A Logic for Virtual Memory

Rafal Kolanski

Sydney Research Lab., National ICT Australia, Australia [1]
School of Computer Science and Engineering, UNSW, Sydney, Australia
rafal.kolanski@nicta.com.au

Abstract

We present an extension to classical separation logic which allows reasoning about virtual memory. Our logic is formalised in the Isabelle/HOL theorem prover in a manner allowing classical separation logic notation to be used at an abstract level. We demonstrate that in the common cases, such as user applications, our logic reduces to classical separation logic. At the same time we can express properties about page tables, direct physical memory access, virtual memory access, and shared memory in detail.

Keywords: Separation Logic, Virtual Memory, Interactive Theorem Proving

1 Introduction

Separation logic [14] has been used for verification of shared mutable data structures at the application level, such as those involved in C programs [18]. While effective, these techniques assume a view of memory as a function from addresses to values. For operating system verification, the situation is more complex. On hardware incorporating the virtual memory abstraction, two different virtual addresses may point to the same physical address. In the majority of operating system code this is not a problem, but the application view is insufficient for verifying the parts involved with the virtual mappings themselves, such as shared memory [16].

The virtual memory abstraction offers flexible, dynamic allocation of physical memory to running processes. It allows each process its own view of physical memory, called a virtual address space, via a set of virtual to physical address mappings. The mappings are stored in memory in a structure called the page table.

Applications usually deal only with their own data, but an operating system additionally manages the application's page table as well as its own. A memory write in this situation can result in the view of memory changing. Most of the time, it does not; in cases when it does, only *part* of the address space changes. Inferring

[1] National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

separation of what may have changed from what has not allows effective reasoning about such memory updates.

The virtual memory mechanism itself has been a target of verification [9], however apart from some work in progress [17] we have not encountered any work dealing with proofs about programs that make essential use of the virtual memory subsystem.

Our contribution in this work is a logic allowing effective reasoning about both the virtual and physical layers of memory. Additionally:

- The abstract layer of our separation logic is similar to traditional separation logic and in fact collapses to traditional separation logic for pure application reasoning, where the active page table is not mapped into virtual memory.
- Separating conjunction extends to multiply-mapped memory addresses.
- The low-level details of page table implementation are independent of the logic. We present an instantiation to a simple one-level page table to demonstrate the concept.
- All of the work presented in this paper is formalised in the Isabelle/HOL theorem prover [13].

Like [18], we use a shallow embedding of separation logic, meaning the constructs of assertions are translated to Isabelle/HOL, rather than being considered distinct types in the logic.

As this is a work in progress, we are using a simplified model: we use a simple machine abstraction and do not take into account memory permissions beyond whether a virtual address is mapped/unmapped. As such permissions are merely extra properties of virtual addresses, they can be easily integrated into our logic. Other work [18] has shown how to integrate actual machine encodings with separation logic. We believe our logic can be likewise connected.

While the normal way to present separation logics [14] is to describe a programming language, assertion language and resulting rules, we focus on the assertion language and properties holding on heap updates only. We do this because, like Tuch et al. [18], we aim to place our logic on top of a classical verification condition generator such as that of Schirmer [15]. The generator outputs higher-order logic statements involving the shallow embedding of our logic's assertions. Thus, the programming language is given, and the usual separation logic rules instead become rules about the semantic effect of the program on the heap.

2 Intuition

Separation logic is traditionally based on a model of memory as a partial function from addresses to values, called a heap. When reasoning about shared mutable data structures [14], such as the representation of memory in a language with pointers, separation logic offers a concise way of defining the disjointness of predicates in memory. The primary mechanism for doing this is the separating conjunction. As shown in Fig. 1, it works by dividing up the heap into two disjoint regions on which each side of the conjunction must hold respectively. For instance, it allows us to

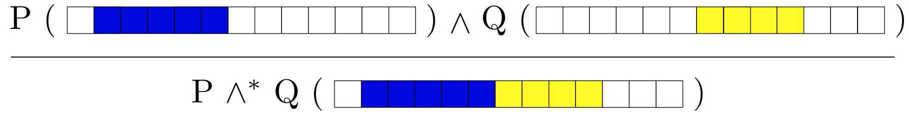

Fig. 1. Separation conjunction.

state two structures do not share memory.

As mentioned in the introduction, virtual memory is a hardware-enforced abstraction allowing each executing process its own view of memory, into which areas of physical memory can be dynamically inserted. It adds a further 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. Direct memory access can be approximated by using a one-to-one virtual-to-physical map.

To save space, virtual to physical mappings are stored in the page table with a granularity of pages whose size is usually dictated by the hardware. The page table resides in physical memory at a location called its root. The full set of virtual-to-physical mappings for a process can thus be lifted from physical memory if the root and type of page table are known.

Each access to a virtual address may result in a *page fault* in cases when the hardware fails to look up an address in the page table, for instance when that page resides on the disk, or has not yet been allocated. The page fault handler decides how to handle such cases. While our logic is aimed at dealing with situations useful in the verification of page fault handlers, we do not present page fault handler verification itself in this paper.

We will henceforth refer to physical memory as the *heap* in the spirit of traditional separation logic, and to the virtual-to-physical map as the *virtual map*.

Our logic is independent of the implementation of the page table. As an example, in this work we use a simple implementation: the one-level page table.

The addition of virtual memory to separation logic raises the issue of what exactly it means for two predicates to be separate, as well as what kind of state space we are to divide into sub-states in order to be able to express their disjointness. We want the ability to make statements on three levels: virtual to physical, physical to values and virtual to values. We also wish to preserve the usefulness of separating assertions in this context, as well as staying close to traditional separation logic notation.

Although the virtual map can be lifted from the heap via the page table, the heap alone is insufficient to make separation statements on all three levels. We observe this in the statement: under separating conjunction, virtual address vp maps to value x and the page table resides somewhere in physical memory. If we split the heap into page table and non-page table subheaps, there is no way to obtain vp's mapping from the non-page table subheap. It also requires carrying around the page table root as part of the state, causing divergence from traditional notation.

Instead, we propose a state consisting of *two maps*: the physical heap and the virtual map. Initially, we establish a valid state in which the virtual map is lifted out of the page table in the heap. Separation conjunction then splits *both* maps. Physical-to-value assertions work on the heap, virtual-to-physical on the virtual

map, and virtual-to-value on the composition of the two, which we will call the *address space*. This has two advantages:

- The page table and heap lifting is independent of our logic.
- Most operations on a single map make sense on a pair of maps, allowing us to use notation identical to standard separation logic and abstract away low-level details.

We begin with explaining basic notation in Sect. 3. In Sect. 4 we discuss the virtual memory abstraction and our specific instantiation of it. In Sect. 5 we introduce our logic, followed by examining its properties in Sect. 6. Finally, we discuss related work in Sect. 7 before concluding.

3 Notation

Our meta-language Isabelle/HOL conforms largely to everyday mathematical notation. This section introduces further non-standard notation and in particular a few basic data types along with their primitive operations.

The space of total functions is denoted by $\Rightarrow$. Type variables are written $'a$, $'b$, etc. The notation $t :: \tau$ means that HOL term t has HOL type τ.

Pairs come with the two projection functions $\mathsf{fst} :: {'a} \times {'b} \Rightarrow {'a}$ and $\mathsf{snd} :: {'a} \times {'b} \Rightarrow {'b}$. *Sets* (type $'a$ set) follow the usual mathematical convention. Intervals are written as follows: $\{m..<n\}$ means $\{i \mid m \leq i < n\}$.

The option type

$$\textbf{datatype}\ {'a}\ option = \mathsf{None} \mid \mathsf{Some}\ {'a}$$

adjoins a new element None to a type $'a$. We use $'a$ *option* to model partial functions, writing $\lfloor a \rfloor$ instead of Some a and $'a \rightharpoonup {'b}$ instead of $'a \Rightarrow {'b}\ option$. The Some constructor has an underspecified inverse called the, satisfying the $\lfloor x \rfloor = x$. Function update is written $f(x := y)$ where $f :: {'a} \Rightarrow {'b}$, $x :: {'a}$ and $y :: {'b}$ and $f(x \mapsto y)$ stands for $f\ (x := \mathsf{Some}\ y)$. Domain restriction is $f \restriction_A$ where $f :: {'a} \rightharpoonup {'b}$ and $(f \restriction_A)\ x = (\mathsf{if}\ x \in A\ \mathsf{then}\ f\ x\ \mathsf{else}\ \mathsf{None})$.

Finite integers are represented by the type $'a$ *word* where $'a$ determines the word length in bits.

Implication is denoted by $\Longrightarrow$ and $[\![\ A_1;\ \ldots;\ A_n\ ]\!] \Longrightarrow A$ abbreviates $A_1 \Longrightarrow (\ldots \Longrightarrow (A_n \Longrightarrow A)\ldots)$.

4 The Virtual Memory Environment

In this section, we describe the memory model our logic is currently based within: the pointer and page table abstractions, as well as their particular instantiations to our simple machine and one-level page table.

4.1 Pointer Abstraction and a Simple Machine

On all hardware platforms we are aware of, virtual and physical pointers are the length of the machine word, thus the same type. We use a generic pointer model

to prevent confusion between them, while still allowing functions working on both. This allows us to restrict parameters to a particular kind of pointer where necessary and overload functionality where not. We begin by wrapping each in a datatype

datatype *'a pptr-t* = PPtr *'a*
datatype *'a vptr-t* = VPtr *'a*

which we enclose in a type class *ptrs* encompassing both types. We define a value extraction function ptr-val satisfying

ptr-val (PPtr x) = x
ptr-val (VPtr x) = x

In order to disambiguate between pointer types in this work, we use *vp* for the virtual and *p* for physical pointers.

Using the above, we define an instantiation of a highly simplified machine: each 32-bit pointer points to a 32-bit word in memory for a total of 2^{32} words in an address space. Note that traditional implementations use 8-bit values. Our three machine types are: *vptr*, *pptr* and *val*, representing virtual pointers, physical pointers and values respectively. On this machine, we define the types of the three memory views described in Sect. 2 to be:

$vmap = vptr \rightharpoonup pptr$
$heap = pptr \rightharpoonup val$
$addr\text{-}space = vptr \rightharpoonup val$

They represent the virtual map, heap and address space types respectively.

4.2 The Page Table Abstraction

We define our page table abstraction using Isabelle's modular reasoning construct called a *locale* [1]. It allows us to define the interface we require of a page table implementation independent of the hardware configuration: the functions each instantiation must provide as well as constraints they must satisfy.

The primary role of a page table is lookups in the virtual map it represents. In order to obtain this map, we require a lifting function ptable-lift. Given a heap and a page table root location, it obtains the virtual map.

ptable-lift::$('pptr \rightharpoonup 'val) \Rightarrow 'pptr \Rightarrow 'vptr \rightharpoonup 'pptr$

Not all page table roots are valid, due to the finiteness of the heap as well as other constraints such as alignment. valid-root defines what is and constitutes a valid location for the page table:

valid-root::$'pptr \Rightarrow bool$

Finally, in order to reason about when the page table is modified, we require information on the page table area, i.e. given a heap and a root pointer, which heap addresses constitute the table:

ptable-area::$('pptr \rightharpoonup 'val) \Rightarrow 'pptr \Rightarrow 'pptr$ set

Note that while the types in a locale are fixed throughout it, they are *abstract*. The *paddr*, *vaddr* and *val* we instantiated our simple machine to *can* instantiate *'paddr*, *'vaddr* and *'val* as their names suggest, but the page table interface specification is completely generic.

Definition 4.1 We require two properties on these functions. Firstly, the relation-

ship between lifting the page table to a virtual map and the page table area: the map must be lifted only out of the page table area. Secondly, that page table area must remain unchanged so long as heap updates don't touch it:

$$\text{valid-root } r \Longrightarrow \text{ptable-lift } h\ r = \text{ptable-lift } (h\restriction_{\text{ptable-area } h\ r})\ r$$
$$[\![\text{valid-root } r;\ p \notin \text{ptable-area } h\ r]\!] \Longrightarrow \text{ptable-area } (h(p \mapsto v))\ r = \text{ptable-area } h\ r$$

Independent of the page table instantiation, theorem 4.2 follows.

Theorem 4.2 *Updating the heap outside the page table area has no effect on the resulting virtual map:*

$$[\![\text{valid-root } r;\ p \notin \text{ptable-area } h\ r]\!] \Longrightarrow \text{ptable-lift } (h(p \mapsto v))\ r = \text{ptable-lift } h\ r$$

Definition 4.3 Using the supplied functions, we additionally define the concept of the page table not being mapped in the address space it defines:

$$\text{ptable-not-mapped } h\ root \equiv \text{let } vmap = \text{ptable-lift } h\ root \text{ in } \forall\, v\ p.\ vmap\ v = \lfloor p \rfloor \longrightarrow p \notin \text{ptable-area } h\ root$$

As mentioned in Sect. 2, 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, its simplicity and contiguity allows for fast experimentation with particular memory layouts. 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:

$$\begin{array}{lcl}
\text{lev1pt-pagesize} & \equiv & 2^{12} \\
\text{lev1pt-size} & \equiv & 2^{32} \text{ div lev1pt-pagesize} \\
\text{get-page } w & \equiv & \text{ptr-val } w >> 12 \\
\text{ptr-remap } vp\ page & \equiv & page \text{ AND NOT } 4095 \text{ OR } vp \text{ AND } 4095 \\
\text{ptable-lift } h\ r\ w & \equiv & \begin{array}{l}\text{case } h\ r + \text{get-page } w \text{ of None} \Rightarrow \text{None} \\ \mid \lfloor addr \rfloor \Rightarrow \text{if } addr\ !!\ 0 \text{ then } \lfloor \text{PPtr (ptr-remap (ptr-val } w)\ addr) \rfloor \text{ else None}\end{array} \\
\text{ptable-area } h\ r & \equiv & \{r..{<}r + \text{lev1pt-size}\}
\end{array}$$

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. This page table satisfies the properties required in Def. 4.1.

5 Extending Separation Logic

Having specified the nature of our memory environment, we will now describe our logic, its properties and relationship to classical separation logic as defined by Reynolds [14]. We will begin by comparing the state structure of the two logics, then follow with introducing all of the traditional separation logic constructs for our new setting, as well as a few constructs specific to our logic.

As mentioned in Sect. 2, our logic is based on a two-map state consisting of a heap and a virtual map. As a two-map is just a pair of maps, most map operations

still make sense. The operations on maps separation logic requires are map override, domain of, disjunction, subset, domain restriction and the empty map:

$$\begin{array}{lcl}
m_1 \mathbin{++} m_2 & \equiv & \lambda x.\ \mathsf{case}\ m_2\ x\ \mathsf{of}\ \mathsf{None} \Rightarrow m_1\ x \mid \lfloor y \rfloor \Rightarrow \lfloor y \rfloor \\
\mathsf{dom}\ m & \equiv & \{a \mid m\ a \neq \mathsf{None}\} \\
m_1 \perp m_2 & \equiv & \mathsf{dom}\ m_1 \cap \mathsf{dom}\ m_2 = \emptyset \\
m_1 \subseteq_m m_2 & \equiv & \forall\, a{\in}\mathsf{dom}\ m_1.\ m_1\ a = m_2\ a \\
m{\restriction}_A & \equiv & \lambda x.\ \mathsf{if}\ x \in A\ \mathsf{then}\ m\ x\ \mathsf{else}\ \mathsf{None}
\end{array}$$

Overriding (++) takes the value of an entry in the first map when it is not defined in the second. The domain of a map (dom) is the set of all defined values. Map disjointness ($\perp$) implies their domains are disjoint. A map is a subset of another ($\subseteq_m$) if all entries in the smaller map have the same values in the larger one. Finally, restricting a map ($\restriction$) is the same as restricting its domain.

We expand these to two-maps, using almost identical notation:

$$\begin{array}{lcl}
(a,\ b) \mathbin{++} (c,\ d) & \equiv & (a \mathbin{++} c,\ b \mathbin{++} d) \\
tdom\ (a,\ b) & \equiv & (\mathsf{dom}\ a,\ \mathsf{dom}\ b) \\
(a,\ b) \perp (c,\ d) & \equiv & a \perp c \wedge b \perp d \\
(a,\ b) \subseteq_t (c,\ d) & \equiv & a \subseteq_m c \wedge b \subseteq_m d \\
(a,\ b){\restriction}_{(c,\ d)} & \equiv & (a{\restriction}_c,\ b{\restriction}_d) \\
\mathsf{empty} & \equiv & (\mathsf{empty},\ \mathsf{empty})
\end{array}$$

We will now introduce the standard separation logic constructs into our new setting.

We begin with the definition of separating conjunction, which our two-map abstraction allows us to express in an identical fashion to the traditional notation:

$$P \wedge^* Q \equiv \lambda s.\ \exists s_0\ s_1.\ s_0 \perp s_1 \wedge s = s_0 \mathbin{++} s_1 \wedge P\ s_0 \wedge Q\ s_1$$

Note that s_0 and s_1 are now pairs of maps, not a heap as in traditional separation logic.

Separation logic also defines a useful concept for dealing with heap updates: separating implication. We say that P separately implies Q on a heap s when for any disjoint extension of s on which P holds, Q holds for s overridden with that extension. We can thus specify that modifying a heap in some way will establish a property. Once more, our version looks identical to the traditional one ($\longrightarrow$ is implication):

$$P \longrightarrow^* Q \equiv \lambda s.\ \forall s'.\ s \perp s' \wedge P\ s' \longrightarrow Q\ (s \mathbin{++} s')$$

In order to make statements about the heap, traditional separation logic also provides several heap assertions: a concrete *maps-to*, as well as a *maps-to-something*:

$$p \mapsto v \equiv \lambda s.\ s\ p = \lfloor v \rfloor \wedge \mathsf{dom}\ s = \{p\}$$

$$p \mapsto - \equiv \lambda s.\ \exists v.\ (p \mapsto v)\ s$$

$$p \hookrightarrow v \equiv p \mapsto v \wedge^* \mathsf{sep\text{-}true}$$

$$p \hookrightarrow - \equiv \lambda s.\ \exists v.\ (p \hookrightarrow v)\ s$$

The first two are domain exact [14], meaning they apply to a specific heap and are false for any extension of it. sep-true and sep-false are assertions defined to be respectively true and false for any heap:

$$\mathsf{sep\text{-}true} \equiv \lambda s.\ \mathsf{True}$$

$$\mathsf{sep\text{-}false} \equiv \lambda s.\ \mathsf{False}$$

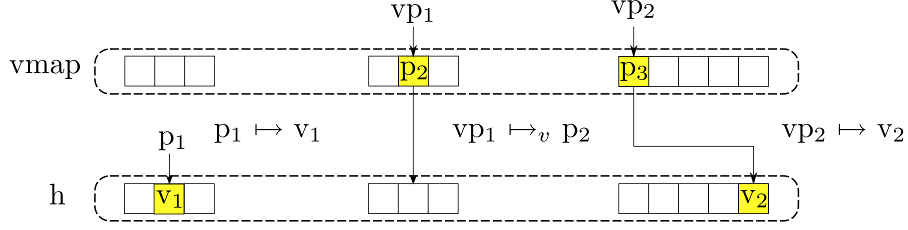

Fig. 2. Maps-to assertions on the heap, virtual map and address space.

The above assertions are standard separation logic assertions expressed on heaps. We will now proceed to describing their equivalents in our logic, expressed on a two-map state consisting of a heap and a virtual map.

Excepting the first domain exact assertion, which is the basis for all the others, we use identical notation to standard separation logic for heap and virtual map assertions, utilising Isabelle's function overloading ability.

In order to maintain the domain exact property of the first *maps-to* assertion, we proceed as follows. Heap assertions (leftmost on Fig. 2) are identical to normal separation logic, with the additional constraint that they only talk about the heap:

$$p \mapsto v \equiv \lambda(h,\ vmap).\ h\ p = \lfloor v \rfloor \wedge \mathsf{dom}\ h = \{p\} \wedge vmap = \mathsf{empty}$$

Virtual map assertions (middle on Fig. 2) work similarly, but only on the virtual map. Unlike the other *maps-to* assertions, virtual addresses do not map to values. Hence, we use $\mapsto_v$ to denote them:

$$vp \mapsto_v p \equiv \lambda(h,\ vmap).\ vmap\ vp = \lfloor p \rfloor \wedge \mathsf{dom}\ vmap = \{vp\} \wedge h = \mathsf{empty}$$

Finally, address space assertions (rightmost on Fig. 2) involve both the heap and virtual map. In order to be domain exact, the *maps-to* assertion uses exactly one member of each, satisfying:

$$vp \mapsto v \equiv \lambda(h,\ vmap).\ \exists\, p.\ vmap\ vp = \lfloor p \rfloor \wedge h\ p = \lfloor v \rfloor \wedge \mathsf{dom}\ vmap = \{vp\} \wedge \mathsf{dom}\ h = \{p\}$$

With our new definitions, separating conjunction works as expected on predicates involving only one of the maps. However, address space predicates require entries from both maps. Thusly, under separating conjunction, if a virtual pointer $vp \mapsto v$ via some physical pointer p, then vp and p can not map to any other values than p and v, respectively:

$$\frac{(vp \mapsto v \wedge^* vp' \mapsto v')\ (h,\ vmap)}{vp \neq vp' \wedge vmap\ vp \neq vmap\ vp'} \qquad \frac{(vp \mapsto v \wedge^* p \mapsto v')\ (h,\ vmap)}{vmap\ vp \neq \lfloor p \rfloor}$$

In other words, mappings of virtual pointers to values will not share physical memory with each other, nor with mappings of physical pointers to values. Heap predicates and virtual map predicates are always disjoint, as they refer to two different maps.

Since virtual pointers can alias (map to the same physical address), we define an additional assertion to denote this case:

$$vp_1 \sim vp_2 \equiv \lambda(h,\ vmap).\ vmap\ vp_1 = vmap\ vp_2$$

We also found that being able to express that a set of pointers is mapped to some values, e.g. $\mathsf{ptable\text{-}area}\ r\ s \mapsto -$ proved very convenient. Again utilising overloading, we define it as:

$\mathsf{pure}\ P \equiv \forall\, s\ s'.\ P\ s = P\ s'$

$[\![(P \wedge^* Q)\ s;\ \mathsf{pure}\ P;\ \mathsf{pure}\ Q]\!] \Longrightarrow P\ s \wedge Q\ s$

$[\![P\ s \wedge Q\ s;\ \mathsf{pure}\ P \vee \mathsf{pure}\ Q]\!] \Longrightarrow (P \wedge^* Q)\ s$

$\mathsf{pure}\ P \Longrightarrow (\lambda s.\ P\ s \wedge Q\ s) \wedge^* R = (\lambda s.\ P\ s \wedge (Q \wedge^* R)\ s)$

$[\![(P \longrightarrow^* Q)\ s;\ \mathsf{pure}\ P]\!] \Longrightarrow P\ s \longrightarrow Q\ s$

$[\![P\ s \longrightarrow Q\ s;\ \mathsf{pure}\ P;\ \mathsf{pure}\ Q]\!] \Longrightarrow (P \longrightarrow^* Q)\ s$

Fig. 3. Pure assertions of our logic

$$
\begin{array}{lcl}
S \mapsto - & \equiv & \lambda s.\ \mathit{fold}\ \mathsf{op} \wedge^* (\lambda x.\ x \mapsto -)\ \Box\ S\ s \\
S \hookrightarrow - & \equiv & \lambda s.\ \forall\, p{\in}S.\ (p \hookrightarrow -)\ s
\end{array}
$$

The first of these simply states that all members of the set S map to some value, iteratively joined by $\wedge^*$.

The final traditional separation logic construct is empty heap assertion heap: $\Box$. Our logic has three, depending on which of the maps we want to be empty:

$$
\begin{array}{lcl}
\Box_p & \equiv & \lambda(h,\ \mathit{vmap}).\ h = \mathsf{empty} \\
\Box_v & \equiv & \lambda(h,\ \mathit{vmap}).\ \mathit{vmap} = \mathsf{empty} \\
\Box & \equiv & \Box_p\ \lfloor\wedge\rfloor\ \Box_v \\
P\ \lfloor\wedge\rfloor\ Q & \equiv & \lambda x.\ P\ x \wedge Q\ x
\end{array}
$$

Where $\lfloor\wedge\rfloor$ is the lifted $\wedge$ operator.

We have now introduced adapted versions of all the separation logic connectives as well as those unique to our logic. This done, we can begin reasoning about its properties.

6 State Updates

In this section, we present various properties of our logic, describe how state updates work and their relationship to standard separation logic.

The basic mechanics of separation logic are intuitive about simple statements, as can be seen in these simple examples: no address may have two values; two different allocated physical addresses may have any value:

$p \mapsto v_1 \wedge^* p \mapsto v_2 = \mathsf{sep\text{-}false}$

$[\![h = [p_1 \mapsto v_1,\ p_2 \mapsto v_2];\ p_1 \neq p_2]\!] \Longrightarrow (p_1 \mapsto v_1 \wedge^* p_2 \mapsto v_2)\ (h,\ \mathsf{empty})$

We can lift these examples to the address space level, where two distinct virtual pointers can only be separated if they map to values via different physical addresses:

$[\![s = ([p_1 \mapsto v_1,\ p_2 \mapsto v_2],\ [vp_1 \mapsto p_1,\ vp_2 \mapsto p_2]);\ vp_1 \neq vp_2;\ p_1 \neq p_2]\!]$
$\Longrightarrow (vp_1 \mapsto v_1 \wedge^* vp_2 \mapsto v_2)\ s$

$[\![s = ([p_1 \mapsto v_1,\ p_2 \mapsto v_2],\ [vp_1 \mapsto p_1,\ vp_2 \mapsto p_2]);\ vp_1 \neq vp_2;\ p_1 = p_2]\!]$
$\Longrightarrow \neg\ (vp_1 \mapsto v_1 \wedge^* vp_2 \mapsto v_2)\ s$

Connectives in our logic conform to the associative, commutative and distributive properties of classical separation logic [14], with identical notation. We have also formalised the pure (Fig. 3) and intuitionistic (Fig. 4) assertions of separation logic (as they apply to our logic) and proved their properties [14], once more with no departure from standard notation.

$\textsf{intuitionistic}\ P \equiv \forall\, s\ s'.\ P\ s \wedge s \subseteq_t s' \longrightarrow P\ s'$

$\textsf{pure}\ P \Longrightarrow \textsf{intuitionistic}\ P$

$[\![\textsf{intuitionistic}\ P;\ \textsf{intuitionistic}\ Q]\!] \Longrightarrow \textsf{intuitionistic}\ (\lambda s.\ P\ s \wedge Q\ s)$

$[\![\textsf{intuitionistic}\ P;\ \textsf{intuitionistic}\ Q]\!] \Longrightarrow \textsf{intuitionistic}\ (\lambda s.\ P\ s \vee Q\ s)$

$(\bigwedge x.\ \textsf{intuitionistic}\ (P\ x)) \Longrightarrow \textsf{intuitionistic}\ (\lambda s.\ \forall\, x.\ P\ x\ s)$

$(\bigwedge x.\ \textsf{intuitionistic}\ (P\ x)) \Longrightarrow \textsf{intuitionistic}\ (\lambda s.\ \exists\, x.\ P\ x\ s)$

$\textsf{intuitionistic}\ (\textsf{sep-true} \wedge^*\ P)$

$\textsf{intuitionistic}\ (\textsf{sep-true} \longrightarrow^*\ P)$

$\textsf{intuitionistic}\ P \Longrightarrow \textsf{intuitionistic}\ (P \wedge^*\ Q)$

$\textsf{intuitionistic}\ Q \Longrightarrow \textsf{intuitionistic}\ (P \longrightarrow^*\ Q)$

$[\![(P \wedge^*\ \textsf{sep-true})\ s;\ \textsf{intuitionistic}\ P]\!] \Longrightarrow P\ s$

$[\![P\ s;\ \textsf{intuitionistic}\ P]\!] \Longrightarrow (\textsf{sep-true} \longrightarrow^*\ P)\ s$

Fig. 4. Intuitionistic assertions of our logic

The properties and simple examples above define a logic about heaps. In order to begin reasoning about programs, we need to consider their role as state transformers and hence examine the mechanics of heap updates and their effects.

In our machine instantiation, updating the heap alone is simply updating the map, though on more realistic instantiations it will become more complex:

$\textsf{heap-update}\ pptr\ val\ h \equiv h(pptr \mapsto val)$

In a virtual memory environment updating the heap can potentially involve changing the page table and thus the virtual map. Hence, when performing a state update at a physical address, we re-lift the page table from the updated heap:

$\textsf{heap-update-p}\ pptr\ val\ root \equiv \lambda(h,\ vmap).\ \textsf{let}\ h' = \textsf{heap-update}\ pptr\ val\ h\ \textsf{in}\ (h',\ \textsf{ptable-lift}\ h'\ root)$

As mentioned in Sect. 2, direct access to the heap is usually limited to system devices. Applications modify the heap through virtual addresses. The extra step involved over a direct heap update is the lookup of the virtual address. The page table is re-lifted as with heap-update-p:

$$\begin{aligned}\textsf{heap-update-v}\ vp\ val\ root\ (h,\ vmap)\ \equiv\ & \textsf{case}\ vmap\ vp\ \textsf{of None} \Rightarrow \textsf{arbitrary}\\ & \mid \lfloor p \rfloor \Rightarrow \textsf{let}\ h' = \textsf{heap-update}\ p\ val\ h\ \textsf{in}\ (h',\ \textsf{ptable-lift}\ h'\ root)\end{aligned}$$

Following Tuch et al. [18], heap updates on the physical layer always succeed. The behaviour of trying to access virtual addresses is more complex. At the hardware level, accessing an unmapped address will cause a *page fault* interrupt, which will redirect control to a page fault handler. The handler's implementation may then allocate pages, map pages, load them from disk, etc. If the fault is resolved successfully, control is returned to the faulting process and the access is attempted again. Hence, there are two cases: either the page is mapped or it is not. In the former, heap-update-v will perform the update just as the hardware would. In the latter, the term heap-update-v will not occur in the verification condition, as transferring control to the page fault handler will only change the necessary registers, not the heap. This means the arbitrary part of heap-update-v will not occur in verification conditions in either case.

At the application level, processes typically run in a simulated memory environ-

ment which abstracts away the inner workings of the memory subsystem. Hence, verification of application code under the assumption of page fault handler correctness can proceed by assuming that all memory implicitly required by the application (e.g. code, stack) is resident and mapped. Page fault handler correctness in this case is a simulation theorem showing that any execution with page faults can be simulated by one in the virtual application setting without faults.

At the operating system level, the virtual memory mechanism is visible. Furthermore, at least some system data structures, particularly in the kernel, need to be permanently resident in memory. To make this property part of the semantics, we can add guards to these critical kernel heap accesses, requiring the address to be mapped. For using the guard technique, see Tuch et al. [18].

Having defined heap updates, we now proceed to reasoning about their properties. Reynolds [14] defines two properties of state mutation, expressible in a shallow embedding (on a *one-heap* state) as:

$$\frac{(p \mapsto - \wedge^* P)\ h}{(p \mapsto v \wedge^* P)\ (\textsf{heap-update}\ p\ v\ h)} \qquad \frac{(p \mapsto - \wedge^* (p \mapsto v \longrightarrow^* P))\ h}{P\ (\textsf{heap-update}\ p\ v\ h)}$$

The former states that a property not dependent on the area of the heap being modified holds after the update. The latter is a weakest-precondition rule useful for backwards reasoning. It states that if P holds for a heap with the entry p set to v, then P will hold after the update, as that is precisely what heap-update does. Henceforth, we will refer to these as the *global* and *weakest-precondition* rules respectively.

As mentioned in Sect. 1, most operating system code falls into the "safe" category of using a one-to-one virtual map. Most application code likewise does not involve memory sharing or physical regions multiply mapped into virtual ones. In the following properties, we will present how our logic reduces down to statements identical to normal separation logic.

In all properties about updates in this section, we assume that r represents a valid root and s a state where the virtual map is the result of lifting the page table:

$\textsf{valid-root}\ r$

$\textsf{ptable-lift}\ (\textsf{fst}\ s)\ r = \textsf{snd}\ s$

Note that for physical addresses $p \mapsto -$ implies p is allocated, while for virtual addresses $vp \mapsto -$ implies that vp is mapped *and* the physical address it is mapped to is allocated.

The easiest to adapt is the physical update version of the global update rule.

Theorem 6.1 *For any physical pointer p that is not part of the page table, the classical global update rule holds:*

$$\frac{p \notin \textsf{ptable-area}\ (\textsf{fst}\ s)\ r \qquad (p \mapsto - \wedge^* P)\ s}{(p \mapsto v \wedge^* P)\ (\textsf{heap-update-p}\ p\ v\ r\ s)}$$

As the page table is not modified, no mappings change, so the the virtual map lifted from the updated heap is identical to the original map. Apart from the requirement on p, our notation looks exactly like that of classical separation logic. We can rephrase the requirement on p in theorem 6.1 as a requirement on the page table area being allocated:

$$\frac{(p \mapsto - \wedge^* \text{ptable-area (fst } s)\ r \mapsto - \wedge^* P)\ s}{(p \mapsto v \wedge^* \text{ptable-area (fst } s)\ r \mapsto - \wedge^* P)\ (\text{heap-update-p } p\ v\ r\ s)}$$

A similar transformation can also be done to the following update theorems, but we omit this form due to space considerations.

The weakest-precondition rule is more interesting, although it is derivable from the global update rule, due to its usefulness in backwards reasoning:

Theorem 6.2 *For any physical pointer p that is not part of the page table, the classical weakest-precondition update rule holds:*

$$\frac{p \notin \text{ptable-area (fst } s)\ r \qquad (p \mapsto - \wedge^* (p \mapsto v \longrightarrow^* P))\ s}{P\ (\text{heap-update-p } p\ v\ r\ s)}$$

Once more the page table is not modified, meaning only one value in physical memory changes. If the old state overridden by the new value satisfies P, then P will hold after the update.

In the interest of brevity, we omit the simpler global update rule for virtual pointers, focusing on the weakest-precondition rule instead. The primary difference from theorem 6.2 is in the constraint of not residing in the page table being applied to the physical address the virtual pointer is mapped to:

Theorem 6.3 *For any virtual pointer vp, mapped to a physical address that is not part of the page table, the classical weakest-precondition update rule holds:*

$$\frac{\text{the (snd } s\ vp) \notin \text{ptable-area (fst } s)\ r \qquad (vp \mapsto - \wedge^* (vp \mapsto v \longrightarrow^* P))\ s}{P\ (\text{heap-update-v } vp\ v\ r\ s)}$$

At the end of the operation, the address space is unchanged except for all the virtual pointers mapped to the physical address at which the heap was changed (via heap-update-v). As we described in Sect. 5 however, vp cannot alias with any other virtual pointer under separation conjunction, thus the modification of the heap will only be visible through one virtual address: vp.

Furthermore, if the page table is not mapped into the address space, as is typical for applications, heap-update-v cannot modify the page table at all:

Theorem 6.4 *The weakest-precondition rule holds for any virtual pointer vp if the page table is not mapped into the address space:*

$$\frac{\text{ptable-not-mapped (fst } s)\ r \qquad (vp \mapsto - \wedge^* (vp \mapsto v \longrightarrow^* P))\ s}{P\ (\text{heap-update-v } vp\ v\ r\ s)}$$

As access to page tables is typically restricted to the operating system, this result demonstrates that for the application domain the update rules are identical to that of classical separation logic under the assumptions of a valid unchanging page table root and a valid state. For applications the page table root usually does not change, and the valid state condition can be checked automatically. Hence, for the application domain our logic reduces to classical separation logic.

Another observation is that applications cannot observe modifications to the heap at locations not mapped into their address space:

Theorem 6.5 *Under the assumption of the page table not being mapped into the current address space, for any physical address p not in the page table area and not mapped into the address space, updating the heap at p is not visible at the virtual memory level:*

$$\frac{p \notin \mathit{ran}\ v \qquad \begin{array}{c} p \notin \textsf{ptable-area}\ h\ r \\ \textsf{ptable-not-mapped}\ h\ r \end{array} \qquad \begin{array}{c} v = \textsf{ptable-lift}\ h\ r \\ (\mathit{new\text{-}h},\ \mathit{new\text{-}vmap}) = \textsf{heap-update-p}\ p\ \mathit{val}\ r\ (h,\ v) \end{array}}{(\mathit{new\text{-}vmap}\ \mathit{oo}\ \mathit{new\text{-}h})\ \mathit{vp} = (v\ \mathit{oo}\ h)\ \mathit{vp}}$$

where oo is the map composition operator:

$$f\ \mathit{oo}\ g \equiv \lambda x.\ \textsf{case}\ f\ x\ \textsf{of None} \Rightarrow \textsf{None} \mid \lfloor y \rfloor \Rightarrow g\ y$$

The vast majority of code, both in applications and operating systems will fall into one of the above categories. However, the applicability of our logic does not end at situations where page tables are not touched. We identify three categories of page table updates, based on the modification of the virtual map: *map* (add entry), *remap* (modify entry) and *unmap* (remove entry). This is a work in progress: we have thus far formalised remap and are at present formalising map and unmap semantics. We will conclude this section with a discussion of remap.

As we are mostly interested in safety invariant preservation, we are concerned about what *does not* change during state updates. Even when updating the page table, most of the virtual map does not change.

Definition 6.6 The set of virtual addresses whose mappings are not affected during an update of the heap:

$$\textsf{ptable-affected}\ f\ h\ \mathit{root} \equiv \textsf{let}\ \mathit{vmap} = \textsf{ptable-lift}\ h\ \mathit{root}\ \textsf{in dom}\ \mathit{vmap} - \mathit{vmap} \cap_m \textsf{ptable-lift}\ (f\ h)\ \mathit{root}$$

$$\text{where } m_1 \cap_m m_2 \equiv \{x \in \textsf{dom}\ m_1.\ m_1\ x = m_2\ x\}$$

During a remap operation, the domain of the virtual map remains the same, but its range might change. Thus, properties not invalidated by the heap modification and the change of virtual map must still hold.

Theorem 6.7 *Updating the heap via some virtual pointer vp preserves a property P if: the domain of the virtual map is not affected, vp's mapping is not affected and P is not invalidated by the new value at vp nor the new contents of the affected virtual map area.*

$$\frac{\begin{array}{c} \mathit{update} = \textsf{heap-update-v}\ \mathit{vp}\ v\ r \qquad \textsf{dom}\ (\textsf{snd}\ (\mathit{update}\ s)) = \textsf{dom}\ (\textsf{ptable-lift}\ (\textsf{fst}\ s)\ r) \\ (\mathit{vp} \mapsto - \wedge^* \textsf{ptable-affected}\ \mathit{update}\ r\ s \rightarrow_v - \wedge^* (\mathit{vp} \mapsto v \wedge^* \textsf{ptable-affected}\ \mathit{update}\ r\ s \rightarrow_v - \longrightarrow^* P))\ s \end{array}}{P\ (\textsf{heap-update-v}\ \mathit{vp}\ v\ r\ s)}$$

7 Related work

The primary focus of this work is enhancement of separation logic, originally conceived by O'Hearn, Reynolds et al. [8,14]. Separation logic has previously been formalised in mechanised theorem proving systems [8,20]. We enhance these abstract models with the ability to reason about properties in a virtual memory environment.

Our exploration of virtual memory semantics is driven by the long-term goal of our research group: a verified operating system microkernel [5] based on L4. Earlier attempts such as UCLA Secure Linux [19], PSOS [11] and KIT [2] lacked the theorem proving technology required to deal with the complexities of a modern microkernel. Like our group, the VeriSoft project [6] is attempting to verify a

microkernel (VAMOS), but their focus is verifying an entire system stack, including compiler and applications. Our focus is on creating an efficient, verified microkernel.

Kernel verification efforts acknowledge the existence of virtual memory; previous work has involved verifying the virtual memory subsystem [9,4,7]. 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 [17]. 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. Hence, we believe our work to be a superset of the Robin approach.

Separation logic has been successfully applied to the verification of context switching code [10,12]. Tuch et al. demonstrated the extension of separation logic to reasoning about C programs involving pointer manipulation [18]. Presently, our work uses a simplified machine model with only one type and does not involve Hoare logic. We believe our framework supports addition of these extensions.

8 Discussion

Although our logic is similar to separation logic and collapses down to separation logic for pure application reasoning, it does not itself constitute a full separation logic. This is due to the fact that writes to the page table are not local actions [3]. In particular, the definitions of virtual-to-physical and virtual-to-value maps-to relations does not include the chunk of memory which contains the virtual mapping itself. As a result, we do not expect the frame rule [14] to hold for state updates involving page table modification.

The issue at the heart of this design decision is the granularity of virtual memory mappings, which causes attempts at pointing to the exact area of memory responsible for mappings to become problematic. For instance, on a 32-bit machine with a page size of 4096 bytes, a single-level page table and a page table entry size of four bytes, addresses 0 and 1 both receive their mappings from the same four-byte entry in the page table. To then say that address 0 maps to a value and *separately* address 1 maps to some value clearly causes a collision on those four bytes. Each entry thus maps 4096 addresses. This is one of the *simplest* setups; lifting this example to a *two-level* page table, the first-level entries map 1024 second-level entries, which in turn map 4096 addresses. Additionally, modern hardware commonly uses variable page sizes (superpages).

Our two-map method of reasoning about virtual memory thus sacrifices separation properties during page table modification in exchange for a simpler model, as well as easy abstraction over different hardware instantiations and multiple page table implementations.

The alternative to our two-map method would be to use a form of sub-byte

addressing, assigning multiple owners to slices of the bytes in each page-table entry. While potentially preserving all properties of separation logic, we believe this approach would make the memory model significantly more complex.

9 Conclusion and Future Work

We have presented an extension of separation logic which allows reasoning about virtual memory and processes running within it. Our logic allows for a convenient representation of predicates on memory at three levels: the virtual map, the physical heap and the virtual address space. The notation abstracts away details to the point of appearing very similar to classical separation logic. Our logic preserves the pure and intuitionistic properties of separation logic, again without exposing the underlying abstraction. Our work has been formalised in the Isabelle/HOL theorem prover.

We have shown that if the page table is not involved in an update or does not map itself, our logic reduces to normal separation logic.

Our work is highly modular. While we chose a simplified machine and page table implementation to aid with fast experimentation, the logic does not depend on the implementation of either. Although our framework does not presently have read/write access rights, it can be easily extended to encompass them. We aim to add this functionality in the near future.

As this is a work in progress, many applications and properties of our logic remain to be explored. The next step is more experimentation in the form of case studies on behaviours of programs in the presence of page table manipulation, possibly refining the model presented here into a complete separation logic as discussed in the previous section. Beyond that, we see the main direction for future work as extending our logic to handle C program verification in the style of Tuch, Klein, and Norrish [18].

Acknowledgements

We thank Gerwin Klein, Michael Norrish, Thomas Sewell and Harvey Tuch for suggestions, discussion and comments on earlier versions of this paper.

References

[1] Ballarin, C., *Locales and locale expressions in isabelle/isar*, in: S. Berardi, M. Coppo and F. Damiani, editors, *TYPES*, Lecture Notes in Computer Science **3085** (2003), pp. 34–50.

[2] Bevier, W. R., *Kit: A study in operating system verification*, IEEE Transactions on Software Engineering **15** (1989), pp. 1382–1396.

[3] Calcagno, C., 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* (2007), pp. 366–378.

[4] Dalinger, I., M. A. Hillebrand and W. J. Paul, *On the verification of memory management mechanisms*, in: D. Borrione and W. J. Paul, editors, *CHARME*, Lecture Notes in Computer Science **3725** (2005), pp. 301–316.

[5] Elphinstone, K., G. Klein, P. Derrin, T. Roscoe and G. Heiser, *Towards a practical, verified kernel*, in: *Proc. 11th Workshop on Hot Topics in Operating Systems*, San Diego, CA, USA, 2007, p. 6, online proceedings at `http://www.usenix.org/events/hotos07/tech/`.

[6] Gargano, M., M. Hillebrand, D. Leinenbach and W. Paul, *On the correctness of operating system kernels*, in: *Proc. 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'05)*, Oxford, UK, 2005, pp. 1–16.

[7] Hillebrand, M., "Address Spaces and Virtual Memory: Specification, Implementation, and Correctness," Ph.D. thesis, Saarland University, Saarbrcken (2005).
URL http://www-wjp.cs.uni-sb.de/publikationen/Hil05.pdf

[8] Ishtiaq, S. S. 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* (2001), pp. 14–26.

[9] Klein, G. and H. Tuch, *Towards verified virtual memory in L4*, in: K. Slind, editor, *TPHOLs Emerging Trends '04*, Park City, Utah, USA, 2004.

[10] Myreen, M. O. and M. J. C. Gordon, *Hoare logic for realistically modelled machine code*, in: O. Grumberg and M. Huth, editors, *TACAS*, Lecture Notes in Computer Science **4424** (2007), pp. 568–582.

[11] Neumann, P. G., R. S. Boyer, R. J. Feiertag, K. N. Levitt and L. Robinson, *A provably secure operating system: The system, its applications, and proofs*, Technical Report CSL-116, SRI International (1980).

[12] Ni, Z., D. Yu and Z. Shao, *Using xcap to certify realistic systems code: Machine context management*, in: K. Schneider and J. Brandt, editors, *TPHOLs*, Lecture Notes in Computer Science **4732** (2007), pp. 189–206.

[13] Nipkow, T., L. Paulson and M. Wenzel, "Isabelle/HOL — A Proof Assistant for Higher-Order Logic," LNCS **2283**, Springer, 2002.

[14] Reynolds, J. C., *Separation logic: A logic for shared mutable data structures*, in: *Proc. 17th IEEE Symposium on Logic in Computer Science*, 2002, pp. 55–74.

[15] Schirmer, N., "Verification of Sequential Imperative Programs in Isabelle/HOL," Ph.D. thesis, Technische Universität München (2006).

[16] Tews, H., *Well-behaved memory on top of virtual memory*, Presentation at the NICTA International Workshop on System Verification, Sydney (2006).

[17] Tews, H., *Formal methods in the robin project: Specification and verification of the nova microhypervisor*, Submitted to the C/C++ Verification Workshop (2007), available from www.cs.ru.nl/∼tews/science.html.

[18] Tuch, H., G. Klein and M. Norrish, *Types, bytes, and separation logic*, in: M. Hofmann and M. Felleisen, editors, *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, Nice, France, 2007, pp. 97–108.

[19] Walker, B., R. Kemmerer and G. Popek, *Specification and verification of the UCLA Unix security kernel*, CACM **23** (1980), pp. 118–131.

[20] Weber, T., *Towards mechanized program verification with separation logic*, in: J. Marcinkowski and A. Tarlecki, editors, *Computer Science Logic – 18th International Workshop, CSL 2004*, Lecture Notes in Computer Science **3210** (2004), pp. 250–264.