Types, Bytes, and Separation Logic

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

Michael Norrish
Canberra Research Lab., National ICT Australia, Australia
michael.norrish@nicta.com.au

Abstract

We present a formal model of memory that both captures the low-level features of C's pointers and memory, and that forms the basis for an expressive implementation of separation logic. At the low level, we do not commit common oversimplifications, but correctly deal with C's model of programming language values and the heap. At the level of separation logic, we are still able to reason abstractly and efficiently. We implement this framework in the theorem prover Isabelle/HOL and demonstrate it on two case studies. We show that the divide between detailed and abstract does not impose undue verification overhead, and that simple programs remain easy to verify. We also show that the framework is applicable to real, security- and safety-critical code by formally verifying the memory allocator of the L4 microkernel.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification

General Terms Languages, Theory, Verification

Keywords Separation Logic, C, Interactive Theorem Proving

1. Introduction

Nearly All Binary Searches and Mergesorts are Broken. This was the title of a short, widely read article by Joshua Bloch on Google's research blog [6]. Bloch had discovered a problem in his implementation of binary search in an array in the standard Java library. The problem had remained unnoticed for nine years of widespread deployment in the Java platform, and Bloch argues that it is in fact about 50 years old as it can be found in most standard text-books. What is remarkable about this error is that many of the text-book and lecture implementations have been formally proven correct.

The problem is that the usual text-book view of state and values in programming language semantics is an oversimplification. The bug described by Bloch is a simple value overflow that occurs in the computation of the average of two values: `int mid = (low + high) / 2`. For sufficiently large values of `low` and `high` this statement is wrong. One version that works for all values is `mid = low + ((high - low) / 2)`.

Even in the light of industrially deployed programs failing because of this error, one is tempted to say that they are not really wrong; that they work as long as `int` behaves like the integers of mathematics, and that people should not be using values that are too large. If the specification of binary search did have this precondition explicitly stated, then this view would be defensible. Of course it does not, because we would rather not think about the messy world of finite values, finite memory, unsafe pointer operations, pointer alignment, and other low level constraints of real programming languages when we are focusing on the interesting parts of verifying a program. This view is not just wishful thinking: it was, and in part still is, also widely believed that dealing with all this detail in program verification is intractable, and that it obscures the algorithmic part where errors are most likely. Nonetheless, it is our thesis and contribution that we can indeed verify real programs in real programming languages with their real semantics, and provide the hard correctness guarantees of formal software verification, without sacrificing soundness and without drowning in detail.

Our verification methodology for programs in languages like C is to use Hoare logic together with an automated verification condition generator. Additionally, for pointer programs, we would like to use separation logic [16, 25], which promises to be an efficient and scalable method. However, existing implementations of separation logic [31, 18] use simplifications that while convenient are not directly suitable for C. For instance, they model the heap as a function from integers to integers, or treat values on the heap as atomic.

In contrast, we model the values and heap of the C programming language in the theorem prover precisely. For overflow problems we use the correct mathematical structure (e.g. finite integers). For pointers, correct models are more complex. C's view of memory is that of an array of bytes coupled with various access restrictions. Further, pointers have to be aligned, dereferencing the null pointer is undefined, etc.

Using such a model directly is intractable for verifying real programs. Our contribution is to reconcile this model with convenient, abstract notation and reasoning. Our framework supports both, normal Hoare logic and a sound separation logic, by instantiating Schirmer's Hoare logic environment [27]. Schirmer provides a generic imperative language with operational semantics, Hoare logic, and generic state in Isabelle/HOL. We instantiate this to C on the one side, and provide optional separation logic notation for assertions on the other side. At the heart of our work stands the observation that although languages like C are not type-safe, and although many programs do not always use pointers in a type-safe

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL'07 January 17–19, 2007, Nice, France.
Copyright © 2007 ACM 1-59593-575-4/07/0001... $5.00.

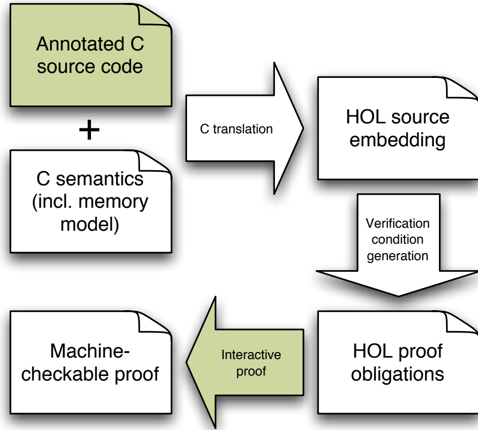

Figure 1. The C verification process. User-supplied components indicated with shading.

way, the majority of pointer operations in any given program are well-behaved and can be treated in the abstract text-book way.

In previous work [28], we showed that the low-level view of the heap as an array of bytes can be soundly unified with a view of multiple abstract heaps of abstract values. We provided an implementation of this unified memory model in Isabelle/HOL and applied it to the verification of page tables in the L4 microkernel. In this paper, we extend this work by making it more concrete and more firmly grounded in reality on the one side, and more abstract and better suited for high-level verification on the other side:

- We harden our implementation so that it accurately reflects the real semantics of a significant, strict subset of the C programming language. C programs as accepted by standard C compilers can be read directly into the theorem prover. We also show how to add a number of low-level semantic restrictions like pointer alignment and automatic NULL-dereference guards without incurring undue verification overhead.
- We extend and generalise the memory model to support separation logic constructs, and we implement a shallow embedding of separation logic with the frame rule on top of this model in Isabelle/HOL. This constitutes the first sound implementation of separation logic for a mainstream programming language.
- We show two case studies in this setting: a simple list reversal, and a full formal verification of the L4 kernel memory allocator. We use the list reversal example to demonstrate the model and to show that easy programs remain easy to verify. The memory allocator study shows the applicability of the model and verification of real, security critical programs. It also gives an example of a program that cannot be verified formally in the traditional setting without losing soundness.
- Finally, we use the two case studies to evaluate separation logic against traditional Hoare logic. To our knowledge this is the first such evaluation of separation logic in one unified setting.

Fig. 1 shows the process of verifying a C program in our setting. The user (the program verifier) provides a normal C program that contains invariant and pre/post annotations in comments. Optionally, the user may also provide an Isabelle/HOL specification with definitions used in these annotations. Building on our model of memory, separation logic, and the semantics of C, the program is then translated into an imperative program in Isabelle/HOL. Schirmer's VCG generates the proof obligations. Proving these is where most of the user-work happens and the process will usually result in changes to the annotations and/or program until a proof is found. The soundness of this setup depends on translating C programs correctly to their Isabelle counterparts, on the correctness of the memory model (both of which we present here), and on the VCG (which Schirmer has proved sound in Isabelle [27]).

Our work is motivated by a project to formally verify the functional correctness of the L4 microkernel [17]. While we concentrate on operating-system-level code and Isabelle/HOL in our case studies and implementation, we believe that this work is widely applicable. The basic technique generalises to other low-level languages and is orthogonal to developing an operational semantics for the statements of the language. We concentrate on a precise model of state, values, and memory. The implementation in the theorem prover uses features specific to Isabelle (e.g. type classes) to optimise proof productivity, but the model should be implementable in different provers like PVS using slightly different mechanisms.

2. 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. 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 $\mathsf{Some}\ a$ and $'a \rightharpoonup {'b}$ instead of $'a \Rightarrow {'b}$ *option*. The Some constructor has an underspecified inverse called the, satisfying $\mathsf{the}\ \lfloor x \rfloor = x$. Function update is written $f(x := y)$ where $f :: {'a} \Rightarrow {'b}$, $x :: {'a}$ and $y :: {'b}$ and $f(x \mapsto y)$ stands for $f\ (x := \mathsf{Some}\ y)$. We use $\{\mapsto\}$ for updating whole sets. 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. For succinctness, we use abbreviations like *word*8 and *word*32. The functions unat and of-nat convert to and from natural numbers (with u for *unsigned*). The notation $\{w..{+}n\}$ stands for the interval starting at the word w with $n{::}nat$ elements, possibly wrapping around to zero.

Hoare triples are written $\{\!|P|\!\}\ c\ \{\!|Q|\!\}$ where P and Q are assertions and c a program. In assertions, we use the syntax $\acute{}x$ to refer to the program variable x in the current state, while $^{\sigma}x$ means x in state σ. Program states can be bound in assertions by $\{\!|\sigma.\ P|\!\}$.

Isabelle supports axiomatic type classes [32] similar to, but more restrictive than Haskell's. The notation $'a{::}ring$ restricts the type variable $'a$ to those types that support the axioms of class *ring*. Type classes can be reasoned about abstractly, with recourse just to the defining axioms. Further, a type τ can be shown to belong to a type class given a proof that the class's axioms hold in τ. All abstract consequences of the class's axioms then follow for τ.

Isabelle theories can be augmented with LaTeX text which may contain references to Isabelle theorems (by name—see chapter 4 of [23]). We use this presentation mechanism to generate the text for most of the definitions and all of the theorems in this paper, taking them directly from the Isabelle proofs.

3. Translation of C

In our domain, C programs are happy to exploit C's low-level features. We cannot pretend that we are verifying a pseudo-Pascal

```
int i = 0, a[2] = {0,0};
int f(void) { i++; return i; }

/* will return either 0 or 1 */
int main(void) { a[i] = f(); return a[0]; }
```

Figure 2. A non-deterministic C program

with integer values of infinite range. In this section, we briefly describe the C dialect that we require, and how our translation of this dialect handles many of C's low-level complications. A number of memory specific details are further discussed in Sect. 4.

Syntactically, our C dialect is a C subset. We must be able to pass implemented C source code to standard compilers (on whose correctness we implicitly rely), but we do also impose some syntactic restrictions on developers. For example, we make some simplifications that have no deep semantic impact, such as requiring all `struct` declarations to occur at the top-level of a translation unit. Rather more significantly, we prohibit side effects in almost all expressions. Assignments become statement-forms, and functions that return values may be called only as the right-hand side of an assignment.

Semantically, we make a number of assumptions about the nature of the environment in which our C code will be executing. These move the C we are verifying away from the standard's ideal of *strict conformance*. When writing applications, such assumptions are often a sign of programmer laziness, and a harbinger of pain when code is ported. At the systems level, these assumptions are unavoidable. We categorise them using the standard's terms for describing varying (and illegal) program behaviours: *implementation-defined*, *unspecified*, and *undefined*.

Implementation-defined behaviours are the easiest to treat. These behaviours are those on which the standard places broad restrictions, but where it also requires that the implementation make a particular (and documented) choice of behaviour.

For example, an implementation must choose the size of its character type (typically 8 bits), the size of other integral types in terms of bytes (`int` values are now typically four bytes), the endianness of integral types, how negative numbers are represented (typically two's-complement), and how integer division behaves with negative arguments. Most of the time, compilers choose to reflect the decisions made by the underlying hardware architecture. As we know the nature of the architecture for which we are developing our code, we add these decisions as extra assumptions to the verification process. This does mean that the same source-code does need to be verified anew for each underlying architecture where it is to be deployed. Depending on the program this could amount to merely re-running the proof.

Unspecified behaviours occur when the standard permits an implementation to vary its behaviour, but does not require any documentation of this variation. Indeed, an implementation is not required to make a particular choice consistently. This allows optimisation to drive the choice of behaviour. The most significant place where behaviour is unspecified is in order of evaluation, of arguments to assignments, of arguments of binary operators such as `+` and `/`, and of arguments to function calls.

Most C expressions that appear to induce different behaviours because of this under-specification are actually examples of *undefined* or illegal behaviour. This is because of the strong restrictions on the way in which programs are allowed to read and write memory. Nonetheless, it is possible to write legal, non-deterministic C programs. One such appears in Fig. 2.

By making all expressions side-effect free, and making assignments and function calls statement forms, we eliminate a whole class of undefined behaviours. By further requiring that functions may only be called in contexts where their return value is ignored, or where it is assigned to a single variable (which restriction the program in Fig. 2 violates), we ensure that all code must have only one possible behaviour, unless we choose to work from deliberate underspecifications, as might happen with an implementation of `malloc`.[1]

For other unspecified behaviours, such as the values of possible padding bits in integer representations, our knowledge of the underlying architecture means we know how the machine will behave, and our verification can exploit this.

Undefined behaviours occur when a program attempts to do something illegal, such as dereferencing a null pointer, or doing an integer division by zero. It is part of the devil's pact that programmers make when they program in C that implementations are not required to trap such events. Instead, implementations are free to do anything at all, and again, there is no requirement that their undefined behaviours be documented. This feature of C makes *testing* difficult, but when verification is performed we can explicitly make sure that undefined behaviours can not occur.

One approach to this verification task is to use explicit *guards*. Guards are extra program annotations, similar to assertions users can provide with the standard `assert` function. In order to be verified with respect to *any* specification, a program must have all of its guards shown to hold when they are encountered. For every possible undefined behaviour, we generate guards that are sufficient to ensure that the undefined behaviour cannot occur. If done manually, this process of annotating programs with guards might lead to guards being omitted, but by generating guards mechanically we can ensure that all those we wish to treat are indeed generated.

Guards that we include prevent division by zero, dereferencing the null pointer, and dereferencing an improperly aligned pointer. One further guard is the test that all memory-writes are to allocated memory. We use this guard to show a form of memory safety in Sect. 5.3 as a proof convenience but it may be omitted from the semantics when considering systems code at the verifier's discretion. A stricter semantics for application verification would require such a guard always. This flexibility in choosing our code's environment is another reason in favour of the guards-based approach: it is straightforward to enable different sorts of guards depending on circumstances. For example, in some kernel verifications, dereferencing the null pointer might be a valid way of accessing the first element of the machine's exception vector. If this was the case, the null pointer guard would need to be disabled. In this way, we can give definition to a variety of behaviours that the standard leaves undefined. Our systems code will not be *strictly conforming*, but indeed this line was crossed when our code relied on implementation-defined aspects of the programming environment.

Finally, generating guards is straightforward: as expressions are translated into the verification environment, forms that may cause difficulties are recorded. When the translation of the enclosing statement has finished, it can be preceded by the appropriate checks on the given expressions. Because of our syntactic restrictions preventing side effects, the repeated evaluation (as guard, and in the original expression) of these sub-expressions is sure to be safe.

Our aim is to make verification part of the engineering process. Crucially, the engineering process does not produce source code in one phase and then stop, with verification taking over from implementation. Rather, implementation continues at the same time. In

[1] Our restriction on the use of function calls may appear draconian. It forbids not only `f(x) + g(y)`, but also `x + f(y)`. Two unexplored options are possible. We might annotate some function calls as side-effect free, and then allow these functions to be called within expressions. Alternatively, rather than have the programmer do so by hand, we might compile such expressions into a linearised form that forced a particular order of evaluation.

order to tie verification effort to the source code that is verified, we annotate source code with Isabelle invariants and specifications.

4. A Unified Memory Model for Pointers

When verifying C programs that may break many of the commonly employed abstractions described in the introduction we face a dilemma. Since convenient simplifications must be discarded we require detailed and cumbersome low-level models. This may increase the difficulty of the verification task, in particular when using an interactive theorem prover, where this detail is exposed to the program verifier. We take soundness to be non-negotiable, yet some abstraction is also required for tractability.

In this and the next section we focus on resolving this difficulty for the C memory model. We describe both a low-level semantic model based on finite byte-addressable memory and connections to two high-level proof abstractions—multiple typed heaps and separation logic. The latter can safely be used when programs remain inside a type-safe fragment. Since most code remains in this fragment in practice, even in the systems domain, we gain soundness while alleviating the additional burden on the program verifier in the common case.

4.1 Semantic model

It is common in language semantics to treat the heap or memory as a partial function $int \rightharpoonup lang\text{-}val$, where *int* is the type of addresses and *lang-val* the type of all language values. While greatly simplifying the formalisation, this makes several assumptions that are not valid in our setting:

- ***Addresses range over an infinite integer type.*** In C addresses are constrained by a finite addressable memory, which affects the semantics of pointer arithmetic and memory allocation, e.g. `*(x+1) = y` may in fact be a null pointer dereference.
- ***Values representations are atomic.*** C language types have representations spanning multiple locations, and it is possible to have value updates at one location affect values in other cells. This calls for a semantic model that both captures values' storage sizes, and reflects these update semantics accurately. E.g. `*x = 0xdeadbeef` affects not only the byte at location `x`, but also the bytes at locations `x + 1`, `x + 2`, and `x + 3`. An additional complication is alignment: per-type restrictions on address validity. For example, 16-bit `short` values may be forced to be stored at even addresses. Expressing alignment conditions in dereferencing and update semantics requires having a constant byte granularity for addressing.
- ***Heap partiality.*** Heap partiality is often used in the heap dereferencing semantics in memory or type-safety checks. Much weaker variants of these properties hold for C programs and it is not always necessary to introduce them in the dereference semantics. This is particularly important in making it possible to verify low-level code that manages details such as the layout of its own address space or implements the functionality of `malloc`.

Earlier work [28] solved a similar problem, where the aim was to reconcile the low-level C memory model and the abstraction of multiple typed heaps. We present that semantic model here, together with a significant extension of the heap abstraction. In particular, we remove the well-formedness invariant previously required, add general support for guards that protect against undefined behaviours, and in Sect. 5 employ it as the basis of a separation logic embedding.

In our model, heap memory state is described as a total function from addresses, represented by a word$n$ type corresponding to the machine address type, to bytes, also a word$n$ type. Arithmetic operations on word$n$ values are modulo 2^n. For example, on a machine with 32-bit addresses and 8-bit bytes the heap memory state will be:

$$\begin{aligned} &addr = word32 \\ &byte = word8 \\ &heap\text{-}mem = addr \Rightarrow byte \end{aligned}$$

Each language type is assigned a unique type in the theorem prover's logic. This allows for both an intuitive definition of language operators as functions in HOL, and the harnessing of the theorem prover's type inferencing mechanism to avoid unnecessary type annotations in assertions and proofs. All such types belong to an axiomatic type class $'a{::}c\text{-}type$ in Isabelle, which introduces constants that connect the low-level byte representation and the HOL values:

$$\begin{aligned} &\mathsf{to\text{-}bytes} :: {'a}{::}c\text{-}type \Rightarrow byte\ \mathsf{list} \\ &\mathsf{from\text{-}bytes} :: byte\ \mathsf{list} \rightharpoonup {'a}{::}c\text{-}type \\ &\mathsf{typ\text{-}tag} :: {'a}{::}c\text{-}type\ itself \Rightarrow \mathsf{typ\text{-}tag} \\ &\mathsf{typ\text{-}info} :: {'a}{::}c\text{-}type\ itself \Rightarrow \mathsf{typ\text{-}info} \end{aligned}$$

The functions to-bytes and from-bytes convert between Isabelle values and lists of bytes suitable for writing to or reading from the raw heap state. The function typ-tag associates a unique *type tag* with each $'a{::}c\text{-}type$, providing a means of treating language types as first-class values in HOL. Finally, typ-info provides enough structure to allow size and alignment information for the type to be calculated. We can then use this to define functions size-of $:: {'a}{::}c\text{-}type\ itself \Rightarrow nat$ and align-of $:: {'a}{::}c\text{-}type\ itself \Rightarrow nat$ respectively.

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

$$\begin{aligned} &\mathsf{from\text{-}bytes}\ (\mathsf{to\text{-}bytes}\ v) = \lfloor v \rfloor \\ &|\mathsf{to\text{-}bytes}\ (x{::}'a)| = \mathsf{size\text{-}of}\ \mathsf{TYPE}('a) \\ &0 < \mathsf{size\text{-}of}\ \mathsf{TYPE}('a) \\ &\mathsf{size\text{-}of}\ \mathsf{TYPE}('a) < \mathsf{addr\text{-}card} \\ &\mathsf{align\text{-}of}\ \mathsf{TYPE}('a)\ \mathsf{dvd}\ \mathsf{addr\text{-}card} \end{aligned}$$

These conditions follow mostly from requirements in the C standard, e.g. fixed size representations, with the exception of the final alignment constraint which we add to make pointer arithmetic better behaved, and which holds on all the C implementations we are aware of. The constant addr-card represents the size of the address space, e.g. 2^{32}.

Finally, we introduce a distinct Isabelle pointer type for each Isabelle type, used to model C pointer types:

$$\textbf{datatype}\ {'a}\ ptr = \mathsf{Ptr}\ addr$$

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. Since the type variable does not appear on the right-hand side it is a phantom type. Nonetheless, the type information is used to constrain the action of various pointer operators by making use of the type information associated with $'a$. The destructor ptr-val retrieves the address from a pointer value. The pointer types $'a{::}c\text{-}type\ ptr$ can be shown to be to be instances of $'a{::}mem\text{-}type$. An example of the use of the phantom type variable comes in the definition of pointer addition, adding a word n to a pointer $p{::}'a\ ptr$:

$$p +_p n = \mathsf{Ptr}\ (\mathsf{ptr\text{-}val}\ p + n * \mathsf{of\text{-}nat}\ (\mathsf{size\text{-}of}\ \mathsf{TYPE}('a)))$$

Another example of the utility of such polymorphic definitions is the pointer alignment guard from Sect. 3:

$$\mathsf{ptr\text{-}aligned}\ (p{::}'a\ ptr) \equiv \mathsf{align\text{-}of}\ \mathsf{TYPE}('a)\ \mathsf{dvd}\ \mathsf{unat}\ (\mathsf{ptr\text{-}val}\ p)$$

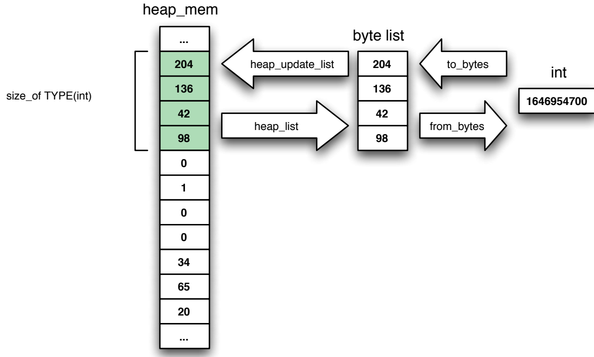

Figure 3. *int* heap representation

In this work we deal primarily with C's base and pointer types but it is possible to extend this to compound types. Array indexing, access, and update semantics can be expressed in terms of the corresponding pointer semantics. C's `struct` types can be translated to Isabelle **record** types, however the models in this paper require extending when fields have their address taken[2] or padding is involved. Similarly, `union` types can be treated as **datatype**s when a tag field can be identified, and otherwise translated to type-casts.

Low-level heap access and update Heap dereferences in expressions, e.g. $*p + 1$ are given a semantics by first lifting the raw heap state with the polymorphic lift function, e.g. $\mathsf{lift}\ s\ p + 1$ where s is the current state. The definition of lift is presented below, together with that of heap-update, providing semantics for heap updates. For example, $*p = *q + 5$ is translated to the state transformer $\lambda s.\ \mathsf{heap\text{-}update}\ p\ (\mathsf{lift}\ s\ q + 5)\ s$. Fig. 3 illustrates the value transformations involved.

$\mathsf{heap\text{-}list} :: \mathit{heap\text{-}mem} \Rightarrow \mathit{nat} \Rightarrow \mathit{addr} \Rightarrow \mathit{byte}\ \mathsf{list}$

$\mathsf{heap\text{-}list}\ h\ 0\ p = []$

$\mathsf{heap\text{-}list}\ h\ (\mathit{Suc}\ n)\ p = h\ p \cdot \mathsf{heap\text{-}list}\ h\ n\ (p + 1)$

$\mathsf{h\text{-}val} :: \mathit{heap\text{-}mem} \Rightarrow {'a}{::}\mathit{c\text{-}type}\ \mathit{ptr} \rightharpoonup {'a}$

$\mathsf{h\text{-}val}\ h\ p \equiv \mathsf{from\text{-}bytes}\ (\mathsf{heap\text{-}list}\ h\ (\mathsf{size\text{-}of}\ \mathsf{TYPE}({'a}))\ (\mathsf{ptr\text{-}val}\ p))$

$\mathsf{lift} :: \mathit{heap\text{-}mem} \Rightarrow {'a}{::}\mathit{c\text{-}type}\ \mathit{ptr} \Rightarrow {'a}$

$\mathsf{lift}\ h \equiv \lambda p.\ \mathsf{the}\ (\mathsf{h\text{-}val}\ h\ p)$

$\mathsf{heap\text{-}update\text{-}list}\ p\ []\ h = h$

$\mathsf{heap\text{-}update\text{-}list}\ p\ (x \cdot xs)\ h = \mathsf{heap\text{-}update\text{-}list}\ (p + 1)\ xs\ (h(p := x))$

$\mathsf{heap\text{-}update}\ p\ v\ h \equiv \mathsf{heap\text{-}update\text{-}list}\ (\mathsf{ptr\text{-}val}\ p)\ (\mathsf{to\text{-}bytes}\ v)\ h$

4.2 Typed heaps

While it is possible to do proofs about programs directly with this semantics, it is rather complicated to do so. Problems quickly arise when the potential for overlapping value representations on the heap and aliasing between lifted typed heaps complicate heap update post-conditions. This is an example of the *aliasing* [8] problem inherent in pointer program proofs being further compounded by the low-level semantic model utilised. We call this representation overlap problem *inter-type aliasing* for the purpose of this paper; see Fig. 4 for an example of this. In this section we present an abstraction from our semantic model, together with appropriate

[2] We also have not required supporting the & operator for local variables yet. These could be modelled explicitly in the *heap-mem*.

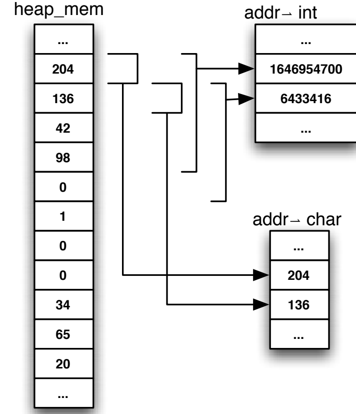

Figure 4. Inter-type aliasing

rewrite rules connecting the models, that avoids reasoning about this kind of aliasing when programs remain in the type-safe fragment of C.

In this type-safe fragment, a convenient model for a type-safe language would be based on explicit multiple typed heaps, one for each language type, e.g. $\mathit{addr} \rightharpoonup \mathit{char}$, $\mathit{addr} \rightharpoonup \mathit{short}$. While this avoids the problem of inter-type aliasing and greatly simplifies proofs, it does not allow us to escape the type-safe world. The lift function provides a view of multiple typed heaps, but values in a lifted $'a$ heap change when the heap is updated through a $'b\ \mathit{ptr}$.

However, inside the type-safe fragment, there is an implicit mapping between memory locations and types, and heap dereferences respect this mapping. We introduce this mapping as an additional state component, and refer to it as the *heap type description*.

$$\mathit{heap\text{-}typ\text{-}desc} = \mathit{addr} \rightharpoonup \mathsf{typ\text{-}tag}\ \mathit{option}$$

The heap type description is a *history* variable, and as such does not affect the semantics of our programs. Since in C this mapping cannot be extracted from the source code, the program verifier adds proof annotations that update the heap type description. The overhead of these annotations is very low relative to the proof effort because the intended type for a region of memory changes infrequently—mainly for `alloc` and `free`.

The heap type description is partial, since it only maps memory actually used by the program. Each value representation has the typ-tag corresponding to its type stored at the base address. The rest of the *heap footprint* of the value is also mapped, but with a None value padding instead of a tag, making heap validity, as defined below, monotonic, a useful property in the development of the separation logic embedding. Without padding we would also have the additional requirement of a well-formedness invariant [28] on the heap type description.

We write $d{,}g \models_t p$ to mean that the pointer p is valid in heap type description d with guard g. The definition is straightforward:

$$\begin{aligned}\mathsf{valid\text{-}footprint}\ d\ x\ t\ n \equiv\ & d\ x = \lfloor\lfloor t\rfloor\rfloor\ \wedge \\ & (\forall y.\ y \in \{x + 1..+n - 1\} \longrightarrow d\ y = \lfloor\mathsf{None}\rfloor)\end{aligned}$$

$$\begin{aligned}d{,}g \models_t p \equiv\ & \mathsf{valid\text{-}footprint}\ d\ (\mathsf{ptr\text{-}val}\ p)\ (\mathsf{typ\text{-}tag}\ \mathsf{TYPE}({'a})) \\ & (\mathsf{size\text{-}of}\ \mathsf{TYPE}({'a}))\ \wedge \\ & g\ p\end{aligned}$$

The guard g strengthens the assertion to restrict validity based on the language's pointer dereferencing rules. For example, alignment

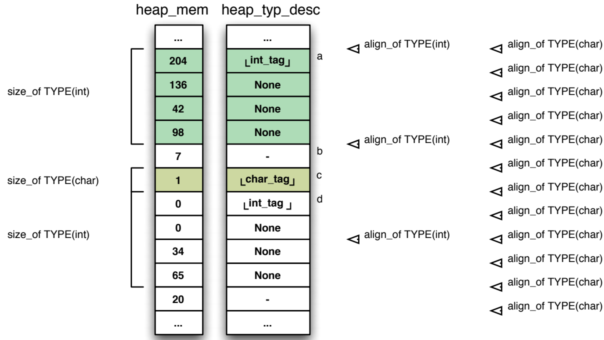

Figure 5. Example heap state

can be captured with d,ptr-aligned $\models_t p$. In the interests of concision, we occasionally omit g hereafter.

An example heap memory and type description state is given in Fig. 5. Here htd,ptr-aligned $\models_t a$ and htd,ptr-aligned $\models_t c$, but $\neg$ htd,ptr-aligned $\models_t b$ and $\neg$ htd,ptr-aligned $\models_t d$. The pointer b::*char ptr* is not valid because the correct heap footprint is missing from the heap type description, and d::*int ptr* is not valid because it is not aligned.

The ptr-tag function updates a heap type description so that it encompasses a pointer, i.e. $g\ p \Longrightarrow$ ptr-tag $p\ d,g \models_t p$:

$$\text{ptr-clear}\ p\ n\ d \equiv d(\{\text{ptr-val}\ p + 1..{+}n - 1\}\{\mapsto\}\text{None})$$

$$\text{ptr-set}\ p\ t\ d \equiv d(\text{ptr-val}\ p \mapsto \lfloor\text{typ-tag}\ t\rfloor)$$

$$\text{ptr-tag}\ (p{::}'a{::}c\text{-}type\ ptr) \equiv$$
$$\text{ptr-set}\ p\ \text{TYPE}('a{::}c\text{-}type) \circ \text{ptr-clear}\ p\ (\text{size-of TYPE}('a{::}c\text{-}type))$$

Heap type description update annotations are written in C comments. The syntax *AUXUPD* takes a guard and a heap type description state transformer as a parameter. E.g. to indicate that the location at p now is the value of a valid pointer of type *addr ptr*, we write $/{*}{*}$ *AUXUPD*: (g, ptr-tag (´p::*addr ptr*)) $*/$.

The following two-stage lifting process provides an abstract heap view for proofs:

1. The first stage results in an intermediate *heap-state*:

$$heap\text{-}state = addr \rightharpoonup \text{typ-tag}\ option \times byte$$

The function lift-state filters out locations that are None in the heap type description, removing values that should not affect the final lifted typed heaps. Equality between lifted heaps is then modulo the heap type description domain.

$$\text{lift-state} \equiv \lambda(h, d)\ x.\ \text{case}\ d\ x\ \text{of None} \Rightarrow \text{None} \mid \lfloor t\rfloor \Rightarrow \lfloor(t, h\ x)\rfloor$$

Lifted validity and heap-list are expressed on *heap-state*s with $d,g \models_s p$ and heap-list-s respectively in the obvious way.

2. The second lifting stage results in typed lifted heaps again. The lift-typ-heap function restricts the heap domain so that the only values affecting the resultant heap are inside the heap footprint of valid pointer values. Equality is now modulo pointer validity.

$$\text{lift-typ-heap}\ g\ s \equiv (\text{from-bytes} \circ \text{heap-list-s}\ s\ (\text{size-of TYPE}('a)) \circ \text{ptr-val})\restriction_{\{p \mid s,g \models_s p\}}$$

Note that it is not sufficient to just restrict the domain to addresses possessing the alignment of the lifted heap's type in order to avoid overlapping value representations, since the alignment of a type can be smaller than its footprint size: in our semantics the relationship between alignment and size is given by align-of t dvd size-of t.

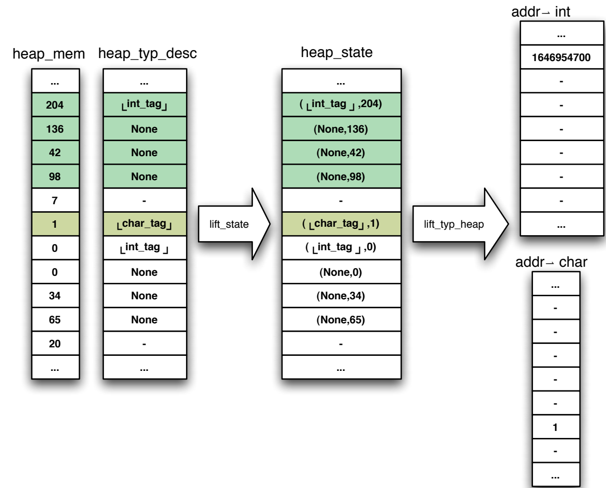

Figure 6. Two-stage lifting

The two stages, shown in Fig. 6, are combined with lift$_\tau$:

$$\text{lift}_\tau\ g \equiv \text{lift-typ-heap}\ g \circ \text{lift-state}$$

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

$$\text{lift}_\tau\ g\ (h, d)\ p = \lfloor v\rfloor \Longrightarrow \text{lift}\ h\ p = v$$

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

$$d,g \models_t p \Longrightarrow \text{lift}_\tau\ g\ (\text{heap-update}\ p\ v\ h, d) = \text{lift}_\tau\ g\ (h, d)(p \mapsto v)$$

$$[\![d,g' \models_t p;\ \text{typ-tag TYPE}('a) \neq \text{typ-tag TYPE}('b)]\!]$$
$$\Longrightarrow \text{lift}_\tau\ g\ (\text{heap-update}\ p\ v\ h, d) = \text{lift}_\tau\ g\ (h, d)$$

These, added to the default simplification set with other heap-related lemmas, do not require manual application. The typ-tag TYPE($'a$) $\neq$ typ-tag TYPE($'b$) condition can be resolved automatically, as long as the type tag definitions for language types are also in the simplification set.

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 the unsafe case, things are no longer automatic, and we require new rules to be proven in terms of the underlying lift and heap-update, as was previously required anyway.

5. Separation Logic

The previous section presents a proof technique that tames inter-type aliasing. However, the problem of *intra-type aliasing* remains, where two valid same-typed pointers may have identical values. Dealing with this often requires explicit anti-aliasing invariants on typed heaps, a problem that has been recognised in the literature [8]. Separation logic [16, 25] provides an approach in which

anti-aliasing information may be expressed implicitly in assertions, potentially simplifying specifications and proofs. In this section we present a development of separation logic based on the preceding memory model.

5.1 Shallow embedding

Below we describe a shallow embedding for separation assertions, where the semantic constructs of assertions are translated to HOL, as opposed to a deep embedding where the syntax of assertions would be considered a distinct type in the logic. There is a tradeoff involved in the choice of embedding approach—shallow embeddings are often more pragmatic and expressive, while deep embeddings allow for language metatheory reasoning and proof optimisations [33]. We opt for the former since our focus here is on the verification task.

We model separation assertions as predicates on *heap-state*s, applied in assertions of the verification environment to the result of the first lifting stage of Sect. 4.2. For example, a loop invariant with the separation assertion P and heap memory and type description state in the variables h and d respectively is written $\{\!|\ P\ (\mathsf{lift\text{-}state}\ (\acute{h}, \acute{d}))\ |\!\}$, which we abbreviate as $\{\!|P^{sep}|\!\}$.

As in the development of Reynolds we have an empty heap predicate:

$$\Box = (\lambda s.\ s = \mathsf{empty})$$

The definition of the singleton heap assertion is more involved in our embedding, and is provided below. $p \mapsto_g v$ asserts that the heap contains exactly one mapping matching the guard g, at the location given by pointer p to value v.

$$\begin{aligned} p \mapsto_g v = \lambda s.\ & \mathsf{lift\text{-}typ\text{-}heap}\ g\ s\ p = \lfloor v \rfloor \wedge \\ & \mathsf{dom}\ s = \{\mathsf{ptr\text{-}val}\ p..+\mathsf{size\text{-}of}\ \mathsf{TYPE}('a)\} \end{aligned}$$

The guard is an addition to the usual $p \mapsto v$ and serves the same purpose as in Sect. 4.2, i.e. strengthening the assertion to aid in discharging guard proof obligations.

There are two significant separation connectives, conjunction and implication:

$$\begin{aligned} s_0 \perp s_1 &= \mathsf{dom}\ s_0 \cap \mathsf{dom}\ s_1 = \emptyset \\ s_0 +\!\!+ s_1 &= \lambda x.\ \mathsf{case}\ s_1\ x\ \mathsf{of}\ \mathsf{None} \Rightarrow s_0\ x \mid \lfloor y \rfloor \Rightarrow \lfloor y \rfloor \\ P \wedge^* Q &= \lambda s.\ \exists s_0\ s_1.\ s_0 \perp s_1 \wedge s = s_1 +\!\!+ s_0 \wedge P\ s_0 \wedge Q\ s_1 \\ P \longrightarrow^* Q &= \lambda s.\ \forall s'.\ s \perp s' \wedge P\ s' \longrightarrow Q\ (s +\!\!+ s') \end{aligned}$$

The definitions are standard, with the intuition behind separation conjunction that $(P \wedge^* Q)\ s$ asserts that s can be partitioned into two subheaps such that P holds on one subheap and Q on the other. This allows for predicates on the heap to be written that conveniently include anti-aliasing information.

Some additional mapping assertions are useful:

$$\begin{aligned} \mathsf{sep\text{-}true} &= \lambda s.\ \mathsf{True} \\ p \mapsto_g - &= \lambda s.\ \exists v.\ (p \mapsto_g v)\ s \\ p \hookrightarrow_g v &= p \mapsto_g v \wedge^* \mathsf{sep\text{-}true} \\ p \hookrightarrow_g - &= \lambda s.\ \exists x.\ (p \hookrightarrow_g x)\ s \end{aligned}$$

The standard commutative, associative, and distributive properties apply to the connectives, and we have formalised pure, intuitionistic, domain, and strictly exact assertions and their properties [25]. We require a stronger definition of the singleton heap assertion for it to be strictly exact. Some of these properties are routinely used in proofs and have been grouped in simplification sets and/or added to the default simplification set.

Since this is a shallow embedding, standard HOL connectives and quantifiers can be freely mixed with the separation connectives, e.g. $\lambda s.\ P\ s \wedge (Q \wedge^* R)\ s$.

A key feature of this embedding is that it avoids the problem of *skewed sharing* [25]. This is essentially the problem of inter-type aliasing in separation logic. An approach like ours, where a history variable is introduced, was suggested as a future direction for separation logic by Reynolds. We have developed this, have a machine-checked formalisation and have deployed it in real-world verification examples (see Sect. 6).

Another notable gain from our development is the harnessing of Isabelle's type inference to avoid explicit type annotations in assertions. Since language types are assigned Isabelle types and pointer types are derived from these, asserting that $p \mapsto v$, where p is a program variable automatically constrains the type of v.

5.2 Lifting proof obligations

Our verification condition generator applies weakest precondition rules to transform Hoare triples to HOL goals that can then be solved by applying theorem prover tactics. In Sect. 4.2, rewrites were given that could automatically lift the raw heap component of these proof obligations, and in this section we provide rules that allow the low-level applications of lift and heap-update to be expressed in terms of separation assertions. This is desirable as reasoning can then use the derived rules for these assertions at the separation logic level, whereas the alternative of unfolding the definitions produces a massively more complex goal and proof.

The approach taken here is quite different to the usual separation Hoare logic proof technique employed in the literature, where a new Hoare logic is developed based on separation logic and individual rules are applied at the Hoare logic level. The advantage of our approach is two-fold: we avoid having to manually apply Hoare rules, a task easily automated, and can take advantage of an existing verification framework and condition generator. On the other hand, applying the rules in this section requires the program verifier to understand the relationship between components of the HOL goals and the original program since this structure is lost during verification condition generation.

The following rule connects lift and separation mapping assertions:

$$\frac{(p \hookrightarrow_g v)\ (\mathsf{lift\text{-}state}\ (h, d))}{\mathsf{lift}\ h\ p = v}$$

An example of how this may be used is the Hoare triple

$$\{\!|(p \hookrightarrow\ x \wedge^* Q)^{sep}|\!\}\ a = *p + *p\ \{\!|a = 2 * x|\!\}.$$

The resulting proof obligation

$$(p \hookrightarrow\ x \wedge^* Q)\ (\mathsf{lift\text{-}state}\ (h, d)) \Longrightarrow 2 * \mathsf{lift}\ h\ p = 2 * x$$

requires the value of x in terms of lift $h\ p$ or vice versa. By deriving from the assumption of the goal that $(p \hookrightarrow\ x)\ (\mathsf{lift\text{-}state}\ (h, d))$ using normal rules of separation logic, the above rule can be applied to solve the goal.

Heap update dereferences produce proof goals of the form

$$\begin{aligned} & P\ (\mathsf{lift\text{-}state}\ (h, d)) \Longrightarrow \\ & \quad Q\ (\mathsf{lift\text{-}state}\ (\mathsf{heap\text{-}update}\ p_0\ v_0\ (\mathsf{heap\text{-}update}\ p_1\ v_1 \\ & \qquad\qquad (\mathsf{heap\text{-}update}\ p_{\ldots}\ \ v_{\ldots}\ (\mathsf{heap\text{-}update}\ p_n\ v_n\ h))), d)) \end{aligned}$$

To reduce heap-updates to a separation assertion on the original state we first require a new separation predicate $g \vdash_s p$:

$$g \vdash_s p \equiv \lambda s.\ s, g \models_s p \wedge \mathsf{dom}\ s = \{\mathsf{ptr\text{-}val}\ p..+\mathsf{size\text{-}of}\ \mathsf{TYPE}('a)\}$$

This is related to the idea of the singleton heap predicate $p \mapsto_g -$, but the implication only works in one direction, $(p \mapsto_g v)\ s \Longrightarrow (g \vdash_s p)\ s$, since it is possible to have both $\mathsf{lift}_\tau\ g\ s\ p = \mathsf{None}$ and a valid footprint at p.

The following rules allow for the reduction of heap-updates:

$$\frac{(g \vdash_s p \wedge^* (p \mapsto_g v \longrightarrow^* P))\ (\mathsf{lift\text{-}state}\ (h, d))}{P\ (\mathsf{lift\text{-}state}\ (\mathsf{heap\text{-}update}\ p\ v\ h, d))}$$

$$\frac{(g \vdash_s p \wedge^* P)\ (\mathsf{lift\text{-}state}\ (h, d))}{(p \mapsto_g v \wedge^* P)\ (\mathsf{lift\text{-}state}\ (\mathsf{heap\text{-}update}\ p\ v\ h, d))}$$

These rules are analogous to the backwards and global reasoning Hoare logic mutation rules [25]. The first rule provides a weakest

precondition style rule that will match any separation assertion, while the second rule may be used on goal assertions that can be manipulated into the matching form.

When the heap type description is modified with ptr-tag the resulting goal needs to be reduced in a similar manner. We provide a rule for this that establishes separation validity from sep-cut, another additional separation predicate. sep-cut simply asserts that the locations in the *heap-state* domain are the supplied interval.

$\mathsf{sep\text{-}cut}\ x\ y \equiv \lambda s.\ \mathsf{dom}\ s = \{x..+y\}$

$$\frac{(\mathsf{sep\text{-}cut}\ (\mathsf{ptr\text{-}val}\ p)\ (\mathsf{size\text{-}of}\ \mathsf{TYPE}(\prime a)) \wedge^* P)\ (\mathsf{lift\text{-}state}\ (h, d))\quad g\ p}{(g \vdash_s p \wedge^* P)\ (\mathsf{lift\text{-}state}\ (h, \mathsf{ptr\text{-}tag}\ p\ d))}$$

The shallow embedding gives us the flexibility to add separation predicates that express information about a region of memory at different levels of abstraction, from $p \mapsto_g v$ to sep-cut. Once the above reduction rules are applied the reasoning can continue using the standard rules of separation logic without requiring additional proof goals or side-conditions.

In deriving these rules we have primarily been concerned with soundness, which we are forced to show prior to Isabelle admitting the rules. Our focus on real-world verification examples has caused us to pay rather less attention to the completeness of our system, particularly as we know that we can always drop down to the lower more concrete levels if required. However, completeness results have been explored elsewhere in the literature [16, 34].

5.3 Frame rule

The separation frame rule [34] is often seen as important to scalability. It allows for deriving a global specification from a local specification of a program's behaviour, with an arbitrary conjoined separation assertion on a part of the heap preserved by the program. In our embedding, this rule would have the form:

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

Unfortunately, such a general rule cannot be expressed in a shallow embedding since (i) the state-space type is program dependent and (ii) c is an arbitrary program in the underlying verification framework for which this rule may not be true. It is however possible to prove this rule for specific programs and state-spaces, and in our development of the frame rule we automate and make generic this process for our C subset.

We make further use of type classes to define $\prime a$::*heap-state-typ*, which provides access and update functions for the heap state and heap type description. A concrete program's state-space is instantiated as a member of this type class using automatic Isabelle/HOL tactics. The frame rule can then be expressed on programs c with a state-space in $\prime a$::*heap-state-typ* as:

$$\frac{\{\!|P^{sep}|\!\}\ c\ \{\!|Q^{sep}|\!\}\qquad \mathsf{mem\text{-}safe}\ c}{\{\!|(P \wedge^* R)^{sep}|\!\}\ c\ \{\!|(Q \wedge^* R)^{sep}|\!\}}$$

The mem-safe c assumption is required as the second problem arises from the dependence of this rule on a form of *memory safety*. In order to prove soundness it is helpful that a program generates a guard failure if it either (i) modifies the heap state or heap type description outside of the initial domain of the heap type description or (ii) depends on the heap type description outside this domain in any expression. This is not the case for the normal output of the C translation stage, since guards are only generated to prevent undefined behaviour as our C semantics understands it.

To show mem-safe c we insert additional guards for memory safety, e.g. for a heap update dereference asserting that the lvalue's heap footprint is contained entirely in the heap type description's

```
word_t reverse(word_t *i) {
  word_t j = 0;
  while (i) {
    word_t *k = (word_t*)*i;
    *i = j;
    j = (word_t)i;
    i = k;
  }
  return j;
}
```

Figure 7. In-place list reversal

domain. It should be noted that we do not require type safety, nor do we require this form of memory safety for most expressions as only heap type description updates may depend on the heap type description, i.e. the output of the C translation stage does not feature heap type description accesses except in such updates.

With the generated guards in place, Isabelle discharges the mem-safe c proof obligation automatically. The additional memory safety proof obligations obtained from the verification condition generator can be discharged with the aid of of a strengthened precondition asserting that the required locations are a subset of the heap type description's domain.

The above presentation of the frame rule is somewhat simplified in that P, Q and R are not functions of the local variable state. We have also derived the more general rule where they are and R does not depend on variables modified by c.

6. Case Studies

This section presents two case studies that demonstrate the support of abstract and low-level reasoning in our setting. Since we have formalised both classical Burstall/Bornat-style pointer reasoning and separation logic in the same framework, we can give a meaningful comparison of their applicability. While our results support the folklore view that separation logic is well suited for specification, but less convenient for verification, we find that the difference becomes smaller for the larger, more realistic case study. Since proofs can be checked, but specifications need to be understood, this advocates the use of separation logic for more complex verification tasks.

6.1 List reversal

This subsection shows a small example program—in-place reversal of a list—with simple, abstract reasoning. The list is somewhat unusual in that it has no content but the pointer values themselves. Lists like these are used in OS code for instance, where they are loosely part of larger data structures. Mehta and Nipkow [19] use the same example in their more abstract setting. We achieve the same style of reasoning and specification with almost the same degree of automation.

Fig. 7 shows the code (where `word_t` is `unsigned int`). Even though the program is very simple, it already makes use of unsafe pointer operations.

Classical setting

The pre/post specification of this program in a classical setting (without separation logic) is:

$\forall zs.\ \{\!|\mathsf{list}\ (\mathsf{lift}_\tau\ \mathcal{H})\ zs\ \acute{}i|\!\}$
$\acute{}reverse\text{-}ret :== PROC\ reverse(\acute{}i)$
$\{\!|\mathsf{list}\ (\mathsf{lift}_\tau\ \mathcal{H})\ (\mathsf{rev}\ zs)\ (\mathsf{Ptr}\ \acute{}reverse\text{-}ret)|\!\}$

The idiom $\acute{}reverse\text{-}ret :== PROC\ reverse(\acute{}i)$ is the standard form for procedure specification in Schirmer's environment. The

variable *reverse-ret* is automatically generated and makes the return value of function *reverse* available to the postcondition. We write $\mathcal{H} = (\acute{}h, \acute{}d)$ for the heap and type description in the current state.

The predicate list relates the heap pointer structure to abstract lists. It takes a heap of *word*32 values, a list of *word*32 values, and the start pointer of the list as arguments. Our definition is analogous to the one in Mehta and Nipkow [19] (we additionally deal with heap partiality):

$$\mathsf{list}\ h\ []\ i = i = \mathsf{NULL}$$
$$\mathsf{list}\ h\ (x{\cdot}xs)\ i = \exists j.\ \mathsf{ptr\text{-}val}\ i = x \wedge x \neq 0 \wedge h\ i = \lfloor j \rfloor \wedge \mathsf{list}\ h\ xs\ (\mathsf{Ptr}\ j)$$

The loop invariant again is almost the same as in Mehta and Nipkow. We use distinct to say that the list *zs* does not contain duplicate addresses and rev to reverse the abstract HOL list:

$$\{\!| \exists xs\ ys.\ \mathsf{list}\ (\mathsf{lift}_\tau\ \mathcal{H})\ xs\ \acute{}i \wedge \mathsf{list}\ (\mathsf{lift}_\tau\ \mathcal{H})\ ys\ (\mathsf{Ptr}\ \acute{}j) \wedge \mathsf{rev}\ zs = \mathsf{rev}\ xs\ @\ ys \wedge \mathsf{distinct}\ (\mathsf{rev}\ zs) |\!\}$$

As mentioned above, the proof of the 3 verification conditions generated for this program is almost completely automatic (5 lines of proof script). However, this has to be viewed in the context of a careful setup of automated simplification rules for the abstraction predicate list that again we share with Mehta and Nipkow. It consists of 13 lemmas, 12 of which are proven automatically. All low-level guards were discharged automatically as well; they did not play any role in the verification of this example.

Separation logic

The specification of the same program in separation logic is similar on the pre/post level, but the abstraction predicate is defined differently.

$$\forall zs.\ \{\!| (\mathsf{list}\ zs\ \acute{}i)^{sep} |\!\}\ \acute{}reverse\text{-}ret :== PROC\ reverse(\acute{}i)\ \{\!| (\mathsf{list}\ (\mathsf{rev}\ zs)\ (\mathsf{Ptr}\ \acute{}reverse\text{-}ret))^{sep} |\!\}$$

$$\mathsf{list}\ []\ i = \lambda s.\ i = \mathsf{NULL} \wedge \Box\ s$$
$$\mathsf{list}\ (x{\cdot}xs)\ i = \lambda s.\ i = \mathsf{Ptr}\ x \wedge x \neq 0 \wedge (\exists j.\ (i \mapsto j \wedge^* \mathsf{list}\ xs\ (\mathsf{Ptr}\ j))\ s)$$

The invariant is a bit shorter, because the separating conjunction takes care of distinctness:

$$\{\!| \exists xs\ ys.\ (\mathsf{list}\ xs\ \acute{}i \wedge^* \mathsf{list}\ ys\ (\mathsf{Ptr}\ \acute{}j))^{sep} \wedge \mathsf{rev}\ zs = \mathsf{rev}\ xs\ @\ ys |\!\}$$

The proof remains easy, but is not automatic any more and requires some manual application of rules for separating conjunction. This is not surprising, since separating conjunction is an existential statement which usually leads to manual intervention. The proof of the 3 verification conditions comes to a total of 32 lines, which we might be able to improve with specialised tactics for separation logic connectives. For example, one of the low-level guards was not solved automatically, because the required precondition was under a separating conjunction and had to be extracted manually. After this, the proof was automatic.

In the separation logic case, there was no specific automation setup for the list data structure apart from the lemma list *xs* NULL $= (\lambda s.\ xs = [] \wedge \Box\ s)$ which brings the total proof effort for this example to 62 lines, as opposed to 89 lines in the classical setting. Most of these 89 lines could be reused for programs on the same data structure, though, which is not the case for the separation logic proof.

6.2 The L4 kernel memory allocator

The case study presented in this section concerns the verification of the kernel memory allocator of the L4 microkernel with and without separation logic in Isabelle/HOL. At the OS kernel level, C library functions like `malloc` and `free` are not available yet, but have to be provided internally. Since this code is run in the

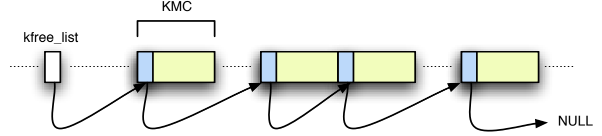

Figure 8. Management data structure of the L4 chunk allocator

privileged mode of the hardware, it is directly security critical. It is also the component that makes most critical use of unsafe C operations (although other tricks like xor of pointers are used elsewhere in L4). Three functions define the interface of this kernel memory allocator:

```
void init(void *start, void *end)
void free(void *address, word_t size), and
void *alloc(word_t size).
```

We have verified `alloc` and `free`, but only have space to show `alloc`; `init` consists of a call to `free`. The full proof document for this case, including code and invariants is available online [29]. The functions are not very large, but the fact that the originals contain more than 50% tracing and debugging code indicates that they were not easy to get right. We did not find any clear bugs in the code during verification which is encouraging for a system with several years of deployment. We did find two preconditions to `free` that are not immediately obvious: the address must be word-aligned (because it is used as a word pointer), and the sum of the address and the size must not overflow. If `free` is only used with values produced by `alloc`, and `init` is used with values as in L4, these conditions will hold.

Currently, L4 is distributed in C++, although no essential use of C++ features is made apart from using classes to structure the code. For a number of reasons (not only verification), we are porting L4 to C. The source code that is verified is taken from the L4Ka::Pistachio distribution, ported, configured for the x86 architecture, and preprocessed. The manual porting is not soundness critical as it is the result of the translation that will eventually run together with the rest of the C port, neither is the configure nor preprocessing part. If they are merely deterministic, we verify exactly what the compiler sees as input.

Fig. 8 shows the internal data structure that is used to manage memory. It is a NULL terminated, singly linked list of chunks of memory of a fixed size KMC (1024 bytes in this case) starting with the global variable `word_t *kfree_list`. The first 4 bytes of each free memory block are used to point to the next one. The blocks are often, but not always, adjacent in memory.

The implementation of `alloc` first searches for a contiguous block of memory of the right size that is correctly aligned. It then removes this block from the free list, zeroes out the memory region, and returns a pointer to it. If no such block is found, the kernel has run out of memory and `alloc` returns NULL. Fig. 9 shows the input to the theorem prover with the invariant annotations removed and formatting adjusted for the presentation.

Classical setting

The data abstraction predicate is very similar to the simple list of Sect. 6.1:

$$\mathsf{k\text{-}list}\ h\ s\ e\ [] = s = e$$
$$\mathsf{k\text{-}list}\ h\ s\ e\ (p{\cdot}ps) = s = \mathsf{Ptr}\ p \wedge s \neq e \wedge (\exists s'.\ h\ s = \lfloor s' \rfloor \wedge \mathsf{k\text{-}list}\ h\ (\mathsf{Ptr}\ s')\ e\ ps)$$

For the specification of `alloc` we first need to define what this operation does in the abstract. For this, we are not interested in the list structure itself, but just in the set of free memory chunks:

```
void * alloc(word_t size) {
  word_t* prev, curr, tmp;
  word_t i;
  size = size >= 1024 ? size : 1024;
  for (prev = (word_t*) &kfree_list, curr = kfree_list;
       curr;
       prev = curr, curr = (word_t*) *curr) {
    if (!((word_t) curr & (size - 1))) {
      tmp = (word_t*) *curr;
      for (i = 1; tmp && (i < size / 1024); i++) {
        if ((word_t) tmp != ((word_t) curr + 1024*i)) {
          tmp = 0;
          break;
        };
        tmp = (word_t*) *tmp;
      }
      if (tmp) {
        *prev = (word_t) tmp;
        for (i = 0; i < (size / sizeof(word_t)); i++) {
          /** AUXUPD:
               (ptr-safe (´curr +_p ´i) ´d, ptr-tag (´curr +_p ´i) ´d) */
          curr[i] = 0;
        }
        return curr;
      }
    }
  }
  return 0;
}
```

Figure 9. `alloc` after configure and preprocessing.

$$
\begin{aligned}
&\mathsf{free\text{-}set}\ \mathit{hp}\ s\ e\ F = \exists\, ps.\ \mathsf{k\text{-}list}\ \mathit{hp}\ s\ e\ ps \wedge \mathsf{distinct}\ ps \wedge F = \mathit{set}\ ps\\
&\mathsf{alloc}\ p\ l\ F \qquad\ = F - \mathit{chunks}\ p\ (\mathsf{ptr\text{-}val}\ p + (l - \mathsf{KMC}))
\end{aligned}
$$

The function *chunks p q* in the definition above refers to a set of locations starting with pointer p, ending with address q, that consists of adjacent memory chunks only. With this, the pre/post specification of `alloc` is:

$$
\begin{aligned}
&\forall F\ \sigma.\ \{\!|\sigma.\ \mathsf{free\text{-}set}\ (\mathsf{lift}_\tau\ \mathcal{H})\ (\mathsf{ptr\text{-}coerce}\ \mathit{kfree\text{-}list\text{-}addr})\ \mathsf{NULL}\ F \wedge\\
&\qquad\quad \mathit{aligned}\ F \wedge \mathsf{KMC}\ |_u\ \acute{}\mathit{size}|\!\}\\
&\quad \acute{}\mathit{alloc\text{-}ret} :== \mathit{PROC}\ \mathsf{alloc}(\acute{}\mathit{size})\\
&\quad \{\!|(\acute{}\mathit{alloc\text{-}ret} \neq \mathsf{NULL} \longrightarrow\\
&\qquad \mathsf{free\text{-}set}\ (\mathsf{lift}_\tau\ \mathcal{H})\ (\mathsf{ptr\text{-}coerce}\ \mathit{kfree\text{-}list\text{-}addr})\ \mathsf{NULL}\\
&\qquad\ (\mathsf{alloc}\ \acute{}\mathit{alloc\text{-}ret}\ (\mathit{max}\ {}^{\sigma}\mathit{size}\ \mathsf{KMC})\ F)) \wedge\\
&\quad\ (\acute{}\mathit{alloc\text{-}ret} = \mathsf{NULL} \longrightarrow \mathcal{H} = {}^{\sigma}\mathcal{H})|\!\}
\end{aligned}
$$

The precondition requires that *kfree-list-addr* describe some set of free memory chunks F and that all addresses in F as well as the requested *size* be aligned with KMC. The postfix *-addr* refers to the fact that the code takes the address of the global variable *kfree-list*; *kfree-list* itself can be accessed by a heap lookup at *kfree-list-addr*. That means, *kfree-list-addr* is of type *word*32 *ptr ptr*. Since k-list expects a *word*32 *ptr*, we use the function ptr-coerce to cast it (as does the C code). Alignment is expressed using the non-overflowing version of divisibility on finite integers with $x\ |_u\ y = (\mathsf{unat}\ x\ \mathsf{dvd}\ \mathsf{unat}\ y)$ and $\mathit{aligned}\ F = (\forall p{\in}F.\ \mathsf{KMC}\ |_u\ p)$. The postcondition refers to the pre-state σ for *size* and the heap.

The *aligned* predicate could be made part of k-list. We have not done so here, because `free` does not require this condition (although `free` preserves it if the base address is aligned with KMC). It is interesting to note that the code, while it tests the size alignment with an assertion (removed by the preprocessor here), does not check base address alignment.

The postcondition has two cases: either some memory was returned ($\acute{}\mathit{alloc\text{-}ret} \neq \mathsf{NULL}$) or the kernel ran out of memory ($\acute{}\mathit{alloc\text{-}ret} = \mathsf{NULL}$). In the latter case, we claim that nothing changes in the heap, including the global variable *kfree-list*. In the former case, we say that the new set of free memory chunks is the same that you would get by evaluating the abstract alloc.

This specification is not completely satisfying. In the success case, we would ideally like to know that nothing else in the heap changes. This "nothing else" is hard to nail down formally. The set F as used in the specification above is too loose, it would miss the heap changes caused by zeroing out the freshly allocated memory. Separation logic will handle this more naturally below.

The proof of the verification conditions for `alloc` takes about 350 lines of proof script with 130 lines of specific supporting lemmas and a further 1400 lines of lemmas shared between and `alloc` and `free` in classical and separation logic settings.

Separation logic

The separation logic version of the abstraction predicate is the following.

$$
\begin{aligned}
&\mathsf{k\text{-}list}_s\ s\ e\ [\,] &&= \lambda h.\ s = e \wedge \Box\ h\\
&\mathsf{k\text{-}list}_s\ s\ e\ (p{\cdot}ps) &&= \lambda h.\ (s = \mathsf{Ptr}\ p \wedge s \neq e) \wedge\\
& && \qquad (\exists\, s'.\ (\mathsf{block}\ s\ s' \wedge^* \mathsf{k\text{-}list}_s\ s'\ e\ ps)\ h)\\
&\mathsf{block}\ s\ s' &&= \lambda h.\ \mathsf{KMC}\ |_u\ \mathsf{ptr\text{-}val}\ s \wedge\\
& && \qquad (s \hookrightarrow \mathsf{ptr\text{-}val}\ s')\ h \wedge\\
& && \qquad \mathsf{sep\text{-}cut}\ (\mathsf{ptr\text{-}val}\ s)\ \mathsf{KMC}\ h\\
&\mathsf{fset}_s\ s\ e\ F &&= \lambda h.\ \exists\, ps.\ \mathsf{k\text{-}list}_s\ s\ e\ ps\ h \wedge F = \mathit{set}\ ps\\
&\mathsf{free\text{-}set}_s\ s\ e\ F &&= \lambda h.\ \exists\, x.\ (\mathsf{ptr\text{-}coerce}\ s \mapsto \mathsf{ptr\text{-}val}\ x \wedge^* \mathsf{fset}_s\ x\ e\ F)\ h
\end{aligned}
$$

The separation logic specification of `alloc` is then:

$$
\begin{aligned}
&\forall F\ \sigma.\ \{\!|\sigma.\ (\mathsf{free\text{-}set}_s\ \mathit{kfree\text{-}list\text{-}addr}\ \mathsf{NULL}\ F)^{\mathit{sep}} \wedge \mathsf{KMC}\ |_u\ \acute{}\mathit{size}|\!\}\\
&\quad \acute{}\mathit{sep\text{-}alloc\text{-}ret} :== \mathit{PROC}\ \mathit{sep\text{-}alloc}(\acute{}\mathit{size})\\
&\quad \{\!|(\acute{}\mathit{sep\text{-}alloc\text{-}ret} \neq \mathsf{NULL} \longrightarrow\\
&\qquad (\mathsf{free\text{-}set}_s\ \mathit{kfree\text{-}list\text{-}addr}\ \mathsf{NULL}\\
&\qquad\ (\mathsf{alloc}\ \acute{}\mathit{sep\text{-}alloc\text{-}ret}\ (\mathit{max}\ {}^{\sigma}\mathit{size}\ \mathsf{KMC})\ F) \wedge^*\\
&\qquad\ \mathsf{zero}\ \acute{}\mathit{sep\text{-}alloc\text{-}ret}\ (\mathit{max}\ {}^{\sigma}\mathit{size}\ \mathsf{KMC}))^{\mathit{sep}}) \wedge\\
&\quad\ (\acute{}\mathit{sep\text{-}alloc\text{-}ret} = \mathsf{NULL} \longrightarrow\\
&\qquad (\mathsf{free\text{-}set}_s\ \mathit{kfree\text{-}list\text{-}addr}\ \mathsf{NULL}\ F)^{\mathit{sep}})|\!\}
\end{aligned}
$$

The precondition here is simpler than in the classical setting. Since alignment is built into fset_s using block, we only need data abstraction and that the requested size is aligned to KMC.

The postcondition has the same two cases that occur in the classical setting. If we have run out of memory ($\acute{}\mathit{alloc\text{-}ret} = \mathsf{NULL}$), we now only say that F does not change (and, by frame rule, nothing else in the heap changes either).

In the success case ($\acute{}\mathit{alloc\text{-}ret} \neq \mathsf{NULL}$), we still state that the new set of free memory chunks is the same that you would get by evaluating the abstract function alloc. Additionally, we now explicitly say that the memory returned is a separate, contiguous block of the right size, filled with zero.

$$
\begin{aligned}
&\mathsf{zero}\ p\ n &&= \mathsf{zero\text{-}block}\ (\mathsf{ptr\text{-}coerce}\ p)\ (\mathsf{unat}\ (n\ \mathsf{div}\ 4))\\
&\mathsf{zero\text{-}block}\ p\ 0 &&= \Box\\
&\mathsf{zero\text{-}block}\ p\ (\mathit{Suc}\ n) &&= (p +_p \mathsf{of\text{-}nat}\ n) \mapsto 0 \wedge^* \mathsf{zero\text{-}block}\ p\ n
\end{aligned}
$$

This zero-block is the sub-formula that can directly be used by client code—it describes the freshly allocated memory. The postcondition is stronger than the one in the classical case.

The proof of the verification conditions induced by this specification takes about 650 lines of proof script with 750 lines of supporting lemmas (plus the 1400 lines shared with the other parts of the verification). This corresponds to 136 lines of code in the original and 30 lines of code after configure and preprocessing. The effort for the separation logic case was higher than in the classical setting, but not significantly so, and we have acquired a much stronger postcondition that was difficult to state before. In the separation logic case, we see potential for improvement in automation. Much of the additional proof text was verbose, but not hard.

It should be noted that we chose `alloc` because it constitutes the worst possible case for this framework: almost every pointer access is unsafe and needs to be reasoned about. In other frameworks this is impossible or leads to unsoundness if applied naïvely, here it is merely more work than usual. Once done, client code does not

need to go to the same level of detail to use the pre/post conditions provided. The complexity is hidden by the framework.

7. Related Work

The directly relevant work in the literature arises from efforts to verify properties of pointer programs. There are two directions that are taken—Hoare logic oriented verification in which partial/total correctness and completeness is the primary goal, with decidability or the existence of an efficient algorithmic procedure considered less important, and the opposite approach that strives for automation, at the expense of only being able to cope with limited language fragments or properties, the domain of software model checking, shape analysis, and separation logic decision procedures.

The idea to use separate heaps for separate pointer types and structure fields in Hoare logic goes back to Burstall [9]. On the abstract level, our multiple typed heaps formalisation is most closely related to Bornat [8] and Mehta and Nipkow's [19] work in Isabelle, although we exploit Isabelle's type inference in a different way. We ground this abstract and efficient reasoning in a detailed C semantics that is directly applicable to concrete programs. Our use of HOL types is similar to that of Blume's [7] encoding of the C type system in ML that utilises phantom typing to express pointer types and operators for the purpose of a foreign-function interface. The Caduceus tool [12] supports Hoare logic verification of C programs, including the type-safe part of pointer arithmetic at this level. We increase the applicability of program verification drastically by supporting the unsafe part as well. Separation logic [16, 25] has been mechanised in theorem proving systems previously [31, 18]. Again, we provide soundness for program verification by grounding these abstract, idealised models in a concrete semantics. We are able to support abstract separation logic notation and unsafe, low-level pointer manipulations at the same time.

On the concrete level, Norrish [24] presents a very thorough and detailed memory model of C. Our formalisation has similarities to exploratory work on C++ in the VFiasco project [15]. The latter two provide a more precise machine model, while the abstractions of the previous paragraph allow for more convenient and efficient reasoning. Our model provides a unified view of both.

The SLAM [2] and BLAST [14] software model checkers apply predicate abstraction and counter-example driven refinement to check some safety properties of C programs automatically, such as correct API usage in device drivers. This is not as useful when verifying fundamental system abstractions, such as memory allocation, that do not rely on other interfaces that can be factored out or that feature inductively defined data structures with complex invariants. It may be possible to tackle some of the guard proof obligations we encounter such as the absence of null pointer dereferences [5] with this technique, and efforts exist that seek to integrate software model checking with proof-based verification [11]. Cook et al [10] extend the automatic theorem prover in software model checkers to support the pointer operations and bit-level arithmetic described in this paper by translating C to propositional logic. Our translation is substantially different, taking advantage of HOL's expressiveness and type system, intended to better support interactive proof.

Shape analysis [20, 26] and separation logic decision procedures [3] can show some structural invariants, such as the absence of loops in linked lists. At this point in time they tend to be specialised for limited language fragments or data structures that either do not meet our needs or do not provide sufficient benefit to offset the implementation effort, but there are promising developments that may improve this situation [1]. The Hob [35] framework takes the refinement view of verification and allows some of these analyses to be used on independent modules or at higher-levels of abstraction, where when applied directly they may fail.

Type-safe C variants like CCured [21] 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. These variants have not gained popularity in real-world systems implementations, due to a combination of runtime overheads and the restrictions imposed.

Our longer-term goal is a verified kernel and earlier work on theorem proving based OS verification includes PSOS [22] and UCLA Secure Unix [30]. A lack of mature mechanised theorem proving technology meant that while designs could be formalised, full implementation proofs were not achieved. Later, KIT [4], part of the CLI stack, 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. The VeriSoft project [13] 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. Most closely related to our case study is the successful verification of the kernel memory allocator from the teaching-based Topsy operating system by Marti et al [18] in Coq. The major difference is the heavy use of pointer arithmetic and casting in L4's memory allocator that we are able to handle confidently and conveniently due to our more detailed semantic model and type encoding. Marti et al model C memory as $nat \Rightarrow int$ and translate the code manually into their tool. Although our model is more detailed, our proofs are significantly shorter.

8. Future Work and Conclusion

Apart from supporting more language features which is underway, we see the main directions for future work as automation and integration. Combining automatic techniques such as decision procedures for separation logic and bit vector arithmetic with interactive proof, as exemplified by Hob [35], should lead to improved productivity and reusability without sacrificing the range of programs or properties we consider.

We have presented a unified, formal framework for the verification of real C programs in an interactive proof assistant without oversimplifications or strong restrictions on the language. Our work is a novel combination of concepts that so far have only been well understood in isolation: Hoare logic verification, separation logic, interactive theorem proving, detailed dynamic and memory models for the C programming language, and low-level pointer operations.

We have demonstrated with two case studies that reasoning on both the abstract and the detailed level is well supported. Our results indicate that while separation logic might not provide the amount of direct automation that we are used to, this difference becomes smaller for larger verification examples.

By grounding separation logic in our memory model, we make verification of systems level code practical: proof obligations relating to issues such as inter-type aliasing disappear when the code to be verified is working within type-safe bounds. When the source code is not so well-behaved, the framework allows verifiers to drop to the lower level of multi-byte values and their encodings. This contrasts with methodologies that would reject such low-level behaviour as ill-formed, making verification impossible.

By importing C code directly into our verification environment, and by tying our logical annotations to the C source, we keep a close connection between the verification and development activities. In particular, we aim to avoid the scenario where the developing code loses its connection with the verification. There is no point in verifying code that is not going to be deployed. We hope our still-nascent methodology, which has verifiers and developers touching the same source files, will make the timely production of verified software possible.

Acknowledgements We thank Kai Engelhardt, Carroll Morgan, Manuel Chakravarty, and Rob van Glabbeek for discussions and for reading drafts of this paper. We are also grateful to David Tsai who started our work on the memory allocator.

References

[1] Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *13th International Symposium on Static Analysis (SAS 2006)*, volume 4134 of *Lecture Notes in Computer Science*, pages 182–203. Springer, 2006.

[2] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN'01, Workshop on Model Checking of Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122, 2001.

[3] J. Berdine, C. Calcagno, and P. O'Hearn. A decidable fragment of separation logic. In *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science, 24th International Conference, Chennai, India, December 16–18, 2004*, 2004.

[4] W. R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.

[5] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. Checking memory safety with Blast. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 3442 of *Lecture Notes in Computer Science*, pages 2–18. Springer, 2005.

[6] J. Bloch. Nearly all binary searches and mergesorts are broken. `http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html`, June 2006.

[7] M. Blume. No-longer-foreign: Teaching an ML compiler to speak C "natively". *Electronic Notes in Theoretical Computer Science*, 59(1), 2001.

[8] 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.

[9] 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.

[10] B. Cook, D. Kroening, and N. Sharygina. Cogent: Accurate theorem proving for program verification. In K. Etessami and S. K. Rajamani, editors, *Proceedings of CAV 2005*, volume 3576 of *Lecture Notes in Computer Science*, pages 296–300. Springer Verlag, 2005.

[11] M. Daum, S. Maus, N. Schirmer, and M. N. Seghir. Integration of a software model checker into Isabelle. In G. Sutcliffe and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning: 12th International Conference, LPAR 2005*, volume 3835 of *Lecture Notes in Artificial Intelligence*, pages 381–395, Montego Bay, Jamaica, October 2005. Springer.

[12] 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.

[13] 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'05)*, pages 1–16, Oxford, UK, 2005.

[14] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with Blast. In *SPIN'03, Workshop on Model Checking Software*, 2003.

[15] 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.

[16] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 14–26, New York, NY, USA, 2001. ACM Press.

[17] L4Ka Team. *L4* eXperimental *Kernel Reference Manual Version X.2*. University of Karlsruhe, Oct. 2001. `http://l4ka.org/projects/version4/l4-x2.pdf`.

[18] N. Marti, R. Affeldt, and A. Yonezawa. Verification of the heap manager of an operating system using separation logic. In *Third workshop on Semantics, Program Analysis, and Computing Environments For Memory Management (SPACE 2006)*, pages 61–72, Jan. 2006.

[19] F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 2005. To appear.

[20] A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '01*, June 2001.

[21] 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.

[22] 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.

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

[24] M. Norrish. *C formalised in HOL*. PhD thesis, Computer Laboratory, University of Cambridge, 1998.

[25] 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.

[26] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 105–118. ACM Press, 1999.

[27] N. Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.

[28] H. Tuch and G. Klein. A unified memory model for pointers. In G. Sutcliffe and A. Voronkov, editors, *12th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR-12)*, volume 3835 of *LNCS*, pages 474–488, 2005.

[29] H. Tuch, G. Klein, and M. Norrish. Verification of the L4 kernel memory allocator. Formal proof document. `http://www.ertos.nicta.com.au/research/l4.verified/kmalloc.pml`, July 2006.

[30] B. Walker, R. Kemmerer, and G. Popek. Specification and verification of the UCLA Unix security kernel. *CACM*, 23(2):118–131, 1980.

[31] T. Weber. Towards mechanized program verification with separation logic. In J. Marcinkowski and A. Tarlecki, editors, *Computer Science Logic – 18th International Workshop, CSL 2004*, volume 3210 of *Lecture Notes in Computer Science*, pages 250–264. Springer, 2004.

[32] M. Wenzel. Type classes and overloading in higher-order logic. In E. L. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics'97*, volume 1275 of *LNCS*, pages 307–322. Springer, 1997.

[33] M. Wildmoser and T. Nipkow. Certifying machine code safety: Shallow versus deep embedding. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics 2004*, volume 3223 of *LNCS*, pages 305–320. Springer, 2004.

[34] H. Yang and P. W. O'Hearn. A semantic basis for local reasoning. In *Foundations of Software Science and Computation Structure*, pages 402–416, 2002.

[35] K. Zee, P. Lam, V. Kuncak, and M. Rinard. Combining theorem proving with static analysis for data structure consistency. In *Int. Workshop on Software Verification and Validation*, 2004.