# A Unified Memory Model for Pointers

Harvey Tuch and Gerwin Klein

National ICT Australia*, Sydney, Australia
School of Computer Science and Engineering, UNSW, Sydney, Australia
{harvey.tuch|gerwin.klein}@nicta.com.au

**Abstract.** One of the challenges in verifying systems level code is the low-level, untyped view of the machine state that operating systems have. We describe a way to faithfully formalise this view while at the same time providing an easy-to-use, abstract and typed view of memory where possible. We have used this formal memory model to verify parts of the virtual memory subsystem of the L4 high-performance microkernel. All formalisations and proofs have been carried out in the theorem prover Isabelle and the verified code has been integrated into the current implementation of L4.

## 1 Introduction

L4 is a second generation, general purpose microkernel [13] that provides the traditional advantages of microkernels while overcoming the performance limitations of previous generations. With implementations in the order of 10,000 lines of C++/assembler it is an order of magnitude smaller than Mach and two orders of magnitude smaller than Linux. The small size and minimalistic design bring L4 into the reach of formal specification and verification and lead to the unique opportunity of bringing the rigour and trustworthiness of formal verification to the very foundation of practical systems that are in current, industrial use. In this paper, we give an overview of a pilot project testing the feasibility of this idea and present a general solution to the problem of verifying low level pointer modifications in system level code.

During this pilot project we encountered a number of OS code specific verification problems. Among them is the question of how to deal with pointer arithmetic and low level memory modifications, as they are common in system level code. Verifying high-level imperative pointer programs is already considered a hard problem. Recent case studies like Mehta and Nipkow's formalisation of the Schorr-Waite graph marking algorithm [14] show that the complexity of the problem can be reduced to an acceptable level for interactive verification if the right abstractions are used. They exploit the idea that in a type safe language a write to memory position of type $S$ cannot influence another memory position of a different type $T$ (ignoring subtypes for the moment), thus drastically reducing

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

the number of cases that need to be considered for each write. Unfortunately, the implementation language of operating systems (C/C++/assembler) usually is not type safe, and despite a plethora of available safe subsets of C, none of them have caught on in the OS community. This is not entirely for the sake of convenience; there are often good reasons to deliberately break the type safety of the implementation language, among them performance and hardware prescribed data structures. Performance enjoys an especially high priority in the microkernel area: a few cache misses and some hundred processor cycles can make the difference between a practical and impractical system. On the other hand, not all OS code is deliberately type unsafe, in fact the vast majority of it is perfectly fine. The approach presented in this paper enables us to achieve a level of abstraction similar to the one of Mehta and Nipkow for these parts, and at the same time (in the background) to use a very detailed memory model that can faithfully formalise the few occasions of indispensable bit-level operations and pointer arithmetic expressions. The formalisation itself is relatively straightforward (with a twist) — the contribution is that it can conveniently describe both levels of abstraction at the same time and do so with minimal overhead for concrete program verification, exploiting Isabelle/HOL's automatic type inference to avoid reasoning about explicit typing predicates for pointers. As mentioned above the approach is not merely academic, but has been tried out in a larger verification project.

After reviewing related work in section 2 and introducing notation in section 3, we present our formalisation of a typed memory abstraction of untyped memory in section 4 together with a small example. In section 5 we give a rough overview of the verification project in which the technique was used to formally verify parts of the L4 microkernel.

## 2 Related Work

Earlier work on OS verification includes PSOS [16] and UCLA Secure Unix [24]. Later, KIT [2] describes verification of process isolation properties down to object code level, but for an idealised kernel with far simpler and less general abstractions than modern microkernels. A number of case studies [6,5,23] describe the IPC and scheduling subsystems of microkernels in PROMELA and verify them with the SPIN model checker. Manually constructed, these abstractions are not necessarily sound, and so while useful for discovering concurrency bugs, they cannot provide guarantees of correctness. The VeriSoft project [8] is attempting to verify a whole system stack, including hardware, compiler, applications, and a simplified microkernel called VAMOS that is inspired by, but not very close to, L4. While the simplifications are appropriate for the goals of VeriSoft, it is doubtful that the VAMOS kernel will show the necessary performance to be relevant for industrial use.

The idea to use separate heaps for separate pointer types and structure fields goes back to Burstall [4]. On the abstract level, our formalisation is most closely related to Bornat [3] and Mehta and Nipkow's [14] work in Isabelle, although

we exploit Isabelle's type inference in a different way. The Caduceus tool [7] supports Hoare logic verification of C programs, including the type safe part of pointer arithmetic. Like all of the above, we do not use any special purpose logics [19,10], but stay with standard Hoare logic, in our case Schirmer's flexible Hoare logic implementation in Isabelle/HOL [20]. On the concrete level, Norrish [18] presents a very thorough and detailed memory model of C. Our formalisation has similarities to exploratory work on C++ in the VFiasco project [9]. The latter two provide a more precise machine model, while the former allows for more convenient and efficient reasoning. Our model provides both.

Type-safe C variants like CCured [15] also take a dual approach to memory type-safety, by statically detecting safe pointer usage and adding runtime checks for those cases where this cannot be verified. Our approach is oriented towards interactive theorem proving, and does not require any change in language semantics or runtime behavior.

## 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 $\tau$.

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

adjoins a new element *None* to a type $'a$. We use $'a\ option$ to model partial functions in the setting of HOL. For succinctness we write $\lfloor a \rfloor$ instead of *Some* $a$. The underspecified inverse *the* of *Some* satisfies *the* $\lfloor x \rfloor = x$. Function update is written $f(x := y)$ where $f :: 'a \Rightarrow 'b$, $x :: 'a$ and $y :: 'b$. Implication is denoted by $\Longrightarrow$ and $[\![ A_1; \ldots; A_n ]\!] \Longrightarrow A$ abbreviates $A_1 \Longrightarrow (\ldots \Longrightarrow (A_n \Longrightarrow A)\ldots)$. Isabelle theories can be augmented with LaTeX text which may contain references to Isabelle theorems (by name — see chapter 4 of [17]). We use this presentation mechanism to generate the text for most of the definitions and all of the theorems in this paper directly from the Isabelle proofs.

## 4 A Typed Heap on Untyped Memory

There are a number of approaches to describing the state space of a program embedded in a theorem prover. In the simple case, without pointers, one most commonly either treats variable names as first-class HOL values, in which case the state may be a function *name* $\Rightarrow$ *value*, or treats the state as a tuple or record type $var_a\text{-}typ \times var_b\text{-}typ \times var_c\text{-}typ \ldots$ When embedding pointer programs it is convenient to use a model similar to the first for memory addressable by pointers. Introducing pointers however also introduces the problem of *aliasing* [3]. Consider a C program fragment with a `long` pointer `foo` and a

`bool` pointer `bar` where you want to show `*foo = 1` after the program fragment `*foo = 1; *bar = true`. The most basic approach is to model the pointers as values of type *addr* and the heap[1] as a function $addr \Rightarrow value$, where *value* is a datatype with alternatives for long, bool, etc. Ignoring fancy syntax, this formally equates to something like $hp\ foo := Long\ 1$; $hp\ bar := Bool\ True$, where *hp* is a heap function. If we evaluate this in standard Hoare logic, we need the precondition $foo \neq bar$ to show that $hp\ foo = Long\ 1$ in the end.

In a type safe language, on the other hand, this precondition is an unnecessary overhead. We already know implicitly that $foo \neq bar$, because they have different types. In the literature [3,14], this is modelled by a state space in which each language type has its own heap. For the example above, that means we now have functions $bool_h :: addr \Rightarrow bool$ and $long_h :: addr \Rightarrow long$ and we can show the Hoare triple $\{\!|\ True\ |\!\}\ long_h\ foo := 1;\ bool_h\ bar := True\ \{\!|long_h\ foo = 1|\!\}$ automatically and without preconditions. The example is simplified, of course. In a more realistic setting we would still have preconditions and invariants about heap layout and pointers being not `null`. The key point, however, is that there is no need to state pair-wise aliasing conditions on all pointers anymore.

As argued in the introduction, OS code does not normally use a type safe language and does not restrict itself to a type safe subset, so we are forced to use a model of the state space close to that of the underlying hardware memory model if we want to preserve soundness. One such extreme treatment of the heap would be to consider it simply as a function mapping addresses to bits, bytes or words. This has the significant advantage that the hardware abstraction step is very small and the model is amenable to reasoning about the type unsafe operations sometimes present in low-level systems code. On the other hand, this complicates the aliasing problem even further, because the pointers could also alias by pointing into the middle of an encoding. Hohmuth et al [9] provide a semantics for C++ types in such a setting.

What we want to achieve is a low-level heap view when necessary, and the more convenient abstraction of multiple typed heaps when possible. Following the reasoning that different types mean unaliased pointers in the type-safe fragment, we require several things. First we need to know which memory locations should correspond to which types. We then need to know that the memory layout provides a disjoint layout of values. Finally, we need a means of using this information to transform the untyped heap into multiple typed heaps, one for each language type, together with rules to reason about updates of the heap when they conform to the state's type information. All this should be provided in such a way that the complexity of encoding, decoding, typing etc, stays under the hood at least for the common case of safe operations. The rest of this section presents our formalisation of such a model together with a mechanisation that aids in proofs about pointer programs when they can be shown to be type-safe, while still allowing us to break type safety when necessary.

[1] In this section we refer to a heap model since this is where valid pointers are restricted to in the rest of our work, but the setting should be generalisable to a model for the entire memory, including local variables.

### 4.1 The Model

We begin with Hohmuth et al's [9] treatment of C++ types and extend it to work with a heap abstraction that allows for effective reasoning about both typed and untyped views of the heap and the effects of updates on the heap. The emphasis is on mechanising the proof process, for example taking advantage of the rewriting support in Isabelle and existing record update rules provided for Isabelle record types, to reduce the proof burden on the program verifier. In this discussion we avoid talking about a specific language embedding with the goal of generality, however our main application is clearly C or a C-like language.

The following type synonyms describe the heap state:

$$\begin{aligned} &\mathit{addr} = \mathit{word32} \\ &\mathit{heap\text{-}mem} = \mathit{addr} \Rightarrow \mathit{word32} \\ &\mathit{heap\text{-}typ\text{-}desc} = \mathit{addr} \Rightarrow \mathit{typ\text{-}tag\ option} \\ &\mathit{heap\text{-}state} = \mathit{heap\text{-}mem} \times \mathit{heap\text{-}typ\text{-}desc} \end{aligned}$$

where *word32* is a type representing 32-bit words[2] imported from the HOL4 system and *typ-tag* is a type with a value for each language type[3] used by the program. For example, a program operating only on boolean, integer and pointer types would have:

$$\textbf{datatype}\ \mathit{typ\text{-}tag} = \mathit{BoolTag} \mid \mathit{IntTag} \mid \mathit{PtrTag}\ \mathit{typ\text{-}tag}$$

Each language type has both a distinct Isabelle type and a distinct *typ-tag* value, which we refer to as its type *tag* below. A good reason for doing things this way is that in a shallow embedding we can avoid reasoning about possibly ill-typed expressions, e.g. `3.14 / &foo`. Instead, Isabelle's type checker and type inference can prevent us from even writing down such invalid expressions. The tag value allows for explicit referencing of language types in Isabelle terms, and is bound to its corresponding Isabelle type later, as described below. We require this to maintain an explicit record of the type of each value stored in memory. Every language type has a fixed size, given by the function $\mathit{typ\text{-}size} :: \mathit{typ\text{-}tag} \Rightarrow \mathit{nat}$.

The *heap-typ-desc* component of the heap state is a partial function that describes the memory layout. We call it the *heap type description*. A tag $t$ at address $a$ indicates that $a$ is the base of the memory footprint for a value of type $t$. Type safe programs respect the program's memory layout in both read and write operations. The heap type description is purely a proof convenience, a history variable, and while it may be affected by, does not itself affect the semantics of successful heap operations and should not be confused with hardware support for tagged memory. The heap type description may be updated anywhere in the program, for example by calls to *malloc* or *free*, or if we were to model local and global variables in the program's initial conditions on function call and return.

[2] In this work we use 32-bit addresses and words but this could be generalised to n-bit address spaces or finer/coarser granularity of addressing fairly easily.

[3] We distinguish between *language* types of the programming language and *Isabelle* types.

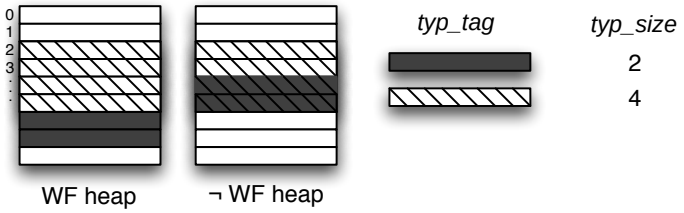


**Fig. 1.** Well-formed heap layout

The predicate *wf-heap* :: *heap-typ-desc* ⇒ *bool* states the well-formedness invariant on the heap type description that is required to establish non-interference of heap updates:

$$\begin{array}{l} \mathit{wf\text{-}heap}\ d \equiv \\ \forall\, x\ y\ t.\ d\ x = \lfloor t \rfloor \wedge 0 < y \wedge y < \mathit{typ\text{-}size}\ t \longrightarrow d\ (n2w\ (w2n\ x + y)) = \mathit{None} \end{array}$$

The type conversions $w2n$ and $n2w$ transform from *word*32 to *nat* and vice versa. Examples of well-formed and not well-formed heap layouts are illustrated in Figure 1.

Each Isabelle type associated with a language type must be an instance of the type class $'a$::*c-type*. The class declares three new constants:

$$\begin{array}{l} \mathit{to\text{-}word} :: {'a}{::}\mathit{c\text{-}type} \Rightarrow \mathit{word32}\ \mathit{list} \\ \mathit{from\text{-}word} :: \mathit{word32}\ \mathit{list} \Rightarrow {'a}{::}\mathit{c\text{-}type}\ \mathit{option} \\ \mathit{typ\text{-}tag} :: {'a}{::}\mathit{c\text{-}type}\ \mathit{itself} \Rightarrow \mathit{typ\text{-}tag} \end{array}$$

The functions *to-word* and *from-word* convert between Isabelle values and lists of words suitable for writing to or reading from the raw heap state. The function *typ-tag* associates a type tag with each $'a$::*c-type*. The type $'a$ *itself* consists of a single element denoted by $\mathit{TYPE}('a)$. This sounds unusual at first, but is easily achieved and part of the Isabelle standard library: since HOL types are non-empty, we can create a polymorphic $'a$ *itself* by taking any element of the type parameter $'a$ as the single occupant of $'a$ *itself*. For each fixed $'a$, any constant of type $'a$ *itself*, say *TYPE*, then refers to this one element. The term $\mathit{TYPE}(\mathit{bool})$ is merely another way of writing the type restriction $\mathit{TYPE}{::}\mathit{bool}$.

The size of the memory footprint of values of a $'a$::*c-type* is given by *size-of* :: $'a$::*c-type itself* ⇒ *nat*:

$$\mathit{size\text{-}of}\ t \equiv \mathit{typ\text{-}size}\ (\mathit{typ\text{-}tag}\ t)$$

The following conditions, captured in the axiomatic type class $'a$::*mem-type*, must hold for any $'a$::*c-type* we want to use in our heap abstraction below.

$$\begin{array}{l} \mathit{from\text{-}word}\ (\mathit{to\text{-}word}\ (x{::}'a)) = \lfloor x \rfloor \\ |\mathit{to\text{-}word}\ (x{::}'a)| = \mathit{size\text{-}of}\ \mathit{TYPE}('a) \end{array}$$

Finally, we introduce a distinct Isabelle pointer type for each Isabelle type.

$$\textbf{datatype } 'a\ ptr = PtrVal\ addr$$

The additional $'a$ on the left-hand side can now be used to associate the pointer type information with pointer values in Isabelle's type system. The destructor *ptr-val* retrieves the address from a pointer value. The pointer types themselves can again be shown to be instances of $'a$::*mem-type*.

We now come to the typed heap abstraction as used by embedded programs. The function *lift* :: *heap-mem* ⇒ $'a$::*c-type ptr* ⇒ $'a$ turns the raw heap into multiple typed heaps — one for each $'a$.

$$\begin{aligned}
&heap\text{-}list :: heap\text{-}mem \Rightarrow addr \Rightarrow nat \Rightarrow word32\ list \\
&heap\text{-}list\ h\ p\ 0 = [] \\
&heap\text{-}list\ h\ p\ (Suc\ n) = h\ p\ \#\ heap\text{-}list\ h\ (p + 1)\ n \\
&h\text{-}val :: heap\text{-}mem \Rightarrow 'a::c\text{-}type\ ptr \Rightarrow 'a\ option \\
&h\text{-}val\ h\ p \equiv from\text{-}word\ (heap\text{-}list\ h\ (ptr\text{-}val\ p)\ (size\text{-}of\ TYPE('a))) \\
&lift :: heap\text{-}mem \Rightarrow 'a::c\text{-}type\ ptr \Rightarrow 'a \\
&lift\ h \equiv \lambda p.\ the\ (h\text{-}val\ h\ p)
\end{aligned}$$

As stated earlier, program expressions operate on the raw heap, ignoring the type tags. This on its own may not be sufficient to faithfully express a language's semantics; *lift h p* will give the value at $p$ where the semantics say a heap access is valid, but we need to establish this validity first. This will usually require a guard or precondition on the statement containing the expression. For example, in C, we would need to know that the memory location had a value of the type in consideration written to it at some earlier point, and that the pointer is correctly aligned. This of course is not limited to this model and also applies to even a simple standard model with multiple typed heaps.

Heap updates are performed with *heap-update* :: *heap-mem* ⇒ $'a$::*c-type ptr* ⇒ $'a$ ⇒ *heap-mem*:

$$\begin{aligned}
&heap\text{-}update\text{-}list\ h\ p\ [] = h \\
&heap\text{-}update\text{-}list\ h\ p\ (x\ \#\ xs) = heap\text{-}update\text{-}list\ (h(p := x))\ (p + 1)\ xs \\
&heap\text{-}update\ h\ p\ v \equiv heap\text{-}update\text{-}list\ h\ (ptr\text{-}val\ p)\ (to\text{-}word\ v)
\end{aligned}$$

Again, heap update statements should be suitably guarded.

We exploit the polymorphism in Isabelle's type system here to avoid explicit definitions of heap abstraction functions for each language type. This may seem a slight gain, but it also enables the simplification rules presented below to be stated once for all types, instead of being reproved for each pair of types.

### 4.2 The Typed View

So far we are able to dereference pointers in our embedded programs, and may be able to do first simple proofs. However, proofs will still have to consider unnecessary aliasing concerns on lifted heaps if we do not know which pointers

respect the heap type description. For example, pointers of different types may still be aliasing the same location, or pointers of the same type may have overlapping memory footprints. The standard way of ruling out these possibilities is an invariant, or ad hoc history variables indicating what the valid pointers of different types are.

Even if we know which pointers are valid, the effect of updates on the lifted heap can only be expressed point-wise: we can determine that pointer $p$ is not affected by an update of pointer $q$ if both are valid. We cannot determine that if the *bool* incarnation of the lifted heap changes, the whole *long* incarnation, as a function, is unaffected.

This means, that if we had, for instance, a heap invariant or abstraction function for a linked list structure that only uses the *long* incarnation of the lifted heap, we would need to prove a separate rule for that abstraction function to show that it remains unchanged under *bool* updates — even if the abstraction function explicitly states that all its pointers are valid.

Utilising the heap type description information we can provide a typed heap abstraction for use in proofs that only depends on the values of the heap at locations valid for the type. We can then prove simplification rules for reasoning about updates to typed heaps once for all language types.

First we introduce the notion of pointer validity. A pointer $p$ of type $'a\ ptr$ is said to be valid under a heap type description $d$ according to the following definition:

$$d \vdash_t p \equiv d\ (\textit{ptr-val}\ p) = \lfloor \textit{typ-tag}\ \textit{TYPE}('a) \rfloor$$

It should be noted that this predicate does not explicitly mention the language type, instead the type is determined automatically by Isabelle's type inference. We found this greatly enhancing the clarity of specifications and proofs.

The heap abstraction function $lift_c$ hides updates to heap locations not corresponding to the valid pointers for a particular typed heap:

$$\begin{aligned} &\textit{lift}_c :: \textit{heap-state} \Rightarrow 'a{::}\textit{c-type}\ \textit{ptr} \Rightarrow 'a\ \textit{option} \\ &\textit{lift}_c\ (h,\ d) \equiv \lambda p.\ \textit{if}\ d \vdash_t p\ \textit{then}\ \textit{h-val}\ h\ p\ \textit{else}\ \textit{None} \end{aligned}$$

Like *lift* $h$, $lift_c\ h$ is polymorphic and returns a heap abstraction of type $'a\ typ\text{-}heap = 'a\ ptr \Rightarrow 'a\ option$. The program text itself can continue to use the functions *lift* and *heap-update*, while pre/post conditions and invariants use the stronger $lift_c$ to make more precise statements. The following conditional rewrite connects the two levels.

$$\textit{lift}_c\ (h,\ d)\ p = \lfloor x \rfloor \Longrightarrow \textit{lift}\ h\ p = x$$

We have proved two further significant rewrite rules that support reasoning about the effects of heap updates on $lift_c$. The first rule states how an $'a\ ptr$ update affects an $'a\ typ\text{-}heap$, the second rule shows that an $'a\ ptr$ update does not affect a $'b\ typ\text{-}heap$ if $'a$ is different from $'b$.

$[\![\mathit{wf\text{-}heap}\ d;\ d \vdash_t p]\!] \Longrightarrow \mathit{lift}_c\ (\mathit{heap\text{-}update}\ h\ p\ v,\ d) = \mathit{lift}_c\ (h,\ d)(p \mapsto v)$

$[\![\mathit{wf\text{-}heap}\ d;\ d \vdash_t p;\ \mathit{typ\text{-}tag}\ \mathit{TYPE}('a) \neq \mathit{typ\text{-}tag}\ \mathit{TYPE}('b)]\!]$
$\Longrightarrow \mathit{lift}_c\ (\mathit{heap\text{-}update}\ h\ p\ v,\ d) = \mathit{lift}_c\ (h,\ d)$

These are added to a simplification set, with other heap related lemmas, in our work, and do not require manual application. Isabelle's simplifier can resolve the $\mathit{typ\text{-}tag}\ \mathit{TYPE}('a) \neq \mathit{typ\text{-}tag}\ \mathit{TYPE}('b)$ condition automatically, as long as the type tag definitions for language types are also in the default simplification set. The heap type description changes relatively infrequently and therefore the proof overhead in showing $\mathit{wf\text{-}heap}\ d$ is low.

For any program that respects the heap type description, we can thus automatically simplify away the fact that the heap is shared and pretend to work on multiple typed heaps. At the same time, we can still capture the semantics of type unsafe operations. In this case things are no longer automatic, and we are required to provide rules for how the lifted heaps changed during the operation. This is a small price to pay for the flexibility and convenience gained on the abstract level. It may even be possible to derive a set of rules that capture common type unsafe operations, for example physical subtyping, although we have not done so in our application.

We can also use type tags to write Isabelle expressions over language types. For example, one might want to express in a specification that only heaps of certain types can change during execution. For this we define the predicate $\mathit{h\text{-}id\text{-}except} :: \mathit{typ\text{-}tag\ set} \Rightarrow \mathit{heap\text{-}state} \Rightarrow \mathit{heap\text{-}state} \Rightarrow \mathit{bool}$ which satisfies the following lemma for lifted heaps of type $'a\ \mathit{typ\text{-}heap}$:

$$[\![\mathit{h\text{-}id\text{-}except}\ ts\ s\ s';\ \mathit{typ\text{-}tag}\ \mathit{TYPE}('a) \notin ts]\!] \Longrightarrow \mathit{lift}_c\ s = \mathit{lift}_c\ s'$$

### 4.3 Example

Picking up the example from the introduction to this section, we show below how it is expressed in our setting.

The state space is now a global program heap of type *heap-state* with two pointers $\mathit{foo} :: \mathit{long\ ptr}$ and $\mathit{bar} :: \mathit{bool\ ptr}$. It is easy to show that *long* and *bool* are instances of the type class *mem-type* by defining the constants *to-word*, *from-word* and *typ-tag* and proving that they satisfy the axioms stated in section 4.1. Using Isabelle's syntax mechanisms to abbreviate $\mathit{lift}_c\ (h,d)\ p = \lfloor v \rfloor$ to $*p = v$ and $h := \mathit{heap\text{-}update}\ h\ p\ v$ to $*p := v$, the Hoare triple we can then state and prove automatically, is the following:

$$\{\!|\mathit{wf\text{-}heap}\ d \wedge d \vdash_t \mathit{foo} \wedge d \vdash_t \mathit{bar}|\!\}\ *\mathit{foo} := 1;\ *\mathit{bar} := \mathit{True}\ \{\!|*\mathit{foo} = 1|\!\}$$

The three preconditions in this statement present only a very small overhead. As long as the program stays in a safe fragment of the language, e.g., when pointers are used like Java references without pointer arithmetic, there is never need to unfold their definition. They can also easily be propagated by the verification condition generator. In contrast to explicit statements of pointer aliasing, they

also only talk about one pointer at a time, not pairwise distinctness or, as it would be applicable in this more detailed setting, distinctness of encoding regions.

While structures can be treated like any other language type in this setting, the formalisation presented here does not yet provide a separate heap for each field of each structure as Bornat does. It is possible to achieve this by another, analogous lifting step on top of $lift_c$ that takes field names into account. We have recently formalised this in Isabelle, but have not used it in our case study yet.

## 5 The L4 Virtual Memory Manager

In this section we describe the case study which motivated the development of this memory model: the virtual memory (VM) management system, one of the three main abstractions L4 provides. Our approach is a classic and pragmatic refinement methodology. We start out from an abstract model of the kernel that we then formally refine towards an implementation. The last step consists of generating C code that implements the same functionality as the original OS code. We based our formalisation on the L4 X.2 API [12] and used the L4Ka::Pistachio [21] implementation on the ARM architecture to resolve ambiguities in addition to discussions with the developers on the *pistachio-core* mailing list.

### 5.1 The abstract model

The VM subsystem of L4 provides a flexible, hierarchical way of manipulating the mapping from virtual to physical memory pages at user-level. Below we sketch our formalisation and show the definition of *unmap*, one of the VM operations.

This model is still a simplification of the current L4 API because the API stipulates two regions per address space (the kernel interface page and user thread control blocks) that we have not modelled, and because the mapping operations in L4 can work on regions of the address space rather than individual pages.

Fig. 2 illustrates the concept of hierarchical mappings. The example maps virtual page $v_1$ in space $n_1$, as well as $v_2$ in $n_2$, and $v_4$ in $n_4$ to the physical page $r_1$. Formally, we use the types $R$ for the physical pages, $V$ for virtual pages, and $N$ for names of address spaces.

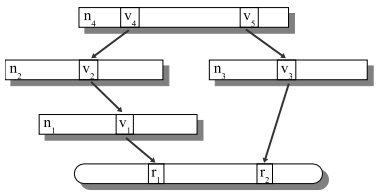


**Fig. 2.** Address Spaces

Mappings $M$, i.e. positions in this picture, are uniquely determined either by a page $V$ in a virtual address space $N$, or by a physical page. An address space is a function from pages $V$ to mappings together with a set of access rights, determining how the mapped page may be used from inside the address space. On a concrete architecture, these will be rights like read/write/execute — here we use the abstract type $AR$. The machine state is then a partial map from address space names to address spaces.

$$\begin{array}{ll} \textbf{datatype} & M = \mathit{Virtual}\ N\ V \mid \mathit{Real}\ R \\ \textbf{types} & \mathit{space} = V \Rightarrow M \times \mathit{AR\ set} \\ & \mathit{state} = N \Rightarrow \mathit{space\ option} \end{array}$$

The concept of *paths* relates these functions to the arrows in Fig. 2: $s \vdash x \rightsquigarrow^1 y$ means that in state $s$ there is a direct path from $x$ to $y$. For this to be true, $x$ must be of the form $\mathit{Virtual}\ n\ v$, the address space $n$ must be defined in state $s$ and it must map the virtual page $v$ to $y$:

$$s \vdash x \rightsquigarrow^1 y = (\exists\, n\ v\ ar\ \sigma.\ x = \mathit{Virtual}\ n\ v \wedge s\ n = \lfloor \sigma \rfloor \wedge \sigma\ v = (y,\ ar))$$

We write $\_ \vdash \_ \rightsquigarrow^* \_$ for the reflexive, transitive closure of the direct path relation.

The operation $\mathit{unmap}\ n\ v\ ar$ reduces the access rights of all pages leading to $\mathit{Virtual}\ n\ v$ by $ar$ (a set of access rights). If $ar$ happens to be $\mathcal{U}$ (the universal set) the operation makes these pages inaccessible. In the definition we use a function *clear* that, given a name $n$, a page $v$, a set of access rights $ar$ and an address space $\sigma$ in state $s$, returns $\sigma$ where the access rights of all $v'$ leading to $\mathit{Virtual}\ n\ v$ have been reduced by $ar$.

$\mathit{unmap} :: N \Rightarrow V \Rightarrow \mathit{AR\ set} \Rightarrow \mathit{state} \Rightarrow \mathit{state}$
$\mathit{unmap}\ n\ v\ ar\ s \equiv \lambda n'.\ \mathsf{case}\ s\ n'\ \mathsf{of}\ \mathit{None} \Rightarrow \mathit{None} \mid \lfloor \sigma \rfloor \Rightarrow \lfloor \mathit{clear}\ n\ v\ ar\ s\ \sigma \rfloor$

$\mathit{clear}\ n\ v\ ar\ s\ \sigma \equiv$
$\lambda v'.\ \mathsf{let}\ (m,\ ar') = \sigma\ v'\ \mathsf{in\ if}\ s \vdash m \rightsquigarrow^*\ \mathit{Virtual}\ n\ v\ \mathsf{then}\ (m,\ ar' - ar)\ \mathsf{else}\ (m,\ ar')$

The other operations of the VM subsystem (flush, map, grant, create, and memory lookup) are modelled in a similar way, modifying paths and access rights accordingly. See our earlier work [11,22] for details on the same formalisation which we have extended with access rights here.

We have shown a number of properties about the reachable states of the VM system, among them that access rights can never increase when a page is mapped to another address space, that there are no loops in the path structure, and that address lookup is a total function. The latter is quite literally an important safety property. Overheating and physical damage may result if two conflicting TLB entries are present for the same virtual address [1, p. B3-26].

### 5.2 The concrete model

The concrete implementation of the address spaces abstraction in L4 is based on two data structures, the *page tables* and the *mapping database* (MDB), as well as

the algorithms for their traversal and manipulation. This is because performance, and in the case of ARM, the hardware, dictates that an efficient translation from virtual to physical addresses and corresponding access rights be available. The page tables are used to achieve the translation, while the MDB keeps track of the mapping relation for the purpose of revocations like the unmap operation shown in the previous section.

The state space consists of the *heap-state* together with some local and static global variables. The page table and MDB data structure abstractions are inductively defined on this raw state, i.e. they are a function and relation that take a *'a typ-heap* as a parameter and when used in pre/post-conditions and invariants this takes on a value of $lift_c$ in the current state. A complete description of the semantics of the concrete operations, specifications and proofs is well beyond the scope of this paper, with the raw theory files alone consisting of over 5000 lines of Isabelle/HOL definitions and proofs. Instead we present the MDB definitions and one example operation here to provide a flavour of the level of detail in the model.

The MDB is a doubly linked list representing the pre-order traversal of a mapping tree, which is essentially the tree described in the abstract model with the arrow directions reversed. Nodes in the MDB are of type *map-node*:

$$\begin{aligned}\textbf{record}\ \textit{map-node} = \ & \textit{map-next} :: \textit{word32} \\ & \textit{map-prev} :: \textit{word32} \\ & \textit{map-pte} :: \textit{pte ptr} \\ & \textit{map-depth} :: \textit{word32}\end{aligned}$$

We use the integer representation of pointer addresses for next and previous pointers as the type *map-node ptr* is not available until after this declaration. The *map-pte* field stores a pointer to the corresponding page table entry and *map-depth* contains the depth of the node in the tree for the pre-order tree representation.

A subtree relation is defined on a typed heap $s :: \textit{map-node typ-heap}$. The term $s \vdash a \rightsquigarrow^T b$ states that $b$ is in $a$'s subtree. This is defined as:

$$s \vdash x \mapsto y = (\exists m.\ s\ x = \lfloor m \rfloor \wedge \textit{get-next}'\ m = \lfloor y \rfloor)$$

$$[\![ s \vdash m \mapsto m';\ \textit{get-depth}'\ (s\ m) < \textit{get-depth}'\ (s\ m') ]\!] \Longrightarrow s \vdash m \rightsquigarrow^T m'$$

$$[\![ s \vdash m \rightsquigarrow^T m';\ s \vdash m' \mapsto ma;\ \textit{get-depth}'\ (s\ m) < \textit{get-depth}'\ (s\ ma) ]\!] \Longrightarrow s \vdash m \rightsquigarrow^T ma$$

where *get-next'* $m$ and *get-depth'* $m$ act as expected, returning the next pointer and depth respectively for a given *map-node*.

It should be clear that proofs about the MDB are localised to just the *map-node typ-heap* and procedures operating on just the MDB can be easily shown not to affect other typed heaps using the previous section's lemmas. *wf-heap* is part of the global invariant in our model.

An example of a procedure in our concrete model's MDB is *map-unlink* used during unmapping to remove a node from the linked list:

**procedures** *map-unlink*(*m*,*mq*) =
$nm := cast\ TYPE(map\text{-}node\ ptr)\ (lift\ h\ mq{\cdot}map\text{-}next)$;
$h := heap\text{-}struct\text{-}update\ h\ m\ map\text{-}next\text{-}update\ (cast\ TYPE(word32)\ nm)$;
$IF\ nm \neq Null\ THEN$
$h := heap\text{-}struct\text{-}update\ h\ nm\ map\text{-}prev\text{-}update\ (cast\ TYPE(word32)\ m)$
$FI$;
$CALL\ map\text{-}free(mq)$

where

$$heap\text{-}struct\text{-}update\ h\ p\ f\ v \equiv heap\text{-}update\ h\ p\ (f\ v\ (lift\ h\ p))$$

An abstraction relation that relates the state spaces of the abstract and concrete models is defined based on the page table and MDB abstractions. We have done most of the simulation proofs to show refinement between these levels in other work, but we still have a small gap to fill prior to having the proofs integrated and completed for the models described here.

### 5.3 Generating high performance C code

On a semantic level our concrete model is intended to be faithful to the semantics of C as understood by our compiler, but we require a simple translation step to turn these Isabelle/HOL definitions into C source code suitable for compilation. This involves traversing the abstract syntax tree of the Isabelle/HOL terms for procedures, generating real C syntax and is fairly straightforward. The number of lines of ML code required to do this is less than 400; hence we have reasonable confidence in this step, which might otherwise be seen as a source of soundness concerns. The generated C source for the *map-unlink* example is:

```
extern "C" inline void
map_unlink(struct map_node* m, struct map_node* mq) {
  struct map_node* nm;
  nm = (map_node *)((*(mq)).map_next);
  (*(m)).map_next = (word32)(nm);
  if ((nm) != (NULL)) {
    (*(nm)).map_prev = (word32)(m);
  }
  map_free(mq);
}
```

The generated code is suitable for passing to *gcc* and with the addition of stub code in the existing L4Ka::Pistachio kernel has been linked to the kernel and can replace the modelled part of the VM subsystem.

The investment for the virtual memory part of this verification pilot project was about 1.5 person years. All specifications and proofs together run to about 14,000 lines. This is significantly more than the effort invested in the VM subsystem in the first place, but it includes exploration of alternatives, determining the right methodology, etc. Our final goal is a verified, high performance implementation of L4, and the results so far have been encouraging.

## 6 Conclusion

We have presented a novel way of modelling memory for imperative pointer programs that allows us to reason abstractly and conveniently about those parts of the program that are type safe and at the same time correctly and precisely about those parts that are not. Both kinds of reasoning can be freely intermixed, using a standard Hoare logic framework.

On the abstract level we can directly express language types inside Isabelle's type system and can therefore enjoy the advantages of type inference as well as avoid explicit type information in specifications and invariants. While we think our model is complete on the low level, we only have shown basic types, pointers and structs on the abstract level. We expect more language features like tagged unions to be expressable.

The model introduces a slight overhead for reasoning about type unsafe operations. Our experience so far in applying the technique to OS kernel verification suggests that this is the right trade-off to make — especially since a model with only low-level reasoning usually very quickly introduces invariants similar to our well-formedness condition.

We showed some important aspects of verifying the VM subsystem of the L4 microkernel in which this technique was applied. We have sketched our abstract model of address spaces with access right restrictions and shown some aspects of refining these operations down to directly executable, high performance C code.

A nice side effect of our memory model is that reasoning about the `malloc` and `free` library functions in C becomes possible. In an abstract setting, their specification is easy: pointers become valid or invalid. Proving their implementation correct, however, is impossible, because they necessarily break the abstraction barrier. In our setting, the specification remains simple, but we are now able to prove in the same framework that their often considerably complex implementation satisfies this specification. We have not done so yet, but are looking forward to taking this on as future work.

*Acknowledgements* We thank Rafal Kolanski, Nicolas Magaude, Michael Norrish, Norbert Schirmer, and Simon Winwood for discussing drafts of this paper.

## References

1. ARM Limited. *ARM Architecture Reference Manual*, June 2000.
2. W. R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.
3. R. Bornat. Proving pointer programs in Hoare Logic. In R. Backhouse and J. Oliveira, editors, *Mathematics of Program Construction (MPC 2000)*, volume 1837 of *LNCS*, pages 102–126. Springer, 2000.
4. R. Burstall. Some techniques for proving correctness of programs which alter data structures. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 23–50. Edinburgh University Press, 1972.

5. T. Cattel. Modelization and verification of a multiprocessor realtime OS kernel. In *Proceedings of FORTE '94, Bern, Switzerland*, October 1994.
6. G. Duval and J. Julliand. Modelling and verification of the RUBIS $\mu$-kernel with SPIN. In *SPIN95 Workshop Proceedings*, 1995.
7. J.-C. Filliâtre and C. Marché. Multi-prover verification of C programs. In *Formal Methods and Software Engineering, 6th International Conference on Formal Engineering Methods, ICFEM 2004, Seattle, USA*, volume 3308 of *LNCS*, pages 15–29. Springer, 2004.
8. M. Gargano, 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 2005)*, Oxford, UK, 2005. to appear.
9. M. Hohmuth, H. Tews, and S. G. Stephens. Applying source-code verification to a microkernel — the VFiasco project. Technical Report TUD-FI02-03-März, TU Dresden, 2002.
10. J. Jensen, M. Joergensen, N. Klarlund, and M. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *PLDI '97*, 1997.
11. G. Klein and H. Tuch. Towards verified virtual memory in L4. In K. Slind, editor, *TPHOLs Emerging Trends '04*, Park City, Utah, USA, 2004.
12. *L4 eXperimental Kernel Reference Manual Version X.2*, 2004. `http://l4hq.org/docs/manuals/`.
13. J. Liedtke. On $\mu$-kernel construction. In *15th ACM Symposium on Operating System Principles (SOSP)*, December 1995.
14. F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 2005. To appear.
15. G. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Trans. Prog. Lang. Syst.*, 27(3):477–526, 2005.
16. P. G. Neumann, 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.
17. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
18. M. Norrish. *C formalised in HOL*. PhD thesis, Computer Laboratory, University of Cambridge, 1998.
19. 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.
20. N. Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In F. Baader and A. Voronkov, editors, *Logic for Programming, AI, and Reasoning*, volume 3452 of *LNAI*, pages 398–414. Springer, 2005.
21. System Architecture Group. The L4Ka::Pistachio microkernel. White paper, University of Karlsruhe, May 2003.
22. H. Tuch and G. Klein. Verifying the L4 virtual memory subsystem. In *Proc. NICTA FM Workshop on OS Verification*, pages 73–97. Technical Report 0401005T-1, National ICT Australia, 2004.
23. P. Tullmann, J. Turner, J. McCorquodale, J. Lepreau, A. Chitturi, and G. Back. Formal methods: a practical tool for OS implementors. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 20–25, 1997.
24. B. Walker, R. Kemmerer, and G. Popek. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM*, 23(2):118–131, 1980.