Mapped Separation Logic

Rafal Kolanski and Gerwin Klein

Sydney Research Lab., NICTA*, Australia
School of Computer Science and Engineering, UNSW, Sydney, Australia

{rafal.kolanski|gerwin.klein}@nicta.com.au

Abstract. We present Mapped Separation Logic, an instance of Separation Logic for reasoning about virtual memory. Our logic is formalised in the Isabelle/HOL theorem prover and it allows reasoning on properties about page tables, direct physical memory access, virtual memory access, and shared memory. Mapped Separation Logic fully supports all rules of abstract Separation Logic, including the frame rule.

1 Introduction

Let memory be a function from addresses to values. Many a paper and semantics lecture start off with this statement. Unfortunately, it is not how memory on any reasonably complex machine behaves. Memory, on modern hardware, is virtualised and addresses pass through a translation layer before physical storage is accessed. Physical storage might be shared between different virtual addresses. Additionally, the encoding of how the translation works (the page table) resides in physical storage as well, and might be accessible through the translation. These two properties make the function model incorrect.

Operating systems can be made to provide the illusion of plain functional memory to applications. This illusion, however, is brittle. Mapping and sharing virtual pages are standard Unix system calls available to applications and they can easily destroy the functional view. What is worse, on the systems software layer, e.g. for the operating system itself, device drivers, or system services, the virtual memory subsystem is fully visible: sharing is used frequently and the translation layer is managed explicitly.

Building on earlier work [11], we show in this paper how Separation Logic [15], a tool for reasoning conveniently about memory and aliasing, can be extended to virtual memory. We do this in a way that preserves the abstract reasoning of Separation Logic as well as its critical locality feature: the frame rule.

The technical contributions in this work are the following:

– a Separation Logic allowing convenient, abstract reasoning about both the virtual and physical layers of memory, supporting the frame rule for arbitrary writes to memory (including the page table),

* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.

– a separating conjunction extending to virtually shared memory,
– a framework that makes the core logic independent of particular page table implementations, and
– a case study on page allocation that demonstrates the logic.

We currently do not take into account memory permissions beyond whether a virtual address is mapped/unmapped. Such permissions are merely extra properties of virtual addresses; they can be easily integrated into our logic.

All of the work presented in this paper is formalised in the Isabelle/HOL theorem prover [12]. We analyse the logic based on a simple, imperative programming language with arbitrary pointer arithmetic. We embed assertions shallowly and the programming language deeply into Isabelle/HOL. The latter means means we cannot use Isabelle's rich proof automation directly for the case study. The point of this paper is to analyse the logic itself and to prove that it supports the frame rule. The case study shows that it can be applied in principle.

For program verification, we plan to switch to a shallow language embedding and connect this logic with earlier work on a precise memory model of C by Tuch et al. [17], including the automatic verification condition generator used there. As this memory model, the context of the present work is the L4.verified project aiming to prove implementation correctness of the seL4 microkernel [7].

2 Intuition

Separation Logic traditionally models memory as a partial function from addresses to values, called the heap. The two central tools of Separation Logic are the separating conjunction and the frame rule. Separating conjunction $P \wedge^* Q$ is an assertion on heaps, stating that the heap can be split into two separate parts on which the conjuncts P and Q hold respectively. We also say P and Q *consume* disjoint parts of the heap. Separating conjunction conveniently expresses anti-aliasing conditions. For an action f, the frame rule allows us to conclude $\{P \wedge^* R\}\, f\, \{Q \wedge^* R\}$ from $\{P\}\, f\, \{Q\}$ for any R. This expresses that the actions of f are local to the heaps described by P and Q, and can therefore not affect any separate heaps described by R. As we shall see below, with virtual memory, both of these tools break and both can be repaired.

Virtual memory is a hardware-enforced abstraction that allows each executing process its own view of memory, into which areas of physical memory can be inserted dynamically. It adds a level of indirection: virtual addresses potentially map to physical addresses. Memory access is ordinarily done through virtual addresses only, although hardware devices may modify physical memory directly.

This indirection is encoded in the heap itself, in a page table data structure. There are many flavours of such encodings, using multiple levels, different sizes of basic blocks (e.g. super pages), dynamically different table depths, and even hash tables. Usually, the encoding and minimum granularity (page size) is dictated by the hardware. The anchor of the page table in physical memory is called its root. Knowing the specific encoding, the full set of virtual-to-physical mappings for a process can be constructed from the heap and the root.

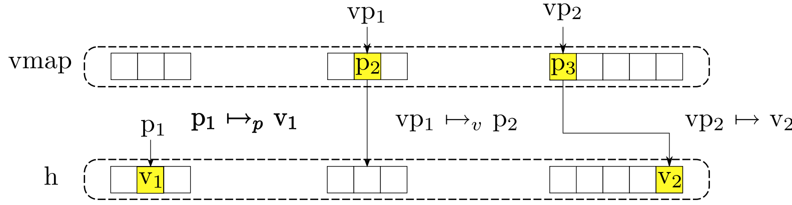

Fig. 1. Maps-to assertions on the heap, virtual map and address space.

Given the additional layer of indirection, two different virtual addresses may resolve to the same physical address, and separating conjunction breaks, because on these addresses it does not end up providing separation. Additionally, although a memory update to the page table may only locally change one byte of physical memory, it might completely change the view the virtual memory layer provides, affecting a whole number of seemingly unrelated virtual addresses. A local action might therefore have non-local effects. This breaks the frame rule. We show how to repair both tools for virtual memory in the remainder of this paper.

We will henceforth restrict our use of the word *heap* to refer to physical memory, and we will call the virtual-to-physical translation the *virtual map*. This virtual map provides the first abstraction layer in our model of virtual memory: for the development of Separation Logic on virtual memory, we do not need to know how the encoding works precisely. We only need to know that there exists a way of decoding, and, as we shall see later, that there is a way of finding out which page table entries contribute to looking up a particular virtual address. As the rest of the development is independent of the encoding, we use only a very simple implementation for the case study presented here: a one-level page table.

For reasoning about virtual memory we wish to make assertions on three levels, shown in Fig. 1: physical-to-value, virtual-to-physical, and virtual-to-value. Given our two partial functions (heap and vmap), this may appear straightforward. The virtual-to-physical assertion corresponds to looking up a virtual address in the page table. The central question that arises is: *which part of the heap should a virtual-to-physical assertion consume?* This question determines the meaning of separation between this and other assertions. Assuming for example a one-level page table where memory location x encodes the lookup of virtual address vp_1 to physical address p_2, there are two cases: (a) the lookup consumes x, or (b) it does not. In case (a) we cannot use separating conjunction to specify separation of vp_1 and another address vp_2 if vp_2 happens to use the same page table entry x. Typically, many virtual addresses share an entry in the encoding, and we clearly do not want to exclude this case. The other extreme, in case (b), would be to say the lookup consumes no resources. In earlier work [11] we came to the result that this model cannot support the frame rule for arbitrary programs: a physical write to the page table would be separate to any virtual-to-physical lookup, but the write might have the non-local effect of changing that apparently separate mapping.

In this paper, we are therefore proposing a solution in between: a page table lookup consumes only *parts* of the page table entry involved. This can be seen as cutting each heap location into multiple slices. If a page table lookup, i.e. a virtual-to-physical mapping, consumes parts of the page table entry involved in the lookup, we can make all memory updates local.

This idea is similar to the model of permissions by Bornat et al. [3] for concurrent threads. The difference is that in our case we extend the domain of the heap to slices, whereas in the permission model, the range is extended. This has the advantage that the development of heap disjointness, merging, separating conjunction and the other standard operators remains unchanged, and therefore should be easy to instantiate to our target model of C [17]. The memory footprint of page table lookups has a direct formulation in our model.

The next section will briefly summarise our formal abstract interface to page table encodings, before Sect. 4 introduces the model and shallow Isabelle/HOL embedding of the assertion language for virtual memory.

3 The Virtual Memory Environment

In this section, we describe the page table abstraction our logic is based on, as well as an instantiation to a simple, one-level page table.

As in previous work [11], we write $\mathsf{VPtr}\ vp$ and $\mathsf{PPtr}\ p$ to distinguish virtual and physical pointers by type in Isabelle/HOL, and provide a destructor $\mathsf{ptr\text{-}val}$ $(\mathsf{PPtr}\ x) = x$ when we need to talk about the physical/virtual address directly.

In Isabelle the abstract concepts of heap, virtual map, and address space are:

$$\textbf{types}\quad heap = pptr \rightharpoonup val \qquad addr\text{-}space = vptr \rightharpoonup val$$
$$vmap = vptr \rightharpoonup pptr$$

"$\rightharpoonup$" denotes a partial function. These types do not include slicing yet. Pointer sizes are parameters of the development. We use 32-bit machine words for pointers as well as values of memory cells in concrete examples.

For the development of the logic in the remainder of the paper, we require the following two functions.

$$\mathsf{ptable\text{-}lift} :: heap \Rightarrow pptr \Rightarrow vmap$$
$$\mathsf{ptable\text{-}trace} :: heap \Rightarrow pptr \Rightarrow vptr \Rightarrow pptr\ \mathsf{set}$$

Both functions take the physical heap and page table root as parameters. The result of ptable-lift is the *vmap* encoded by the page table, whereas ptable-trace returns for each virtual address *vp* the set of locations in the page table that are involved in the lookup of *vp*. In contrast to our earlier work, we do not require an explicit formulation of the page table area in memory.

We will later require five constraints on these two functions. We defer the presentation of these to Sect. 5 when the associated concepts have been introduced.

As mentioned above, we use a simple one-level page table as an example instantiation. It is a contiguous physical memory structure consisting of an array of machine word pointers, where word 0 defines the physical location of page 0 in the address space, word 1 that of page 1 and so forth. While inefficient in terms

of storage, it is simple to present and experiment with. The table is based on an arbitrarily chosen page size of 4096, i.e. 20 bits for the page number and 12 for the offset. Page table lookup works as expected: we extract the page number from the virtual address, go to that offset in the page table and obtain a physical frame number which replaces the top 20 bits of the address:

$$
\begin{array}{lcl}
\textsf{get-page}\ vp & \equiv & \textsf{ptr-val}\ vp >> 12 \\
\textsf{ptr-remap}\ (\textsf{VPtr}\ vp)\ pg & \equiv & \textsf{PPtr}\ (pg\ \textsf{AND NOT}\ 0xFFF\ \textsf{OR}\ vp\ \textsf{AND}\ 0xFFF) \\
\textsf{ptable-lift}\ h\ r\ vp & \equiv & \textsf{case}\ h\ (r + \textsf{get-page}\ vp)\ \textsf{of None} \Rightarrow \textsf{None} \\
 & & \mid \lfloor addr \rfloor \Rightarrow \\
 & & \quad \textsf{if}\ addr\ !!\ 0\ \textsf{then}\ \lfloor \textsf{ptr-remap}\ vp\ addr \rfloor\ \textsf{else None} \\
\textsf{ptable-trace}\ h\ r\ vp & \equiv & \textsf{case}\ h\ (r + \textsf{get-page}\ vp)\ \textsf{of None} \Rightarrow \emptyset \\
 & & \mid \lfloor addr \rfloor \Rightarrow \{r + \textsf{get-page}\ vp\}
\end{array}
$$

AND, OR and NOT are bitwise operations on words. The operator $>>$ is bitwise right-shift on words. The term $x\ !!\ n$ stands for bit n in word x. We use bit 0 to denote whether a mapping is valid. The optional value type has two constructors: None for no value and $\lfloor value \rfloor$ otherwise.

4 Separation Logic Assertions on Virtual Memory

In this section, we describe how the classical Separation Logic assertions can be extended to hold for a model with virtual memory.

As mentioned in Sect. 2, the idea is to extend the domain of the heap to slices. One page table entry can, at most, be responsible for all virtual addresses in the machine. Therefore, the smallest useful granularity of slicing up one physical address is that of one slice per virtual address. For each physical address p, we need to be able to address the slice of p responsible for virtual address $0, 1, \ldots :: vptr$ etc. Thus, the domain of the heap becomes $pptr \times vptr$ and (p, vp) stands for the vp-slice of physical address p.

In our shallow embedding, Separation Logic assertions are predicates on this new heap and the root of the page table:

$$
\begin{array}{ll}
\textbf{types} & fheap = (pptr \times vptr) \rightharpoonup val \\
 & map\text{-}assert = (fheap \times pptr) \Rightarrow bool
\end{array}
$$

With this model, the basic definition of heap merge (++), domain, and disjointness ($\perp$) remain completely standard:

$$
\begin{array}{lcl}
h_1 ++ h_2 & \equiv & \lambda x.\ \textsf{case}\ h_2\ x\ \textsf{of None} \Rightarrow h_1\ x \mid \lfloor y \rfloor \Rightarrow \lfloor y \rfloor \\
\textsf{dom}\ h & \equiv & \{x \mid h\ x \neq \textsf{None}\} \\
h_1 \perp h_2 & \equiv & \textsf{dom}\ h_1 \cap \textsf{dom}\ h_2 = \emptyset
\end{array}
$$

The Separation Logic connectives for empty heap, true, conjunction, and implication stay almost unchanged. We additionally supply the page table root r.

$$
\begin{array}{ll}
\square & \equiv \lambda(h, r).\ h = \textsf{empty} \\
\top & \equiv \lambda(h, r).\ \textsf{True} \\
P \wedge^* Q & \equiv \lambda(h, r).\ \exists h_0\ h_1.\ h_0 \perp h_1 \wedge h = h_0 ++ h_1 \wedge P\ (h_0, r) \wedge Q\ (h_1, r) \\
P \longrightarrow^* Q & \equiv \lambda(h, r).\ \forall h'.\ h \perp h' \wedge P\ (h', r) \longrightarrow Q\ (h ++ h', r)
\end{array}
$$

Since the definitions are almost unchanged it is unsurprising that the usual properties such as commutativity and associativity and distribution over lifted normal conjunction continue to hold.

The interesting, new assertions are the three *maps-to* statements that we alluded to in Fig. 1. Corresponding to the traditional Separation Logic predicate is the physical-to-value mapping:

$$p \mapsto_p v \equiv \lambda(h,\, r).\ (\forall\, vp.\ h\ (p,\, vp) = \lfloor v \rfloor) \wedge \mathsf{dom}\ h = \{p\} \times \mathcal{U}$$

For this assertion, we require that all slices of the heap at physical address p map to the value v and that the domain of the heap is exactly the set of all pairs with p as the first component ($\mathcal{U}$ is the universe set). This predicate would typically be used for direct memory access in devices or for low-level kernel operations.

The next assertion is the virtual-to-physical mapping:

$$\begin{aligned} vp \mapsto_v p \equiv \lambda(h,\, r).& \\ &\mathsf{let}\ heap = \mathsf{h\text{-}view}\ h\ vp;\ vmap = \mathsf{ptable\text{-}lift}\ heap\ r \\ &\mathsf{in}\ vmap\ vp = \lfloor p \rfloor \wedge \mathsf{dom}\ h = \mathsf{ptable\text{-}trace}\ heap\ r\ vp \times \{vp\} \end{aligned}$$

where $\mathsf{h\text{-}view}\ fh\ vp \equiv \lambda p.\ fh\ (p,\, vp)$. Here, we first lift the page table out of the heap to get the abstract *vmap* which provides us with the translation from *vp* to *p*. Additionally, we assert that the domain of the heap is the *vp* slice of all page table entries that are involved in the lookup. With h-view we project out the *vp* slice for all addresses so that the page table lift function can work on a traditional $pptr \rightharpoonup val$ heap.

Putting the two together, we arrive at the virtual-to-value mapping that corresponds to the level that most of the OS and user code will be reasoning at.

$$vp \mapsto v \equiv \lfloor \exists \rfloor p.\ vp \mapsto_v p \wedge^* p \mapsto_p v$$

where $\lfloor \exists \rfloor x.\ P\ x \equiv \lambda s.\ \exists\, x.\ P\ x\ s$. The predicate implies that the lookup path is separate from the physical address p the value v is at. This is the case for all situations we have encountered so far and, as the case study shows, works for page table manipulations as well. It is possible to define a weaker predicate without the separation, but this creates a special case for the assignment rule: if a write changes the page table for the address the write goes to, the post condition would have to take the change in the translation layer into account directly.

The usual variations on the maps-to predicate can be defined in the standard manner again for virtual-to-value mappings:

$$\begin{aligned} &p \mapsto - \equiv \lfloor \exists \rfloor v.\ p \mapsto v \\ &p \hookrightarrow v \equiv p \mapsto v \wedge^* \top \\ &S\ \{\mapsto\}\ - \equiv \mathsf{fold\ op} \wedge^*\ (\lambda x.\ x \mapsto -)\ \Box\ S \end{aligned}$$

The latter definition refers to a finite set S of addresses. It states that all of the elements map separately by folding the $\wedge^*$ operator over S and the maps-to predicate. There are analogous variations for physical-to-value and virtual-to-physical mappings.

We have proved that our basic mappings $\mapsto_p$ and $\mapsto_v$ are domain exact [15]. Note that, although not domain exact due to the existential quantifier on p, the virtual-to-value mapping is still precise [13]:

$$\mathsf{precise}\ P \equiv \forall\, Q\ R.\ (P \wedge^*\ Q\ \lfloor\wedge\rfloor\ R) = ((P \wedge^*\ Q)\ \lfloor\wedge\rfloor\ (P \wedge^*\ R))$$

```
datatype aexp =
    HeapLookup aexp
  | BinOp (val ⇒ val ⇒ val) aexp aexp
  | UnOp (val ⇒ val) aexp
  | Const val

datatype com =
    SKIP
  | aexp := aexp
  | com; com
  | IF bexp THEN com ELSE com
  | WHILE bexp DO com

datatype bexp =
    BConst bool
  | BComp (val ⇒ val ⇒ bool) aexp aexp
  | BBinOp (bool ⇒ bool ⇒ bool) bexp bexp
  | BNot bexp
```

Fig. 2. Syntax of the heap based WHILE language.

That is, it distributes over conjunction in both directions. We write $P \lfloor\wedge\rfloor Q$ for the lifted conjunction of assertions P and Q: $P \lfloor\wedge\rfloor Q \equiv \lambda x.\ P\ x \wedge Q\ x$

5 The Logic

This section introduces a simple, heap based programming language with pointer arithmetic to analyse how the assertions presented above can be developed into a full Separation Logic. For the meta-level proofs in this paper, we provide a deep embedding of the language into Isabelle/HOL. The language is standard, with skip, if, while, and assignment. The WHILE and IF statements potentially read from virtual memory in their guards. Assignment is the most interesting: it accesses memory through the virtual memory layer and can potentially modify this translation by writing to the page table. We leave out the simpler physical write access for the presentation here. It would be easy to add and does not increase the complexity of the language.

Fig. 2 shows the Isabelle datatypes that make up the syntax of the language. Note that the left hand side of assignments can be an arbitrary arithmetic expression. For simplicity, we identify values and pointers in this language and admit arbitrary HOL functions for comparison and arithmetic expressions. The program states are the same states that the separation assertions work on. To keep the number of statements small, we only provide the more complicated case of virtual access: even low-level page table manipulations are translated.

The semantics of boolean and arithmetic expressions is shown in Fig. 3. We write $[\![B]\!]_b$ for the meaning of boolean expression B as a partial function from program state to *bool*. Analogously $[\![A]\!]$ is a partial function from state to values. All of these are straightforward, only heap lookup deserves more attention.

Heap lookup only succeeds if the address the argument of the lookup evaluates to virtually maps to a value. This means that the appropriate slices that are involved in the page table lookup as well as the full cell of the target in physical memory must be available in the domain of the heap. In any execution, there will always be full memory cells available, but using our assertions from above we

$$
\begin{array}{lcl}
[\![\mathsf{Const}\ c]\!]\ s & = & \lfloor c \rfloor \\
[\![\mathsf{HeapLookup}\ vp]\!]\ s & = & \mathsf{case}\ [\![vp]\!]\ s\ \mathsf{of\ None} \Rightarrow \mathsf{None} \\
 & & \mid \lfloor v \rfloor \Rightarrow \mathsf{if}\ (\mathsf{VPtr}\ v \hookrightarrow -)\ s\ \mathsf{then\ as\text{-}view\ ptable\text{-}lift}\ s\ (\mathsf{VPtr}\ v)\ \mathsf{else\ None} \\
[\![\mathsf{BinOp}\ f\ e_1\ \ e_2]\!]\ s & = & \mathsf{case}\ ([\![e_1]\!]\ s,\ [\![e_2]\!]\ s)\ \mathsf{of} \\
 & & (\lfloor v_1 \rfloor,\ \lfloor v_2 \rfloor) \Rightarrow \lfloor f\ v_1\ \ v_2 \rfloor \mid \text{-} \Rightarrow \mathsf{None} \\
[\![\mathsf{UnOp}\ f\ e]\!]\ s & = & \mathsf{case}\ [\![e]\!]\ s\ \mathsf{of\ None} \Rightarrow \mathsf{None} \mid \lfloor v \rfloor \Rightarrow \lfloor f\ v \rfloor
\end{array}
$$

$$
\begin{array}{lcl}
[\![\mathsf{BConst}\ b]\!]_b\ s & = & \lfloor b \rfloor \\
[\![\mathsf{BComp}\ f\ e_1\ \ e_2]\!]_b\ s & = & \mathsf{case}\ ([\![e_1]\!]\ s,\ [\![e_2]\!]\ s)\ \mathsf{of} \\
 & & (\lfloor v_1 \rfloor,\ \lfloor v_2 \rfloor) \Rightarrow \lfloor f\ v_1\ \ v_2 \rfloor \mid \text{-} \Rightarrow \mathsf{None} \\
[\![\mathsf{BBinOp}\ f\ e_1\ \ e_2]\!]_b\ s & = & \mathsf{case}\ ([\![e_1]\!]_b\ s,\ [\![e_2]\!]_b\ s)\ \mathsf{of} \\
 & & (\lfloor v_1 \rfloor,\ \lfloor v_2 \rfloor) \Rightarrow \lfloor f\ v_1\ \ v_2 \rfloor \mid \text{-} \Rightarrow \mathsf{None} \\
[\![\mathsf{BNot}\ b]\!]_b\ s & = & \mathsf{case}\ [\![b]\!]_b\ s\ \mathsf{of\ None} \Rightarrow \mathsf{None} \mid \lfloor v \rfloor \Rightarrow \lfloor \neg\ v \rfloor
\end{array}
$$

Fig. 3. Semantics of arithmetic and boolean expressions.

are able to make finer-grained distinctions during program proofs. The function $\mathsf{as\text{-}view\ ptable\text{-}lift}\ (fh,\ r)\ vp = (\mathsf{let}\ hp = \mathsf{h\text{-}view}\ fh\ vp\ \mathsf{in\ ptable\text{-}lift}\ hp\ r \rhd hp)\ vp$ uses h-view to read the value from the heap after the address has been translated, and $f \rhd g \equiv \lambda x.\ \mathsf{case}\ f\ x\ \mathsf{of\ None} \Rightarrow \mathsf{None} \mid \lfloor y \rfloor \Rightarrow g\ y$ to compose the two partial functions.

Finally, Fig. 4 shows a big-step operational semantics of commands in the language. We write $\langle c,s \rangle \rightarrow s'$ if command c, started in s, evaluates to s'. As usual, the semantics is the smallest relation satisfying the rules of Fig. 4. Non-termination is modelled by absence of the transition from the relation. In contrast to this, we model memory access failure explicitly by writing $\langle c,s \rangle \rightarrow \mathsf{None}$, and we do not allow accessing unmapped pages. On hardware, this would lead to a page fault and the execution of a page fault handler. This can also be modelled by making the assignment statement a conditional jump to the page fault handler, or with an abstraction layer showing that the page fault handler always establishes a known good state by mapping pages in from disk. For the verification of the seL4 micro-kernel [6,7] (the larger context of this work), we intend to show absence of page faults in the kernel itself, hence our stricter model. Excluding the conditional jump is no loss of generality as we already have IF statements in the language.

As with arithmetic expressions, the most interesting rule in Fig. 4 is assignment which involves heap access and which we will explain below. In the other rules, we abbreviate $[\![b]\!]_b\ s = \lfloor \mathsf{True} \rfloor$ with $<b>\ s$, and $[\![b]\!]_b\ s = \lfloor \mathsf{False} \rfloor$ with $\neg<b>\ s$. Failure in any part of the execution leads to failure of the whole statement.

The assignment rule in the top right corner of Fig. 4 requires that both the arithmetic expressions for the left and right hand side evaluate without failure. The left hand side is taken as a virtual pointer, the right hand side as the value being assigned. The assignment succeeds if the virtual address vp is mapped and allocated. We use the notation $s\ [vp,- \mapsto v]$ to update with v all slices in the heap belonging to the physical address that vp resolves to. Since heap writes always update such full cells, all heaps in executions will only ever consist of full cells and are thus consistent with the usual, non-sliced view of memory. Slices are a tool for assertions and proofs only.

$$\langle\mathsf{SKIP},s\rangle \to \lfloor s\rfloor \qquad \frac{[\![lval]\!]\ s = \lfloor vp\rfloor \quad [\![rval]\!]\ s = \lfloor v\rfloor \quad (\mathsf{VPtr}\ vp \hookrightarrow -)\ s}{\langle lval := rval\ ,s\rangle \to \lfloor s\ [\mathsf{VPtr}\ vp,- \mapsto_v\ v]\rfloor}$$

$$\frac{[\![lval]\!]\ s = \mathsf{None} \lor [\![rval]\!]\ s = \mathsf{None}}{\langle lval := rval\ ,s\rangle \to \mathsf{None}} \quad \frac{[\![lval]\!]\ s = \lfloor vp\rfloor \quad \neg\ (\mathsf{VPtr}\ vp \hookrightarrow -)\ s}{\langle lval := rval\ ,s\rangle \to \mathsf{None}}$$

$$\frac{\langle c_0,s\rangle \to \lfloor s''\rfloor \quad \langle c_1,s''\rangle \to s'}{\langle c_0\ ;\ c_1,s\rangle \to s'} \qquad \frac{\langle c_0,s\rangle \to \mathsf{None}}{\langle c_0\ ;\ c_1,s\rangle \to \mathsf{None}}$$

$$\frac{<b>\ s \quad \langle c_0,s\rangle \to s'}{\langle\mathsf{IF}\ b\ \mathsf{THEN}\ c_0\ \mathsf{ELSE}\ c_1,s\rangle \to s'} \qquad \frac{\neg<b>\ s \quad \langle c_1,s\rangle \to s'}{\langle\mathsf{IF}\ b\ \mathsf{THEN}\ c_0\ \mathsf{ELSE}\ c_1,s\rangle \to s'}$$

$$\frac{[\![b]\!]_b\ s = \mathsf{None}}{\langle\mathsf{IF}\ b\ \mathsf{THEN}\ c_0\ \mathsf{ELSE}\ c_1,s\rangle \to \mathsf{None}} \qquad \frac{\neg<b>\ s}{\langle\mathsf{WHILE}\ b\ \mathsf{DO}\ c,s\rangle \to \lfloor s\rfloor}$$

$$\frac{<b>\ s \quad \langle c,s\rangle \to \lfloor s''\rfloor \quad \langle\mathsf{WHILE}\ b\ \mathsf{DO}\ c,s''\rangle \to s'}{\langle\mathsf{WHILE}\ b\ \mathsf{DO}\ c,s\rangle \to s'}$$

$$\frac{<b>\ s \quad \langle c,s\rangle \to \mathsf{None}}{\langle\mathsf{WHILE}\ b\ \mathsf{DO}\ c,s\rangle \to \mathsf{None}} \qquad \frac{[\![b]\!]_b\ s = \mathsf{None}}{\langle\mathsf{WHILE}\ b\ \mathsf{DO}\ c,s\rangle \to \mathsf{None}}$$

where

$$\begin{array}{ll} (h,\ r)\ [vp,- \mapsto_v\ v] & \equiv \mathsf{case\ vmap\text{-}view}\ (h,\ r)\ vp\ \mathsf{of}\ \lfloor p\rfloor \Rightarrow (h\ [p,- \mapsto v],\ r) \\ \mathsf{vmap\text{-}view}\ (h,\ r)\ vp & \equiv \mathsf{ptable\text{-}lift}\ (\mathsf{h\text{-}view}\ h\ vp)\ r\ vp \\ h\ [p,- \mapsto v] & \equiv \lambda p'.\ \mathsf{if\ fst}\ p' = p\ \mathsf{then}\ \lfloor v\rfloor\ \mathsf{else}\ h\ p' \end{array}$$

Fig. 4. Big-step semantics of commands.

Having shown the semantics, we can now proceed to defining Hoare triples. Validity is the usual:

$$\{\!|P|\!\}\ c\ \{\!|Q|\!\} \equiv \forall s\ s'.\ \langle c,s\rangle \to s' \land P\ s \longrightarrow (\exists r.\ s' = \lfloor r\rfloor \land Q\ r)$$

We do not define a separate syntactic Hoare calculus. Instead, we define validity only and then derive the Hoare rules as theorems in Isabelle/HOL directly. Fig. 5 shows the rules we have proved for this language. Again, the rules for IF, WHILE, etc., are straightforward and the same as in a standard Hoare calculus. We write $P\ \lfloor\longrightarrow\rfloor\ Q \equiv \forall s.\ P\ s \longrightarrow Q\ s$ for lifted implication, and $\ll b \gg\ s$ to denote that $[\![b]\!]_b\ s \neq \mathsf{None}$. The precondition P in the IF and WHILE case must be strong enough to guarantee failure free evaluation of the condition b. The lifting rules for conjunction and disjunction, as well as the weakening rule are easy to prove, requiring only the definition of validity and separating conjunction. The interesting cases are the assignment rule and the frame rule.

The assignment rule is more complex looking than the standard rule of Separation Logic. Here, both the left and right hand side of the assignment are arbitrary expressions, potentially including heap lookups that need to be evaluated first. This is the reason for the additional P conjunct separate from the basic pointer mapping. It is not an artifact of virtual memory, but one of expression evaluation only. In essence, this is a rule schema. P can be picked to be just strong enough for the evaluation of the expressions to succeed. For instance, if the left hand side were to contain one heap lookup for location x only,

$$\{\!|P|\!\}\ \mathsf{SKIP}\ \{\!|P|\!\} \qquad \frac{\{\!|P|\!\}\ c\ \{\!|Q|\!\} \quad P' \lfloor\longrightarrow\rfloor P \quad Q \lfloor\longrightarrow\rfloor Q'}{\{\!|P'|\!\}\ c\ \{\!|Q'|\!\}}$$

$$\frac{\{\!|P \lfloor\wedge\rfloor <b>|\!\}\ c_1\ \{\!|Q|\!\} \quad \{\!|P \lfloor\wedge\rfloor \neg<b>|\!\}\ c_2\ \{\!|Q|\!\}}{\{\!|P \lfloor\wedge\rfloor \ll b \gg|\!\}\ \mathsf{IF}\ b\ \mathsf{THEN}\ c_1\ \mathsf{ELSE}\ c_2\ \{\!|Q|\!\}}$$

$$\frac{\{\!|P \lfloor\wedge\rfloor <b>|\!\}\ c\ \{\!|P|\!\} \quad P \lfloor\longrightarrow\rfloor \ll b \gg}{\{\!|P|\!\}\ \mathsf{WHILE}\ b\ \mathsf{DO}\ c\ \{\!|P \lfloor\wedge\rfloor \neg<b>|\!\}} \qquad \frac{\{\!|P|\!\}\ c_1\ \{\!|Q|\!\} \quad \{\!|Q|\!\}\ c_2\ \{\!|R|\!\}}{\{\!|P|\!\}\ c_1\ ;\ c_2\ \{\!|R|\!\}}$$

$$\frac{\{\!|P|\!\}\ c\ \{\!|Q|\!\} \quad \{\!|R|\!\}\ c\ \{\!|S|\!\}}{\{\!|P \lfloor\wedge\rfloor R|\!\}\ c\ \{\!|Q \lfloor\wedge\rfloor S|\!\}} \qquad \frac{\{\!|P|\!\}\ c\ \{\!|Q|\!\} \quad \{\!|R|\!\}\ c\ \{\!|S|\!\}}{\{\!|P \lfloor\vee\rfloor R|\!\}\ c\ \{\!|Q \lfloor\vee\rfloor S|\!\}}$$

$$\{\!|(\mathsf{VPtr}\ vp \mapsto - \wedge^* P) \lfloor\wedge\rfloor [\![l]\!] = \lfloor vp\rfloor \lfloor\wedge\rfloor [\![r]\!] = \lfloor v\rfloor|\!\}\ l := r\ \ \{\!|\mathsf{VPtr}\ vp \mapsto v \wedge^* P|\!\}$$

$$\frac{\{\!|P|\!\}\ c\ \{\!|Q|\!\}}{\{\!|P \wedge^* R|\!\}\ c\ \{\!|Q \wedge^* R|\!\}}$$

Fig. 5. The proof rules for Mapped Separation Logic.

we would choose P to be of the form $x \mapsto r$. The rule is sound for too strong and too weak P: either the precondition is merely stronger than it needs to be, or P is too weak to support the expression evaluation conjunct and the precondition as a whole becomes false.

The proof of the assignment rule proceeds by unfolding the definitions and observing that the postcondition is established by the memory update, noting the fact that the ptable-trace from the precondition fully accounts for all page table entries that are relevant in the postcondition and that P is preserved, because it is separate from the heap in which the update occurs. Since the physical address of the write is separate from the page table lookup for this address and from P, the translation layer for their heaps is not affected by the write. To reason about page table updates we need a slightly stronger rule that unfolds the virtual-to-value mapping and lets us talk about the physical address p:

$$\begin{gathered}\{\!|(\mathsf{VPtr}\ vp \mapsto_v p \wedge^* p \to_p - \wedge^* P) \lfloor\wedge\rfloor [\![l]\!] = \lfloor vp\rfloor \lfloor\wedge\rfloor [\![r]\!] = \lfloor v\rfloor|\!\}\\ l := r\\ \{\!|\mathsf{VPtr}\ vp \mapsto_v p \wedge^* p \mapsto_p v \wedge^* P|\!\}\end{gathered}$$

Reasoning with the new mapping predicates is similar to abstract-predicate style reasoning [14] if we never unfold their definitions in client proofs. As we will see in the case study, we only need to do this locally if we are reasoning about changes to the page table and we are interested in the page that is being modified. This obviously heavily depends on the page table encoding. Application level reasoning can proceed fully abstractly.

The second interesting proof rule is the frame rule that allows global reasoning based on local proofs. In many ways it can be seen as the core of Separation

$$\frac{\mathsf{ptable\text{-}lift}\ (h_0\ {+}{+}\ h_1)\ r\ vp = \lfloor p \rfloor \qquad h_0 \perp h_1}{\mathsf{ptable\text{-}lift}\ h_0\ r\ vp = \lfloor p \rfloor \lor \mathsf{ptable\text{-}lift}\ h_0\ r\ vp = \mathsf{None}} \qquad \frac{\mathsf{ptable\text{-}lift}\ h_0\ r\ vp = \lfloor p \rfloor \qquad h_0 \perp h_1}{\mathsf{ptable\text{-}lift}\ (h_0\ {+}{+}\ h_1)\ r\ vp = \lfloor p \rfloor}$$

$$\frac{\mathsf{get\text{-}page}\ vp = \mathsf{get\text{-}page}\ vp' \qquad \mathsf{ptable\text{-}lift}\ h\ r\ vp = \lfloor val \rfloor \qquad \mathsf{ptable\text{-}lift}\ h'\ r\ vp' = \lfloor val' \rfloor}{\mathsf{ptable\text{-}trace}\ h\ r\ vp = \mathsf{ptable\text{-}trace}\ h'\ r\ vp'}$$

$$\frac{p \notin \mathsf{ptable\text{-}trace}\ h\ r\ vp \qquad \mathsf{ptable\text{-}lift}\ h\ r\ vp = \lfloor p \rfloor}{\mathsf{ptable\text{-}lift}\ (h(p \mapsto v))\ r\ vp = \lfloor p \rfloor}$$

$$\frac{p \notin \mathsf{ptable\text{-}trace}\ h\ r\ vp \qquad \mathsf{ptable\text{-}lift}\ h\ r\ vp = \lfloor p \rfloor}{\mathsf{ptable\text{-}trace}\ (h(p \mapsto v))\ r\ vp = \mathsf{ptable\text{-}trace}\ h\ r\ vp}$$

Fig. 6. The page table interface.

Logic. Calcagno et al. [4] provide an abstract framework for identifying a logic as Separation Logic and distill out the central property of locality. In their setting, our separation algebra is the common heap monoid where the binary operation is heap merge lifted to the $heap \times pptr$ type. Our definition of separating conjunction then coincides with the one in the framework, and to show that our logic is a Separation Logic, we only need to show that all actions in the programming language are local. Locality is equivalent to the combination of safety monotonicity and the frame property [4]. In our setting, these two are:

Lemma 1 (Safety Monotonicity). *If a small state provides enough resources to run a command c, then so does a larger state. Formally, the converse is easier to state: If* $\langle c,(h\ {+}{+}\ h',\ r)\rangle \rightarrow \mathsf{None}$ *and* $h \perp h'$ *then* $\langle c,(h,\ r)\rangle \rightarrow \mathsf{None}$.

Lemma 2 (Frame Monotonicity). *Execution of a command can be traced back to a smaller part of the state as long as the command executes at all on the smaller state. If* $\neg\ \langle c,(h_0,\ r)\rangle \rightarrow \mathsf{None}$ *and* $\langle c,(h_0\ {+}{+}\ h_1,\ r)\rangle \rightarrow \lfloor (h',\ r) \rfloor$ *and* $h_0 \perp h_1$ *then* $\exists\, h_0{}'.\ h' = h_0{}'\ {+}{+}\ h_1 \land \langle c,(h_0,\ r)\rangle \rightarrow \lfloor (h_0{}',\ r) \rfloor$.

We have not formalised the full abstract, relational treatment of the framework by Calcagno et al., but shown the above two properties and the implied frame rule directly by induction on the evaluation of commands.

Fig. 6 gives the page table interface constraints that we promised in Sect. 3. These rules need to be proved about ptable-lift and ptable-trace for a new page table instantiation in order to use the abstract logic presented before. The first two rules are the frame and monotonicity property on ptable-lift. The third rule states that if the domain of ptable-lift does not change, neither does ptable-trace. The last two rules state that updates to the heap outside the trace affect neither the lifting nor the trace of the page table. We have proved the rules in Fig. 6 for the one-level page table instantiation in the examples.

6 Case Study

In this section, we present a small page allocation and assignment routine one might see in operating system services, mapping a *frame*, the physical equivalent

```
1. fte := ft_free_list;
2. IF fte != NULL THEN
3.      ft_free_list := *ft_free_list;
4.      frame := &fte - &frame_table;
5.      *(ptable + (a2p page_addr)) := f2a frame OR valid_pmask;
6.      ret_val := 0
7. ELSE ret_val := -1
```

Fig. 7. A simple page table manipulating program.

of a page, to some address in virtual memory. The program appears in Fig. 7 with simplified syntax. Frame availability information is stored in a frame table, which contains one entry per frame in the system, marking it as used or unused. In our program, line 1 attempts to find a free frame's entry in the free frame list. An empty list causes an erroneous return at line 7. Line 3 removes the head of the list. Line 4 calculates the number of the frame from its entry. Upon successfully allocating a frame, line 5 updates the page table with the appropriate entry mapping page-addr to the new frame.

We have proved that the program conforms to the following specification:

$$\{\!|\mathsf{vars} \wedge^* \mathsf{in\text{-}pt\ page\text{-}addr} \wedge^* \mathsf{frame\text{-}list}\ (f{\cdot}fs) \wedge^* \mathsf{pt\text{-}alloc\ page\text{-}addr} \wedge^* \mathit{ret\text{-}val}_v \mapsto -|\!\}$$
$$\mathsf{program\ page\text{-}addr}$$
$$\{\!|\mathsf{vars} \wedge^* \mathsf{in\text{-}pt\ page\text{-}addr} \wedge^* \mathsf{frame\text{-}list}\ fs \wedge^* \mathsf{VPtr}\ f \mapsto - \wedge^* \mathit{ret\text{-}val}_v \mapsto 0 \wedge^* \mathsf{page\text{-}mapped\ page\text{-}addr}|\!\}$$

where:

$\mathsf{vars} \equiv \mathit{ptable}_v \mapsto \mathit{pt} \wedge^* \mathit{frame}_v \mapsto - \wedge^* \mathit{frame\text{-}table}_v \mapsto \mathit{frame\text{-}table} \wedge^* \mathit{fte}_v \mapsto -$
$\mathsf{in\text{-}pt\ page\text{-}addr}\ (h, r) \equiv (\mathsf{VPtr}\ \mathit{pt} + \mathsf{a2p\ page\text{-}addr} \mapsto_v r + \mathsf{a2p\ page\text{-}addr})\ (h, r)$
$\mathsf{pt\text{-}alloc\ page\text{-}addr} \equiv \lambda(h, r).\ (r + \mathsf{get\text{-}page\ page\text{-}addr} \rightarrow_p -)\ (h, r)$
$\mathsf{frame\text{-}list}\ xs \equiv \mathsf{list}\ xs\ \mathit{ft\text{-}free\text{-}list}_v\ (\mathsf{fte\text{-}property}\ \mathit{frame\text{-}table})$
$\mathsf{list}\ []\ h\ P \equiv h \mapsto 0$
$\mathsf{list}\ (x{\cdot}xs)\ h\ P \equiv \lambda s.\ x \neq 0 \wedge (h \mapsto x \wedge^* \mathsf{list}\ xs\ (\mathsf{VPtr}\ x)\ P \wedge^* P\ x)\ s$
$\mathsf{fte\text{-}property}\ \mathit{start}\ \mathit{ptr} \equiv \mathsf{entire\text{-}frame}\ (\mathsf{PPtr}\ (\mathsf{f2a}\ (\mathit{ptr} - \mathit{start})))\ \{\rightarrow_p\}\ -$
$\mathsf{entire\text{-}frame}\ p \equiv \{p..\mathsf{PPtr}\ (\mathsf{pptr\text{-}val}\ p + 0xFFF)\}$

The functions a2p and f2a convert addresses to page numbers and frame numbers to addresses by respectively dividing and multiplying by 4096.

Since the language in this paper is purely heap based, we use $\mathit{var}_v \mapsto \mathit{var}$ to denote that a variable *var* has a specific value and to represent *var* in Fig. 7. We assume that the page table is accessible from *ptable*, and the frame table lies at *frame-table*. Further, we have a non-empty free list starting at *first-free*, where each address in the list indicates the presence of a frame. Finally, we require that the page table entry that is used to resolve a lookup of page-addr is allocated, and accessible from our address space. We additionally require that page-addr is aligned to the page size. As a result of executing the program, the free frame list becomes shorter and the page at page-addr is fully accessible. The latter is:

$\mathsf{page\text{-}mapped}\ vp \equiv \mathsf{entire\text{-}page}\ vp\ \{\mapsto\}\ - \wedge^* \mathsf{consume\text{-}slices}\ vp\ (\mathcal{U} - \mathsf{entire\text{-}page}\ vp)$
$\mathsf{consume\text{-}slice}\ vp\ sl \equiv \lambda(h, r).\ \mathsf{dom}\ h = \mathsf{ptable\text{-}trace}\ (\mathsf{h\text{-}view}\ h\ sl)\ r\ vp \times \{sl\}$
$\mathsf{consume\text{-}slices}\ vp\ S \equiv \mathsf{fold\ op} \wedge^* (\mathsf{consume\text{-}slice}\ vp)\ \square\ S$
$\mathsf{entire\text{-}page}\ vp \equiv \{vp..\mathsf{VPtr}\ (\mathsf{vptr\text{-}val}\ vp + 0xFFF)\}$

All our separation logic statements work on heaps of a precise size. Using up the entire page table entry in our precondition means we must likewise use it all in the postcondition. A page table entry maps 4096 addresses, but is made up of more slices. Stating that those addresses are mapped does not consume all the slices. The difference in heap size is made up by consume-slices.

The actual mapping-in step from line 5 of our program performs a write to the page table at pt + a2p page-addr. In addition to our assignment rule, we have shown another property that allows us to to conclude from the post-state of line 5 that the page at page-addr is now completely mapped:

$$\text{entire-frame (PPtr } (\mathit{f2\ frame})) \rightarrow - \wedge^* \text{ pt-map page-addr } \mathit{frame}$$
$$\lfloor\longrightarrow\rfloor \text{ page-mapped page-addr}$$

$$\text{pt-map page-addr } \mathit{frame} \equiv$$
$$\lambda(h,\ r).\ (\text{PPtr (pptr-val } r + \text{a2p page-addr}) \mapsto_p \text{f2a } \mathit{frame} \text{ OR } 1)\ (h,\ r)$$

This rule is the only place in the case study where we had to unfold page table definitions and reason directly about the encoding. All other reasoning used Separation Logic rules only.

In order to check successful interaction with the newly mapped page, we added an extra segment to our program.

```
*page_addr := 0xFF;
*(page_addr + 3) := *page_addr + 2
```

If executed on a state with offsets 0 and 3 in the page mapped and allocated:

$$\{\!|\text{page-addr} \mapsto - \wedge^* \text{ page-addr} + 3 \mapsto -|\!\}$$

it results in those offsets set to `0xFF` and `0x101`:

$$\{\!|\text{page-addr} \mapsto 0xFF \wedge^* \text{ page-addr} + 3 \mapsto 0x101|\!\}$$

These two offsets are part of the page at page-addr; the program fragment does not change anything else. As we can rewrite

$$\text{page-mapped } p = (p \mapsto - \wedge^* p + 3 \mapsto - \wedge^* \text{ page-mapped } \{p,\ p + 3\}\ p)$$

we can invoke the frame rule and include the rest of the state.

Our case study shows that our logic allows abstract reasoning, even in the presence of page table manipulation. Examples like this would occur when verifying OS code directly. The logic is easier to use for application code that might support sharing, but has no direct access to the page table.

7 Related work

The primary focus of this work is enhancement of Separation Logic, originally conceived by O'Hearn, Reynolds et al. [9,15]. Separation logic has previously been formalised in mechanised theorem proving systems [18,1]. We enhance these models with the ability to reason about properties on virtual memory.

Previous work in OS kernel verification has involved verifying the virtual memory subsystem [10,5,8]. Reasoning about programs *running* under virtual memory, however, especially the operating systems which control it, remains

mostly unexplored. The challenges of reasoning about virtual memory are explored in the development of the Robin micro-hypervisor [16]. Like our work, the developers of Robin aim to use a single semantics to describe all forms of memory access which simplifies significantly in the well-behaved case. They focus on reasoning about "plain memory" in which no virtual aliasing occurs and split it into read-only and read-write regions, to permit reading the page table while in plain memory. They do not use Separation Logic notation. Our work is more abstract. We do not explicitly define "plain memory". Rather the concept emerges from the requirements and state.

Alkassar et al. [2] have proved the correctness of a kernel page fault handler, albeit not at the Separation Logic level. As in our setting, they use a single level page table and prove that the page fault handler establishes the illusion to the user of a plain memory abstraction, swapping in pages from disk as required. The proof establishes simulation between those two layers, with the full complexity of the page table encoding visible at the lower level for the whole verification. The work presented in this paper focuses on a logic for reasoning about virtual memory instead. The aim is to make such verifications easier and more productive.

Tuch et al. demonstrated the extension of Separation Logic to reasoning about C programs involving pointer manipulation [17]. We believe our framework is orthogonal and can be instantiated readily to the C model.

8 Conclusion and Future Work

We have presented an extension of Separation Logic which allows reasoning about virtual memory and processes running within. The logic fully supports the frame rule as well as the other Separation Logic proof rules and allows for a convenient representation of predicates on memory at three levels: the virtual map, physical heap and virtual address space. We have presented a small case study that demonstrates the applicability of the logic to OS level page table code as well as client code using the page table mechanism.

For analysing the logic in this paper we chose a simplified machine and page table instance. The logic does not depend on the implementation of either.

We have significantly extended our previous work on virtual memory and have managed to fully hide the complexity of virtual memory reasoning for code that does not directly modify the page table. We also have shown that high-level reasoning is still possible for code that does. The concepts in the logic are close to the mental model kernel programmers have of virtual memory.

Practical implementations of virtual memory in hardware use a cache, the translation lookaside buffer (TLB). To include this concept in our model, we could add standard cache consistency protocols such as TLB flushes when appropriate.

The next steps are to fully instantiate this model to the C programming language and apply it to the verification of the seL4 microkernel.

Acknowledgements We thank Michael Norrish, Harvey Tuch, Simon Winwood, and Thomas Sewell for discussions and comments on earlier versions of this paper as well as Hongsoek Yang for sharing his insight into Separation Logic.

References

1. R. Affeldt and N. Marti. Separation logic in Coq. `http://savannah.nongnu.org/projects/seplog`, 2008.
2. E. Alkassar, N. Schirmer, and A. Starostin. Formal pervasive verification of a paging mechanism. In *Proc14th Int'l Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, LNCS, Budapest, Hungary, Apr. 2008. Springer. to appear.
3. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL '05: Proc32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 259–270. ACM, 2005.
4. C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS '07: Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science*, pages 366–378. IEEE Computer Society, 2007.
5. I. Dalinger, M. A. Hillebrand, and W. J. Paul. On the verification of memory management mechanisms. In D. Borrione and W. J. Paul, editors, *CHARME*, volume 3725 of *LNCS*, pages 301–316. Springer, 2005.
6. P. Derrin, K. Elphinstone, G. Klein, D. Cock, and M. M. T. Chakravarty. Running the manual: An approach to high-assurance microkernel development. In *Proc. ACM SIGPLAN Haskell WS*, Portland, OR, USA, Sept. 2006.
7. K. Elphinstone, G. Klein, P. Derrin, T. Roscoe, and G. Heiser. Towards a practical, verified kernel. In *Proc. 11th Workshop on Hot Topics in Operating Systems*, page 6, San Diego, CA, USA, May 2007.
8. M. Hillebrand. *Address Spaces and Virtual Memory: Specification, Implementation, and Correctness*. PhD thesis, Saarland University, Saarbrücken, 2005.
9. S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 14–26. ACM, 2001.
10. G. Klein and H. Tuch. Towards verified virtual memory in L4. In K. Slind, editor, *TPHOLs Emerging Trends '04*, Park City, Utah, USA, 2004.
11. R. Kolanski. A logic for virtual memory. In R. Huuck, G. Klein, and B. Schlich, editors, *Proc. 3rd Int'l Workshop on Systems Software Verification (SSV'08)*, ENTCS, pages 55–70. Elsevier, Feb. 2008. to appear.
12. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
13. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL '04: Proc. 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 268–280. ACM, 2004.
14. M. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL '05: Proc. 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 247–258. ACM, 2005.
15. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. 17th IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.
16. H. Tews. Formal methods in the Robin project: Specification and verification of the Nova microhypervisor. In *C/C++ Verification Workshop, Technical Report ICIS-R07015*, pages 59–68, Oxford, UK, July 2007. Radboud University Nijmegen.
17. H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *POPL '07*, pages 97–108. ACM, 2007.
18. T. Weber. Towards mechanized program verification with separation logic. In J. Marcinkowski and A. Tarlecki, editors, *Computer Science Logic – 18th Int'l Workshop, CSL 2004*, volume 3210 of *LNCS*, pages 250–264. Springer, 2004.