

Developer-oriented Correctness Proofs

A Case Study of Cheney’s Algorithm

Holger Gast

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

Abstract. This paper examines the problem of structuring proofs in functional software verification from a novel perspective. By aligning the proofs with the operational behaviour of the program, we allow the formalization of the underlying concepts and their properties to reflect informal correctness arguments. By splitting the proof along the different aspects of the code, we achieve re-use of both theories and proof strategies across algorithms, thus enabling reasoning by analogy as employed in software construction. We demonstrate the viability and usefulness of the approach using a low-level C implementation of Cheney’s algorithm.

1 Introduction

Proofs in functional software verification are usually complex and technically involved. In the case of automatic verification, they require the maintenance of abstract state in ghost-variables (e.g. [1, 2]), auxiliary intermediate assertions (e.g. [3]), and suitable axioms and triggers to guide the specific prover ([4]; e.g. [2, §4.3], [1, §7]). While interactive software verification can in principle lead to more readable and understandable proofs (e.g. [5, §1.2], [6]) even in the most carefully structured larger case studies (e.g. [7–10]), the presentation is limited to the specifications and invariants, while proofs are expressly excluded ([7, 8]), or are only surveyed at a very high level (e.g. [10]). The implicit underlying strategy is to reduce most occurring goals to forms that the available automatic provers can handle (e.g. [7, 10]). Even allowing for necessary abbreviations for space reasons, the gap between the concrete code and the formal proof is still tremendous and often needs to be bridged at a technical level.

This paper proposes to approach the problem of finding and structuring correctness proofs from the developer’s (or software engineer’s) perspective. The immediate motivation for this choice is the fact that developers seem to be able to produce code that is basically correct (except for failures in unforeseen circumstances), so their mode of reasoning can be considered overall successful. By emulating it at a more formal level, one would arrive at proofs that are more precise versions of the developers’ correctness arguments, thus bridging the gap between the formal and the informal.

We show that it is possible carry out this agenda by verifying a moderately complex algorithm, a C implementation of Cheney’s collector [11]. Herein, we apply the following strategies derived from a developer’s point of view.

- Developers rely on experience with similar algorithms. For a novel problem, they identify familiar parts or aspects and apply the corresponding coding idioms and reasoning patterns. We therefore structure the proof around previously known aspects: graphs of memory objects, including reachability, are shared with the Schorr-Waite algorithm [12] and the collector’s work queue is treated as a linked lists from [13, 14]. The algorithm’s distinguishing forwarding pointers reduce to the *map* datatype from the Isabelle/HOL library.
- A major strength of interactive theorem proving is the possibility to develop self-contained theories independently of concrete verification conditions (e.g. [5], [10], [6]). As an experiment, we have therefore formalized the mentioned aspects by considering the algorithm’s code and expected behaviour alone, without looking at the concrete verification conditions. The derived lemmas thus follow a developer’s mental execution of the code, and the actual verification was found to consist of applying these lemmas.
- We unify the overall proof by applying the *split/join* reasoning pattern throughout. The pattern complements the separation lemmas of classical approaches [6] and extends the automatic unfoldings of [15]: before the code manipulates a memory object, the pre-condition must be split such that separation lemmas prove the parts unchanged; for the post-condition, the split parts are joined together. Beyond automatic unfoldings, the formulation of split/join theorems reflects the information available at their point of application (e.g. §3.2) and boundary parameters (§2.5–2.7, §3.2) are introduced to enable the split. The reasoning pattern resembles a developer’s drawing of pointer diagrams to make explicit the manipulated entities.

The proof is carried out in lightweight separation, which is described in detail in [12, 16, 13] and is developed as a conservative extension of Isabelle/HOL. For the purposes of this paper, it can be treated as a usual verification environment based on Hoare logic. We introduce the notation and concepts as needed. The proof document is available from the author’s homepage [17].

The contributions of the paper are the following:

- The aspect-oriented proof style enables the re-use of extensive and non-trivial generic theories across algorithms that implement different specifications and only share the specific aspects of their correctness arguments. The re-use also comprises reasoning patterns building on the re-used theories.
- Since the aspects are mostly independent of one another, the different parts of the proof can equally be developed, understood, and maintained independently. We have found this useful in adjusting the details of the formalization during development, as is usually necessary for any larger case study.
- All parts of the proof exhibit a direct link to an informal understanding of the code and the high-level steps are connected by a few straightforward arguments about equalities, sets, and maps.

Overview Section 2 formalizes the different independent aspects of Cheney’s algorithm and exhibits the theory re-use from earlier developments. Section 3 assembles the aspects into a common invariant and correctness proof of the overall algorithm. Section 4 surveys related work. Section 5 concludes.

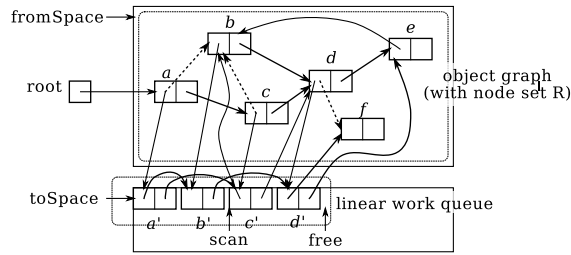


Fig. 1. Overview over the Collector State

2 Dissecting Cheney’s Algorithm

This section summarizes Cheney’s collector [11, 9, 18] and formalizes its three main aspects: the object graph, the work queue, and the forwarding map. The final aspect captures that the *to-space* is large enough for receiving all necessary copies of objects.

Following the earlier studies [9, 18], we restrict the algorithm to objects with two fields, which may contain pointers or atomic data, and to a single root reference. The machine representation of pointers, values, and objects follows [9].

2.1 The Three Main Aspects of the Algorithm

Figure 1 depicts the general idea of Cheney’s collector. The algorithm works on two equal-size half-spaces, the *from-space* and the *to-space*. (They are depicted here in different sizes to save vertical space.) At the beginning of a collection, the *from-space* contains all objects allocated by the program, while the *to-space* is empty. The collector’s task is to copy all objects still required by the program into the *to-space* and to declare the *from-space* unused.

The algorithm’s first aspect is the graph formed by the objects in the *from-space*, with the pointers in their fields as the successors. The graph is accessed from a single cell *root*, which represents the root set in the program’s run-time stack. The central point here is the notion of reachability: an object is required by the program iff it is reachable in the graph starting from the root cell.

Further, the algorithm uses the copied objects between the addresses *toSpace* and *free* as a work queue. These objects are divided by the *scan* pointer: the objects before *scan*, i.e. *a'*, *b'*, are processed completely, i.e. their fields already point to copies of their original successor objects. The objects after *scan*, i.e. *c'*, *d'*, have been copied but their fields have not been updated.

The final aspect of the algorithm is the forwarding map: objects in the *from-space* that have already been copied, i.e. objects *a* to *d*, contain a forwarding pointer to their copies in their first word. These pointers are crucial for handling aliasing and cycles. For instance, when processing *c'*, the algorithm finds that its successor *a* has already been copied because its first field is a pointer to the *to-space*, and it therefore sets the first field of *c'* to *a'*. By the same mechanism, the cycle from *e* to *b* will lead to a cycle from the later copy *e'* to *b'*.

```

1 void collect(void **r) {
2   void *tmp = fromSpace;
3   fromSpace = toSpace;
4   toSpace = tmp;
5   free = toSpace;
6   scan = free;
7
8   copy_ref(r);
9   while (scan != free) {
10    copy_ref((void**)scan);
11    copy_ref((void**)(scan + 4));
12    scan = scan + 8;
13  }
14 }

1 void copy_ref(void **p) {
2   if (*((int*)p) & 1 == 0 &&
3       *(void**)p != null) {
4     void *obj = *p;
5     int fwd = *(int*) obj;
6     if (fwd & 1 == 0 &&
7         toSpace <= (void*)fwd &&
8             (void*)fwd < toSpace+spaceSz){
9       *(void**)p = (void*)fwd;
10    } else {
11      void *newObj = free;
12      free = free + 8;
13      *(int*)newObj = *(int*)obj;
14      *(int*)(newObj + 4) =
15          *(int*)(obj + 4);
16      *(void**)obj = newObj;
17      *(void**)p = newObj;
18    } } }

```

Fig. 2. The Collector’s Code

2.2 Implementation

Figure 2 shows the collector’s code (in the C dialect of [12, 16]). It is organized around the idea of copying *references*, i.e. memory words that contain either an atomic value or a pointer to an object. Values and pointers are distinguished by the least-significant bit in their byte-representation [9]. Both the root cell and the two fields of an object constitute references in this sense, which is expressed in the factoring the basic step into the `copy_ref` function.

The function `copy_ref` mainly performs a case distinction. Lines 2–3 identify pointers by checking the bit-representation of the reference. Nothing is done for atomic values and the `null` pointer. Lines 4–8 read the first word of the referenced object and check for a forwarding pointer. If the object `obj` has already been copied, reference `p` is set to the existing copy (Line 9). The final case in Lines 11–17 allocates an object at `free`, copies `obj`, and sets `p` to the new object.

At an informal level, `copy_ref` is correct since it maintains the situation of Figure 1 and guarantees that after execution the reference `p` is handled completely, i.e. it contains the “correct” atomic value or the “correct” pointer to the copy of the original object. We will subsequently make this general idea precise enough for formal verification while maintaining the link to the informal view.

The `collect` function drives the overall process. It is called when the to-space is exhausted and the objects still used by the program must be copied to the empty from-space. Lines 2–6 reverse the roles of the half-spaces and initialize the collection. Line 8 copies the root reference, Lines 9–13 process newly copied objects until no more such objects remain. Intuitively, the loop invariant should be a formalization of Figure 1, which we achieve in the remainder of the paper.

2.3 Values, Pointers, and Objects

We will now make the informal view presented so far precise enough for formal verification, while maintaining close links to the informal arguments. We start by introducing some basic definitions that abstract over the low-level representation of values and pointers (`to-int` and `to-ptr` convert the byte-representation into a word [19] or the wrapper type `addr` [12], respectively).

$$\begin{aligned} \text{is-atomic } v &\equiv \text{to-int } v \text{ AND } 1 \neq 0 \vee \text{to-ptr } v = \text{null} \\ \text{is-ptr } v &\equiv \text{to-int } v \text{ AND } 1 = 0 \wedge \text{to-ptr } v \neq \text{null} \end{aligned}$$

The memory objects handled by the collector consist of two adjacent machine words. The constant `obj-fields` yields the byte-representations of these words in a list (`[]` denotes a HOL list; `rd ctx p t M` reads the byte-representation of type t at address p in memory M ; \oplus is address offset; ctx is the context containing the type definitions; `gctx` is a global context fixed for the development).

$$\text{obj-fields } p \equiv [\text{rd gctx } p \text{ TInt } M, \text{rd gctx } (p \oplus 4) \text{ TInt } M]$$

Lightweight separation [12] requires a symbolic formalization of the memory layout in the form of *covers*. An object is covered by a block of 8 consecutive bytes.

$$\text{obj-cover } p \equiv \text{block } p \ 8$$

To access the fields, we prove that an object may be split [12] into its two fields ($\langle p:t \rangle$ is a block at p with the size of type t ; \parallel is the disjoint union of covers).

$$\text{obj-cover } p = \langle p:\text{TInt} \rangle \parallel \langle p \oplus 4:\text{TInt} \rangle$$

The core idea of lightweight separation is to prove the result of memory accesses unchanged after a modification by the program by reasoning about the disjointness of covers [16]. We therefore show (automatically) that the memory accessor `obj-field p` depends only on the object at p .

$$\text{declare_accessor "obj-fields } p" "obj-cover } p"$$

Since the constants introduced subsequently have straightforward accessed regions and the proofs are automatic, we omit the corresponding declarations.

2.4 The Structure of the Half-Spaces

The collector's overall memory layout is given by global variables `toSpace`, `fromSpace`, `spaceSz`, and `free`. We therefore define constants to access them, according to this template (`rdv` is a `rd` with the address and type of a variable):

$$\text{toSpace } M \equiv \text{to-ptr } (\text{rdv } (\text{in-globals } \text{gctx}) \text{ "toSpace" } M)$$

The layout obeys the following invariant: `from-` and `to-space` are given by non-`null` pointers such that the spaces are contiguous and non-overlapping ($\{a..<b\}$ is the half-open interval $[a, b)$; `to-Ptr` converts an `addr` to its byte-representation).

$$\begin{aligned} \text{space-vars-inv } M &\equiv \\ &\text{fromSpace } M \neq \text{null} \wedge \text{toSpace } M \neq \text{null} \wedge \\ &\text{is-Ptr } (\text{to-Ptr } (\text{fromSpace } M)) \wedge \text{is-Ptr } (\text{to-Ptr } (\text{toSpace } M)) \wedge \\ &\text{fromSpace } M \leq \text{fromSpace } M \oplus \text{spaceSz } M \wedge \\ &\text{toSpace } M \leq \text{toSpace } M \oplus \text{spaceSz } M \wedge \\ &\{ \text{fromSpace } M .. < \text{fromSpace } M \oplus \text{spaceSz } M \} \\ &\cap \{ \text{toSpace } M .. < \text{toSpace } M \oplus \text{spaceSz } M \} = \{ \} \end{aligned}$$

Pointers into the to-space play a special role of forwarding pointers, and we introduce the following constants:

$\text{toSpace-range } M \equiv \{ \text{toSpace } M .. < \text{toSpace } M \oplus \text{spaceSz } M \}$
 $\text{to-space-ref } p \ M \equiv \text{is-ptr}(\text{rd gctx } p \ \text{TInt } M) \wedge \text{to-ptr}(\text{rd gctx } p \ \text{TInt } M) \in \text{toSpace-range } M$

2.5 Queue Structure

We now turn to the structure of the work queue. The algorithm splits the objects between `toSpace` and `free` into two groups at `scan` and advances `scan` linearly through the objects. It thus treats the copied objects as two linear lists, one of “jobs already done” and one of “jobs that need to be done”.

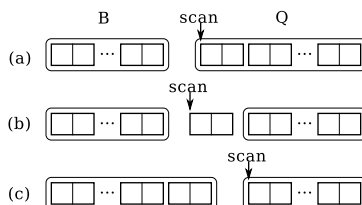
Linear lists are, of course, a common structure that developers are familiar with. We have therefore developed a generic library of lists [14], building on [13]. The library is expressed as a *locale* [20], Isabelle’s form of parameterized theories. The list locale takes two parameters: the cover `node` for the list nodes and a function `succ` reading the *next*-pointer from a node. The node cover is, of course, the `obj-cover` from §2.3; the *next*-link is computed from the objects’ size:¹

$\text{queue-succ } p \ M \equiv p \oplus 8$

The library furthermore assumes that the next-link only depends on the object under consideration, which is proven automatically. Re-using the library as a locale instance `q` takes 20 lines of straightforward Isabelle code; the queue-structure can then be expressed by the constant `nodes` in `q`: there are nodes `B` (for “black” [9]) and `Q` (for “queue”) such that

$q.\text{nodes}(\text{toSpace } M) (\text{scan } M) \ B \ M \wedge q.\text{nodes}(\text{scan } M) (\text{free } M) \ Q \ M \wedge \text{set } B \cap \text{set } Q = \{\}$

The library also provides theorems for reasoning about the defined lists. For instance, the loop in `collect` starts with configuration (a) below, then takes one object off the front of list `Q` in (b), works on it, and adds it to list `B` in (c).



These are very common operations lists. The library therefore provides theorem (1) for unfolding the first node of a lists and theorem (2) for folding a node into the end of a list. (The complementary theorems are available, too. The parameter `succ` will be replaced by `queue-succ` in the specific local instance.)

$$\frac{p \neq q}{\text{nodes } p \ q \ x \ M = (\exists y \ s. \ \text{nodes}(\text{succ } p \ M) \ q \ y \ M \wedge x \ s = p \ \# \ y \ s \wedge p \notin \text{set } y \ s)} \quad (1)$$

$$\frac{\text{nodes } p \ r \ y \ M \quad \text{succ } r \ M = q \quad q \notin \text{set } (y \ s @ [r])}{\text{nodes } p \ q \ (y \ s @ [r]) \ M} \quad (2)$$

¹ For objects of arbitrary length, the successor could be computed by reading the object’s header field. The used list structure is thus not overly general.

Note also that premise of (1) and the first premise of (2) directly relate to the while-test and operations of `collect`. The second premise of (2) is given by the disjointness condition in the above formalization of the work queue. By applying the theorems as suggested, both the structure of the queue and the correctness proofs are thus related to a developer’s familiar concept of lists.

2.6 The Object Graph

Developers naturally think of memory objects a set of entities that are linked by pointers. In programming languages with garbage collection, the runtime system guarantees that all objects reachable from the program variables will always be retained in memory and that there are no dangling pointers.

We make this general idea more precise by using a generic library from an earlier case study of the Schorr-Waite algorithm [12]. The locale parameters here are the object’s memory layout, which is given by the constant `obj-cover` (§2.5), and a function that reads the successor-pointers from a given object. That function is defined directly using `obj-fields` (§2.3): the object successors are precisely the pointers stored in the object fields. (Note that `is_ptr null` is false.)

$$\text{obj-succs } p \ M \equiv \text{map to-ptr (filter is_ptr (obj-fields } p \ M))$$

Creating the instance `g` of the object graph locale takes 16 lines. The library then provides extensive automatic proof support [12, §5.1] for establishing disjointness of sets of objects in the graph by set-theoretic arguments about their base-pointers and Burstall-style [21] disjointness proofs on individual fields.

We can now introduce and motivate the split/join proof strategy announced in §1. For the queue structure in §2.5, it was natural to split off the “current” node from the list `Q` to modify its content while standard separation lemmas [6] prove the remainder of `Q` unchanged. Analogously in the case of an object graph, assertions about reachability may be invalidated by modifications to individual objects. It therefore becomes necessary to split the graph at particular objects.

The splitting of lists into fragments is enabled by an explicit end node. The graph library thus introduces a *boundary set* `Q` into the definition of reachability: `reachable P Q R M` denotes that in memory state `M`, there are paths starting in set `P` to all nodes in set `R` that never touch `Q`. In particular, `Q` and `R` are disjoint.

The collector uses the reachability predicate in two places: first, the specification (§3.1) defines the set of objects `R` reachable in the initial state (with an empty boundary set). Second, and more interestingly, we can capture that all objects that have not been copied so far (predicate `forw-obj`, see §2.7) are reachable from the work queue, without ever crossing an already copied object:

$$\text{g.reachable (g.Succs (set } Q) \ M \cup \ S) \ \{p \in \ R. \ \text{forw-obj } p \ M\} \ \{p \in \ R. \ \neg \ \text{forw-obj } p \ M\} \ M$$

This formalization is particularly interesting as it mirrors the following conjunct of the Schorr-Waite loop invariant ([12, Figure 4], [6, i4,i6]): all unmarked nodes are reachable from the stack without crossing marked nodes (or `null` pointers).

$$\text{reachable } (\{t\} \cup \text{set (map } (\lambda n. \ \text{to-ptr (cell-r-rd } n \ M)) \ S)) \\ \{n. \ n = \ \text{null} \ \vee \ n \in \ N \ \wedge \ \text{marked } n \ M\} \ \{n \in \ N. \ \neg \ \text{marked } n \ M\} \ M$$

At this point, we thus see the re-use of a proof strategy between algorithms that are only linked by the aspects of a work queue and reachability in object graphs.

Using the boundary set, the library provides the splitting theorem (3) ([12, (44)]): the reachable set R can be split at some subset D into parts $R1$ and $R2$ and D . Note that both $R1$ and $R2$ are disjoint from D , such that nodes in D can be manipulated without influencing reachability within $R1$ and $R2$.

$$\frac{\text{reachable } \Gamma \text{ P Q R M} \quad D \subseteq R \quad D \cap Q = \{\}}{(\exists R1 \text{ R2. reachable } \Gamma \text{ P (Q} \cup \text{D) R1 M} \wedge \text{reachable } \Gamma \text{ (Succs } \Gamma \text{ D M) (Q} \cup \text{D) R2 M} \wedge R = R1 \cup R2 \cup \text{D})} \quad (3)$$

Conversely, two parts of a object graph thus created can be joined together:

$$\frac{\text{reachable P Q R M} \quad \text{reachable P' Q R' M}}{\text{reachable (P} \cup \text{P') Q (R} \cup \text{R') M}} \quad (4)$$

Both theorems apply in the verification of the `copy_ref` function: before setting the forwarding pointer of `obj` in Line 16, the reachability of un-copied nodes is split using (3) to expose the object `obj`. At the end, the split parts are joined by (4), using the fact that `obj` has now been copied.

This reasoning succeeded immediately, because it mirrors the Schorr-Waite proof [12, §5.2.4]. The proximity to an informal argument is also interesting: one would draw a pointer diagram, highlight the object to be manipulated, and think about the reachability by paths that might cross the object.

The specification of copying collectors rests on the notion of a *graph isomorphism* [18, 22, 9], i.e. a one-to-one mapping from the original object graph to its copy that respects the pointer structure. The definition can be given at the level of the existing graph library as follows, and it will be equally re-usable.

First, the isomorphism will map addresses, while object fields can contain either pointers or atomic values. Following [18], we hide the case distinction by an auxiliary function *pointer map*, which is the identity on atoms.

```
pmap  $\varphi \equiv \lambda v.$  if is-ptr v
      then (if to-ptr v  $\in$  dom  $\varphi$  then Some (to-Ptr (the ( $\varphi$  (to-ptr v)))) else None)
      else Some v
```

Then, a *morphism* is a mapping between object sets that respects the successor relations; an *isomorphism* is an injective morphism (\circ is function composition).

```
morph  $\varphi$  A fieldsA B fieldsB  $\equiv$ 
  dom  $\varphi = A \wedge$  ran  $\varphi = B \wedge$ 
  ( $\forall a \in A.$  fieldsB (the ( $\varphi$  a)) = map (the  $\circ$  pmap  $\varphi$ ) (fieldsA a))
iso  $\varphi$  A fieldsA B fieldsB  $\equiv$  morph  $\varphi$  A fieldsA B fieldsB  $\wedge$  inj-on  $\varphi$  A
```

2.7 The Forwarding Pointers

The forwarding pointers established by the algorithm are used to handle aliasing and cycles. Beyond this technical role, they bear a relation to the specification (cf. §3.1) as they constitute the graph isomorphism that a copying collector constructs [18, 22]. As this aspect touches all parts of Figure 1, it is the most complex one. Nevertheless, it can be tackled using the proposed split/join strategy and arguments following the algorithm's operational behaviour.

of the algorithm, as gleaned from the code.

$$\begin{array}{c}
\text{Forw } \varphi \text{ (R - \{p\}) B C Q M0 M} \\
\text{p} \in \text{R} \quad \text{forw-obj p M} \\
\text{obj-fields (to-ptr (rd ctx p TInt M)) M} = \text{obj-fields p M0} \\
\text{to-ptr (rd ctx p TInt M)} \notin \text{B} \cup \text{C} \cup \text{Q} \\
\text{same-static ctx gctx} \\
\hline
\text{Forw (extend-forw ctx } \varphi \text{ p M) R B C (Q} \cup \{ \text{to-ptr (rd ctx p TInt M)} \} \text{) M0 M}
\end{array} \tag{6}$$

The final code location concerned with the forwarding map is Line 12 of `collect`, which advances `scan` and thus moves the “current” object to the “black” part of the work queue (§2.5). At this point, both fields of the object at `scan` must be “completely processed”. Using the function `pmap` from §2.6, we can express this assertion concisely: a reference is “done”, under a given mapping φ , if in current state M its previous content in M' has been overwritten with the correct word:

$$\text{done-ref p } \varphi \text{ M' M} \equiv (\text{pmap } \varphi \text{ (rd gctx p TInt M')} = \text{Some (rd gctx p TInt M)})$$

The constant is, however, more than an auxiliary. Consider the two calls in the `collect` loop. Each guarantees in its post-condition `done_ref` for its passed object field, but each possibly extends the map φ . It therefore became clear immediately that (7) was needed at the end of the loop body; by proving it directly, we could check the background theory about the algorithm before attempting the verification. (\leq_m from the Isabelle library defines extensions of maps.)

$$\frac{\text{done-ref p } \varphi \text{ M' M} \quad \varphi \leq_m \varphi'}{\text{done-ref p } \varphi' \text{ M' M}} \tag{7}$$

Theorem (8) then verifies the correctness of Line 17 of `collect`: the forwarding map is left intact if both object fields are “done”:

$$\frac{\begin{array}{c} \text{Forw } \varphi \text{ R B \{p\} Q M0 M} \\ \text{obj-fields p M'} = \text{obj-fields (the (inv-map } \varphi \text{ p)) M0} \\ \text{done-ref p } \varphi \text{ M' M} \quad \text{done-ref (p} \oplus 4 \text{) } \varphi \text{ M' M} \end{array}}{\text{Forw } \varphi \text{ R (insert p B) \{\} Q M0 M}} \tag{8}$$

We wished to be sure early in proof development that the overall goal of constructing the graph isomorphism (§2.6) would be attained. Immediately after deriving the definition of `Forw` from Figure 1, we therefore proved (9) to ensure that the consistency conditions are strong enough: when the algorithm’s work queue becomes empty after the loop, and all objects have been forwarded, then the forwarding map is an isomorphism.

$$\frac{\text{Forw } \varphi \text{ R B \{\} \{\} M0 M} \quad \forall \text{p} \in \text{R. forw-obj p M}}{\text{iso } \varphi \text{ R (\lambda p. obj-fields p M0) B (\lambda p. obj-fields p M)}} \tag{9}$$

In summary, even the complex aspect of the forwarding map has been derivable from the code and informal arguments, without referring to the generated verification conditions. The search for `split/join` theorems has led to understandable reasoning steps, whose application during the verification is clear.

2.8 Remaining Free Space

The collector copies objects out of the from-space into the to-space. Since both spaces have the same size, the to-space will be large enough to receive all copies. In order to be satisfied that this reasoning will be sufficient, we proved (10) before proceeding further (`region-size A` is the size of the region covered by `A`; \triangleright denotes allocatedness [16]). Its premises reflect the situation in Line 11 of `copy_ref`: as an invariant, the space occupied by all non-copied objects is no greater than the remaining free-block (i.e. the remainder of the to-space after `free`); both the free-block and the non-copied objects are allocated; and some object `p` among these has not been copied. The conclusion asserts that one object can be extracted to maintain the invariant and to split the free-block; furthermore moving the free-pointer forward will leave it inside the to-space, ready for further allocations.

$$\begin{array}{l}
 \text{region-size } (g.\text{nodes } \{p \in R. \neg \text{forw-obj } p \ M\}) \leq \text{region-size } (\text{free-block } M) \\
 M \triangleright \text{free-block } M \parallel g.\text{nodes } \{p \in R. \neg \text{forw-obj } p \ M\} \\
 p \in R \ \neg \text{forw-obj } p \ M \\
 \hline
 \text{region-size } (g.\text{nodes } \{p' \in R. p' \neq p \wedge \neg \text{forw-obj } p' \ M\}) + 8 \leq \text{region-size } (\text{free-block } M) \wedge \\
 \text{free-block } M = \text{obj-cover } (\text{free } M) \\
 \parallel \text{block } (\text{free } M \oplus 8) (\text{spaceSz } M - (\text{free } M \ominus \text{toSpace } M) - 8) \wedge \\
 \text{free } M < \text{free } M \oplus 8 \wedge \text{free } M \oplus 8 \leq \text{toSpace } M \oplus \text{spaceSz } M
 \end{array}
 \tag{10}$$

3 Assembling the Correctness Proof

Section 2 has formalized the aspects of Cheney’s algorithm along with theorems for reasoning about them, and has indicated precisely how to apply the theorems in verification. It remains to assemble the available parts into the correctness proof. This mode of presentation reflects our way of proceeding: we finished most of the base work before being concerned with the verification conditions.

3.1 The Specification of `collect`

The pre-condition of the collector is the expected one: the memory consists of the two half-spaces (§2.4) and the objects `R` reachable from the root `r` (§2.6) are contained in the (exhausted) to-space. Finally, the two spaces are laid out according to §2.4, the global definitions in the current context match those in `gctx`, and the auxiliary (or logical) variable `M0` is bound to the initial state

(\blacktriangleright denotes that the cover describes the entire memory layout):

$$\begin{array}{l}
 M \blacktriangleright \ll r:\text{TInt} \gg \parallel \text{gc-vars} \parallel \text{to-block } M \parallel \text{from-block } M \wedge \\
 g.\text{reachable } (\text{ref-set } r \ M) \ \{ \} \ R \ M \wedge g.\text{nodes } R \preceq \text{to-block } M \wedge \\
 \text{space-vars-inv } M \wedge \text{same-static } \text{ctx } g\text{ctx} \wedge M0 ::= M
 \end{array}$$

The post-condition specifies the same memory layout, but the object graph is copied to the new to-space into a set `R'` of objects, whose graph structure is isomorphic to that of the initial graph `R` [18, 22]:

$$\begin{array}{l}
 M \blacktriangleright \ll R0:\text{TInt} \gg \parallel \text{gc-vars} \parallel \text{to-block } M \parallel \text{from-block } M \wedge \text{space-vars-inv } M \wedge \\
 (\exists \varphi \ R'. \text{iso } \varphi \ R \ (\lambda p. \text{obj-fields } p \ M0) \ R' \ (\lambda p. \text{obj-fields } p \ M) \wedge \\
 g.\text{nodes } R' \preceq \text{to-block } M)
 \end{array}$$

$$\begin{aligned}
& \text{gc-inv root sc sc' R S B C Q } \varphi \text{ M0 M } \equiv \\
& \text{space-vars-inv M } \wedge \\
& \text{is-Ptr (to-Ptr (free M)) } \wedge \text{ free M } \in \{ \text{toSpace M .. toSpace M } \oplus \text{ spaceSz M } \} \wedge \\
& \text{q.nodes (toSpace M) sc B M } \wedge \text{ q.nodes sc' (free M) Q M } \wedge \\
& \text{set B } \cap (\text{set C } \cup \text{set Q}) = \{ \} \wedge \text{set C } \cap \text{set Q} = \{ \} \wedge \\
& (\text{set B } \cup \text{set C } \cup \text{set Q}) \subseteq \{ .. < \text{free M} \} \wedge \\
& \text{Forw } \varphi \text{ R (set B) (set C) (set Q) M0 M } \wedge \\
& \text{g.reachable (ref-set root M0) } \{ \} \text{ R M0 } \wedge \\
& \text{g.nodes R } \preceq \text{ from-block M } \wedge \\
& \text{to-block M} = \text{g.nodes (set B } \cup \text{set C } \cup \text{set Q) } \parallel \text{ free-block M } \wedge \\
& \text{g.reachable (g.Succs (set Q) M US) } \{ p \in \text{R. forw-obj p M} \} \{ p \in \text{R. } \neg \text{forw-obj p M} \} \text{ M } \wedge \\
& \text{region-size (g.nodes } \{ p \in \text{R. } \neg \text{forw-obj p M} \}) \leq \text{region-size (free-block M) } \wedge \\
& (\forall p \in \text{R. } \neg \text{forw-obj p M} \longrightarrow \text{obj-fields p M} = \text{obj-fields p M0})
\end{aligned}$$

Fig. 3. The Collector's Invariant

3.2 The Loop Invariant and Proof of collect

The main invariant of `collect` (Figure 3), which together with the memory layout from the pre-condition forms the loop invariant, relates the parts of Figure 1 and has four further parameters `S`, `C`, `sc` and `sc'`, which act as boundaries in the formulation of split/join theorems. The invariant gathers the aspects from §2: it captures the half-spaces (§2.4) with the additional `free` pointer and the queue structure (§2.5), whose fragments are delimited by the boundary parameters. In the forwarding map, it leaves `C` open in the same way as §2.7. It keeps the definition of `R` as the set of objects reachable in the initial state `M0`, and the split of the (swapped) from- and to-spaces. Finally, the invariant on sizes (§2.8) and the reachability of un-copied nodes (§2.6) guarantee that the algorithm can make progress. The un-copied objects remain unmodified from the initial state, which again imitates the invariant of the Schorr-Waite algorithm [12, 6].

Like the separate aspects, the invariant enjoys split and join theorems that guide the verification. The split theorem (11) is applied at the beginning of the `collect` loop body, where the invariant holds and the queue is not empty. It moves the object `scan` to the current nodes `C` and makes explicit the information that was known before: its successors may lead to un-copied nodes, and its fields are just copied from some original object (given by φ^{-1}), such that the successors are contained in `R` (by reachable sets being closed).

$$\frac{\text{gc-inv root (scan M) (scan M) R S B [] Q } \varphi \text{ M0 M } \quad \text{scan M} \neq \text{free M}}{\exists \text{Q'. gc-inv root (scan M) (queue-succ (scan M) M) R} \\
\text{(S } \cup \text{set (obj-succs (scan M) M)) B [scan M] Q' } \varphi \text{ M0 M } \wedge \quad (11) \\
\text{Q} = \text{scan M} \# \text{Q' } \wedge \text{scan M} \notin \text{set Q' } \wedge \\
\text{obj-fields (scan M) M} = \text{obj-fields (the (inv-map } \varphi \text{ (scan M))) M0 } \wedge \\
\text{set (obj-succs (scan M) M)} \subseteq \text{R}}$$

After the loop body, the `scan` object is re-integrated into the overall structure by the join theorem (12). Therein, `M'` designates the memory state before the loop body, such that the first three premises mirror the conclusion of the split

theorem (11) and the advancing of `scan` (Line 12). The real proof obligation is in the last line: both fields of the object must be “done” (§2.7). In the conclusion, the `scan` object is now “black”.

$$\begin{array}{l}
\text{gc-inv root (scan M')} \text{ (scan M) R (set (obj-succs (scan M') M')) B [scan M'] Q } \varphi \text{ M0 M} \\
\text{obj-fields (scan M')} \text{ M}' = \text{obj-fields (the (inv-map } \varphi \text{ (scan M')))} \text{ M0} \\
\text{scan M} = \text{scan M}' \oplus 8 \\
\text{done-ref (scan M')} \varphi \text{ M}' \text{ M} \quad \text{done-ref (scan M}' \oplus 4) \varphi \text{ M}' \text{ M} \\
\hline
\text{gc-inv root (scan M) (scan M) R \{\} (B @ [scan M']) \} \parallel \text{Q } \varphi \text{ M0 M}
\end{array} \tag{12}$$

The proof of both theorems (11) and (12) are direct using the split/join theorems for the constituent aspects from §2. For instance, the last two premises of (12) are obviously needed to apply (8).

The verification of `collect` is now clear: in the beginning, `gc-inv` holds trivially for the empty queue and empty forwarding map. The helper `copy_ref` will be proven in §3.3 to maintain `gc-inv` and to guarantee that its argument is a `done-ref` (§2.7) after execution. For `collect`, we therefore split `gc-inv` at the beginning of the loop body by (11) and join it after the two calls by (12) (using (7)).

In summary, the overall correctness proof is only an application of reasoning steps derived from informal arguments of the code’s behaviour and consistency conditions in §2, and the developer’s intention of factoring out the copying of a single reference is reflected in the proof structure.

3.3 The `copy_ref` Function

Informally, the auxiliary function `copy_ref` must process a single reference completely, i.e. convert it to a `done-ref` (§2.7), without violating `gc-inv`. Its specification merely makes this notion precise. The pre-condition assumes the invariant memory layout and `gc-inv`, and binds a few auxiliary variables to initial values.

$$\begin{array}{l}
\exists \text{'B Q. gc-inv root sc sc' R S B C Q } \varphi \text{ M0 M} \wedge \text{proper-ref p R M} \wedge \\
\text{M} \blacktriangleright \text{gc-vars} \parallel \text{g.nodes R} \parallel \text{g.nodes (set B} \cup \text{set Q)} \parallel \text{free-block M} \parallel \langle \text{p:TInt} \rangle \parallel \text{F} \wedge \\
\text{same-static ctx gctx} \wedge \text{P0} := \text{p} \wedge \text{M1} ::= \text{M} \wedge \text{wf-cover F}
\end{array}$$

The post-condition asserts `gc-inv` (albeit with a possibly changed queue) and an unmodified layout (although the free-block may be different). Furthermore, it has processed the given reference as required (§2.7), has at most extended the forwarding map, and has not modified the space variables, the pointer `scan` and `F`, which contains the remainder of the memory (the frame conjunct [12]).

$$\begin{array}{l}
\exists \text{'B Q } \varphi'. \text{gc-inv root sc sc' R S B C Q } \varphi' \text{ M0 M} \wedge \\
\text{M} \blacktriangleright \text{gc-vars} \parallel \text{g.nodes R} \parallel \text{g.nodes (set B} \cup \text{set Q)} \parallel \text{free-block M} \parallel \langle \text{P0:TInt} \rangle \parallel \text{F} \wedge \\
\text{done-ref P0 } \varphi' \text{ M1 M} \wedge \text{map-le } \varphi \varphi' \wedge \\
\text{frame (space-vars} \parallel \text{gvar-block "scan"} \parallel \text{F)} \text{ M1 M}
\end{array}$$

As in the case of `collect`, the correctness proof of `copy_ref` follows the indications from §2: in the first case (Lines 7–9), only the reference `p` is modified, and the post-condition can be derived by extracting knowledge about the copied pointer from `gc-inv`. The second case (Lines 11–17) is the interesting one: we split the graph `R` by (3), the forwarding map by (5), and the free-block by (10). To derive the post-condition, the complementary join theorems (4) and (6) are invoked to re-assemble `gc-inv` and the post-condition.

4 Related Work

We have proposed strategies for structuring a correctness proof from a developer’s perspective. To the best of our knowledge, neither the alignment of the proof with different aspects of the algorithm, nor the unifying strategy of high-level split/join theorems with dedicated boundary parameters have been discussed before. Further, no other study has attempted theory re-use across algorithms through generic formalizations of shared aspects. We now discuss related case studies on garbage collectors, focussing on recent low-level implementations.

Myreen [7] verifies Cheney’s collector by refinement, proving successively more detailed specifications correct relative to the previous one. The intention is to re-use higher levels, which express the core of copying collection, to verify other collectors. This re-use is, however, not demonstrated, and it applies only to algorithms with the same specification. Our approach of factoring the proof into aspects has enabled the re-use of extensive theories across algorithms with different specifications. Myreen explicitly excludes the discussion of the proofs beyond mentioning that his approach is a reduction to set-theoretic arguments.

Varming/Birkedal [23, §4.1] formalize the earlier pen-and-paper development [18] in Isabelle/HOLCF based on higher-order separation logic. Their structure of the invariants and specifications is complementary to ours, in that they use the separating conjunction to specify disjoint parts of the heap independently, while we formalize the unifying aspects across the entire heap. Their definitions of the invariants refer to the memory content directly, and no intermediate levels are introduced. The overall structure of the substantial development is not discussed.

McCreight [9, Ch. 6] gives a detailed proof of Cheney’s collector in separation logic. He structures the specifications and invariants carefully by a number of auxiliary predicates and motivates their definitions in relation to the code (e.g. [9, §6.2, Fig. 6.7]). His loop invariant is developed in several steps (Figs. 6.16, 6.17, 6.18, 6.21). Following the idea of local reasoning, each definition leaves open a parameter for the remainder of the memory, which is instantiated in the subsequent definitions. The development is thus, again, complementary to ours by splitting the assertions along the memory layout, as prescribed by separation logic. Independent lemmas about the auxiliary predicates are not discussed, and McCreight mentions (§6.3.3; p. 122; §6.4.3) that the actual proofs involve a substantial amount of manual, low-level manipulation.

Hawblitzel and Petrank [2] verify several practical collectors using the Boogie/Z3 tool-suite. The collectors are written in BoogiePL, are translated to assembly language, and can be applied to existing benchmarks. Their specifications follow the framework [22], but exclude the central aspect of reachability [2, §4.1.1]. The proofs are discussed briefly along the verification conditions, which are also related back to the code. For the SMT prover to succeed, however, a substantial amount of further annotations as well as detailed technical considerations on triggers [2, §4.3] are necessary.

Mehta and Nipkow [6] verify the related Schorr-Waite graph marking algorithm. Their elegant Isar proof [6, §7.2] is discussed down to individual verification conditions. However, no attempt at structuring the proof further is made

and the level of detail is limited by the used high-level language. Hubert and Marché [10] verify a C implementation of the same algorithm. Their description of the proof in [10, §4] is limited to the explanation of the invariant.

5 Conclusion

We have approached the verification of a low-level C implementation of Cheney’s collector from a developer’s perspective: by formalizing and reasoning about the different aspects of the algorithm independently, by choosing the aspects to be familiar from other contexts, and by unifying the development using the introduced split/join reasoning pattern, a strong relationship between the proof and an informal understanding of the code’s operational behaviour was maintained throughout. The verification then consisted in applying the derived lemmas, and the overall proof appears as a precise version of informal correctness arguments.

The development [17] consists of 2680 lines, which is comparable to [7] and substantially smaller than [9]. Of these lines, 1230 are re-used from previous studies (lists: 330; object graphs: 650 (+750 ML); byte-level memory: 250). The new part consists of 290 lines for basic definitions and library instantiation (§2.4–§2.6); 560 for proofs about the aspects (§2.5–2.8); 600 for the verification conditions (§3). This relative distribution underlines the degree of re-use, as well as the largely independent treatment of the identified aspects of the algorithm.

The proposed approach has shown several benefits. First, we have achieved high-level proof re-use across different algorithms: two aspects, the work queue and reachability in object graphs, were solved completely by generic theories from previous developments. Isabelle locales were found to be a suitable mechanism for accomplishing such re-use. Second, the strong relation between the proof structure and the code greatly aided the development, since proofs could be derived from the informal correctness arguments as used by developers. Finally, the fact that the reasoning about the different aspects is independent has enhanced the maintainability of the proof during development.

We have chosen an interactive prover for our development because of its support for structured theory development. The application of the derived lemmas was, however, mostly automatic and consisted in discharging side-conditions in set-theory. As future work, we therefore propose to investigate the application of our approach in the context of the Boogie and SMT solver integration [5] available with the current Isabelle distribution.

References

1. 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
2. Hawblitzel, C., Petrank, E.: Automated verification of practical garbage collectors. SIGPLAN Not. **44**(1) (2009) 441–453

3. Zee, K., Kuncak, V., Rinard, M.: Full functional verification of linked data structures. *SIGPLAN Not.* **43**(6) (2008) 349–361
4. Moskal, M.: Programming with triggers. In Dutertre, B., Strichman, O., eds.: *SMT '09: Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, ACM (2009)
5. 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
6. Mehta, F., Nipkow, T.: Proving pointer programs in higher-order logic. *Inf. Comput.* **199**(1–2) (2005) 200–227
7. Myreen, M.O.: Reusable verification of a copying collector. In: *Proceedings of the Third international conference on verified software: theories, tools, experiments (VSTTE '10)*. Volume 6217 of LNCS., Springer (2010)
8. 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
9. McCreight, A.: *The Mechanized Verification of Garbage Collector Implementations*. PhD thesis, Department of Computer Science, Yale University (2008)
10. 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)
11. Cheney, C.J.: A nonrecursive list compacting algorithm. *Commun. ACM* **13** (1970) 677–678
12. Gast, H.: Reasoning about memory layouts. *Formal Methods in System Design* **37**(2-3) (2010) 141–170
13. Gast, H., Trieflinger, 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)
14. Gast, H.: Verifying the L4 kernel allocator in lightweight separation (2010) <http://www-pu.informatik.uni-tuebingen.de/users/gast/proofs/kalloc.pdf>.
15. 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)
16. 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)
17. Gast, H.: A developer-oriented proof of Cheney's algorithm (2011) <http://www-pu.informatik.uni-tuebingen.de/users/gast/proofs/cheney.pdf>.
18. Torp-Smith, N., Birkedal, L., Reynolds, J.C.: Local reasoning about a copying garbage collector. *ACM Trans. Program. Lang. Syst.* **30**(4) (2008) 1–58
19. 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)
20. 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
21. Burstall, R.: Some techniques for proving correctness of programs which alter data structures. In Meltzer, B., Michie, D., eds.: *Machine Intelligence*. Number 7. Edinburgh University Press (1972)
22. McCreight, A., Shao, Z., Lin, C., Li, L.: A general framework for certifying garbage collectors and their mutators. *SIGPLAN Not.* **42**(6) (2007) 468–479
23. Varming, C., Birkedal, L.: Higher-order separation logic in Isabelle/HOLCF. *Electron. Notes Theor. Comput. Sci.* **218** (2008) 371–389