Compositional Verification of a Baby Virtual Memory Manager

Alexander Vaynberg and Zhong Shao

Yale University

Abstract. A virtual memory manager (VMM) is a part of an operating system that provides the rest of the kernel with an abstract model of memory. Although small in size, it involves complicated and interdependent invariants that make monolithic verification of the VMM and the kernel running on top of it difficult. In this paper, we make the observation that a VMM is constructed in layers: physical page allocation, page table drivers, address space API, etc., each layer providing an abstraction that the next layer utilizes. We use this layering to simplify the verification of individual modules of VMM and then to link them together by composing a series of small refinements. The compositional verification also supports function calls from less abstract layers into more abstract ones, allowing us to simplify the verification of initialization functions as well. To facilitate such compositional verification, we develop a framework that assists in creation of verification systems for each layer and refinements between the layers. Using this framework, we have produced a certification of BabyVMM, a small VMM designed for simplified hardware. The same proof also shows that a certified kernel using BabyVMM's virtual memory abstraction can be refined following a similar sequence of refinements, and can then be safely linked with BabyVMM. Both the verification framework and the entire certification of BabyVMM have been mechanized in the Coq Proof Assistant.

1 Introduction

Software systems are complex feats of engineering. What makes them possible is the ability to isolate and abstract modules of the system. In this paper, we consider an operating system kernel that uses virtual memory. The majority of the kernel makes an assumption that the memory is a large space with virtual addresses and a specific interface that allows the kernel to request access to any particular page in this large space. In reality, this entire model of memory is in the imagination of the programmer, supported by a relatively small but important portion of the kernel called the virtual memory manager. The job of the virtual memory manager is to handle all the complexities of the real machine architecture to provide the primitives that the rest of the kernel can use. This is exactly how the programmer would reason about this software system.

However, when we consider verification of such code, current approaches are mostly monolithic in nature. Abstraction is generally limited to abstract data types, but such abstraction can not capture changes in the semantics of computation. For example, it is impossible to use abstract data types to make virtual memory appear to work like physical memory without changing operational semantics. To create such abstraction, a change of computational model is required. In the Verisoft project[11, 18], the abstract virtual memory is defined by creating the CVM model from VAMP architecture. In AIM[7], multiple machines are used to define interrupts in the presence of a scheduler.

These transitions to more abstract models of computation tend to be quite rare, and when present tend to be complex. The previously mentioned VAMP-CVM jump in Verisoft abstracts most of kernel functionality in one step. In our opinion, it would be better to have more abstract computation models, with smaller jumps in abstraction. First, it is easier to verify code in the most abstract computational model possible. Second, smaller abstractions tend to be easier to prove and to maintain, while larger abstractions can be still achieved by composing the smaller ones. Third, more abstractions means more modularity; changes in the internals of one module will not have global effects.

However, we do not commonly see Hoare-logic verification that encourages multiple models. The likely reason is that creating abstract models and linking across them is seen as ad-hoc and tedious additional work. In this paper we show how to reduce the effort required to define models and linking, so that code verification using multiple abstractions becomes an effective approach. More precisely, our paper makes the following contributions:

- We present a framework for quickly defining multiple abstract computational models and their verification systems.
- We show how our framework can be used to define safe cross-abstraction linking.
- We show how to modularize a virtual memory manager and define abstract computational models for each layer of VMM.
- We show a complete verification of a small proof-of-concept virtual memory manager using the Coq Proof Assistant.

The rest of this paper is organized as follows. In Section 2, we give an informal overview of our work. In Section 3, we discuss the formal details of our verification and refinement framework. In Section 4, we specialize the framework for a simple C-like language. In Section 5, we certify BabyVMM, our small virtual memory manager. Section 6 discusses the Coq proof, and Section 7 presents related work and concludes.

2 Overview and Plan for Certification

We begin the overview by explaining the design of BabyVMM, our small virtual memory manager. First, consider the model of memory present in simplified hardware (Figure 1). The memory is a storage system, which contains cells that can be read from or written to by the software. These cells are indexed by addresses. However, to facilitate indirection, the hardware includes a system called address translation (AT), which, when enabled, will cause all requests for specific addresses from the software to be translated. The AT system adds special registers to the memory system - one to enable or disable AT, and the other to point where the software-managed AT tables are located in memory. The fact that these tables are stored in memory is one of the sources of complexity in the AT system - updating AT tables requires updating in-memory tables, a process which goes through AT as well.

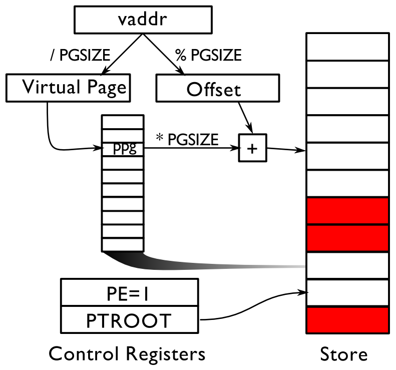

Fig. 1. Address Translated Hardware (HW) Model of Memory

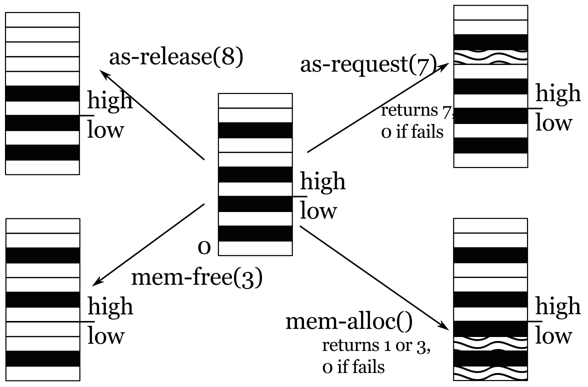

Fig. 2. Address Space (AS) Model of Memory

Because AT is such a complicated, machine-dependent, and general mechanism, BabyVMM creates an abstraction that defines specific restrictions on how AT will be used, and presents a simpler view of AT to the kernel. Although the abstract models of memory may differ depending on the features that the kernel may require, BabyVMM defines a very basic model, to which we refer as the address space (AS) model of memory (Figure 2). The AS model replaces the small physical memory with a larger virtual address space with allocatable pages and no address translation. The space is divided into high and low areas, where the low area is actually a window into physical memory (a pattern common in many kernels). Because of this distinction, the memory model has two sets of allocation functions, one for the "high" memory area where the programmer requests a specific page for allocation, and one for the "low" memory area, where the programmer can not pick which page to allocate.

The virtual memory manager is the system that creates such an abstraction. We have implemented a small VMM, which we call BabyVMM. Its code is given in Appendix A, with the diagram of functions and function calls shown in Figure 3.

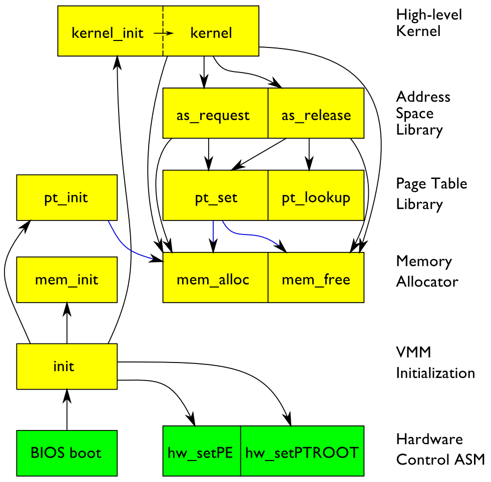

Fig. 3. Architecture of BabyVMM

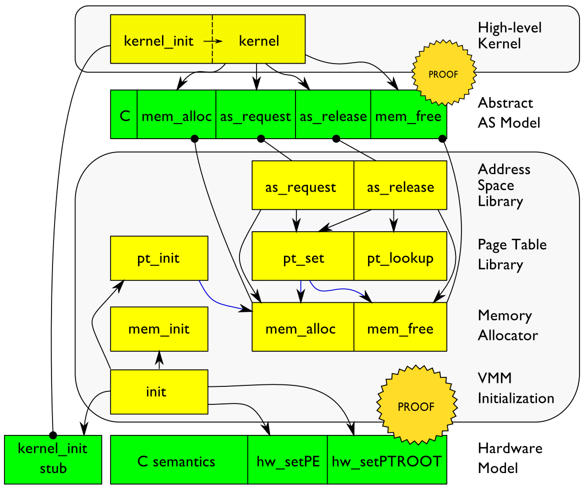

Fig. 4. Kernel Certified in the Abstraction Provided by BabyVMM

Since the address space model of memory is simpler to reason about, we would like to have the rest of the kernel certified over it, and then link the certified kernel with BabyVMM, which we will certify over the actual hardware. The plan of this certification can be seen in Figure 4. The green boxes are assumptions of the primitives, while yellow is the code. Notice how function calls (arrows) no longer go through one module to another, but instead go to the primitives that abstract the code. However, there are new connections (lines with circles) between these primitives and code, which are cross-abstraction links. These links are the tricky part, as we will have to show that the specification in one model is compatible with the implementation in another. An additional tricky part is the call of `kernel_init` from `init`, which requires a call from

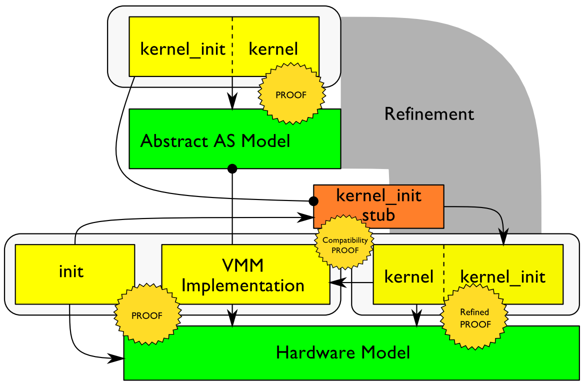

Fig. 5. Diagram of Refinement

the code in the more concrete model of memory to call a function that works over the abstract model of memory.

In practice, the way this works can be explained by the diagram in Figure 5. From the relation between the AS and Hardware models, we construct a refinement. The refinement is a set of properties over the relation between the abstract and concrete models that the programmer has to provide. Once the refinement is established, from its definition our framework creates a specification translator, that will convert any specification over the AS model into the specification over the hardware model of memory. Moreover, the refinement will guarantee that the entire kernel remains sound under the translated specification, when the calls to abstract primitives of the AS model are replaced with calls to the actual functions of VMM, which we have shown to be compatible.

This means that in the diagram, the certification of the kernel is an immediate result of the refinement, with no additional proof obligations. The proof of compatibility of the kernel and the VMM implementation is just the proofs of implementation between the functions and the primitives they implement.

The upcall of the `init` to the `kernel_init` is handled in a similar way. Instead of proving that we can safely make the call to `kernel_init`, we create a stub with a particular specification. We then show that the refined specification of the `kernel_init` is compatible with the specification of the stub, which means that it is safe to replace the call to the stub with the call to the actual kernel.

The above presented a general idea of how we certify systems with refinement in the PCC framework, where we can use two machines. Our technique however is compositional, allowing us to make smaller and simpler steps. Instead of treating entire VMM as a single abstraction, we can break it up into individual layers, such as memory allocator, hardware page table manager, and a library that provides `as_request` and `as_release` API to the high-level kernel. Moreover, we can simplify the certification of certain pieces by creating restricted subsets of the hardware. In the example of the kernel, the allocator initialization and the page table initialization functions all run only when the hardware does not have AT enabled, and thus we can restrict the hardware to remove AT.

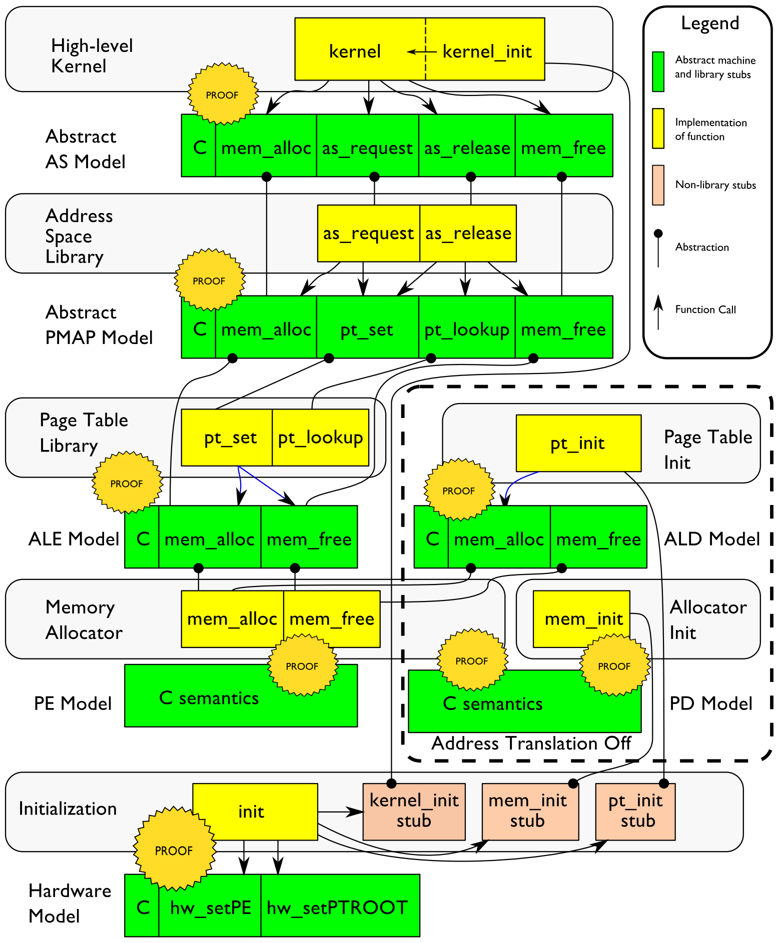

Fig. 6. Complete Plan for VMM Certification

$$
\begin{array}{rcl@{\qquad}rcl}
(\mathit{State})\ \mathbb{S} & \in & \Sigma & (\mathit{State\ Predicate})\ \mathtt{p} & \in & \Sigma \to \mathit{Prop} \\
(\mathit{Operation})\ \iota & \in & \Delta & (\mathit{State\ Relation})\ \mathtt{g} & \in & \Sigma \to \Sigma \to \mathit{Prop} \\
(\mathit{Cond})\ b & \in & \beta & (\mathit{Operational\ Semantics})\ \mathtt{OS} & \in & \{\iota \rightsquigarrow (\mathtt{p},\mathtt{g})\}^* \\
(\mathit{CondInterp})\ \Upsilon & \in & \beta \to \Sigma \to \mathit{Prop} & (\mathit{Language\,/\,Machine})\ \mathcal{M} & \in & (\Sigma,\Delta,\beta,\Upsilon,\mathtt{OS})
\end{array}
$$

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

Fig. 7. Abstract State Machine

Thus, when we consider these simplifications in turn, we come up with the complete plan for certification of the kernel and the VMM implementation shown in Figure 6, which we will now put into action.

$$
\begin{array}{ll}
\mathit{id} & \triangleq (\lambda\mathbb{S}.\,\mathsf{True}, \quad \lambda\mathbb{S}.\lambda\mathbb{S}'.\,\mathbb{S}' = \mathbb{S}) \\
\mathit{fail} & \triangleq (\lambda\mathbb{S}.\,\mathsf{False}, \quad \lambda\mathbb{S}.\lambda\mathbb{S}'.\,\mathsf{False}) \\
\mathit{loop} & \triangleq (\lambda\mathbb{S}.\,\mathsf{True}, \quad \lambda\mathbb{S}.\lambda\mathbb{S}'.\,\mathsf{False}) \\
(\mathtt{p},\mathtt{g}) \circ (\mathtt{p}',\mathtt{g}') & \triangleq (\lambda\mathbb{S}.\,\mathtt{p}\;\mathbb{S} \wedge \forall\mathbb{S}'.\,\mathtt{g}\;\mathbb{S}\;\mathbb{S}' \rightarrow \mathtt{p}'\;\mathbb{S}', \quad \lambda\mathbb{S}.\lambda\mathbb{S}''.\,\exists\mathbb{S}'.\,\mathtt{g}\;\mathbb{S}\;\mathbb{S}' \wedge \mathtt{g}'\;\mathbb{S}'\;\mathbb{S}'') \\
(\mathtt{p},\mathtt{g}) \underset{c}{\oplus} (\mathtt{p}',\mathtt{g}') & \triangleq (\lambda\mathbb{S}.\,(\mathtt{p}\;\mathbb{S} \wedge c\;\mathbb{S}) \vee (\mathtt{p}'\;\mathbb{S} \wedge \neg c\;\mathbb{S}), \lambda\mathbb{S}.\lambda\mathbb{S}'.\,(c\;\mathbb{S} \wedge \mathtt{g}\;\mathbb{S}\;\mathbb{S}') \vee (\neg c\;\mathbb{S} \wedge \mathtt{g}'\;\mathbb{S}\;\mathbb{S}')) \\
(\mathtt{p},\mathtt{g}) \supseteq (\mathtt{p}',\mathtt{g}') & \triangleq \forall\mathbb{S}.\,\mathtt{p}\;\mathbb{S} \rightarrow \mathtt{p}'\;\mathbb{S} \wedge \forall\mathbb{S},\mathbb{S}'.\,\mathtt{g}'\;\mathbb{S}\;\mathbb{S}' \rightarrow \mathtt{g}\;\mathbb{S}\;\mathbb{S}'
\end{array}
$$

Fig. 8. Combinators and Properties of Actions

$$
\begin{array}{rlll}
(\textit{Meta-program}) & \mathbb{P} & ::= (\mathbb{C},\mathbb{I}) & \\
(\textit{Proc}) & \mathbb{I} & ::= \mathtt{nil} \mid \iota \mid [\mathtt{l}] \mid \mathbb{I}_1;\mathbb{I}_2 & \\
 & & \quad\mid (b?\;\mathbb{I}_1 + \mathbb{I}_2) & \\
(\textit{Proc Heap}) & \mathbb{C} & ::= \{\mathtt{l} \rightsquigarrow \mathbb{I}\}^* & \\
(\textit{Labels}) & \mathtt{l} & ::= n \text{ (nat numbers)} & \\
(\textit{Spec Heap}) & \Psi,\mathcal{L} & ::= \{\mathtt{l} \rightsquigarrow (\mathtt{p},\mathtt{g})\}^* &
\end{array}
\qquad
\begin{array}{ll}
[\![\mathbb{C},\mathtt{a}]\!]^0_{\mathcal{M}} & := \mathit{loop} \\
[\![\mathbb{C},\mathtt{nil}]\!]^n_{\mathcal{M}} & := \mathit{id} \\
[\![\mathbb{C},\iota]\!]^n_{\mathcal{M}} & := (\mathcal{M}(\iota)) \\
[\![\mathbb{C},[\mathtt{l}]]\!]^n_{\mathcal{M}} & := [\![\mathbb{C},\mathbb{C}(\mathtt{l})]\!]^{n-1}_{\mathcal{M}} \\
[\![\mathbb{C},\mathbb{I};\mathbb{I}']\!]^n_{\mathcal{M}} & := [\![\mathbb{C},\mathbb{I}]\!]^n_{\mathcal{M}} \circ [\![\mathbb{C},\mathbb{I}]\!]^n_{\mathcal{M}} \\
[\![\mathbb{C},(b?\;\mathbb{I}_1+\mathbb{I}_2)]\!]^n_{\mathcal{M}} & := [\![\mathbb{C},\mathbb{I}_1]\!]^n_{\mathcal{M}} \underset{\mathcal{M}(b)}{\oplus} [\![\mathbb{C},\mathbb{I}_2]\!]^n_{\mathcal{M}}
\end{array}
$$

Fig. 9. Syntax and Semantics of the Meta-Language

3 Certifying with Refinement

Our framework for multi-machine certification is defined in two parts. First, we create a machine-independent verification framework that will allow us to define quickly and easily as many machines for verification as we need. Second, we will develop our notion of refinements which will allow us to link all the separate machines together.

3.1 A Machine-Independent Certification Framework

Our Hoare-logic based framework is parametric over the definition of operational semantics of the machine, and is sound no matter what machine semantics it is parameterized with. To begin defining such a framework, we first need to understand what exactly is a machine on which we can certify code. The definition that we use is given in Figure 7. Our notion of the machine consists of the following parts:

– State type (Σ). Define the set of all possible states in a machine.
– Operations (Δ). This is a set of names of all operations that the machine supports. The set can be infinite, and defined parametrically.
– Conditionals (β). Defines a type of expressions that are used for branching.
– Conditional Interpreter (Υ). Converts conditionals into state predicates.
– The operational semantics OS. This is the main portion of the machine definition. It is a set of actions $(\mathtt{p},\mathtt{g})$ named by all operations in the machine.

The most important bit of information in the machine are the semantics (OS). The semantics of operations are defined by a precondition ($\mathtt{p}$), which shows when the operation is safe to execute, and by a state relation ($\mathtt{g}$) that defines the set of possible states that the operation may result in. We will refer to the pair of $(\mathtt{p},\mathtt{g})$ as an action of the operation. Later we will also use actions to define the specification of programs.

$$\frac{\forall \mathtt{l} \in \mathtt{dom}(\mathbb{C}).\, \mathcal{M}, \Psi \cup \mathcal{L} \vdash \mathbb{C}(\mathtt{l}) : \Psi(\mathtt{l})}{\mathcal{M}, \mathcal{L} \vdash \mathbb{C} : \Psi}\ (\textsc{code}) \qquad \frac{\mathcal{M}, \Psi \vdash \mathbb{I} : (\mathtt{p}', \mathtt{g}') \quad (\mathtt{p}, \mathtt{g}) \supseteq (\mathtt{p}', \mathtt{g}')}{\mathcal{M}, \Psi \vdash \mathbb{I} : (\mathtt{p}, \mathtt{g})}\ (\textsc{weak})$$

$$\frac{\mathcal{M}, \Psi \vdash \mathbb{I}' : (\mathtt{p}', \mathtt{g}') \quad \mathcal{M}, \Psi \vdash \mathbb{I}'' : (\mathtt{p}'', \mathtt{g}'')}{\mathcal{M}, \Psi \vdash (b?\, \mathbb{I}' + \mathbb{I}'') : (\mathtt{p}', \mathtt{g}') \underset{\mathcal{M}(b)}{\oplus} (\mathtt{p}'', \mathtt{g}'')}\ (\textsc{split}) \quad \frac{\mathcal{M}, \Psi \vdash \mathbb{I}' : (\mathtt{p}', \mathtt{g}') \quad \mathcal{M}, \Psi \vdash \mathbb{I}'' : (\mathtt{p}'', \mathtt{g}'')}{\mathcal{M}, \Psi \vdash \mathbb{I}';\mathbb{I}'' : ((\mathtt{p}', \mathtt{g}') \circ (\mathtt{p}'', \mathtt{g}''))}\ (\textsc{seq})$$

$$\frac{}{\mathcal{M}, \Psi \vdash \iota : \mathcal{M}(\iota)}\ (\textsc{perf}) \qquad \frac{}{\mathcal{M}, \Psi \vdash [\mathtt{l}] : \Psi(\mathtt{l})}\ (\textsc{call}) \qquad \frac{}{\mathcal{M}, \Psi \vdash \mathtt{nil} : id}\ (\textsc{nil})$$

Fig. 10. Static Semantics of the Meta-Language

Because the type of actions is somewhat complex, we define action combinators in Figure 8, including composition and branching. The same figure also shows the weaker than relation between actions.

Although, at this point we have defined our machines, it does not have any notion of computation. To make use of the machine, we will need to define a concept of programs, as well as what it means for the particular program to execute.

The definition of the program is given in Figure 9. The most important definition in that figure is that of the procedure, $\mathbb{I}$. The procedure is a bit of program logic that sequences together calls to the operations of a machine (ι), or to other procedures [l] (loops are implemented as recursive calls). Procedures also include a way to branch on a condition. The procedures can be given a name, and placed in the procedure heap $\mathbb{C}$, where they can be referenced from other procedures through the [l] constructor. The procedure heap together with a program rest (the currently executing procedure) makes up the program that can be executed.

The meaning of executing a program is given by the indexed denotational semantics shown on the right side of Figure 9. The meaning of the program is an action that is constructed by sequencing operations. As programs can be infinite, the semantics are indexed by the depth of procedure inclusion.

We use the static semantics (Figure 10) to approximate the action of a procedure. These semantics are similar to the denotational semantics of the meta-language, except that the specifications of called procedure are looked up in the table (Ψ). This means that the static semantics works by the programmer approximating the actions of (specifying) the program, and then making sure that the actual action of the program is within the specifications. These well-formed procedures are then grouped into a well-formed module using the CODE rule, which forms the concept of a certified module $\mathcal{M}, \mathcal{L} \vdash \mathbb{C} : \Psi$, where every procedure in $\mathbb{C}$ is well-formed under specification in Ψ. The module also defines a library ($\mathcal{L}$) which is a set of specifications of stubs, i.e. procedures that are used by the module, but are not in the module. These stubs can then be eliminated by providing procedures that match the stubs (see Section 3.2). For a program to be completely certified, all stubs must either be considered valid primitives or eliminated.

The soundness and correctness theorems need to show that if we checked the well-formedness of our specification, then the specifications are indeed the approximations of the actual actions. The first such theorem checks this property for the individual procedure for the maximum depth of n for procedure inclusion, and assumes that all the functions have been checked for depth $n-1$.

Theorem 1 (Indexed Partial Correctness of Procedures).
If $\mathcal{M}, \Psi \vdash \mathbb{I} : \mathtt{a}$ and $\mathcal{M}, \emptyset \vdash \mathbb{C} : \Psi$ and $\forall \mathtt{f} \in \mathsf{dom}(\mathbb{C}).\ \Psi(\mathtt{f}) \supseteq [\![\mathbb{C}, \mathbb{C}(\mathtt{f})]\!]^{n-1}_{\mathcal{M}}$, then $\mathtt{a} \supseteq [\![\mathbb{C}, \mathbb{I}]\!]^{n}_{\mathcal{M}}$.

Pf.
If $n = 0$, then $[\![\mathbb{C}, \mathbb{I}]\!]^{n}_{\mathcal{M}} = loop$. Since for any $\mathtt{a}$, $\mathtt{a} \supseteq loop$, the result is trivial.
If $n > 0$, proceed by induction on $\mathbb{I}$.
Most cases are trivial, but rely on properties of $\supseteq$.
The trivial, interesting case is $\mathbb{I} = [\mathtt{l}]$.
By definition of $[\![\mathbb{C}, [\mathtt{l}]]\!]^{n}_{\mathcal{M}} = [\![\mathbb{C}, \mathbb{C}(\mathtt{l})]\!]^{n-1}_{\mathcal{M}}$.
By our assumption $\Psi(\mathtt{l}) \supseteq [\![\mathbb{C}, \mathbb{C}(\mathtt{l})]\!]^{n-1}_{\mathcal{M}}$, as needed.

□

Using this theorem, we can build up the correctness of modules to an arbitrary depth, as shown by the next theorem.

Theorem 2 (Indexed Partial Correctness for Modules).
If $\mathcal{M}, \emptyset \vdash \mathbb{C} : \Psi$, then for any index n, for every label $\mathtt{f} \in \mathsf{dom}(\mathbb{C})$, $\Psi(\mathtt{f}) \supseteq [\![\mathbb{C}, \mathbb{C}(\mathtt{f})]\!]^{n}_{\mathcal{M}}$.

Pf. By induction on n. If $n = 0$, result is trivial by theorem 1.
If $n > 0$, by IH, $\forall \mathtt{l} \in \mathtt{dom}(\mathbb{C}).\ \Psi(\mathtt{l}) \supseteq [\![\mathbb{C}, \mathbb{C}(\mathtt{l})]\!]^{n-1}_{\mathcal{M}}$
Apply theorem 1 to every $\mathbb{C}(\mathtt{l})$, to get the needed result.

□

Thus, if we show a module that is free of stubs to be well-formed, then we know that the program will obey the specifications we have given to the code.

3.2 Linking

When we certify using modules, it will be very common that the module will require stubs for the procedures of another module. Linking two modules together should replace the stubs in both modules for the actual procedures that are now present in the linked code. The general way to accomplish this is by the following linking lemma:

Theorem 3 (Linking).

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

where $\Psi_1 \perp \Psi_2 \triangleq \forall \mathtt{l} \in \mathtt{dom}(\Psi_1).\ (\mathtt{l} \notin \mathtt{dom}(\Psi_2) \ \vee \ \Psi_1(\mathtt{l}) = \Psi_2(\mathtt{l}))$.

Pf. By inversion on the two assumption, and then applying CODE rule to get the final result.

□

However, the above rule does not always apply immediately. When the two modules are developed independently, it is possible that the stubs of one module are weaker than the specifications of the procedures that will replace the stubs, which breaks the linking lemma. To fix this, we strengthen the library.

Theorem 4 (Stub Strengthening).
If $\mathcal{M}, \mathcal{L} \vdash \mathbb{C} : \Psi$, then for any $\mathcal{L}'$ s.t. $\forall \mathtt{l} \in \mathtt{dom}(\mathcal{L}).\ \mathcal{L}(\mathtt{l}) \supseteq \mathcal{L}'(\mathtt{l})$ and $\mathtt{dom}(\mathcal{L}') \cap \mathtt{dom}(\Psi) = \emptyset$, the following holds: $\mathcal{M}, \mathcal{L}' \vdash \mathbb{C} : \Psi$.

$$\frac{}{T_{\mathtt{a}}(\mathcal{M}_A(\iota_A)) \supseteq \mathcal{M}_C(\iota_A)} \qquad \frac{}{T_{\mathtt{a}}(id_A) \supseteq id_C} \qquad \frac{(\mathtt{p}_A,\mathtt{g}_A) \supseteq (\mathtt{p'}_A,\mathtt{g'}_A)}{T_{\mathtt{a}}(\mathtt{p}_A,\mathtt{g}_A) \supseteq T_{\mathtt{a}}(\mathtt{p'}_A,\mathtt{g'}_A)}$$

$$\frac{}{T_{\mathtt{a}}((\mathtt{p}_A,\mathtt{g}_A) \underset{\mathcal{M}_A(b)}{\oplus} (\mathtt{p'}_A,\mathtt{g'}_A)) \supseteq (T_{\mathtt{a}}(\mathtt{p}_A,\mathtt{g}_A) \underset{\mathcal{M}_C(b)}{\oplus} T_{\mathtt{a}}(\mathtt{p'}_A,\mathtt{g'}_A))}$$

$$\frac{}{T_{\mathtt{a}}((\mathtt{p}_A,\mathtt{g}_A) \circ (\mathtt{p'}_A,\mathtt{g'}_A)) \supseteq (T_{\mathtt{a}}(\mathtt{p}_A,\mathtt{g}_A) \circ T_{\mathtt{a}}(\mathtt{p'}_A,\mathtt{g'}_A))}$$

Fig. 11. Requirements for Relation of Actions

Pf. Induct over every procedure in the code heap, one by one. For every call to the stub, use the weakening rule to weaken the new stronger stub to the previous weaker specification.

□

This theorem allows us to strengthen the stubs to match the specs of procedures, enabling the linking lemma. Of course, if the specs of the real procedures are not stronger than the specs of the stubs, then the procedures do not correctly implement what the module expects, and linking is not possible.

3.3 The Refinement Framework

Up to this point, we have only considered what happens to the code that is certified over a single machine. However, the purpose of our framework is to facilitate multi-machine verification. For this purpose, we construct the refinement framework that will allow us to refine certified modules in one machine to certified modules in another. The most general notion of refinement in our framework can be defined by the following:

Definition 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 over the abstract certified module Acc, such that for all $\mathbb{C}_A$, Ψ'_A, Ψ_A, the following holds

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

This definition is not a rule, but a template for other definitions. To define a refinement, one has to provide the particular $T_{\mathbb{C}}$, T_{Ψ}, Acc together with the proof that the rule holds. However, instead of trying to define these translations directly, we will automatically generate them from the relations between the particular pairs of machines.

Code-preserving Refinement In our plan for certification of BabyVMM, the syntax of the language does not change, but only the state, the semantics, and the stubs change. Thus, any pair of machines $\mathcal{M}_A$ and $\mathcal{M}_C$ we use for certification will have the property $\mathcal{M}_A.\Delta = \mathcal{M}_C.\Delta$ and $\mathcal{M}_A.\beta = \mathcal{M}_C.\beta$. This means that when we will refine the procedures of our code, the procedures and proc heaps can stay exactly the same, and only the specifications change. Using this restriction, we can define a refinement rule using only a predicate $T_{\mathtt{a}}$ that defines such a change in the specifications.

Definition 2 (Code-Preserving Refinement).
Code-preserving refinement can be defined from abstract machine $\mathcal{M}_A$ to the concrete machine $\mathcal{M}_C$, when the two machines share the same set of operations (Δ). The refinement is defined by an action translation function $T_{\mathtt{a}}$, which must satisfy the properties in Figure 11.

We can show that by this small definition, we can generate a complete refinement rule. To do so, we define $T_{\mathbb{C}}(\mathbb{C}) := \mathbb{C}$ and $T_{\Psi}(\Psi) := \{T_{\mathtt{a}}(\Psi(\mathtt{l})) \mid \mathtt{l} \in \mathtt{dom}(\Psi)\}$. Now we show that the REFINE rule holds for these transformation functions.

Lemma 1 (Code-Preserving Refinement Valid).
Given $T_{\mathtt{a}}$ for which the properties above are true, the following is valid:

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

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

Pf. Assume that $\mathcal{M}_A, \Psi' \vdash \mathbb{C} : \Psi$.
It is adequate to show that for all $\mathtt{l} \in \mathtt{dom}(\Psi)$, if $\mathcal{M}_A, \Psi' \vdash \mathbb{C}(\mathtt{l}) : \Psi(\mathtt{l})$, then $\mathcal{M}_C, T_{\Psi}(\Psi') \vdash \mathbb{C}(\mathtt{l}) : T_{\mathtt{a}}(\Psi(\mathtt{l}))$.
Proceed by induction on derivation of $\mathcal{M}_A, \Psi' \vdash \mathbb{C}(\mathtt{l}) : \Psi(\mathtt{l})$.
Each case is handled by application of the relevant properties of $T_{\mathtt{a}}$ and an application of induction hypothesis when applicable.
The [l] case is trivial since $T_{\Psi}(\Psi')(\mathtt{l}) = T_{\mathtt{a}}(\Psi'(\mathtt{l}))$ by definition of T_{Ψ}.

□

Thus, by defining a single $T_{\mathtt{a}}$, we produce a functioning refinement.

Representation Refinement The code preserving refinement has to be defined by creating a direct transformation of an abstract specification to the concrete one, and showing that such transformation obeys certain properties. However, such specification transformations can be automatically created from the relation between the states of the two machines ($\mathcal{M}_A$, $\mathcal{M}_C$), which we call `repr`.

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

Using `repr`, we can define our specification translation function:

$$T_{A-C}(\mathtt{p}, \mathtt{g}) \triangleq \begin{array}{l}(\lambda \mathbb{S}_C.\, \exists \mathbb{S}_A.\, \mathtt{repr}\ \mathbb{S}_A\ \mathbb{S}_C \wedge \mathtt{p}\ \mathbb{S}_A, \\ \quad \lambda \mathbb{S}_C.\, \lambda \mathbb{S}'_C.\, \forall \mathbb{S}_A.\, \mathtt{repr}\ \mathbb{S}_A\ \mathbb{S}_C \to \forall \mathbb{S}'_A.\, \mathtt{g}\ \mathbb{S}_A\ \mathbb{S}'_A \to \mathtt{repr}\ \mathbb{S}'_A\ \mathbb{S}'_C)\end{array}$$

This operation creates an concrete action from an abstract action. Informally it works as follows. There must be at least one abstract state related to the starting concrete state for which the abstract action applies. The action starting from state $\mathbb{S}_C$ results in set containing $\mathbb{S}'_C$, only if for all related abstract states for which the abstract action is valid result in sets of abstract states that contain a state related to $\mathbb{S}'_C$. Essentially, the resulting concrete action is an intersection of all abstract actions that do not fail.

To make this approach work, we require several properties over the machines and the `repr`. First, the refined semantics of abstraction operations have to be weaker than the semantics of their concrete counterparts, e.g.

$$\forall \iota_A \in \mathcal{M}_A.\, T_{A-C}(\mathcal{M}_A(\iota_A)) \supseteq \mathcal{M}_C(\iota_A)$$

Second, the refinement must preserve the branch choice, e.g. if the refined program chooses left branch, then abstract program had to choose the left branch in all states related by `repr` as well. This property is ensured by requiring the following:

$$\forall b. \forall \mathbb{S}, \mathbb{S}'. (\exists \mathbb{S}_C. \mathtt{repr}(\mathbb{S}, \mathbb{S}_C) \wedge \mathtt{repr}(\mathbb{S}', \mathbb{S}_C)) \rightarrow (\mathcal{M}(b)\, \mathbb{S} \leftrightarrow \mathcal{M}(b)\, \mathbb{S}')$$

We now can show that T_{A-C} is a valid specification translation that supports all the properties needed for the code preserving refinement lemma to work.

Lemma 2 **(`repr`-refinement valid).**
Given `repr` with proofs of the two properties above, the following is valid:

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

where $T_\Psi(\Psi) := \{T_{A-C}(\Psi(\mathtt{l})) \mid \mathtt{l} \in \mathtt{dom}(\Psi)\}$

Pf. For each property listed in Figure 11, show that T_{A-C} satisfies the property. The property that relates two machines is provided by the first condition of representation-refinement. Proving the property for the branching requires branch-preservation condition. Once these properties are proven, Lemma 1 gives us the needed result.

□

Thus we can produce a refinement between machines that use same program by producing a `repr`, making sure that the `repr` preserves branching correct, and then checking the compatibility of operations. Once this information is provided, we can refine our abstract certified modules to automatically generate the concrete specifications as well as automatically generate a proof that the module is well formed under these new concrete specifications.

Other Refinements We have proven other refinements that are not needed for the current verification, but may become important if the current work is extended.

4 Certifying C Code

Since BabyVMM is written in C, we define a formal specification of a tiny subset of the C language using our framework. This C machine will be parameterized by the specific semantics of the memory model, as our plan required. We will also utilize the C machine to further speed up the creation of refinements.

$$
\begin{array}{rll}
(\mathit{State}) & \mathbb{S} & ::= (M, S) \\
(\mathit{Memory}) & M & ::= \textit{(any type over which } load(M, \mathtt{l}) \textit{ and } store(M, \mathtt{l}, w) \textit{ are defined)} \\
(\mathit{Stack}) & S & ::= nil \mid Call(list\ w) :: S \mid Data(\{\mathtt{v} \leadsto w\} :: S) \mid Ret(w) :: S \\
(\mathit{Expressions}) & \mathsf{e} & ::= \mathrm{se} \mid *(\mathsf{e}) \\
(\mathit{StackExpr,\ Cond}) & \mathrm{se}, b & ::= w \mid \mathtt{v} \mid binop(bop, \mathsf{e}_1, \mathsf{e}_2) \mid unop(uop, \mathsf{e}_1) \\
(\mathit{Binary\ Operators}) & bop & ::= + \mid - \mid * \mid / \mid \% \mid == \mid < \mid <= \mid >= \mid > \mid != \mid \&\& \mid \| \\
(\mathit{Unary\ Operators}) & uop & ::= ! \\
(\mathit{Variables}) & \mathtt{v} & ::= \textit{(a decidable set of names)} \\
(\mathit{Words}) & w & ::= n \text{ (integers)} \\
(\mathit{Operation}) & \iota & ::= \mathtt{v} := \mathsf{e} \mid *(\mathsf{e}_{loc}) := \mathsf{e} \mid \mathrm{fcall}(\mathrm{list}\ \mathsf{e}) \mid \mathrm{ret}(\mathsf{e}) \mid \mathrm{args}(\mathrm{list}\ \mathtt{v}) \mid \mathrm{readret}(\mathtt{v})
\end{array}
$$

Operation (ι) =	Action ($\mathcal{M}(\iota)$) =
$\mathtt{v} := \mathsf{e}$	$(\lambda\mathbb{S}.\,\exists S', F, w.\,\mathbb{S}.S = Data(F) :: S' \land eval(\mathsf{e}, \mathbb{S}) = w,$ $\lambda\mathbb{S}, \mathbb{S}'.\,\exists S', F, w.\,\mathbb{S}.S = Data(F) :: S' \land eval(\mathsf{e}, \mathbb{S}) = w \land$ $\mathbb{S}'.M = \mathbb{S}.M \land \mathbb{S}'.S = Data(F\{v \leadsto w\}) :: S')$
$*(\mathsf{e}_{loc}) := \mathsf{e}$	$(\lambda\mathbb{S}.\,\exists l, w.\,eval(\mathsf{e}, \mathbb{S}) = w \land eval(\mathsf{e}_{loc}, \mathbb{S}) = l \land \exists M'.\,M' = store(M, l, w),$ $\lambda\mathbb{S}, \mathbb{S}'.\,\exists l, w.\,eval(\mathsf{e}, \mathbb{S}) = w \land eval(\mathsf{e}_{loc}, \mathbb{S}) = l \land$ $\mathbb{S}'.M = store(\mathbb{S}.M, l, w) \land \mathbb{S}'.S = \mathbb{S}.S)$
$\mathrm{fcall}([e_1, \ldots, e_n])$	$(\lambda\mathbb{S}.\,\exists v_1, \ldots, v_n.\,eval(e_1, \mathbb{S}) = v_1 \land \ldots \land eval(e_n, \mathbb{S}) = v_n,$ $\lambda\mathbb{S}, \mathbb{S}'.\,\exists v_1, \ldots, v_n.\,eval(e_1, \mathbb{S}) = v_1 \land \ldots \land eval(e_n, \mathbb{S}) = v_n \land$ $\mathbb{S}'.M = \mathbb{S}.M \land \mathbb{S}'.S = Call([v_1, \ldots, v_n]) :: \mathbb{S}.S)$
$\mathrm{args}([\mathtt{v}_1, \ldots, \mathtt{v}_n])$	$(\lambda\mathbb{S}.\,\exists w_1, \ldots, w_n, S'.\,\mathbb{S}.S = Call([w_1, \ldots, w_n]) :: S',$ $\lambda\mathbb{S}, \mathbb{S}'.\,\exists w_1, \ldots, w_n, S'.\,\mathbb{S}.S = Call([w_1, \ldots, w_n]) :: S' \land$ $\mathbb{S}'.M = \mathbb{S}.M \land \mathbb{S}'.S = Data(\{\mathtt{v}_1 \leadsto w_1, \ldots, \mathtt{v}_n \leadsto w_n\}) :: S')$
$\mathrm{readret}(\mathtt{v})$	$(\lambda\mathbb{S}.\,\exists S', w.\,\mathbb{S}.S = Ret(w) :: Data(D) :: S',$ $\lambda\mathbb{S}, \mathbb{S}'.\,\exists S', w.\,\mathbb{S}.S = Ret(w) :: Data(D) :: S' \land$ $\mathbb{S}'.M = \mathbb{S}.M \land \mathbb{S}'.S = Data(D\{\mathtt{v} \leadsto w\}) :: S')$
$\mathrm{ret}(\mathsf{e})$	$(\lambda\mathbb{S}.\,\exists w.\,eval(\mathsf{e}, \mathbb{S}) = w,\ \lambda\mathbb{S}, \mathbb{S}'.\,\mathbb{S}'.M = \mathbb{S}.M \land \mathbb{S}'.S = Ret(eval(\mathsf{e}, \mathbb{S})) :: \mathbb{S}.S)$

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

$$
\Upsilon(b) ::= \lambda\mathbb{S}.\,eval(b, \mathbb{S}) \neq 0
$$

Fig. 12. Primitive C-like machine

4.1 The Semantics of C

To define our C machine in terms of our verification framework, we need to give it a state type, a list of operations, and the semantics of those operations expressed as actions. All of these are given in Figure 12.

The state of the C machine includes two components, the stack and the memory. The stack is an abstract C stack that consists of a list of frames, which include call, data, and return frames. In the current version, the stack is independent from memory (one can think of it existing within a statically defined part of the loaded kernel). The memory model is a parameter in the C machine, meaning that it can make use of any memory model as long as it defines load and store operations. The syntax of the C machine is different from the usual definition, in that it relies on the meta-machine for its control flow by using the meta-machines call and branch. Our definition of C adds atomic operations that perform state updates. Thus the operations include two types assignments - one to stack and one to memory, and 4 operations to manipulate stack for call and return, which push and pop the frames.

Because control flow is provided by a standard machine, the code has to be modified slightly. For example, a function call of the form $r = f(x)$ will split into a sequence of three operations: $fcall([x]);[f];readret([r])$, the first setting up a call frame, the second making the call, and the third doing the cleanup. Similarly, the body of the function $f(x)\{body;return(0);\}$ will become $args([x]);body;ret(0)$, as the function must first move the arguments from the call frame into a data frame. Loops have to be desugared into recursive procedures with branches. These modifications are entirely mechanical, and hence we can claim that our machine supports desugared linearized C code.

4.2 Refinement in C machines

C machines at different abstraction layers differ only in their memory models, with the stack being the same. We can use this fact to generate refinements between the C machines using only the representation relation between memory models. This relation ($M_1 \preceq M_2$) can be completely arbitrary as long as these conditions hold:

$$\forall l, v. load(M_1, l) = v \to load(M_2, l) = v$$
$$\forall l, v, M_1'. \left(M_1' = (store(M_1, l, v))\right) \to \left(M_1' \preceq (store(M_2, l, v))\right)$$

The above properties make sure that the load and store operations of memory behave in a similar way. We construct the `repr` between C machine as follows:

$$\texttt{repr} := \lambda \mathbb{S}_A, \mathbb{S}_C.\ (\mathbb{S}_A.S = \mathbb{S}_C.S)\ \wedge\ (\mathbb{S}_A.M \preceq \mathbb{S}_C.M)$$

With `repr` defined, we can show that the two properties needed for `repr`-refinement to be valid hold in all cases. First, we show the following lemma:

Lemma 3 (C refinement op weakening).

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

Pf. First, note that for any expression `e`, if *eval*(e) works in $\mathcal{M}_{M1}$, the it will result in same value in $\mathcal{M}_{M2}$ due to property of loads in related memory.
Proceed by cases of ι. Most cases will rely on *eval* and stack updates, and hence they are trivial. The only interesting case is $*(\texttt{e}_{loc}) := \texttt{e}$. The property of *eval* guarantees that both expressions will compute to same values. The use the store property to conclude that the results are related by `repr`.

□

The next lemma shows that the `repr` does not break branches.

Lemma 4 (C refinement branch ok).

$$\forall b. \forall \mathbb{S}, \mathbb{S}'. (\exists \mathbb{S}_C. \texttt{repr}(\mathbb{S}, \mathbb{S}_C) \wedge \texttt{repr}(\mathbb{S}', \mathbb{S}'_C)) \to (\mathcal{M}(b)\ \mathbb{S} \leftrightarrow \mathcal{M}(b)\ \mathbb{S}')$$

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($addr$)	$addr/$PGSIZE	gets page of address
Off($addr$)	$addr\%$PGSIZE	offset into page of address
LowPg(pg)	$0 \leq pg <$ NPAGES	valid page in low memory area
HighPg(pg)	NPAGES $\leq pg <$ VPAGES	valid page in high memory area
LowAddr(pg)	$\mathrm{LowPg}(\mathrm{Pg}(addr)) \wedge addr\%8 = 0$	valid page in low memory area
HighAddr(pg)	$\mathrm{HighPg}(\mathrm{Pg}(addr)) \wedge addr\%8 = 0$	valid page in high memory area

Fig. 13. Page Definitions

Pf. The b are expressions that can not refer to memory. Since `repr` guarantees that stacks are the same, that means that $\mathbb{S}$ and $\mathbb{S}'$ must have equal stack, and thus $\mathcal{M}(b)\mathbb{S} = \mathcal{M}(b)\mathbb{S}'$.

□

At this point we have shown that if we have two machines with memory models $M1$ and $M2$, such that $M1 \leq M2$, we can automatically generate the `repr` for the two machines. We have shown that this `repr` satisfies all the conditions necessary to form a `repr`-refinement, which means that we can use the following rule for refining code:

Corollary 1 (C Refinement).
For any two memory models $M1$ and $M2$, s.t. $M1 \leq M2$, the following refinement works for C machines instantiated with $M1$ and $M2$.

$$\frac{\mathcal{M}_{M1}, \mathcal{L} \vdash \mathbb{C} : \Psi}{\mathcal{M}_{M2}, T_{M1-M2}(\mathcal{L}) \vdash \mathbb{C} : T_{M1-M2}(\Psi)} \; M1-M2$$

Pf. By lemma 3 and lemma 4 our `repr` defines a valid `repr`-refinement.
Then apply Lemma 2 to get the result.

□

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 (in Figure 6).

5 Virtual Memory Manager

At this point, we have all the machinery necessary to start building our certified memory manager according to the plan. The first step is to formally define and give relations between the memory models that we will use in our certification. Then we will certify the code of the modules that make up the VMM. These modules will then be refined and linked together, resulting in the conclusion that the entire BabyVMM is certified.

5.1 The Memory Models

There are several memory models that are used in the verification of this work, which we have already laid out in our plan. In this section, we will give formal definitions of these models.

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

Notation	`Definition`
$load(M, vaddr)$	$D(trans_M(vaddr))$
$store(M, vaddr, w)$	$(D\{(trans_M(vaddr)) \leadsto w\}, \texttt{PE}, \texttt{PTROOT})$

where

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

Function	Specification
`hw_setPE`	$(\lambda\mathbb{S}.\, \exists S'.\, \mathbb{S}.S = Call([]) :: S' \wedge ValidPT(M.\texttt{PTROOT}),$ $\lambda\mathbb{S}, \mathbb{S}'.\, \exists S'.\, \mathbb{S}.S = Call([]) :: S' \wedge \mathbb{S}'.S = Ret(0) :: S') \wedge$ $\mathbb{S}'.M.\texttt{PE} = true \wedge \mathbb{S}'.M.\texttt{PTROOT} = \mathbb{S}.M.\texttt{PTROOT} \wedge \mathbb{S}'.M.D = \mathbb{S}.M.D)$
`hw_setPTROOT`	$(\lambda\mathbb{S}.\, \exists S', nr.\, \mathbb{S}.S = Call([nr]) :: S' \wedge ValidPT(nr),$ $\lambda\mathbb{S}, \mathbb{S}'.\, \exists S', nr.\, \mathbb{S}.S = Call([nr]) :: S' \wedge \mathbb{S}'.S = Ret(0) :: S' \wedge$ $\mathbb{S}'.M.\texttt{PE} = \mathbb{S}.M.\texttt{PE} \wedge \mathbb{S}'.M.D = \mathbb{S}.M.D \wedge \mathbb{S}'.M.\texttt{PTROOT} = nr)$

Fig. 14. Address Translated Hardware (HW) Memory Model and Library ($\mathcal{L}_{HW}$)

First, Figure 13 gives several predicates and constants that are used in our memory definitions. 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. This is reflected in the predicates `PGSIZE` as well as Pg($addr$) and Off($addr$) that allow for computations of pages and addresses. The definition of `NPAGES` and `VPAGES` are needed to define the precise size of physical memory and virtual space. This is important to be able to define whether a particular page number is in low or high memory areas, which is defined by predicates such as `LowPg`(pg).

Address Translated Memory Model Figure 14 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 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

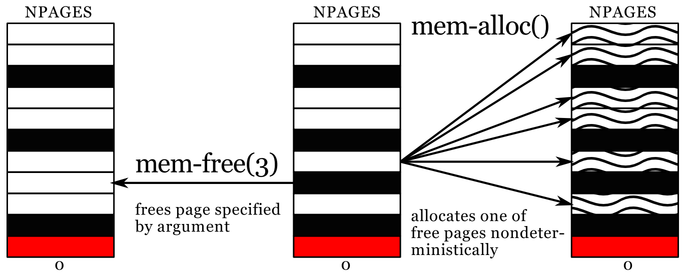
NPAGES
NPAGES
NPAGES
mem-alloc()
mem-free(3)
frees page specified by argument
allocates one of free pages nondeterministically
0
0
0

Fig. 15. Informal Diagram of $\mathcal{M}_{ALE}$

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.

To deal with the registers contained in the memory, we rely on the primitive library $\mathcal{L}_{HW}$, which contains the stubs of two functions: `hw-setPE` and `hw-setPTROOT`. These functions can not be directly defined in C, as they need to update the registers. In real operating systems they are written in inline assembly, and linked with the code. As we currently do not target the assembly language, we simply define these as trusted primitives. The `hw_setPE` primitive is used to switch the address translation on, and the `hw_setPTROOT` to update the PTROOT register. The specifications of these primitives make them appear as functions, which means that we use them by setting up a call frame, calling the primitives, and then removing the return frame.

Our plan also features PE and PD memory models, which are restrictions of the HW memory model. PE requires that AT is always on and that the translation uses identity map for all addresses in physical memory. PD memory model requires that AT is off, and is used to simplify the memory model for initialization. These specializations of the memory models are straightforward simplifications of the HW model.

Allocated Memory Model The model that is more abstract than the hardware models is the allocated memory model named ALE when AT is enabled, and ALD when AT is disabled.

This memory model brings the notion of allocation to the memory. Before reading or writing to a memory location, the semantics will check that the address used is in the allocated page. The pages are allocated using `mem_alloc` and `mem_free` stubs, shown in Figure 15. The allocation of pages is non-deterministic, meaning that the programmer can not rely on getting a specific page allocated. When AT is on, it works the same way as in HW model, which means that page tables must be marked allocated, or the memory operation will fail.

The formal definition of ALE model is given in Figure 16. In addition to regular data storage (D), the model includes an allocation table, (A), which is used by the load and store operations. The operations also include predicates to make sure that the page tables are allocated and that page tables create a direct-mapped window into the physical memory as needed by our plan.

$$\begin{array}{rl}
(\textit{Memory System}) & M ::= (D, A) \\
(\textit{Data Store}) & D ::= \{addr \leadsto w \mid \mathrm{LowAddr}(addr)\}^* \\
(\textit{Page Allocation Table}) & A ::= \{pg \leadsto \texttt{bool} \mid \mathrm{LowPg}(pg)\}^*
\end{array}$$

Notation	Definition
$load(M, va)$	$M.D(trans(M, va))$ if $PTalloc\ M \wedge dm\ M \wedge M.A(Pg(trans(M, va)) = true$
$store(M, va, w)$	$(M.D\{trans(M, va) \leadsto w\}, M.A)$ if $PTalloc\ M \wedge dm\ M \wedge M.A(Pg(trans(M, va)) = true$

where

$$trans(M, va) := \begin{cases} va & \text{if } \mathrm{Pg}(va) < \texttt{NPAGES} \\ M.D(\texttt{PTROOT} + \mathrm{Pg}(va) * 8) * \texttt{NPAGES} + \mathrm{Off}(va) & \text{otherwise} \end{cases}$$

$$PTalloc(M) := \forall pg.\, (\mathrm{LowPage}(pg) \vee \mathrm{HighPage}(pg)) \rightarrow M.A(\mathrm{Pg}(\texttt{PTROOT} + pg * 8)) = true$$

$$dm(M) := \forall vp.\, \mathrm{LowPg}(vp) \rightarrow \mathrm{LowAddr}(\texttt{PTROOT} + vp * 8) \wedge M.D(\texttt{PTROOT} + vp * 8) = vp$$

Function	Specification
mem_alloc	$(\lambda\mathbb{S}.\, \exists S'.\, \mathbb{S}.S = Call([]) :: S',$ $\lambda\mathbb{S}, \mathbb{S}'.\, \exists S'.\, (\mathbb{S}.S = Call([]) :: S') \wedge ((\mathbb{S}'.S = Ret(0) :: S' \wedge \mathbb{S}'.M = \mathbb{S}.M) \vee$ $(\exists pg.\, \mathbb{S}'.S = Ret(pg) :: S' \wedge \mathbb{S}'.M.A = \mathbb{S}.M.A\{pg \leadsto true\} \wedge$ $\wedge \mathbb{S}.M.A(pg) = false \wedge \forall l.\, \mathbb{S}.M.A(Pg(l)) = true \rightarrow (\mathbb{S}'.M.D(l) = \mathbb{S}.M.D(l))))$
mem_free	$(\lambda\mathbb{S}.\, \exists S', pg.\, \mathbb{S}.S = Call([pg]) :: S' \wedge \mathbb{S}.M.A(pg) = true,$ $\lambda\mathbb{S}, \mathbb{S}'.\, \exists S', pg.\, \mathbb{S}.S = Call([pg]) :: S' \wedge \mathbb{S}'.S = Ret(0) :: S' \wedge$ $\mathbb{S}'.M.A = \mathbb{S}.M.A\{pg \leadsto false\} \wedge$ $\forall l.\, \mathbb{S}'.M.A(Pg(l)) = true \rightarrow \mathbb{S}'.M.D(l) = \mathbb{S}.M.D(l)$ $)$

Fig. 16. Allocated Memory Model (M_{ALE}) Semantics and Library ($\mathcal{L}_{ALE}$)

As the C language lacks its own primitives for modification of the allocation tables, and we want to decouple them from the memory writes, we have defined a stub library ($\mathcal{L}_{ALE}$) that contains primitives to modify the allocation tables. The library includes two primitives: `mem_alloc` for allocating a page, and `mem_free` for deallocation. These stubs have fairly complex specifications. For example, the specification of `mem_alloc` includes a disjunction that shows that it either returns a 0 and does not do anything or allocated some free page and returns the number of that page, all while preserving the information in all the allocated pages. These stubs mask themselves as functions, which allows us to replace these primitives with actual functions when we will link the code.

Our verification plan also includes the ALD model of memory, which is a parallel of the ALE memory model, except that AT is off, which allows some simplifications in the semantics, but the main idea behind the model does not change.

Page Map Memory Model The next machine in the stack of abstraction is PMAP. PMAP machine abstracts the hardware-defined pagetables into a virtual page map, hiding the actual pagetables. The machine also encodes the invariants of our page mapping,

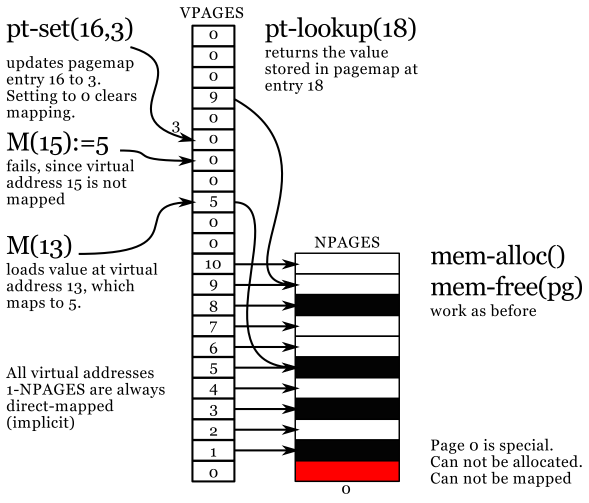
pt-set(16,3)
updates pagemap entry 16 to 3. Setting to 0 clears mapping.
3
M(15):=5
fails, since virtual address 15 is not mapped
M(13)
loads value at virtual address 13, which maps to 5.
All virtual addresses 1-NPAGES are always direct-mapped (implicit)
VPAGES
0
0
0
9
0
0
0
0
5
0
0
10
9
8
7
6
5
4
3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

Fig. 17. Page Map Model of Memory

$$
\begin{array}{rll}
(\textit{Global Storage System}) & M & ::= (D, A, PM) \\
(\textit{Allocatable Memory}) & D & ::= \{addr \rightsquigarrow w \mid \mathrm{LowPg}(Pg(addr)) \wedge addr\%8 = 0\}^* \\
(\textit{Page Allocation Table}) & A & ::= \{pg \rightsquigarrow \texttt{bool} \mid \mathrm{LowPg}(pg)\}^* \\
(\textit{Page Map}) & PM & ::= \{pg \rightsquigarrow pg' \mid \mathrm{HighPg}(pg)\}^*
\end{array}
$$

Notation	Definition	
$load(M, va)$	$M.D(trans(M, va))$	if $M.A(Pg(trans(M, va))) = true$
$store(M, va, w)$	$(M.D\{trans(M, va) \rightsquigarrow w\}, M.A, M.PM)$	if $M.A(Pg(trans(M, va))) = true$

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

Label	Specification
mem_alloc	$(\lambda\mathbb{S}.\,\exists S'.\,\mathbb{S}.S = Call([]) :: S',$ $\lambda\mathbb{S},\mathbb{S}'.\,\exists S'.\,(\mathbb{S}.S = Call([]) :: S') \wedge ((\mathbb{S}'.S = Ret(0) :: S' \wedge \mathbb{S}'.M = \mathbb{S}.M) \vee$ $(\exists pg.\,\mathbb{S}'.S = Ret(pg) :: S' \wedge \mathbb{S}'.M.A = \mathbb{S}.M.A\{pg \rightsquigarrow true\} \wedge \mathbb{S}'.M.PM = \mathbb{S}.M.PM \wedge$ $\wedge \mathbb{S}.M.A(pg) = false \wedge \forall l.\,\mathbb{S}.M.A(Pg(l)) = true \rightarrow (\mathbb{S}'.M.D(l) = \mathbb{S}.M.D(l))))$
mem_free	$(\lambda\mathbb{S}.\,\exists S', pg.\,\mathbb{S}.S = Call([pg]) :: S' \wedge \mathbb{S}.M.A(pg) = true,$ $\lambda\mathbb{S},\mathbb{S}'.\,\exists S', pg.\,\mathbb{S}.S = Call([pg]) :: S' \wedge \mathbb{S}'.S = Ret(0) :: S' \wedge \mathbb{S}'.M.PM = \mathbb{S}.M.PM \wedge$ $\mathbb{S}'.M.A = \mathbb{S}.M.A\{pg \rightsquigarrow false\} \wedge \forall l.\,\mathbb{S}'.M.A(Pg(l)) = true \rightarrow \mathbb{S}'.M.D(l) = \mathbb{S}.M.D(l))$
pt_set	$(\lambda\mathbb{S}..\exists S', vp, pp.\,\mathbb{S}.S = Call([vp, pp]) :: S' \wedge \mathrm{HighPg}(vp) \wedge \mathrm{LowPg}(pp)$ $\lambda\mathbb{S},\mathbb{S}'.\,\exists S', vp, pp.\,\mathbb{S}.S = Call([vp, pp]) :: S' \wedge \mathbb{S}'.S = Ret(0) :: S' \wedge \mathbb{S}'.M.A = \mathbb{S}.M.A \wedge$ $\mathbb{S}'.M.PM = \mathbb{S}.M.PM\{vp \rightsquigarrow pp\} \wedge \forall l.\,\mathbb{S}'.M.A(Pg(l)) = true \rightarrow \mathbb{S}'.M.D(l) = \mathbb{S}.M.D(l))$
pt_lookup	$(\lambda\mathbb{S}.\,\exists S', vp.\,\mathbb{S}.S = Call([vp]) :: S' \wedge \mathrm{HighPg}(vp),$ $\lambda\mathbb{S},\mathbb{S}'.\,\exists S', vp.\,\mathbb{S}.S = Call([vp]) :: S' \wedge \mathbb{S}'.S = Ret(\mathbb{S}.M.PM(vp)) :: S' \wedge \mathbb{S}'.M = \mathbb{S}.M)$

Fig. 18. PMAP Memory Model (M_{PMAP}) and Library ($\mathcal{L}_{PMAP}$)

and also protects the page mappings from modifications by regular load and stores, making the translation system simpler. The intuitive view of this model of memory is given in Figure 17.

The formal definition of the PMAP memory model is given in Figure 18. The formal definition is fairly similar to the one in the ALE model of memory. To facilitate abstract translation, we have added an additional table *PM*, which contains mappings of high pages to low pages, which exists outside the main memory store. The translation function *trans* is modified to use the PM for translating high addresses, no longer looking up addresses in memory. The translation function automatically uses identity for low addresses thereby removing the need to check that the addresses inserted into the page tables are correct. Such formal definition makes PMAP appear as a concise machine-independent definition of the address translation.

We still need the stub library ($\mathcal{L}_{PMAP}$) to perform the tasks that can not be directly controlled with the load and store operations accessible directly through C code. The library retains the stubs for the allocation tables, whose specification is similar to those in the previous library, except there is an additional condition that shows that the memory preserves the page map as well. The stub library also defines two new stubs, `pt_set` and `pt_lookup`, for controlling the entries in the page map. Since the translation system is completely abstract, these two stubs are the only ways to set the virtual addresses in this model.

Address Space Memory Model The most abstract memory model in our plan is M_{AS}, which will be used for the certification of the high-level kernel. We have already presented the informal view of this memory model in Section 2.

The formal definition is in Figure 19. This memory model is different from the other memory models presented in the paper. The first major change is that the data store now includes the entire virtual space. This means that in this model, storing information in virtual addresses actually uses those addresses in the store, which also means that there is no more address translation, but a large address space.

This address space still has an allocation system that the programs must follow, e.g. can not access an address that is not allocated. But other than the allocation table and the actual data store, no additional state is defined by this model.

Because of the allocation system, the AS memory model needs stubs to control the allocation tables, which are defined by the ($\mathcal{L}_{AS}$). However, the semantics of the low memory space (which is a window into physical memory) and the high memory space are different, and thus we define two pairs of allocation predicates. The low memory area is controlled by the already familiar `mem_alloc` and `mem_free`, whose semantics are similar to those of the ALE model, with only an additional precondition that makes sure that the address is in the low space. To allocate and free pages in the high memory area, the library provides `as_request` and `as_release` stubs. The major difference in their specifications are that `as_request` takes a parameter that specifies the page number of the page that the programmer wants to allocate. Such semantics is needed in the high memory area, as the kernel may want to use specific addresses for its purposes.

$$
\begin{array}{rl}
(\textit{Memory System}) & M ::= (D, A) \\
(\textit{Data Store}) & D ::= \{addr \leadsto w \mid \mathrm{HighAddr}(addr) \vee \mathrm{LowAddr}(addr)\}^* \\
(\textit{Page Allocation}) & A ::= \{pg \leadsto \texttt{bool} \mid \mathrm{HighPg}(pg) \vee \mathrm{LowPg}(pg)\}^*
\end{array}
$$

Notation	`Definition`
$load(M, va)$	$M.D(addr)$ if $M.A(Pg(addr)) = true$
$store(M, va, w)$	$(M.D\{addr \leadsto w\}, M.A)$ if $M.A(Pg(addr)) = true$

Label	Specification
mem_alloc	$(\lambda\mathbb{S}.\exists S'.\mathbb{S}.S = Call([]) :: S',$ $\lambda\mathbb{S},\mathbb{S}'.\exists S'.\ (\mathbb{S}.S = Call([]) :: S') \wedge ((\mathbb{S}'.S = Ret(0) :: S' \wedge \mathbb{S}'.M = \mathbb{S}.M) \vee$ $(\exists pg.\mathrm{LowPg}(pg) \wedge \mathbb{S}'.S = Ret(pg) :: S' \wedge \mathbb{S}'.M.A = \mathbb{S}.M.A\{pg \leadsto true\} \wedge$ $\wedge \mathbb{S}.M.A(pg) = false \wedge \forall l.\mathbb{S}.M.A(Pg(l)) = true \rightarrow (\mathbb{S}'.M.D(l) = \mathbb{S}.M.D(l))))$
mem_free	$(\lambda\mathbb{S}.\exists S', pg.\mathbb{S}.S = Call([pg]) :: S' \wedge \mathrm{LowPg}(pg) \wedge \mathbb{S}.M.A(pg) = true,$ $\lambda\mathbb{S},\mathbb{S}'.\exists S', pg.\mathbb{S}.S = Call([pg]) :: S' \wedge \mathbb{S}'.S = Ret(0) :: S' \wedge$ $\mathbb{S}'.M.A = \mathbb{S}.M.A\{pg \leadsto false\} \wedge$ $\forall l.\mathbb{S}'.M.A(Pg(l)) = true \rightarrow \mathbb{S}'.M.D(l) = \mathbb{S}.M.D(l))$
as-request	$(\lambda\mathbb{S}.\exists S', vpg.\mathbb{S}.S = Call([vpg]) :: S' \wedge \mathrm{HighPg}(vpg),$ $\lambda\mathbb{S},\mathbb{S}'.\exists S', vpg.\mathbb{S}.S = Call([vpg]) :: S' \wedge$ $((\mathbb{S}'.S = Ret(0) :: S' \wedge \mathbb{S}'.M = \mathbb{S}.M) \vee$ $(\mathbb{S}'.S = Ret(vpg) :: S' \wedge \mathbb{S}.M.A(vpg) = false \wedge \mathbb{S}'.M.A = \mathbb{S}.M.A\{vpg \leadsto true\} \wedge$ $\forall l.\mathbb{S}.M.A(Pg(l)) = true \rightarrow (\mathbb{S}'.M.D(l) = \mathbb{S}.M.D(l)))$ $)$ $)$
as-release	$(\lambda\mathbb{S}.\exists S', vpg.\mathbb{S}.S = Call([vpg]) :: S' \wedge \mathbb{S}.M.A(pg) = true \wedge \mathrm{HighPg}(vpg),$ $\lambda\mathbb{S},\mathbb{S}'.\exists S', vpg.\mathbb{S}.S = Call([vpg]) :: S' \wedge \mathbb{S}'.S = Ret(0) :: S' \wedge$ $\mathbb{S}'.M.A = \mathbb{S}.M.A\{vpg \leadsto false\} \wedge$ $\forall l.\mathbb{S}'.M.A(Pg(l)) = true \rightarrow \mathbb{S}'.M.D(l) = \mathbb{S}.M.D(l))$

Fig. 19. Address Space (M_{AS}) Memory Model and Library ($\mathcal{L}_{AS}$)

5.2 Relation between Memory Models

Our plan calls for creation of the refinements between the memory models. In Section 4.2, we have shown that we can generate a valid refinement by creating a relation between the memory states, and then showing that abstract loads and stores are preserved by this relation. These relations and proofs of preserving the memory operations are fairly lengthy and quite technical, and thus we leave the mathematical detail to our Coq implementation, opting for a visual description shown in Figure 20.

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 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

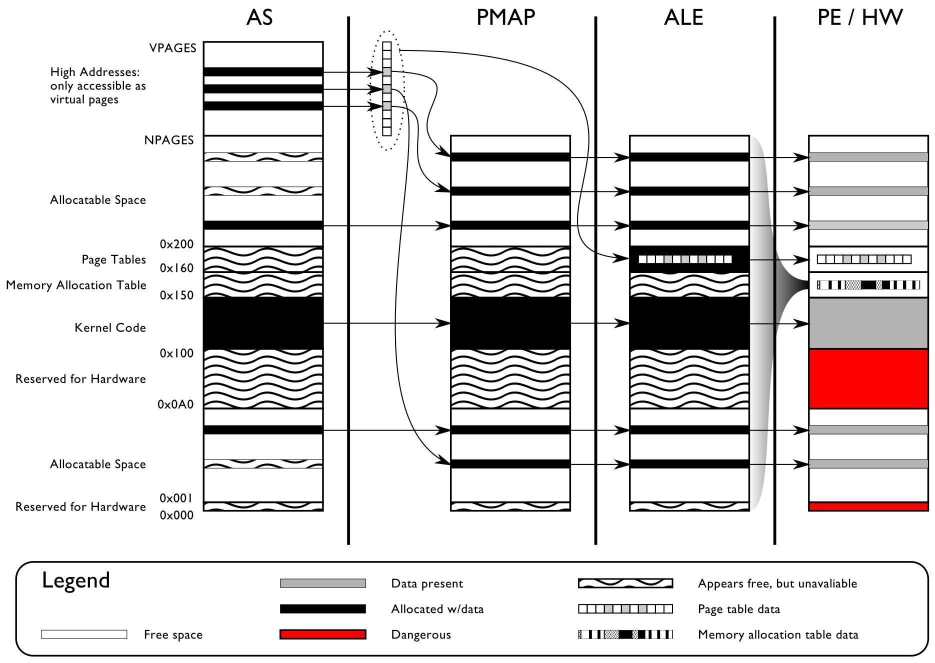

Fig. 20. Relation between Memory Models

appear to be unusable. The ALE mem_alloc primitive will never allocate pages from these wavy areas, protecting them 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 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 not go into detail about the preservation of load and stores, as these proofs are mostly straightforward, given the relations.

This description does not give the details of relations between several models for which our plan does have refinements. However, in the case of ALD-PD models, the relation parallels the ALE-PE relation. and the relation between PE and HW as well as PD and HW is fairly trivial as these PE and PD are just restrictions of the HW, and hence the relation is just a restriction as well. The formal `repr` relations between these models can be found in the Coq proof.

5.3 Certification and Linking of BabyVMM

We have verified all the functions of the virtual memory on the appropriate memory models. This means that we have defined appropriate specifications for our functions, and certified our code. We also make an assumption that a kernel is certified in the AS model. The result is the following certified modules:

$$\begin{array}{lll} \mathcal{M}_{PE}, \mathcal{L}_{PE} \vdash \mathbb{C}^{mem} : \Psi^{mem}_{PE} & \mathcal{M}_{ALE}, \mathcal{L}_{PMAP} \vdash \mathbb{C}^{as} : \Psi^{as}_{PMAP} & \mathcal{M}_{PD}, \mathcal{L}_{PD} \vdash \mathbb{C}^{meminit} : \Psi^{meminit}_{PD} \\ \mathcal{M}_{ALE}, \mathcal{L}_{ALE} \vdash \mathbb{C}^{pt} : \Psi^{pt}_{ALE} & \mathcal{M}_{AS}, \mathcal{L}_{AS} \vdash \mathbb{C}^{kernel} : \Psi^{kernel}_{AS} & \mathcal{M}_{ALD}, \mathcal{L}_{ALD} \vdash \mathbb{C}^{ptinit} : \Psi^{ptinit}_{ALD} \end{array}$$

However, the `init` function makes calls to other procedures that are certified in more abstract machines. Thus to certify `init` over the $\mathcal{M}_{HW}$ machine, we will need to create stubs for these procedures, which have to be carefully crafted to be valid for the refined specifications of the actual procedures. Thus, the specification of `init` results in the following:

$$\mathcal{M}_{HW}, \mathcal{L}_{HW} \cup \left\{ \text{kernel\_init} \rightsquigarrow \mathsf{a}^{kernel-init}_{HW}, \text{ mem\_init} \rightsquigarrow \mathsf{a}^{meminit}_{HW}, \text{ pt\_init} \rightsquigarrow \mathsf{a}^{ptinit}_{HW} \right\} \vdash \mathbb{C}^{init} : \Psi^{init}_{HW}$$

With all the modules verified, we proceed to link them together. The first step is to refine the kernel. We use our AS-PMAP refinement rule to get the refined module:

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

Then we show that the specs of functions and the primitives of the PMAP machine are proper implementation of the refined specs of $\mathcal{L}_{AS}$, more formally,

$$T_{AS-PMAP}(\mathcal{L}_{AS}) \supseteq \mathcal{L}_{PMAP} \cup \Psi^{as}_{PMAP}$$

Using library strengthening and the linking lemma, we produce a certified module that is the union of the refined kernel and address space library:

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

Applying this process to all the modules over all refinements, we link all parts of the code, except `init` certified over $\mathcal{M}_{HW}$. For readability, we hide chains of refinements. For example, T_{AS-HW} is actually $T_{AS-PMAP} \circ T_{PMAP-ALE} \circ T_{ALE-PE} \circ T_{PE-HW}$.

$$\begin{array}{l} \mathcal{M}_{HW}, \mathcal{L}_{HW} \vdash \mathbb{C}^{kernel} \cup \mathbb{C}^{as} \cup \mathbb{C}^{pt} \cup \mathbb{C}^{mem} \cup \mathbb{C}^{meminit} \cup \mathbb{C}^{ptinit} : \\ \quad T_{AS-HW}(\Psi^{kernel}_{AS}) \cup T_{PMAP-HW}(\Psi^{as}_{PMAP}) \cup T_{ALE-HW}(\Psi^{pt}_{ALE}) \cup \\ \quad T_{PE-HW}(\Psi^{mem}_{PE}) \cup T_{PD-HW}(\Psi^{meminit}_{PD}) \cup T_{ALD-HW}(\Psi^{ptinit}_{ALD}) \end{array}$$

To get the initialization to link with the refined module, we must make sure that the stubs that we have developed for init are compatible with the refined specifications of the actual functions. This means that we prove the following:

$$\mathsf{a}^{kernel-init}_{HW} \supseteq T_{AS-HW}(\Psi^{kernel}_{AS})(\texttt{kernel-init})$$

$$\mathsf{a}^{meminit}_{HW} \supseteq T_{PD-HW}(\Psi^{meminit}_{PD})(\texttt{mem-init}) \qquad \mathsf{a}^{ptinit}_{HW} \supseteq T_{ALD-HW}(\Psi^{ptinit}_{ALD})(\texttt{pt-init})$$

Using these properties, we apply stub strengthening to the init module:

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

This certification is now linkable to the rest of the VMM and kernel, to produce the final result that we need:

$$\begin{aligned}\mathcal{M}_{HW}, \mathcal{L}_{HW} \vdash \mathbb{C}^{kernel} \cup \mathbb{C}^{as} \cup \mathbb{C}^{pt} \cup \mathbb{C}^{mem} \cup \mathbb{C}^{meminit} \cup \mathbb{C}^{ptinit} \cup \mathbb{C}^{init} :&\\ T_{AS-HW}(\Psi^{kernel}_{AS}) \cup T_{PMAP-HW}(\Psi^{as}_{PMAP}) \cup T_{ALE-HW}(\Psi^{pt}_{ALE}) \cup&\\ T_{PE-HW}(\Psi^{mem}_{PE}) \cup T_{PD-HW}(\Psi^{meminit}_{PD}) \cup T_{ALD-HW}(\Psi^{ptinit}_{ALD}) \cup \Psi^{init}_{HW}&\end{aligned}$$

This result means that given a certified kernel in the AS model, we can refine it to the HW model of memory by linking it with VMM implementation. Furthermore, it is safe to start this kernel by calling the `init` function, which will perform the setup, and then call the `kernel-init` function, the entry point of the high-level kernel.

6 Coq Implementation

All portions of this system have been implemented in the Coq Proof Assistant[5]. The portions of the implementation directly related to the BabyVMM verification, including C machines, refinements, specs, and related proofs (excluding frameworks) took about 3 person-months to verify. The approximate line counts for unoptimized proof are:

– Verification and refinement framework - 3000 lines
– Memory models - 200-400 lines each
– `repr` and compatibility between models - 200-400 lines each
– Compatibility of stubs and implementation - 200-400 lines per procedure
– Code verification - less than 200 lines per procedure (half of it boilerplate).

7 Related Work and Conclusion

The work presented here is a continuation of the work on Hoare-logic frameworks for verification of system software. The verification framework evolved from SCAP[8] and GCAP[3]. Although our framework does not mention separation logic[17], information hiding[16], and local action[4] explicitly, these methods had great influence on the design of the meta-language and the refinements. The definition of `repr` generalizes the work on certified garbage collector[15] to fit our concept of refinement. The project's motivation is the modular and reusable certification of the CertiKOS kernel[10].

The well-known work in OS verification is L4.verified[12, 6], which has shown a complete verification of an OS kernel. Their methodology is different, but they have considered verification of virtual memory[13, 14]. However, their current kernel verification does not abstract virtual memory, maintaining only the invariant that allows the kernel to function, and leaving the details to the user level.

The Verisoft project [9, 2, 1, 11, 18] is the work that is closest to ours. We both aim for pervasive verification of OS by doing foundational verification of all components. Both works utilize multiple machines, and require linking. As both projects aim for certification of a kernel, both have to handle virtual memory. Although Verisoft uses multiple machine models, they use them sparingly. For example, the entire microkernel, excluding assembly code, is specified in a single layer, with correctness shown as a single simulation theorem between the concurrent user thread model (CVM) and the instruction set. The authors mention that the proof of correctness is a more complex part of Verisoft. Such monolithic approach is susceptible to local modifications, where a small change in one part of microkernel may require changes to the entire proof.

Our method for verification defines many more layers, with smaller refinement proofs between them, and composes them to produce larger abstractions, ensuring that the verification is more reusable and modular. Our new framework enables us to create abstraction layers with less overhead, reducing the biggest obstacle to our approach. We have demonstrated the practicality of our approach by certifying BabyVMM, a small virtual memory manager running on simplified hardware, using a new layer for every non-trivial abstraction we could find.

Acknowledgements. We thank anonymous referees for suggestions and comments on an earlier version of this paper. 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. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

1. E. Alkassar, M. A. Hillebrand, D. C. Leinenbach, N. W. Schirmer, A. Starostin, and A. Tsyban. Balancing the load: Leveraging a semantics stack for systems verification. *Journal of Automated Reasoning: OS Verification*, 42:389–454, 2009.
2. E. Alkassar, N. Schirmer, and A. Starostin. Formal pervasive verification of a paging mechanism. *Proc. TACAS'08*, pages 109–123, 2008.
3. H. Cai, Z. Shao, and A. Vaynberg. Certified self-modifying code. In *Proc. PLDI'07*, pages 66–77, New York, NY, USA, 2007. ACM.
4. C. Calcagno, P. O'Hearn, and H. Yang. Local action and abstract separation logic. In *Proc. LICS'07*, pages 366–378, July 2007.
5. Coq Development Team. The Coq proof assistant reference manual. The Coq release v8.0, Oct. 2005.
6. K. Elphinstone, G. Klein, P. Derrin, T. Roscoe, and G. Heiser. Towards a practical, verified kernel. In *Proc. HoTOS'07*, San Diego, CA, USA, May 2007.
7. X. Feng, Z. Shao, Y. Dong, and Y. Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *Proc. PLDI'08*, pages 170–182. ACM, 2008.
8. X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni. Modular verification of assembly code with stack-based control abstractions. In *PLDI'06*, pages 401–414, June 2006.
9. M. Gargano, M. A. Hillebrand, D. Leinenbach, and W. J. Paul. On the correctness of operating system kernels. In *TPHOLs'05*, 2005.
10. L. Gu, A. Vaynberg, B. Ford, Z. Shao, and D. Costanzo. Certikos: A certified kernel for secure cloud computing. In *Proc. APSys'11*. ACM, 2011.
11. T. In der Rieden. *Verified Linking for Modular Kernel Verification*. PhD thesis, Saarland University, Computer Science Department, Nov. 2009.
12. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *Proc. SOSP'09*, pages 207–220, 2009.

13. G. Klein and H. Tuch. Towards verified virtual memory in l4. In *TPHOLs Emerging Trends '04*, Park City, Utah, USA, Sept. 2004.
14. R. Kolanski and G. Klein. Mapped separation logic. In *Proc. VSTTE'08*, pages 15–29, 2008.
15. A. McCreight, Z. Shao, C. Lin, and L. Li. A general framework for certifying garbage collectors and their mutators. In *Proc. PLDI'07*, pages 468–479, 2007.
16. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL'04*, pages 268–280, Jan. 2004.
17. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. LICS'02*, pages 55–74, July 2002.
18. A. Starostin. *Formal Verification of Demand Paging*. PhD thesis, Saarland University, Computer Science Department, Mar. 2010.

A C code of the BabyVMM

Hardware-Specified Constants and Functions (hw.h)

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

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

Code of the Memory Allocator (mem.c)

```
#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;
}
```

Code of the Page Table System

```
#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++;
  }
}
```

Code of the Address Spaces (as.c)

```
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
}
```

Code of the Initialization (init.c)

```
void init()
```

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