EASST

Proceedings of the
Ninth International Workshop on
Automated Verification of Critical Systems
(AVOCS 2009)

High-level Proofs about Low-level Programs

Holger Gast and Julia Trieflinger

15 pages

High-level Proofs about Low-level Programs

Holger Gast and Julia Trieflinger

Wilhelm-Schickard-Institut für Informatik
Eberhard Karls Universität Tübingen, Tübingen, Germany
gast@informatik.uni-tuebingen.de
trieflin@informatik.uni-tuebingen.de
http://www-pu.informatik.uni-tuebingen.de/users/gast/
http://www-pu.informatik.uni-tuebingen.de/users/trieflin/

Abstract: Functional verification of low-level code requires abstractions over the memory model to be effective, since the number of side-conditions induced by byte-addressed memory is prohibitive even with modern automated reasoners. We propose a flexible solution to this challenge: assertions contain explicit memory layouts which carry the necessary side-conditions as invariants. The memory-related proof obligations arising during verification can then be solved using specialized automatic proof procedures. The remaining verification conditions about the content of data structures directly reflect a developer's understanding. The development is formalized in Isabelle/HOL.

Keywords: verification of C code, pointer programs, precise memory models

1 Introduction

The functional verification of low-level C code has recently attracted much attention (e.g. [Tuc08b, TKN07b, CMST09, RH09]). The central challenge in these applications consists in proving the disjointness of memory objects in the C memory model. Unlike in strongly typed languages like Java or C#, the inequality of pointers in C does not imply the disjointness of the memory regions occupied by the referenced objects: pointer arithmetic, pointer casts, and internal pointers to struct fields allow almost arbitrary overlaps. The strategy proposed in the literature is to maintain a typed view on the untyped memory: Tuch et al. [Tuc08b, TKN07b] employ a variant of separation logic; Cohen et al. [CMST09] maintain a set of disjoint objects in a ghost variable; Rakamarić et al. [RH09] use a static analysis to identify parts of C programs that obey the split-heap model [Bur72]. The actual verification is then performed on the typed view.

The disjointness of regions is, however, only one aspect of reasoning about low-level programs. Another aspect concerns the invariants and side-conditions associated with data structures. For instance, allocated blocks in C are always contiguous in memory, i.e. overflow in the pointer arithmetic inside them is excluded. This property, in turn, enables pointers into an array to be compared by the less than operator. While it is always possible to augment assertions with suitable side-conditions, these need to be handled explicitly although they are, in fact, invariants that continue to hold through all operations. Reasoning about less precise memory models is more efficient partly because the invariants are implicit. We show that it is possible to associate the invariants with descriptions of low-level memory layouts instead.

The following function, which initializes raw memory, demonstrates the point.

```
void init(char *p, char *q) {
   char *r = p;
   while (r != q)
     *r++ = 0;
}
```

For functional verification, the user would like to give the following natural loop invariant. It asserts that r runs between p and q and that the bytes from p to r have already been initialized (*a denotes reading from address a):

$$\mathsf{p} \le \mathsf{r} \wedge \mathsf{r} \le \mathsf{q} \wedge \; (\forall \mathsf{a}.\; \mathsf{p} \le \mathsf{a} \wedge \mathsf{a} < \mathsf{r} \longrightarrow (\mathsf{*a}) = 0)$$

The crucial point to observe is that this invariant requires reasoning about pointer inequalities. To establish the invariant, $\mathsf{p} \le \mathsf{q}$ has to be given as a precondition of the init function. However, a programmer would never initialize a memory region without this property, so that the precondition appears as a merely technical necessity.

Building on the lightweight separation method [Gas08], we propose, instead, to make memory layouts explicit in assertions and to associate side-conditions with them. In the example, the layout will contain a ptr-block p q. It describes the memory region between addresses p and q and also includes the side-condition $\mathsf{p} \le \mathsf{q}$. Furthermore, the proposed approach simplifies proofs about layouts [Gas09] since fewer side-conditions arise (see Section 3).

The purpose of this paper is to show how side-conditions and invariants can be maintained implicitly with the memory layout, and that handling them implicitly leads to natural proof obligations. In low-level programs it then becomes straightforward to switch between typed and untyped views. Furthermore, we show that the reasoning also covers composite objects such as structs, arrays, and linked lists. The current paper thus extends the earlier work [Gas08, Gas09] to byte-addressed memory and machine-level representations of values.

The work presented is carried out in Isabelle/HOL to ensure soundness. The proofs are therefore partly interactive. However, we structure them to enable a direct comparison with automated approaches: for each algorithm, we first prove auxiliary theorems, which would become axioms in other methods. The verification itself is then essentially automatic (cf. Section 2).

Organization Section 2 gives an overview over lightweight separation. Section 3 describes the maintenance of invariants associated with memory layouts. Section 4 applies the resulting framework to two examples. Section 5 discusses related work. Section 6 concludes.

Isabelle Notation The notation of Isabelle/HOL mostly uses standard mathematical conventions. A few exceptions need to be mentioned. Functions, as usual in higher-order logic, are curried. The function type is denoted by $\Rightarrow$. Application of function f to argument a is written by juxtaposition $f\ a$. Definitions of constants are written by $\equiv$. An interval $[a,b)$ over an ordered domain is denoted by $\{a..<b\}$. For readability, we render theorems, which in Isabelle are expressed by meta-level implication $\Longrightarrow$, as inference rules.

2   An Overview of Lightweight Separation

Assertions in separation logic [ORY01] capture both the memory content and the memory layout, and thus the disjointness of memory regions, at the same time. Although fragments can be used for automatic verification of shape-like properties [BCC$^+$07], reasoning in separation logic in general requires extensive user interaction [App06, Tuc08b, McC09], except when algorithms are closely tied to the shape of data structures [Tue09]. Lightweight separation [Gas08] alleviates this problem by splitting the layout and the content in assertions. The layout is then used to derive the disjointness of memory regions automatically. This section gives an overview of lightweight separation. For more details, the reader is referred to [Gas08].

2.1   Language and Hoare Logic

We consider a low-level, C-like language with side-effecting expressions, an unrestricted address operator, pointer arithmetic, and pointer casts. Its operational semantics is inspired by [Nor98]. Compared with the detailed model of C given there, we make four main simplifications: expressions are evaluated left-to-right; we neglect alignment and padding; we do not support unstructured control flow (`break`, `continue`, `goto`, `switch`); finally, we do not differentiate between allocated and initialized memory [Nor98, §3.1.2].

We use a linear, byte-addressed memory model. Addresses `addr` are isomorphic to 32-bit words [Daw07]. For syntactical reasons, we define `null` as the address 0 and also use that constant in the language syntax. Memory is a partial function from allocated addresses to bytes. It is defined by the following Isabelle/HOL record:

```
record memory =
  m-dom :: "addr set"
  m-cnt  :: "addr ⇒ byte"
  m-valid :: bool
```

The domain and content are given in the fields `m_dom` and `m_cnt`, respectively. The flag `m_valid` is a history variable, which records illegal accesses to unallocated memory but does not influence the semantics.

A static context Γ contains the definitions of types, structs, local variables, and functions. Since local variables are memory-allocated, the context only gives their addresses, while their content is stored in the memory defined above.

The Hoare logic captures fault-avoiding partial correctness. In particular, the generated proof obligations ensure that all accesses are legal and that the memory remains valid throughout the execution. Assertions are predicates on static contexts and memory states. Side-effecting expressions are handled by Kowaltowski's approach [Kow77], in which the postconditions in Hoare-triples for expressions are assertions about the post-state and the result value. For the control structures and function calls, we follow Schirmer's formalization [Sch05].

The Hoare logic uses forward-style reasoning to emulate the proofs in separation logic: from a given pre-condition, the post-condition is computed as follows. The Hoare rules introduce into the post-condition *inverse operators*. Consider for illustration an assignment x = e, where x is a variable and e is side-effect-free. Suppose that the pre-condition is P. In a HOL formalization, P is a predicate on the context Γ and the memory state M. The generated post-condition then is as

follows (where rd-var Γ x M fetches the value stored in variable x):

$$\lambda\,\Gamma\,M.\,\exists M'.\,P\,\Gamma\,(\text{STORE-VAR}\,\Gamma\,\text{x}\,\text{M'}\,\text{M})\,\wedge\,\text{rd-var}\,\Gamma\,\text{x}\,M = e \tag{1}$$

The inverse operator STORE-VAR modifies the current state M after the assignment by replacing the region occupied by variable x with the content of the existentially quantified pre-state M'. For a general assignment e = e', where both e and e' may have side-effects, several inverse operators may be introduced. Furthermore, any access to memory generates a proof obligation that the accessed region is allocated. In summary, our rules generalize Floyd's assignment axiom $\{\,P\,\}\,x = e\,\{\,\exists x'.\,P[x'/x]\wedge x = e\,\}$, which handles only the variable case.

Note that although the formalization (1) uses higher-order logic, the resulting post-conditions are first-order if the preconditions are first-order: the β-redex in the post-condition only denotes a replacement that could also be carried out by a first-order verification condition generator.

2.2 Reasoning about Disjointness

An assertion containing inverse operators is not useful, because it refers to some existentially quantified, previous state rather than the current state. Suppose, for example, that pre-condition P in (1) contains an assertion rd-var Γ y M $= Y$, where M refers to the pre-state of the execution. The generated post-condition then contains rd-var Γ y (STORE-VAR Γ x M' M) $= Y$, where now M refers to the post-state and M' is existentially quantified. The central concern is to prove the memory region accessed by rd-var Γ y M disjoint from the region modified by STORE-VAR Γ x M' M. In that case, we can simplify the assertion to rd-var Γ y M $= Y$, which refers to the current state M, the post-state of the execution, alone.

Towards that end, lightweight separation captures memory layouts by *covers*. A cover A is a predicate on address sets, i.e. it describes memory regions. All covers used subsequently will be *well-formed* covers, which accept a single memory region and can therefore be used interchangeably with that region. For example, var-block Γ x is the region occupied by variable x. The disjointness operator A ∥ B accepts the union of the regions given by covers A and B, but only if these regions are disjoint. The assertion $M \blacktriangleright A$ states that A *covers* the domain, i.e. the allocated addresses, of memory M. The weaker variant $M \vartriangleright A$ asserts that region A is allocated in M.

With these definitions, memory layouts can be specified. For instance, a memory containing only the two disjoint variables x and y is captured by

M ▶ var-block Γ x ∥ var-block Γ y

From this assertion, the automated tactics from [Gas08] can prove the disjointness of the regions occupied by x and y, and the inverse operator in the running example can be removed.

The approach scales directly to inductively defined data-structures, arrays with dynamic size, user-defined memory layouts for special data structures, and predicates and functions accessing part of the memory. New cover constants are introduced by the command declare-cover. New memory-accessing constants are specified by declare-accessor. For each accessor, the user has to specify, in the form of a cover, the memory region that the predicate depends on. This, of course

generates a proof obligation: the accessor may not read any memory outside the specified region. A tactic `accesses` provided by the verification environment discharges this proof obligation automatically in almost all cases.

Nested structures in memory can be *unfolded* automatically [Gas09] to prove memory regions disjoint. In this process, covers in the memory layout are replaced by more detailed descriptions of the same memory region. For instance, to prove a single array element disjoint from a field in a struct, the array is split into slices to exhibit the sought element; the struct, likewise, is replaced by the disjoint union of its fields. The required disjointness is then obvious and can be used as before.

The application of Hoare rules and the removal of inverse operators is implemented in a tactic `step`. Using the above reasoning, it thus executes the program symbolically and computes a readable post-condition for the given pre-condition. It stops after each statement to enable the user to inspect the result and to compare it with their intuition.

In summary, lightweight separation allows the layout and content of memory to be handled independently and automatically: users can state assertions about the content in classical higher-order logic. Contrary to the case of the substructural separation logic, properties about these assertions can be proven using the automated reasoners available in Isabelle/HOL. The only new obligations are to specify the memory layout and to characterize the accessed region for newly introduced functions reading from memory. The commands `declare-cover` and `declare-accessor` automate this process in most cases.

3   Invariants of Covers

The language treated in the earlier presentation [Gas08] is based on a less precise memory model that uses natural numbers for addresses and stored values. The purpose of the current paper is to show that the automation available there carries over to the precise byte-addressed memory model from Section 2.1. The main challenge is the exclusion of overflows in the address arithmetic. This section addresses the challenge by associating side-conditions with covers.

3.1   Implicit Invariants

The necessity for associating invariants with covers is best illustrated by the example of arrays. In the old memory model with infinite address space, an array slice with base p between indices i and j simply covers the addresses between its start and end, as computed by pointer arithmetic (where $\oplus_{\Gamma,t}$ denotes C-style pointer offset by a multiple of the size of type t).

$$\text{array } \Gamma \, t \, p \, i \, j \equiv \lambda S. \ \ S = \{ \, p \oplus_{\Gamma,t} i \, ..< p \oplus_{\Gamma,t} j \, \} \ \wedge \ i \leq j \ \wedge \ t \neq \text{void}$$

The following unfolding rule, which is the basis for automated reasoning about disjointness within arrays, then splits the array slice at an intermediate index j.

$$\frac{i \leq j \quad j \leq k}{\text{array } \Gamma \, t \, a \, i \, k = \text{array } \Gamma \, t \, a \, i \, j \ \| \ \text{array } \Gamma \, t \, a \, j \, k} \tag{2}$$

This natural and straightforward equation does not carry over directly to the more precise memory model considered here: if the pointer arithmetic overflows, the left-hand-side may become empty, while the right-hand-side consists of one empty and one non-empty memory region.

The solution is to introduce side-conditions that prevent these overflows into the definition of array above. Section 3.3 will define arrays in the new model along with other structured types. As a result, rule (2) can, again, be proven and used for unfolding.

Re-interpretations are unfoldings that do not split the covered memory region itself, but only change the cover constant describing the region. They often replace a cover with strong invariants by a cover with weaker invariants, as is the the case when converting from a typed to an untyped view. In these cases, the unfolding rules can take the form [Gas09]:

$$\frac{p_1 \cdots p_n}{\text{is-valid}\, A} \longrightarrow A = B$$

The predicate is-valid asserts that A accepts at least one address set, which entails that the side-conditions associated with A hold and can therefore be used in the proof of the equality.

3.2 Memory Blocks

For low-level programs, it is sometimes necessary to reason about the raw, byte-addressed memory. At this level, we work with the word representation of addresses directly. A memory region is therefore given by its start address a and its length n as a machine word:

block a n $\equiv$ (λS. S = {a .. < a $\oplus$ n} $\wedge$ a $\leq$ a $\oplus$ n)

The block cover thus accepts an address set starting from a up to a $\oplus$ n, exclusively. Herein, $\oplus$ denotes address offset in two's complement arithmetic. Overflows in the addition are excluded by the side-condition a $\leq$ a $\oplus$ n. Note that for uniformity, empty blocks with n=0 are included in the definition. Furthermore, from a $\neq$ null we can conclude that all pointers p in the block are non-null, which is important for reasoning about internal pointers to structs. Note also that it would be possible to include an additional side-condition a $\neq$ null into the definition of block to reflect the C-standard's guarantee that the address null will never be allocated.

Equivalently, a block can be delimited by its start- and end addresses, which is useful for reasoning about access by pointer arithmetic:

ptr-block p q $\equiv$ (λS. S = { p .. < q } $\wedge$ p $\leq$ q)

These definitions solve the example from Section 1: the function's precondition includes a conjunct M ▶ ptr-block p q $\|\cdots$, from which we can deduce is-valid(ptr-block p q), hence p $\leq$ q.

For high-level reasoning, we need to express assertions about typed objects. Since types in C mainly serve to determine the size of memory objects and values, we wish to capture a typed object by an untyped block where the length is the type's size. However, the sizes of types are computed as natural numbers to avoid overflows (see also [Tuc08b]). In converting the type size to a machine word expected by block (via of-nat), a large type might map to a small block size. The is-small-type predicate therefore checks that the type's size can be represented as a machine word. The definition of a typed block is then:

typed-block Γ a t $\equiv$ (λS. block a (of-nat (sz-of-ty Γ t)) S $\wedge$ is-small-type Γ t)

3.3 Structured Types

The handling of structured types in low-level settings is non-trivial [Tuc08b], because the memory model itself does not exclude aliasing: a pointer to `int` may point to a field in a struct, which in turn may be allocated as a local variable. We now show how these challenges can be met in lightweight separation, while maintaining the natural invariants associated with the types' structure.

Structs The basic handling of structs is straightforward: a memory object of struct type *S* is captured directly by a typed-block Γ p S. To expose the fields contained in a struct, we introduce a constant field-block that describes a single field (where field-off and field-ty look up the offset and type, respectively, of the field in the struct's definition in Γ):

field-block Γ p t f $\equiv$ typed-block Γ (p $\oplus$ (of-nat (field-off Γ t f))) (field-ty Γ t f)

Our verification environment allows the user to define new struct types in C syntax. It generates an accessor function for each field. Furthermore, the environment proves an unfolding theorem of the following form, which allows to prove disjointness of the fields.

typed-block Γ p (struct S) = field-block Γ p (struct S) f$_1$ $\|$ field-block Γ p (struct S) f$_2$ $\|$ $\cdots$

Note that a general unfolding theorem cannot be formulated: while the side-conditions associated with the block for the entire struct imply those for the fields, the sum of the field sizes may not be representable as a machine word. The theorem can, however, be proven automatically if the struct size is known, as is the case for specific struct definitions.

Arrays Arrays, like in C, are defined by pointer arithmetic. The memory region occupied by an array slice with indices $[i,j)$ can therefore be expressed directly using ptr-blocks, as in the first line of the following definition. The entire definition must include, however, further invariants:

$$
\begin{aligned}
\text{array } \Gamma \text{ t p i j} \equiv\ & \lambda\text{S. ptr-block (p } \oplus_{\Gamma,t} \text{ i) (p } \oplus_{\Gamma,t} \text{ j) S } \wedge \\
& 0 \leq_s \text{i } \wedge \text{ i } \leq_s \text{ j } \wedge \\
& \text{unat j * (sz-of-ty } \Gamma \text{ t)} \leq \text{unat max-word } \wedge \\
& \text{t } \neq \text{ TVoid } \wedge \text{ wf-ty } \Gamma \text{ t}
\end{aligned}
\tag{3}
$$

The central goal is to reduce reasoning about arrays to reasoning about the index ranges that the verified program manipulates. The invariants associated with ptr-blocks are, however, too weak for this purpose, since overflows in the address arithmetic would lead to the wrong memory region. As in related work (e.g. [BPS09]), we consider arrays with non-negative indices (line 2). For these, line 3 excludes overflows in the address arithmetic. Finally, the last line adds two requirements of the C standard: the element type t is not `void` and it is well-formed, i.e. its size can be computed.

With this definition, the natural unfolding rule (2), which was used in the less precise model with an unbounded address space, continues to hold for the low-level memory model. Only the premises of the rule are now signed word comparisons. Accordingly, the automated reasoning about disjointness of array slices and array elements works as expected.

When accessing arrays by pointer arithmetic, array elements must be re-interpreted as typed blocks, which is achieved by the following theorem:

$$
\text{is-valid(array } \Gamma \text{ t a j (j+1))} \longrightarrow \text{array } \Gamma \text{ t a j (j+1)} = \text{typed-block } \Gamma \text{ (a } \oplus_{\Gamma,t} \text{ j) t}
\tag{4}
$$

The treatment of invariants in this rule highlights our approach: while the side-conditions of the array imply those of the typed block, the other direction does not hold directly, because the conjuncts in line 2–4 of (3) cannot be proven. The additional validity assumption provides the necessary information. As a result, the unfolding applies only in a context where the pointer arithmetic takes place inside an array, which is just the requirement stated in the C standard.

4 Applications

Sections 2 and 3 have introduced a verification environment for low-level programs written in a C-like language. Its foremost feature is the explicit formalization of memory layouts and a flexible mechanism for automated reasoning about the disjointness of memory regions. This section applies the environment to two examples that demonstrate the interaction of typed and untyped views on the memory and the handling of low-level data structures.

For readability, we have simplified slightly the notation of assertions, since the actual statements need to respect the restrictions of the Isabelle parser. In particular, variables in layouts denote their variable blocks and «p : t» is a typed block. In both cases, the context Γ is implicit.

4.1 Struct-Copy

The first example that we consider is copying structs by copying their byte representations. The annotated function `copy_point` is shown in Figure 1. It receives pointers to structs `src` and `dst`, which are defined by `struct point { int x; int y; }`. In the loop, these base pointers are cast into `char`-pointers, the objects are then accessed as `char`-arrays. The precondition asserts that the referenced structs are allocated and captures the (typed) initial values in auxiliary variables SRC, DST, X, and Y. The postcondition asserts that the typed view on `src` remains unchanged and that the typed content `dst` is same as that of `src`.

The purpose of the example is to demonstrate how typed and untyped views onto memory can be handled automatically at the same time. The challenge manifests itself in the loop invariant in Figure 1. The invariant describes the memory layout, with the two structs and the local variables `src`, `dst`, and `i`. Furthermore, it limits `i` to index a byte inside the struct and asserts that the fields in `src`-struct, as well as the parameters, are unchanged. The final conjunct concerns the low-level access: the bytes between indices 0 and `i` have been already copied from `src` to `dst`.

Figure 2 shows the different levels of reasoning involved in proving the maintenance of the loop invariant. At the top, the typed views on the structs `src` and `dst` each consist of two 4-byte fields. At the bottom, the same memory regions are re-interpreted as byte arrays of size 8. The shown index `i` is the current position of the copying process. The depicted `j` ranges over the copied indices up to `i`. The shading indicates memory regions about which the invariant makes an assertion: the partial result of the copying process and the typed view of `src`.

Consider now the assignment in the loop body, which in Figure 2 is depicted by the dashed arrow. To show that the loop invariant is maintained, we have to prove that writing to element `i`, indicated by the cross, does not affect the assertions about the already copied region in the untyped view, as well as the assertion about `src->x` and `src->y` in the typed view. This is shown automatically by proving the respective memory regions disjoint as follows.

pre: M ► «src : struct point» ‖ «dst : struct point» ∧ point-known Γ ∧
    src = SRC ∧ dst = DST ∧ src → x = X ∧ src → y = Y

post: M ► «SRC : struct point» ‖ «DST : struct point» ∧
    SRC → x = X ∧ SRC → y = Y ∧ DST → x = X ∧ DST → y = Y

```
void copy_point(struct point *src, struct point *dst) {
  int i;
  i = 0;
  [inv M ► «src : struct point» ‖ «dst : struct point» ‖ src ‖ dst ‖ i ∧
    point-known Γ ∧ 0 ≤ₛ i ∧ i ≤ₛ sz-of-ty Γ (struct point) ∧
    src = SRC ∧ dst = DST ∧ «src → x» = X ∧ «src → y» = Y ∧
    (∀ j. 0 ≤ₛ j ∧ j <ₛ i ⟶ rd Γ (dst ⊕_{Γ,char} j) char M = rd Γ (src ⊕_{Γ,char} j) char M)
  ]
  while (i < sizeof(struct point)) {
    * ((char *)dst + i) = * ((char *)src + i);
    i++;
  }
}
```
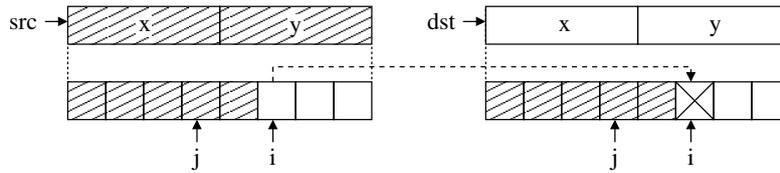
Figure 1: Byte-wise copy of structs

Figure 2: Memory layouts in struct copy

The first task is to locate the written array element i in the memory layout. Towards that end, the automatic tactics re-interpret the typed view given in the loop invariant as an array of bytes using theorem (5).

$$\frac{\text{point-known } \Gamma}{\text{typed-block } \Gamma \, p \, (\text{struct point}) = \text{array } \Gamma \, \text{char } p \, 0 \, 8} \tag{5}$$

The tactics then identify element i by applying (2) and (4). The element j is located in the same manner. Since both elements reside in the same array, the tactics then split that array, using (2) and the upper bound on j, to prove them disjoint.

After element i has been located, the remaining proof obligations are straightforward: the fields src→x and src→y are not contained in dst, so they are obviously disjoint from element i. The same holds for element j in the untyped view of src. Since local variables are memory-allocated, incrementing i in the next statement again generates similar proof conditions, which are solved in the same way as those above.

It remains to derive the postcondition from the invariant and the negated loop test. The following theorem provides the key argument:

$$\frac{M \rhd \text{typed-block } \Gamma \, a \, t}{\text{rd } \Gamma \, a \, t \, M = \text{rd-bytes } \Gamma \, a \, (\text{sz-of-ty } \Gamma \, t) \, M}$$
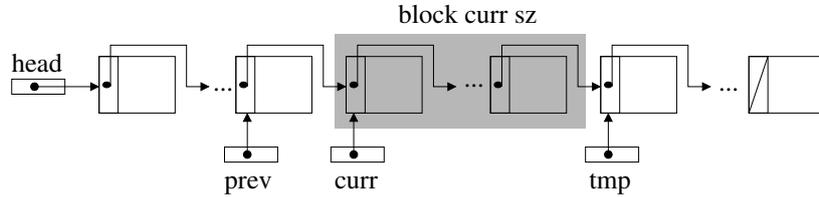
Figure 3: Allocator list structure

This theorem replaces the typed view on a memory region by its byte-representation obtained by function rd-bytes. It is then sufficient to unfold the definitions of the constants that access the fields x and y to show that the representations are equal. In conclusion, this example demonstrates how our approach enables effective, high-level reasoning about low-level memory content.

4.2 Low-level List Structure

As our second example, we use the `kalloc` routine of the L4 microkernel as verified by Tuch [Tuc08a, Chapter 6]. Here, we focus on the innermost loop which contains the essence of the algorithm and whose invariant subsumes that of the outer loop.

Figure 3 illustrates the idea of the algorithm. The data structure of the allocator consists of a singly-linked list of 1 kb *chunks*. The first 4 bytes of each chunk contain the pointer to the next chunk. The elements of the list are ordered by their start addresses. The variable `head` stores the pointer to the first list element. During allocation, the algorithm's outer loop [Tuc08a, Chapter 6] scans the list with a pointer `curr`. A pointer `prev` points to the preceding chunk, or initially to the variable `head`. At each element, the inner loop then tries to concatenate chunks starting at `curr` that are contiguous in memory, until the desired size has been obtained. The pointer `tmp` in this process points to the end of the already found block of size $sz \leq size$. If insufficient contiguous space is available at `curr`, the inner loop stops and the outer loop continues its search.

The algorithm's code in Figure 4 is taken from [Tuc08a]. The variable `kfree_list` herein points to the variable `head` in Figure 3. The code deviates from the original in three minor ways: first, we rewrite the `for`-loop and `break` control structures because our language does not support them as yet. Second, to avoid reasoning about divisibility of words, we replace Tuch's counter `i` of the number of found chunks by their accumulated size `sz`. Third, our language does not support unsigned integers yet, so we work with signed `int` values throughout.

For the verification, we capture the data structure from Figure 3 by a few definitions. The cover (Section 2.2) ds-cover Γ p q M ("ds" for "data structure") contains the set of addresses occupied by all chunks in the linked list starting at p and ending at q. It enumerates that address set inductively as follows (where rd Γ p t M reads a value of type t from address p):

$$\frac{p = q \qquad S = \{\}}{\text{ds-cover } \Gamma \text{ p q M S}} \qquad \frac{p \neq q \quad p \neq \text{null} \quad (\text{block p } 1024 \parallel \text{ds-cover } \Gamma \text{ (rd } \Gamma \text{ p (void*) M) q M) S}}{\text{ds-cover } \Gamma \text{ p q M S}}$$

If p = q, then the set is empty. Otherwise, the set contains a chunk and the remainder of the data structure is obtained by reading the next pointer from the beginning of p. The side-condition

```
int sz = 1024;
tmp = *(void **)curr;
[inv M  ▶  kfree-list  ‖  size  ‖  sz  ‖  prev  ‖  curr  ‖  tmp
        ‖  «kfree-list:void*»  ‖  ds-cover Γ (*kfree-list) null M  ∧
   ds-inv Γ kfree-list M  ∧  0 ≤ₛ size  ∧
   prev ≠ null  ∧  curr ≠ null  ∧  *prev = curr  ∧
   in-list Γ curr (*kfree-list) null M  ∧
   (prev = kfree-list  ∨  in-list Γ prev (*kfree-list) null M)  ∧
   M ▷ «prev:void*»  ‖  tmp  ‖  sz  ∧
   (tmp ≠ null  ⟶  in-list Γ tmp curr null M  ∧
                  ds-cover Γ curr tmp M = block curr sz)
]
while (tmp != null && sz < size) {
    if (tmp != (curr + sz)) {
        tmp = null;
    } else {
        tmp = *(void**)tmp;
        sz = sz + 1024;
    }
}
```

Figure 4: Inner loop of `kalloc`

p≠null ensures that null can never be allocated, as requested by the C standard. Since the algorithm works with pointers into the chunk list, we define in the same way a predicate in-list Γ a p q M which checks that chunk a is reachable from p without going beyond q. Finally, the predicate ds-inv Γ kfree-list M assures that the chunks in the list are sorted by their start addresses.

With these predicates, the loop invariant in Figure 4 captures the situation from Figure 3 directly. The first conjunct describes the memory layout. It contains the local variables, the `head` variable (Figure 3) pointed to by kfree-list, and the chunks of the data-structure itself. The next three lines concern the pointers established by the outer loop: both prev and curr are not null, the pointer referenced by prev points to curr, and curr points into the chunk list. The pointer prev, on the other hand, either points to the `head` variable or into the chunk list.

The next line demonstrates the expressiveness of lightweight separation: since we do not know where exactly prev points to, it is not straightforward to show that * prev = curr is maintained despite the assignments in the loop body. Usually, a case-distinction would be required. Here, we can simply introduce a new view on the memory's layout which contains just the necessary disjointness assertions.

The last conjunct captures concisely the already found contiguous block between curr and tmp, which is shaded in Figure 3: if the search has not been interrupted by setting tmp=null, then the sequence of chunks is equivalent to a single contiguous memory block of size sz.

With this setup, the invariant is established initially from the invariant of the outer loop, which contains all but the last conjunct. The maintenance of the loop invariant is also straightforward: by only using tactic `step`, we reach the end of the loop body. Since `step` has proven that all but the last conjunct of the invariant are not affected by the loop body, they are maintained trivially. The last conjunct depends on the if-branch taken in the body. If tmp has been set to null to stop the search, it is satisfied immediately. In the other case, we first have to show that tmp is still in

the list, which is obvious by the definition of in-list. Finally, we need to join a new chunk into the already found block, which is the crucial proof obligation of the algorithm. It shows how lightweight separation can be used for manual proofs about memory layouts. We have to prove

$$\text{ds-cover } \Gamma \text{ curr (rd } \Gamma \text{ old-tmp (void*) M) M} = \text{ block curr (old-sz + 1024)} \qquad (6)$$

from ds-cover Γ curr old-tmp M = block curr old-sz . The idea of the proof is to split off the last block at old-tmp from cover in (6), which is possible by generic unfolding lemmata about the acyclic list structure. This yields a single element list from old-tmp to tmp, which can be converted into a block old-tmp 1024. It remains to show:

$$\text{block curr old-sz } \| \text{ block (curr } \oplus \text{ old-sz) 1024} = \text{block curr (old-sz + 1024)}$$

Joining the adjacent blocks on the left-hand-side finishes the proof.[1]

Having completed the proof, we would like to remark on the apparent complexity of the invariant. Towards that end, we compare it with the invariant given by Tuch [Tuc08a, §6.5] for the same algorithm. The first observation is that both are approximately of the same size and have a similar structure.[2] The main difference is that, because Tuch uses separation logic, he has to specify explicitly the different parts of the list that the algorithm works with: the part up to curr, between curr and tmp, and from tmp to null. In the proof of the invariant, all three parts then have to be manipulated. In our invariant, the overall data structure is kept as a whole. In the proof, the unfolding tactics then split the memory layout automatically to solve memory-related proof obligations. In this manner, all but the last conjunct are maintained directly.

If the loop finishes with tmp ≠ null, the algorithm removes the chunks between curr and tmp from the list structure by an assignment *prev = tmp; (not shown in Figure 4). In Tuch's loop invariant, the required split is already present. With ours, the removal is accomplished directly by lemmata from the list library. They allow a ds-cover to be split at points designated by the in-list assertions from the invariant, which capture the relative positions of prev, curr, and tmp.

In summary, our approach has enabled us to move many proof obligations from the verification of the specific `kalloc` algorithm into a generic library of lemmata about the allocator's data structure. They are, indeed, independent of the algorithm and follow a developer's intuition about the data structure. Since they are formulated in classical higher-order logic, their proofs can take advantage of the automatic reasoners available in Isabelle/HOL. The library with definitions, theorems, and proofs has just above 370 lines of Isabelle code. The verification of the loop itself consists in 165 lines, of which 50 are the statement of the lemma, and 19 are immediate invocations of `step`.

5 Related Work

Several authors have studied the application of separation logic to low-level code. Appel and Blazy [AB07] give a logic for C-minor. Myreen and Gordon [MFG07] target machine code.

---

[1] In an unbound address space, these steps would be obvious. Since addresses are words, overflows in the address arithmetic have to be excluded. This can be derived from the fact that the chunks in the data structure are sorted by their start addresses.

[2] However, Tuch crucially exploits the break statement in his language, which allows the invariant to be broken after setting tmp to null in the inner loop. Without break, the invariant would be twice the size [TKN07a, Sec. 1, p.8].

This work focuses on the formalization, rather than the application, of the Hoare logic. Lin et al. [LC$^+$07] verify a garbage collector using a separation logic. They report that a major challenge was the manipulation of separation logic assertions [LC$^+$07, §5.2]. Their tactics are able to match equal separation conjuncts, but do not provide any unfolding. Appel [App06] and McCreight [McC09] address the support for interactive reasoning in separation logic, but do not aim at automatic reasoning.

Tuch [Tuc08b] demonstrates how structs and fixed-size arrays can be handled in separation logic. He uses a deep embedding, i.e. he defines a datatype that captures the recursive structure explicitly. He models the semantics of C memory in detail, including alignment and padding. The automation available is limited to a tactic that unfolds the structure of types on demand. In contrast, we use a shallow embedding where new data structures can be defined in a flexible way, and rely on automated reasoning support.

Cohen et al. [CMST09] maintain the memory layout in a ghost variable. This variable contains a set of disjoint objects identified by their (typed) base pointers. They introduce statements split and join that allow the programmer to expose or to hide the internal structure of objects. With this setup, disjointness of memory regions can be derived from pointer inequality as in a typed memory model. They support structs, bit-fields, unions, and fixed-size arrays, but not arrays with dynamic size and no recursively defined data structures. They also rely on programmer assistance in maintaining the layout, while we automated the reasoning. Since only a single layout can be expressed, the multiple views used in Section 4 cannot be realized directly.

Berdine et al. [BCC$^+$07] use fully automatic proofs about a fragment of separation logic to implement a shape analysis. Their algorithm depends, however, on the fact that in the considered fragment only finitely many unfolding steps apply in each goal. Recently, it has been shown that the fragment can in some cases also accommodate reasoning about the content of data structures [Tue09, BPS09].

6 Conclusion

We have shown that it is possible to reason about low-level programs using high-level arguments and proofs that follow a developer's understanding. The basis of our approach is the automated reasoning about the disjointness of memory regions in the lightweight separation method. We have extended the existing mechanisms to low-level programs by including appropriate invariants into the definitions of memory layouts. One of the main challenges has been the finiteness of the address space, which entails the possibility of overflows in the address arithmetic. With the invariants, the theorems about the layouts become natural and intuitive.

Lightweight separation splits the assertions about the layout from assertions about the memory content. This split has proven particularly useful in low-level programs that use typed and untyped views on the memory at the same time (Section 4.1). Furthermore, it has enabled us to verify code that works on a low-level data structure without actually exposing the internal memory layout of the data structure (Section 4.2). The arising proof obligations about the list structure have been solved by generic lemmata from a library, which is independent of a particular algorithm being verified and thus re-usable in further projects. Furthermore, the theorems reflect the intuitive understanding of lists.

The presented approach is very flexible, as it covers a wide range of data structures. C types such as structs and arrays of dynamic size are directly supported by the verification environment. Furthermore, the user can describe application-specific data structures, including recursively defined data types, in a straightforward manner. Once introduced, these memory structures are handled automatically just like the built-in ones (Section 2.2, 4.2).

There are three areas of future work: first, we plan to incorporate the details of the C memory model for bit-fields, unions, padding, and alignment. Second, we propose to pursue further the formulation of generic theorems about data structures: the list structure shown in Section 4.2 is not, in principle, different from any singly-linked list. It will therefore be possible to provide a generic library of theorems about linked lists using Isabelle's locale mechanism. Finally, we plan to use existing automatic provers for discharging proof obligations. Towards that end, note that all theorems about memory layouts used during verification are first-order. The main challenge is therefore to replace the special purpose ML tactics by the provers' search strategies.

Bibliography

[AB07]   A. W. Appel, S. Blazy. Separation Logic for Small-Step C minor. In Schneider and Brandt (eds.), *TPHOLs*. LNCS 4732, pp. 5–21. Springer, 2007.

[App06]   A. W. Appel. Tactics for separation logic. http://www.cs.princeton.edu/~appel/papers/septacs.pdf, Jan. 2006.

[BCC+07]   J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, H. Yang. Shape analysis of composite data structures. In *CAV 2007*. LNCS 4590. Springer, Heidelberg, 2007.

[BNUW09]   S. Berghofer, T. Nipkow, C. Urban, M. Wenzel (eds.). *Theorem Proving in Higher Order Logics 22nd International Conference (TPHOLs 2009)*. LNCS 5674. Springer, 2009.

[BPS09]   M. Botincan, M. Parkinson, W. Schulte. Separation Logic Verification of C Programs with an SMT Solver. In *4th International Workshop on Systems Software Verification (SSV 2009)*. Electronic Notes in Theoretical Computer Science. Elsevier Science B.V., 2009.

[Bur72]   R. Burstall. Some Techniques for Proving Correctness of Programs which Alter Data Stuctures. In Meltzer and Michie (eds.), *Machine Intelligence*. Volume 7. Edinburgh University Press, 1972.

[CMST09]   E. Cohen, M. Moskal, W. Schulte, S. Tobies. A Precise Yet Efficient Memory Model for C. In *4th International Workshop on Systems Software Verification (SSV 2009)*. ENTCS. Elsevier Science B.V., 2009.

[Daw07]   J. E. Dawson. Isabelle Theories for Machine Words. In *Seventh International Workshop on Automated Verification of Critical Systems (AVOCS'07)*. ENTCS. September 2007.

[Gas08]   H. Gast. Lightweight Separation. In Ait Mohamed et al. (eds.), *Theorem Proving in Higher Order Logics 21st International Conference, TPHOLs 2008*. LNCS 5170. Springer, 2008.

[Gas09]   H. Gast. Reasoning about Memory Layouts. In Cavalcanti and Dams (eds.), *FM 2009: Formal Methods,Second World Congress*. LNCS 5850. Springer, 2009.

[Kow77]   T. Kowaltowski. Axiomatic aproach to side effects and general jumps. *Acta Informatica* 7:357–360, 1977.

[LC⁺07]   C.-X. Lin, Y.-Y. Chen, , L. Li, B. Hua. Garbage Collector Verification for Proof-Carrying Code. *JCST* 22(3):426–437, May 2007.

[McC09]   A. McCreight. Practical Tactics for Separation Logic. In [BNUW09].

[MFG07]   M. O. Myreen, A. C. J. Fox, M. J. C. Gordon. Hoare Logic for ARM Machine Code. In Arbab and Sirjani (eds.), *FSEN*. LNCS 4767, pp. 272–286. Springer, 2007.

[Nor98]   M. Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, 1998. Technical Report UCAM-CL-TR-453.

[ORY01]   P. W. O'Hearn, J. C. Reynolds, H. Yang. Local Reasoning about Programs that Alter Data Structures. In *Proceedings of the 15th International Workshop on Computer Science Logic*. LNCS 2142, pp. 1–19. Springer, 2001.

[RH09]   Z. Rakamaric, A. J. Hu. A Scalable Memory Model for Low-Level Code. In Jones and Müller-Olm (eds.), *Verification, Model Checking, and Abstract Interpretation, 10th International Conference (VMCAI 2009)*. LNCS 5403. Springer, 2009.

[Sch05]   N. Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2005.

[TKN07a]   H. Tuch, G. Klein, M. Norrish. Verification of the L4 kernel memory allocator. Formal proof document. Technical report, NICTA, 2007. http://www.ertos.nicta.com.au/research/l4.verified/kmalloc.pml.

[TKN07b]   H. Tuch, G. Klein, M. Norrish. Types, Bytes, and Separation Logic. In Hofmann and Felleisen (eds.), *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*. Pp. 97–108. Nice, France, Jan. 2007.

[Tuc08a]   H. Tuch. *Formal Memory Models for Verifying C Systems Code*. PhD thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, Aug 2008.

[Tuc08b]   H. Tuch. Structured Types and Separation Logic. In *3rd International Workshop on Systems Software Verification (SSV 08)*. Feb. 2008.

[Tue09]   T. Tuerk. A Formalisation of Smallfoot in HOL. In [BNUW09].