot Heiser. Towards a practical, verified kernel. In *Proc. 11th Workshop on Hot Topics in Operating Systems*, San Diego, CA, USA, May 2007.

[11] N. Falliere, L.O. Murchu, and E. Chien. W32. stuxnet dossier. *Symantec Security Response,[Online], Accessed*, 14, 2010.

[12] Xinyu Feng, Zhaozhong Ni, Zhong Shao, and Yu Guo. An open framework for foundational proof-carrying code. In *TLDI'07*, pages 67–78, 2007.

[13] Xinyu Feng and Zhong Shao. Modular verification of concurrent assembly code with dynamic thread creation and termination. In *Proc. ICFP'05*, pages 254–267, 2005.

[14] Xinyu Feng, Zhong Shao, Alexander Vaynberg, Sen Xiang, and Zhaozhong Ni. Modular verification of assembly code with stack-based control abstractions. In *PLDI'06*, pages 401–414, 2006.

[15] Xinyu Feng, Zhong Shao, Alexander Vaynberg, Sen Xiang, and Zhaozhong Ni. Modular verification of assembly code with stack-based control abstractions. In *PLDI'06*, pages 401–414, June 2006.

[16] Mauro Gargano, Mark A. Hillebrand, Dirk Leinenbach, and Wolfgang J. Paul. On the correctness of operating system kernels. In *TPHOLs'05*, 2005.

[17] L. Gu, A. Vaynberg, B. Ford, Z. Shao, and D. Costanzo. Certikos: A certified kernel for secure cloud computing. In *Proceedings of 2nd ACM SIGOPS Asia-Pacific Workshop on Systems (APSys'11)*. ACM, 2011.

[18] Y. Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Transactions on Computational Logic (TOCL)*, 1(1):77–111, 2000.

[19] Y. Gurevich, B. Rossman, and W. Schulte. Semantic essence of asml. *Theoretical Computer Science*, 343(3):370–412, 2005.

[20] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, October 1969.

[21] Thomas In der Rieden. *Verified Linking for Modular Kernel Verification*. PhD thesis, Saarland University, Computer Science Department, November 2009.

[22] S. Jones, T. Nordin, and D. Oliva. C–: A portable assembly language. *Implementation of Functional Languages*, pages 1–19, 1998.

[23] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *Proc. SOSP'09*, pages 207–220, 2009.

[24] Rafal Kolanski and Gerwin Klein. Mapped separation logic. In *Proc. VSTTE'08*, pages 15–29. Springer-Verlag, 2008.

[25] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Prin. of Prog. Lang. (POPL'06)*, pages 42–54, New York, NY, USA, 2006. ACM Press.

[26] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[27] Jochen Liedtke. On micro-kernel construction. In *Proc. ACM Symposium on Operating Systems Principles (SOSP'95)*, pages 237–250, 1995.

[28] N. Marti, R. Affeldt, and A. Yonezawa. Formal verification of the heap manager of an operating system using separation logic. *Formal Methods and Software Engineering*, pages 400–419, 2006.

[29] Andrew McCreight, Zhong Shao, Chunxiao Lin, and Long Li. A general framework for certifying garbage collectors and their mutators. In *Proc. PLDI'07*, pages 468–479, 2007.

[30] David Moore, Colleen Shannon, and k Claffy. Code-red: a case study on the spread and victims of an internet worm. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, IMW '02, pages 273–284, New York, NY, USA, 2002. ACM.

[31] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: a realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, May 1999.

[32] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Proc. 1998 Int'l Workshop on Types in Compilation: LNCS Vol 1473*, pages 28–52. Springer-Verlag, 1998.

[33] George Necula. Proof-carrying code. In *Proc. 24th ACM Symp. on Principles of Prog. Lang.*, pages 106–119. ACM Press, January 1997.

[34] George Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proc. 2nd USENIX Symp. on Operating System Design and Impl.*, pages 229–243, 1996.

[35] Zhaozhong Ni and Zhong Shao. Certified assembly programming with embedded code pointers. In *POPL'06*, pages 320–333, 2006.

[36] Zhaozhong Ni, Dachuan Yu, and Zhong Shao. Using XCAP to certify realistic systems code: Machine context management. In *TPHOLs'07*, 2007.

[37] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *POPL'04*, pages 268–280, January 2004.

[38] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. *ACM Trans. Program. Lang. Syst.*, 31(3):1–50, 2009.

[39] M. Parkinson and G. Bierman. Separation logic and abstraction. In *ACM SIGPLAN Notices*, volume 40, pages 247–258. ACM, 2005.

[40] Wolfgang Paul, Manfred Broy, and Thomas In der Rieden. The verisoft project. URL: `http://www.verisoft.de`, 2006.

[41] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. LICS'02*, pages 55–74, July 2002.

[42] S. Schneider. *The B-method: an introduction*. Palgrave, 2001.

[43] D. Seal. *ARM architecture reference manual*. Pearson Education, 2000.

[44] M. Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1996.

[45] G. Slabodkin. Software glitches leave navy smart ship dead in the water. *Government Computer News*, 13:33727–1, 1998.

[46] Artem Starostin. *Formal Verification of Demand Paging*. PhD thesis, Saarland University, Computer Science Department, March 2010.

[47] Harvey Tuch, Gerwin Klein, and Gernot Heiser. Os verification — now! In *Proc. HotOS-X*, June 2005.

[48] Dachuan Yu, Nadeem A. Hamid, and Zhong Shao. Building certified libraries for PCC: Dynamic storage allocation. In *Proc. 2003 European Symposium on Programming (ESOP'03)*, April 2003.