

# Structuring Interactive Correctness Proofs by Formalizing Coding Idioms

Holger Gast

Wilhelm-Schickard-Institut für Informatik  
University of Tübingen  
`gast@informatik.uni-tuebingen.de`

**Abstract.** This paper examines a novel strategy for developing correctness proofs in interactive software verification for C programs. Rather than proceeding backwards from the generated verification conditions, we start by developing a library of the employed data structures and related coding idioms. The application of that library then leads to correctness proofs that reflect informal arguments about the idioms. We apply this strategy to the low-level memory allocator of the L4 microkernel, a case study discussed in the literature.

## 1 Introduction

Interactive theorem proving offers several recognized benefits for functional software verification. From a foundational perspective, it enables the definition of the language semantics and the derivation of the Hoare logic, which ensures that the verification system is sound (relative to the defined semantics) (e.g. [1–3]). The background theories for reasoning about the machine model can likewise be derived, rather than axiomatized [4, §1.4][1], thus avoiding known practical issues of inconsistencies [5, §7]. From the perspective of applications, interactive provers offer strong support for the development of theories of the application domain [4, §1.3], which are not restricted to special classes of properties [6, §2.3]. In particular, they can address algorithmic considerations [7, §7.2], such as geometric questions [8, §4.4] or properties of defined predicates [9, §4.3].

However, interactive software verification incurs the obvious liability of requiring the user to guide the proof in some detail and to conceive a proof structure matching the intended correctness argument. This is the case even more in the development of background theories applicable to several algorithms. The necessity of strategic planning and human insight involved is often perceived as a major obstacle to the practical applicability of interactive proving.

This paper proposes to address the challenge of structuring correctness proofs by focusing on the idioms and coding patterns connected with the data structures found in the verified code. The benefit to be gained from this approach is clear: users can bring to bear their insight and experience as software engineers on the formal development, and the proof’s structure will follow the informal correctness arguments used by developers, thus making it more understandable and hence maintainable.

We demonstrate and evaluate this strategy using a study of the memory allocator of the L4 microkernel, which has previously been verified by Tuch et al. [10, 11] and thus affords a point of comparison. Although the allocator merely maintains a sorted singly-linked list of free blocks, we have found even a simplified fragment of the code surprisingly hard to verify in a previous attempt [12], owing to the many technical aspects introduced by the low-level memory model. This impression is confirmed by the level of detail present in the original proof [13].

The benefit of the strategy proposed now can therefore be gauged by whether the found proof matches the essentially simple structure of the algorithm, thus appearing as a detailed version of an informal correctness argument. This goal also relates to a peculiarity of interactive software verification. Differing from mechanized mathematics, no effort is spent on making the proof more concise or elegant once it is found—its mechanically verified existence is sufficient.

For this reason, the paper’s structure reflects the development of the proof. Section 2 gives an overview of the allocator and points out the coding idioms that make the code seem straightforward. Section 3 then formalizes these idioms in a library of singly-linked lists and their standard manipulations. Section 4 gives the correctness proof on the basis of that library, with an emphasis on the direct match between the library theorems and informal correctness arguments.

### 1.1 An Overview of Lightweight Separation

We conduct the proof within the lightweight separation verification system [14, 15]. It is developed as a conservative extension of Isabelle/HOL and permits the verification of low-level programs in a C dialect inspired by [1].

The idea of lightweight separation is to complement the standard formulation of assertions in HOL with explicit formal representations of the memory layout. Towards that end, assertions usually contain a conjunct  $M \blacktriangleright A$  where  $M$  is the current memory state and  $A$  is a *cover*, which is a predicate on address sets. A cover is *well-formed* if it accepts at most one address set. We call the address set accepted by a well-formed cover  $A$  the *memory region covered by A*. For instance, the following constant captures a block of  $n$  bytes at  $a$ . It describes the address set and excludes overflows in address arithmetic, making the block contiguous ( $\{a..<b\}$  denotes a half-open interval  $[a, b)$  in Isabelle/HOL;  $\oplus$  is address offset).

$$\text{block } a \ n \equiv \lambda S. S = \{a..< a \oplus n\} \wedge a \leq a \oplus n$$

$M \blacktriangleright A$  states  $A$  covers the allocated region of  $M$ . For  $M \triangleright A$ ,  $A$  is allocated in  $M$ . The *subcover* relation  $A \preceq B$  states that the region of  $A$  is contained in the region of  $B$ . The memory layout is described by nested cover expressions combined by the *disjointness* operator  $A \parallel B$ . The system provides covers for standard constructs, such as variables and blocks whose size is given by a type. New constants for covers can be defined as needed. In particular, one can define covers for inductive data structures using Isabelle’s built-in `inductive` command.

The lightweight separation tactics then prove, by symbolic manipulation of cover expressions [14], the allocatedness of memory accessed by the program and the disjointness of regions read in assertions and modified by programs. If necessary, they unfold given layouts to expose their constituent parts [15].

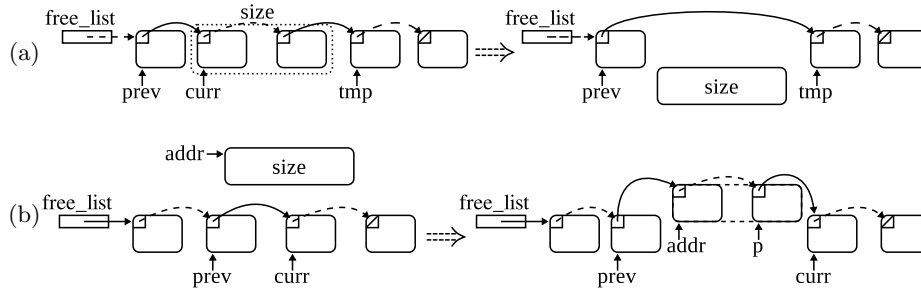


Fig. 1. Allocation and Deallocation in the Free List

## 2 The L4 Memory Allocator

The memory allocator of the L4 microkernel [13] is responsible for the low-level allocation of memory blocks. The interface consists of two routines `alloc` and `free` that enable client code to obtain and release memory blocks. We now describe and analyze their overall structure.

### 2.1 Data Structure and Routines

The microkernel allocator offers basic services to higher-level allocators and handles memory blocks as multiples of 1kb. Internally, it maintains a *free list* of 1kb *chunks*, whose first word is a pointer to the next chunk. The chunks in the list are ordered by their start addresses to enable efficient compaction during allocation. The routines `alloc` and `free` (Appendix A) in essence cut out or splice in sequences of chunks at the correct position within the free list.

The `alloc` routine advances a pointer `curr` forward through the free list (Fig. 1 (a); dashed arrows indicate the possible crossing of several chunks). At each chunk, a nested loop advances a pointer `tmp` to check whether the sequence of adjacent chunks starting at `curr` matches the requested size. If this is the case, the routine removes the sequence from the list, initializes it with 0-bytes, and returns it to the caller as a raw block of memory. The `prev` pointer is required to splice out the returned chunks and always lags one step behind the `curr` pointer.

The `free` routine dually splices a returned block of memory back into the free list (Fig. 1 (b)). Since the block's size may be a multiple of 1kb, the routine first creates a list structure inside the raw memory block. Then, it searches for the place where the new list fragment must be inserted to maintain sortedness. Finally, it links the new fragment into the free list. Like the `alloc` routine, it maintains a `prev` pointer to perform that final pointer manipulation.

### 2.2 Idioms for List Operations

The routines' code (Appendix A) appears straightforward after this explanation of its purpose. The reason for the simple reading is that the code only applies

```

ListNode *p = list_head;
while (p != NULL && not found) {
    perform check & return/break;
    p = p->next;
}
(a)

ListNode **prev = &list_head;
ListNode *p = list_head;
while (p != NULL && not found) {
    perform check & return/break;
    prev = &p->next;
    p = p->next;
}
(b)

```

**Fig. 2.** Iteration Through Lists

well-known patterns and idioms: the experienced developer recognizes these and uses them in arguments about the code’s functionality and correctness.

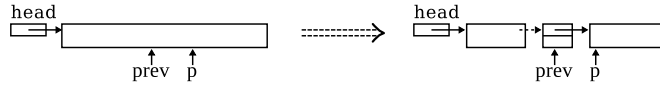
The first and most basic idiom is the search for a particular point in a list. The coding pattern is shown in Fig 2 (a): some pointer `p` is initialized to the first node, and is advanced repeatedly by dereferencing the `next` pointer of the node. The checks in the while test or body are usually used alternatively. Following the terminology of the C++ STL [16], we will call the pointer `p` an *iterator*. Informally, the iteration works without failure because `p` never leaves the list structure: it is “properly” initialized and “properly” advanced to the next node in the list. Since it points to a node after the test for `NULL`, the iterator `p` can be dereferenced in the checks without causing a memory fault.<sup>1</sup>

If a modification is intended after the search, the idiom must be extended by some reference to the predecessor node of `p`, in order to insert or remove nodes at `p` by pointer manipulation. There are several variants of such an extension. The L4 allocator uses the one shown in Fig. 2 (b), which makes use of C’s ability of taking the addresses of arbitrary memory objects. The constraints associated with `prev` are that `*prev = p` and that `prev` points either to the list-head variable or to the `next` field of some node in the list.<sup>2</sup>

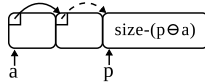
After the loop, the manipulation of the list structure involves the assignment `*prev = q`, where `q` is either some successor node of `p` for the removal of `p` or a new node to be inserted before `p`. To show that the resulting pointer structure is the desired linked list, informal arguments usually use pointer diagrams: the reached situation is shown in Fig. 3 on the left. If `prev = &head`, the argument is simple. Otherwise, one needs to expose the node containing the `prev` pointer in the diagram, possibly followed by extracting node `p` from the remaining list. Then, one draws the algorithm-specific pointer operations, e.g. those of Fig. 1, and argues that the expected list structure results. Note that the case distinction on `prev = &head`, which is necessary in the argument, is not present in the code.

<sup>1</sup> These are also the requirements for the STL’s most basic *forward iterator* [16].

<sup>2</sup> A common alternative uses a sentinel head node, such that `prev` is a node pointer and `p` is inlined as `prev->next` (e.g. the `slist` library of `g++`). This variant has the advantage of unifying the reasoning by avoiding the case distinction about the exact target of `prev`. The library in §3 supports this variant as well.



**Fig. 3.** Extraction of a Node in the Follow Iterator Idiom



**Fig. 4.** Establishing the Successor Structure

### 2.3 Idioms for Aligned Low-level Memory Access

The iterator idiom, once identified, can also be applied to algorithms that do not handle list-like data structures. The `zero_mem` function (Appendix A), for example, initializes a memory block by writing machine words, i.e. by setting groups of 4 bytes at a time. Its loop advances the pointer  $(\text{int}^*)p+i$  as an iterator in steps of 4, by incrementing  $i$  in each loop iteration. The pointer is properly initialized by setting  $i=0$ . Advancing the iterator by  $i++$  leaves it within the bounds of the raw memory block, because the block’s size is a multiple of 4.

The first loop of `dealloc` (§4.3) similarly establishes a list structure in a memory block (Fig. 4) by advancing a pointer  $p$ , initialized to start address  $a$ , in  $1\text{kb}$ -steps. The proof obligations are the same as for the iterator in `zero_mem`.

The correctness arguments in both cases therefore consist of the familiar “initializing” and “advancing” an iterator, and can be carried out analogously to the list case. Although the formulation of the invariants is quite different, the proofs thus still reflect the common structure. For space reasons, we will not discuss them further.

## 3 A Library of List Manipulations

This section captures the idioms from §2.2 in a generic library of singly-linked lists. Using this library, the correctness proof in §4 will be structured according to informal arguments, after the allocator’s free list has been proven an instance of the general case. For space reasons, we omit derived constructs, such as the typed abstraction of the nodes’ contents as HOL values and the variant of follow iterators mentioned in Footnote 2. The library consists of 750 lines and has been re-used in two further case studies (§6).

### 3.1 Parameters and Assumptions

The library is formulated as an Isabelle locale [17] to abstract over the specific structure of list nodes. Locales can depend on parameters and state assumptions about these. The list library has three parameters (1) (“ $\dots$ ” denotes a type constraint): `node` is a cover (§1.1) for a single list node and `succ` reads the successor

(or “next”-) link from a given node in a given memory state. Both usually depend on type definitions, so a global context  $\text{gctx}$  with these definitions is added.

$$\text{node} :: \text{addr} \Rightarrow \text{cover} \quad \text{succ} :: \text{addr} \Rightarrow \text{memory} \Rightarrow \text{addr} \quad \text{gctx} :: \text{ctx} \quad (1)$$

$$\text{accesses} (\text{succ } p) M (\text{node } p) \quad \text{node } p S \Longrightarrow p \in S \quad \text{wf-cover} (\text{node } p) \quad (2)$$

The theory makes three natural assumptions (2) about these parameters: reading the successor of a node depends only on that node ( $\text{accesses } f M A$  states that memory-reading function  $f$ , when applied to state  $M$ , depends only on the region covered by  $A$ ); the base pointer of a node is contained in its footprint; finally, the node cover must be well-formed (§1.1). Note that these assumptions are implicit in pointer diagrams and are validated by the usual list structures in programs.

### 3.2 List Structure

The standard approach to lists (e.g. [18]) is to define a predicate to enumerate the nodes in list fragments. An inductive definition is given by the introduction rules (3). A parallel definition for  $\text{cover } p q M$ , the memory region of the list, is straightforward. ( $[]$  is the empty list,  $\#$  denotes the “cons” operation)

$$\frac{p = q \quad xs = []}{\text{nodes } p q xs M} \quad \frac{p \neq q \quad \text{nodes} (\text{succ } p M) q ys M \quad xs = p \# ys}{\text{nodes } p q xs M} \quad (3)$$

Already at this point, the library yields a benefit in the form of useful properties, such as the nodes of a list being distinct (4).

$$\frac{\text{nodes } p q xs M}{\text{distinct } xs} \quad (4)$$

Due to their parallel definitions, the nodes and the cover of a list are closely related. In particular, if a list is allocated, then it consists of a sequence of nodes (5) and—since  $\text{null}$  is never allocated—it cannot contain  $\text{null}$  as a node (6).

$$\frac{M \triangleright \text{cover } p q M}{\exists xs. \text{nodes } p q xs M} \quad (5)$$

$$\frac{M \triangleright \text{node } p}{p \neq \text{null}} \quad \frac{\text{nodes } p q xs M \quad M \triangleright \text{cover } p q M}{\text{null} \notin \text{set } xs} \quad (6)$$

We have noted in §2 that informal arguments by pointer diagrams address the “extraction” of nodes from a list and the resulting “overall” list. We now reflect the graphical arguments in the form of theorems to make their application straightforward: for every change in the pointer diagram, the formal proof contains an application of the corresponding theorem. For space reasons, we omit the parallel development for  $\text{cover}$ .

Theorems (7) and (8) enable the extraction and integration of the first node of a list. Note how the pre-condition  $p \neq q$  reflects the check of the idiomatic while loops from §2.2. To save a separate application of (4), (7) yields the derived information that the nodes were originally distinct. The complementary

theorems (9) and (10) manipulate the last node of a list. The final rules (11) and (12) reflect splitting and joining at a given node of the list, as is necessary for Fig. 1. The last premises of (10) and (12) ensure that no cycles have been created. In the frequent case where  $q$  is null, they can be proven by (6); the library provides specialized rules for this case to simplify proofs further.

$$\frac{p \neq q}{\text{nodes } p \ q \ xs \ M = (\exists ys. \text{nodes } (\text{succ } p \ M) \ q \ ys \ M \wedge xs = p \# \ ys \wedge p \notin \text{set } ys)} \quad (7)$$

$$\frac{\text{nodes } r \ q \ ys \ M \quad \text{succ } p \ M = r \quad p \neq q}{\text{nodes } p \ q \ (p \# \ ys) \ M} \quad (8)$$

$$\frac{q = \text{succ } r \ M \quad r \in \text{set } xs}{\text{nodes } p \ q \ xs \ M = (\exists ys. \text{nodes } p \ r \ ys \ M \wedge xs = ys \ @ \ [r] \wedge q \notin \text{set } xs)} \quad (9)$$

$$\frac{\text{nodes } p \ r \ ys \ M \quad \text{succ } r \ M = q \quad q \notin \text{set } (ys \ @ \ [r])}{\text{nodes } p \ q \ (ys \ @ \ [r]) \ M} \quad (10)$$

$$\frac{r \in \text{set } xs}{\text{nodes } p \ q \ xs \ M = (\exists ys \ zs. \text{nodes } p \ r \ ys \ M \wedge \text{nodes } r \ q \ zs \ M \wedge xs = ys \ @ \ zs \wedge q \notin \text{set } xs)} \quad (11)$$

$$\frac{\text{nodes } p \ q \ xs \ M \quad \text{nodes } q \ r \ ys \ M \quad r \notin \text{set } xs}{\text{nodes } p \ r \ (xs \ @ \ ys) \ M} \quad (12)$$

### 3.3 Iterators

In principle, the definitions and theorems from §3.2 are sufficient state loop invariants and perform proofs about list manipulating programs (e.g. [18]). However, this approach invariably has to consider the set of list nodes. The idioms of §2.2, on the other hand, focus on the “current” and the “next” node, which aligns the informal reasoning with the local list structure—the inductive argument about the iterator referencing one of the list’s nodes is left implicit.

We can obtain proofs that reflect the informal reasoning by formalizing the idea of an “iterator” itself. In the STL concept [16], an iterator into some data structure always points to one of its elements or is a special one-past-the-end iterator. In the case of fragments of singly-linked lists, this idea is expressed by the following definition.

$$\text{iter } a \ p \ q \ M \equiv \exists xs. \text{nodes } p \ q \ xs \ M \wedge (a \in \text{set } xs \vee a = q)$$

The loop invariant for the iteration idiom can then simply contain the conjunct  $\text{iter } p \ \text{head} \ \text{null} \ M$ , thus hiding the list structure as desired. Furthermore, the informal arguments about “initializing” and “advancing” an iterator from §2.2 are reflected by theorems (13), and these are used to establish the invariant and prove its preservation after the loop body. When the sought node in the list has been found, it can be exposed by (14), followed by (8), without leaving the iterator idiom.

$$\frac{M \triangleright \text{cover } p \ q \ M}{\text{iter } p \ p \ q \ M} \quad \frac{\text{iter } a \ p \ q \ M \quad a \neq q}{\text{iter } (\text{succ } a \ M) \ p \ q \ M} \quad (13)$$

$$\frac{\text{iter } r \ p \ q \ M}{\text{nodes } p \ q \ xs \ M = (\exists ys \ zs. \text{nodes } p \ r \ ys \ M \wedge \text{nodes } r \ q \ zs \ M \wedge xs = ys \ @ \ zs \wedge q \notin \text{set } xs)} \quad (14)$$

### 3.4 Follow Iterators

Whenever a list manipulation is intended after iteration, one has to keep an auxiliary pointer to the node preceding the current one (§2.2). Since the pattern is so frequent, we introduce another abstraction to capture it. Since the `prev` pointer lags one step behind a `cur` pointer, we choose the term *follow iterator*.

The locale for follow iterators extends that of iterators by introducing parameters that abstract over the structure of the “successor field”, i.e. the memory object containing the “next” pointer. By the idiom, this structure must be the same as that of the head variable. The structure is given by a cover `succ-field`. The function `rd-succ-field` is used for reading its content. The offset of the field within the node is given by `succ-field-off`.

```

succ-field    :: "addr ⇒ cover"
succ-field-off :: "word32"
rd-succ-field :: "addr ⇒ memory ⇒ addr"

```

The locale’s assumptions describe the following relationships between these parameters: the special accessor reads the information gained by `succ` (§3.2) and depends only on the given region, which must be contained in the corresponding list node and must be well-formed (§1.1).

$$\begin{aligned}
& \text{rd-succ-field } (p \oplus \text{succ-field-off}) M = \text{succ } p M \\
& \text{accesses } (\text{rd-succ-field } p) M (\text{succ-field } p) \\
& \text{succ-field } p \preceq \text{node } (p \oplus \text{-succ-field-off}) \quad \text{wf-cover } (\text{succ-field } p)
\end{aligned}$$

The follow iterator abstraction is then defined as expected (§2.2, Fig. 1): `cur` is an iterator, while `prev` points to a link to `cur`; further, `prev` either points to the head variable or is itself an iterator within the list.

$$\begin{aligned}
& \text{follow-iter } \text{prev } \text{cur } \text{head } p \ q \ M \equiv \\
& \text{iter } \text{cur } p \ q \ M \wedge \text{rd-succ-field } \text{prev } M = \text{cur} \wedge \\
& (\text{prev} = \text{head} \vee (\text{prev} \oplus \text{-succ-field-off} \neq q \wedge \text{iter } (\text{prev} \oplus \text{-succ-field-off}) p \ q \ M))
\end{aligned}$$

This newly defined construct establishes another layer of abstraction over the raw list structure, in that it enables the now familiar reasoning patterns in a self-contained system: theorems (15) and (16) capture the initializing and advancing of the iterator and thus replace (13) in the proofs. It is worth checking that the additional premises reflect the initializations from the idiomatic code (§2.2, Fig. 2 (b)), thus making the application of theorems straightforward.

$$\frac{M \triangleright \text{cover } p \ q \ M \quad \text{prev} = \text{head} \quad \text{cur} = p \quad \text{cur} = \text{rd-succ-field } \text{prev } M}{\text{follow-iter } \text{prev } \text{cur } \text{head } p \ q \ M} \quad (15)$$

$$\frac{\text{cur}' \neq q \quad \text{cur} = \text{succ } \text{cur}' \ M \quad \text{prev} = \text{cur}' \oplus \text{succ-field-off}}{\text{follow-iter } \text{prev } \text{cur } \text{head } p \ q \ M} \quad (16)$$

Furthermore, the reasoning about the modification after the search from Fig. 3 can now be expressed in a single theorem (17). The prerequisite case distinction from the informal argument of §2.2 can be introduced by the (tautological)

rule (18) by a single tactic invocation, saving explicit terms in the proof.

$$\frac{\text{follow-iter prev cur head p q M} \quad \text{prev} \neq \text{head}}{\text{nodes p q xs M} = (\exists \text{ys zs. nodes p prev ys M} \wedge \text{nodes cur q zs M} \wedge \text{xs} = \text{ys} \text{ @ prev} \# \text{zs} \wedge \text{q} \notin \text{set xs})} \quad (17)$$

$$\frac{\text{follow-iter prev cur head p q M}}{\text{prev} = \text{head} \vee \text{prev} \neq \text{head}} \quad (18)$$

The `follow-iter` abstraction thus encapsulates all information necessary to perform the split. This is evident in the proof of (17), which is based on a combination of the elementary lemmas (11), (12), (7), and (4) about the list structure (§3.2). While that proof still follows an informal argument by pointer diagram, the formalization in `follow-iter` and (17) enables the user to link the concrete proof to the code’s intention directly. Furthermore, it saves a lot of detailed and cumbersome manipulation of formulae, which we struggled with in [12], and makes the proof more readable and thus more maintainable.

## 4 The Correctness Proof

This section gives the correctness proof of the allocator. The proof is structured by the application of the library from §3 and thus follows the informal arguments used in §2. The proof script is available from the author’s homepage [19].

### 4.1 Formalizing the Allocator’s Free List

We first instantiate the list library from §3 for the allocator’s free list. Even though the library seems to suggest some “typed” concept of lists, the allocator’s data structure fits directly: after instantiating the parameters as follows and discharging the library’s assumptions by 40 lines of straightforward tactics, the developed constants and reasoning patterns are available. («» delineates program syntax in HOL, here that of types. The system contains a pre-processor.)

```

node p           ≡ block p 1024
succ p M         ≡ to_ptr (rd gctx p «void*» M)
succ-field p     ≡ typed-block gctx p «void*»
rd-succ-field a M ≡ to_ptr (rd gctx a «void*» M)
succ-field-off   ≡ 0

```

We then introduce an abbreviation `kfree-list` for reading the global head variable `kfree_list` and define the invariant `free-list-inv`: the chunks in the list are ordered by their base addresses and they are aligned to 1kb. The `free-list-cover` summarizes the memory occupied by the data structure.<sup>3</sup>

```

kfree-list ctx M ≡ to_ptr (rdv (in-globals ctx) "kfree-list" M)
free-list-inv ctx M ≡ (∃ C. nodes (kfree-list ctx M) null C M ∧ sorted C ∧
  (∀ p ∈ set C. aligned p 1024))
free-list-cover ctx M ≡ var-block (in-globals ctx) "kfree-list" || cover (kfree-list ctx M) null M

```

<sup>3</sup> We note in passing that the introduced information hiding is maintained for clients by the theorem `accesses (free-list-inv ctx) M (free-list-cover ctx M)`: the lightweight separation framework will prove that the free list is not influenced by the clients’ memory manipulations and thus solves the frame problem (e.g. [20]) in a natural fashion.

## 4.2 Allocation

The `alloc` routine searches for a contiguous block of memory that is large enough to fit the requested size (§2.1). Its specification is translated from [13]: the pre-condition requires that the free list data structure is intact and the memory does contain the free list. Furthermore, the requested size must be a multiple of 1kb.

$M \blacktriangleright \text{free-list-cover ctx } M \wedge \text{free-list-inv ctx } M \wedge 0 < \text{size} \wedge 1024 \text{ udvd } \text{size} \wedge \text{size} = \text{SIZE}$

The post-condition distinguishes between success and failure. In both cases, the data structure itself is preserved. If the allocation is successful, an aligned block of 0-initialized memory has been extracted from the free list. The auxiliary (or logical) variable `SIZE` links the pre- and post-conditions as usual.

$\text{free-list-inv ctx } M \wedge$   
 $(\text{return} \neq \text{null} \longrightarrow M \blacktriangleright \text{free-list-cover ctx } M \parallel \text{block return SIZE} \wedge$   
 $\quad \text{aligned return } 1024 \wedge \text{zero-block ctx return SIZE } M) \wedge$   
 $(\text{return} = \text{null} \longrightarrow M \blacktriangleright \text{free-list-cover ctx } M)$

The nested loops (Appendix A) of `alloc` advance the pointers `curr` and `tmp`, where the inner loop leaves `curr` unchanged. The outer loop invariant is therefore the same as the following inner loop invariant, except that Lines 3–4 are missing:

```

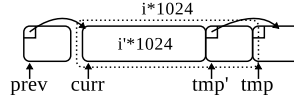
1 free-list-inv ctx M ∧ size = SIZE ∧ 0 < size ∧ 1024 udvd size ∧ i ≤ size div 1024 ∧
2 follow-iter prev curr «&kfree-list» kfree-list null M ∧
3 curr ≠ null ∧
4 (tmp ≠ null → cover curr tmp M = block curr (i * 1024) ∧ iter tmp curr null M) ∧
5 M ▶ free-list-cover ctx M ∥ size ∥ prev ∥ curr ∥ tmp ∥ i ∧
6 M ▷ typed-block prev «void*» ∥ size ∥ prev ∥ curr ∥ tmp ∥ i

```

Line 1 preserves the pre-condition and states that `i` will not exceed the bound given by the `size` parameter. Line 2 invokes the follow iterator idiom (§2.2) from the library (§3.4). Line 3 preserves the test result of the outer loop. Line 4 uses the notation for memory layouts (§1.1, §3.2) to state that a contiguous block of memory is found between `curr` and `tmp`. Line 5 extends the initial memory layout by the local variables; Line 6 adds that `prev` is not a local variable while leaving open whether it refers to the variable `kfree-list` or a list node.

The structure of the correctness proof is now already clear: the initializations before both loops leave precisely the situation where theorems (13) and (15) about the initialization of iterators apply. For the preservation of the outer invariant, the pointer assignments in the body match the idiom (§2.2) such that (16) is sufficient. All of these steps thus reflect the idiomatic, informal view, and the proof is merely a more precise form of argument.

For the preservation of the inner invariant, the then-branch is trivial. In the else-branch, only Line 4 needs to be newly established. In the conceptual view of §2.2, the code advances the iterator `tmp`; correspondingly (13) solves the `iter`-part immediately. The remainder of Line 4 contains the core of the algorithm: we have to prove that the found block is still contiguous, using that  $\text{tmp} = \text{curr} + i * 1024$  by the if-test. Fig. 5 depicts the proof obligation, using primed variables for the pre-state of the loop body. The figure also contains the idea of the proof: on the right-hand side of the equation from Line 4, we split off one chunk at the end of the list by (9); on the left-hand side, we split the contiguous block at address `tmp'`. This strategy can be expressed by 8 lines of tactics.



**Fig. 5.** Extending the Found Block with a New Chunk

The final proof obligation concerns the returning of an allocated memory block after the inner loop. Here, the assignment `*prev=tmp` splices out a sequence of chunks (Fig. 1). Since that assignment matches the idiom from §2.2, we can use (17) to perform the split of the list, after a case distinction by (18). Then, Line 4 of the invariant yields the memory layout of the post-condition. The argument takes 30 lines of tactics for both cases together; the application of the theorems reflects the informal manipulation of pointer diagrams in all steps.

### 4.3 Deallocation

The `free` routine takes a block of memory and integrates it into the allocator’s free list (Fig. 1). Its specification, again translated from [13], requires that the free list is intact and allocated and that the block’s size is a multiple of 1kb.

$M \triangleright \text{free-list-cover ctx } M \parallel \text{block a size} \wedge \text{free-list-inv ctx } M \wedge$   
 $0 < \text{size} \wedge 1024 \text{ udvd size} \wedge \text{aligned a } 1024$

It guarantees that the passed block has been merged into the free list.

$M \triangleright \text{free-list-cover ctx } M \wedge \text{free-list-inv ctx } M$

The function `free` consists of two loops. The first one establishes the pointer structure within the passed memory block, the second splices the created list into the free list at the correct position.

The first loop uses an iterator-like construct to establish the list structure within the raw memory block (§2.3; Fig. 4). We have developed a thin wrapper around Isabelle’s Word library [21] to enable the idiomatic reasoning about initializing and advancing iterators. The proof that the overall block maintains the shape of Fig. 4, i.e. an initial list of elements with a trailing raw block, can be proven along the graphical intuition, by using essentially the same steps as the derivation from Fig. 5 in §4.2.

The invariant of the second loop is again typical of a search loop (§2.2, §3.4):

- 1  $\exists B. M \triangleright \text{free-list-cover ctx } M \parallel \text{cover a p } M \parallel \text{node p} \parallel \text{a} \parallel \text{size} \parallel \text{p} \parallel \text{prev} \parallel \text{curr} \wedge$
- 2  $\text{free-list-inv ctx } M \wedge 0 < \text{size} \wedge 1024 \text{ udvd size} \wedge \text{aligned a } 1024 \wedge \text{aligned p } 1024 \wedge$
- 3  $\text{cover a p } M \parallel \text{node p} = \text{block a } (p \oplus 1024 \ominus a) \wedge$
- 4  $\text{nodes a p } B \wedge \text{sorted } B \wedge (\forall b \in \text{set } B. a \leq b \wedge b < p \wedge \text{aligned b } 1024) \wedge$
- 5  $\text{follow-iter prev curr} \ll \&\text{kfree-list} \gg \text{kfree-list nill } M \wedge$
- 6  $(\text{prev} = \ll \&\text{kfree-list} \gg \vee \text{prev} < a) \wedge a \leq p$

Lines 1–2 maintain the information of the pre-condition; Lines 3–4 keep the result of the first loop (Fig. 4). Line 5 captures `curr` as a follow iterator (§3.4) for the search, while Line 6 characterizes the nodes that `curr` has already passed as having strictly smaller start addresses that `a`.

Since the loop matches the idiom (Fig. 2 (b)), its correctness proof follows the reasoning already discussed for `alloc` in §4.2: (15) and (16) yield initialization

and preservation; Line 6 follows from the while-test. After the loop, the new sequence of nodes  $a \dots p$  is spliced into the free list before  $\text{curr}$ , again making use of (17) and (18) to split the overall list structure before the pointer updates.

## 5 Related Work

To the best of the author’s knowledge, the proposal of developing background theories by formalizing idioms and coding patterns has not been discussed previously. We therefore focus on similar case studies and on approaches to structuring interactive proofs beyond the discharging of generated verification conditions.

Tuch et al. [10, 13] give two proofs of the L4 memory allocator, one using separation logic and one using a typed view on the raw memory. Their development shows the intricacy of reasoning about byte-addressed finite memory. Our own proof clearly benefits from Isabelle’s Word library [21] contributed by the L4 verification project. In his analysis [11, §6.6], Tuch suggests that with further experience in similar proofs, a set of re-usable libraries could be constructed to aid in future developments. He proposes to collect lemmas that have been found useful, and to improve automation for separation logic assertions. Differing from ours, his approach is thus goal-directed, starting from the verification conditions. Although proof sizes in different systems are not directly comparable, it is interesting that our proof is significantly shorter (by a factor of 2) even though Tuch et al. prove only the immediately necessary theorems.

Marti et al. [22] verify the heap manager of the Topsy operating system, which is also based on an untyped singly-linked list. The paper focuses on the developed verification environment and therefore the actual proof is discussed only at the level of the defined predicates and the function specifications. An instance of forward reasoning appears in [22, §4.2], where a central theorem for compacting two list nodes is derived beforehand and is shown to apply to an example Hoare triple of an expected format. The structure of the greater part of the proof ( $\approx 4500$  lines of Coq) is not analyzed further.

Böhme et al. [4] investigate the advantages of interactive theorem proving for software verification. In [4, §1.3], they observe that the introduction of suitable abstractions with well-developed theories can make interactive proofs feasible where automated provers fail because they have to unfold the definitions. They demonstrate the claim by a case study on an implementation of circular singly-linked lists, but do not formulate strategies to develop general theories.

Concerning the question of structuring interactive correctness proofs, Myreen [23, §5.2] verifies Cheney’s garbage collector using a refinement argument. The first two layers capture the specification and abstract implementation of copying garbage collection; they can thus be read as the common structure of different collectors. Our proposal of formalizing idioms addresses, on the other hand, cross-cutting issues of different algorithms. McCreight’s proof [24] of the same algorithm introduces carefully chosen separation logic predicates that reflect the structure of pointer diagrams, and diagrammatic arguments are used to illustrate the proof strategies. However, their translation into a proof script involves a

substantial amount of technical formula manipulation [24, §6.3.3, p. 122, §6.4.3]. Both the defined predicates and the proof strategies are specific to the algorithm.

A different approach to interactive proving has been proposed by Tuerk [25] and Chlipala [3]. They use a restricted form of separation logic, inspired by Smallfoot [26]. Besides pure assertions, verification conditions then consist of implications between iterated spatial conjunctions, which are cancelled syntactically one-by-one, possibly using user-supplied unfolding rules. This process reduces the verification conditions to pure assertions, which are solved mostly automatically by the built-in tactics of the interactive provers.

## 6 Conclusion

Interactive software verification enables the development of theories independently of concrete verification conditions, with a view to making proofs readable, maintainable, and possibly re-usable. This paper has proposed to structure such theories around the idioms and coding patterns employed by developers, and to formulate the definitions and theorems to reflect informal arguments about the code, e.g. in the form of pointer diagrams.

We have demonstrated this strategy using the frequent case of singly-linked lists. Besides their basic structure (§3.2), we have introduced the higher-level idioms of *iterators* (§3.3) for read-only searches and *follow iterators* (§3.4) for searching and modifying lists. The developed library is formulated as an Isabelle locale and can be instantiated for different concrete list structures. We have applied the library to the untyped free list of the L4 memory allocator [10, 13]. It was interesting to find during the development that the reasoning patterns embodied in the library made the overall proof [19] much more straightforward than the previous partial attempt [12], even though several additional points, such as alignment and the initialization of allocated memory had to be considered.

The proposed strategy has shown several benefits: first, all verification conditions regarding the list structure were solved by library theorems, and their application in each case reflected informal arguments by pointer diagrams. The chosen theorem names preserve this link in the proof script [19], thus contributing to its maintainability. Second, the analogies between the allocator's routines could be exploited by having a common ground for expressing them (§4.2, §4.3). Third, although no specific effort was made, the script is substantially smaller than the original one [13, 11], which can be attributed to the simple application of library theorems due to their matching the coding idioms.

Finally, the library's genericity has enabled its re-use for the work queues of the Schorr-Waite graph marking algorithm [15] and Cheney's collector [27, 28]. Both algorithms use a non-standard successor link, involving a case-distinction and pointer arithmetic, respectively. The correctness proofs are nevertheless covered by the library theorems (§3.2). Between the two algorithms, we have re-used a theory of object graphs [15, §5.1] that is also structured around expected common manipulations. This further example suggests that the strategies proposed now will be applicable beyond the chosen case study.

## A Source Code

```
void *alloc(unsigned int size) {
  void **prev = &kfree_list;
  void *curr = kfree_list;
  while (curr != null) {
    void *tmp = *(void **)curr;
    unsigned int i = 1;
    while (tmp != null &&
           i < size / 1024) {
      if (tmp != curr + i * 1024) {
        tmp = null;
      } else {
        tmp = *(void **)tmp;
        i++;
      }
    }
    if (tmp != null) {
      *prev = tmp;
      zero_mem(curr, size);
      return curr;
    }
    prev = (void **)curr;
    curr = *(void **)curr;
  }
  return null;
}

void free(void *a, unsigned int size) {
  void *p;
  void **prev;
  void *curr;
  p = a;
  while (p < a + (size - 1024)) {
    *(void **)p = p + 1024;
    p = *(void **)p;
  }
  prev = &kfree_list;
  curr = kfree_list;
  while (curr != null && (a > curr)) {
    prev = (void **)curr;
    curr = *(void **)curr;
  }
  *prev = a;
  *(void **)p = curr;
}

void zero_mem(void *p, unsigned int n) {
  unsigned int i = (unsigned int)0;
  while (i < n / 4) {
    *((int*)p+i) = 0;
    i++;
  }
}
```

## References

1. Norrish, M.: C formalised in HOL. PhD thesis, University of Cambridge (1998) Technical Report UCAM-CL-TR-453.
2. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. PhD thesis, Technische Universität München (2005)
3. Chlipala, A.: Mostly-automated verification of low-level programs in computational separation logic. In: Proceedings of the ACM SIGPLAN 2011 Conference on Programming Language Design and Implementation (PLDI'11). (2011)
4. Böhme, S., Moskal, M., Schulte, W., Wolff, B.: HOL-Boogie—An interactive prover-backend for the Verifying C Compiler. *J. Autom. Reason.* **44** (2010) 111–144
5. Banerjee, A., Barnett, M., Naumann, D.A.: Boogie meets regions: a verification experience report. In Shankar, N., Woodcock, J., eds.: VSTTE'08. Volume 5295 of LNCS., Springer (2008) 177–191
6. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. *Communications of the ACM (CACM)* **53**(6) (2010) 107–115
7. Zee, K., Kuncak, V., Rinard, M.: Full functional verification of linked data structures. *SIGPLAN Not.* **43**(6) (2008) 349–361
8. Walter, C.L.D.: Certifiable specification and verification of C programs. In: Formal Methods. Formal Methods (FM 2009), Springer (2009)
9. Hubert, T., Marché, C.: A case study of C source code verification: the Schorr-Waite algorithm. In Aichernig, B.K., Beckert, B., eds.: SEFM, IEEE (2005)

10. Tuch, H., Klein, G., Norrish, M.: Types, bytes, and separation logic. In Hofmann, M., Felleisen, M., eds.: Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07). (2007)
11. Tuch, H.: Formal Memory Models for Verifying C Systems Code. PhD thesis, School of Computer Science and Engineering, University of NSW (2008)
12. Gast, H., Trieblinger, J.: High-level Reasoning about Low-level Programs. In Roggenbach, M., ed.: Automated Verification of Critical Systems 2009. Volume 23 of Electronic Communications of the EASST., EASST (2009)
13. Tuch, H., Klein, G., Norrish, M.: 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>.
14. Gast, H.: Lightweight separation. In Ait Mohamed, O., Munoz, C., Tahar, S., eds.: Theorem Proving in Higher Order Logics 21st International Conference (TPHOLs 2008). Volume 5170 of LNCS., Springer (2008)
15. Gast, H.: Reasoning about memory layouts. Formal Methods in System Design **37**(2-3) (2010) 141–170
16. Austern, M.H.: Generic Programming and the STL — using and extending the C++ Standard Template Library. Addison-Wesley (1998)
17. Kammüller, F., Wenzel, M., Paulson, L.C.: Locales—A sectioning concept for Isabelle. In Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L., eds.: TPHOLs. Volume 1690 of LNCS., Springer (1999) 149–166
18. Mehta, F., Nipkow, T.: Proving pointer programs in higher-order logic. Inf. Comput. **199**(1–2) (2005) 200–227
19. Gast, H.: Verifying the L4 kernel allocator in lightweight separation (2010) <http://www-pu.informatik.uni-tuebingen.de/users/gast/proofs/kalloc.pdf>.
20. Kassios, I.T.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Misra, J., Nipkow, T., Sekerinski, E., eds.: FM. Volume 4085 of LNCS., Springer (2006) 268–283
21. Dawson, J.E.: Isabelle theories for machine words. In: 7th International Workshop on Automated Verification of Critical Systems (AVOCS'07). Volume 250 of ENTCS. (2009)
22. Marti, N., Affeldt, R., Yonezawa, A.: Formal verification of the heap manager of an operating system using separation logic. In Liu, Z., He, J., eds.: ICFEM. Volume 4260 of LNCS., Springer (2006) 400–419
23. Myreen, M.O.: Formal verification of machine-code programs. PhD thesis, University of Cambridge (2009) UCAM-CL-TR-765.
24. McCreight, A.: The Mechanized Verification of Garbage Collector Implementations. PhD thesis, Department of Computer Science, Yale University (2008)
25. Tuerk, T.: A formalisation of Smallfoot in HOL. In Berghofer, S., Nipkow, T., Urban, C., Wenzel, M., eds.: Theorem Proving in Higher Order Logics 22nd International Conference (TPHOLs 2009). Volume 5674 of LNCS., Springer (2009)
26. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P., eds.: FMCO. Volume 4111 of LNCS., Springer (2005)
27. Gast, H.: Developer-oriented correctness proofs: A case study of Cheney's algorithm. In Qin, S., Qiu, Z., eds.: Proceedings of 13th International Conference on Formal Engineering Methods (ICFEM 2011). LNCS, Springer (2011) (to appear).
28. Gast, H.: A developer-oriented proof of Cheney's algorithm (2011) <http://www-pu.informatik.uni-tuebingen.de/users/gast/proofs/cheney.pdf>.