Electronic Communications of the EASST
Volume 53 (2012)

Proceedings of the 12th International Workshop on Automated Verification of Critical Systems (AVoCS 2012)

Semi-automatic Proofs about Object Graphs in Separation Logic

Holger Gast

15 pages

Guest Editors: Gerald Lüttgen, Stephan Merz
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Semi-automatic Proofs about Object Graphs in Separation Logic

Holger Gast

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

Abstract: Published correctness proofs of garbage collectors in separation logic to date depend on extensive manual, interactive formula manipulations. This paper shows that the approach of symbolic execution in separation logic, as first developed by Smallfoot, also encompasses reasoning about object graphs given by the reachability of objects. This approach yields semi-automatic proofs of two central garbage collection algorithms: Schorr-Waite graph marking and Cheney's collector. Our framework is developed as a conservative extension of Isabelle/HOL. Our verification environment re-uses the Simpl framework for classical Hoare logic.

Keywords: Separation logic, symbolic execution, garbage collectors

1 Introduction

Symbolic execution in separation logic [BCO05b, BCO05a], has now been adopted by many authors as an effective approach to functional software verification. Several tools [JSP10, DJ08, BPS09] aim at fully-automated verification using SMT-solvers for pure side-conditions. In the area of interactive theorem proving, the studies [Chl11, AB07, McC09, Tue09] have shown that with some user assistance, higher-order constructs are admissible in the pure part of the specification, and low-level programs with complex specifications can be verified. Here, symbolic execution is guided by the user in proof scripts of the underlying prover.

In the important area of verifying garbage collectors, however, proofs remain cumbersome and detailed. McCreight et al. [MSLL07, McC08] report that despite the use of specialized tactics [McC09], proofs still involve a fair amount of low-level interactive manipulation of spatial formulae [McC08, §6.3.3; p. 122; §6.4.3]. Varming and Birkedal [VB08] likewise employ detailed proof scripts. In his recent refinement-based approach, Myreen [Myr10] reduces the proof size by using a lightweight variant of separation logic and expressing the verification conditions in set theory, with the disadvantage of losing the potential benefits of separation logic.

This paper proposes to overcome the outlined gap by implementing a framework in which garbage collection algorithms can be verified by symbolic execution, in the mostly-automated style of Tuerk [Tue09] and Chlipala [Chl11].

Symbolic execution in separation logic rests on the availability of suitable abstraction predicates to specify the memory content, together with structural unfolding rules for reasoning about these predicates according to the heap manipulations performed by the program. Programs that work on linked lists, for instance, usually add and remove elements at the beginning of a list. Given a predicate node p v yielding an HOL record value v for a list node and a predicate list p xs q for the list between p and q with content xs, the required unfoldings are ($\barwedge$ is a conjunction

with a pure assertion inside a spatial assertion; $\bar{\exists}$ is existential quantification in heap assertions; # denotes cons on Isabelle/HOL lists; $(\!|\ |\!)$ is a record value):

$$\frac{p \neq q}{\begin{array}{l}\text{list p xs q} = (\bar{\exists}\text{v xs'. node p v} \star \text{list (node-next v) xs' q} \mathbin{\bar{\wedge}} \text{(xs = node-data v \# xs')}) \\ \text{node p v} = (\bar{\exists}\text{d n. p+off-data}\mapsto\text{d} \star \text{p+off-next}\mapsto\text{n} \mathbin{\bar{\wedge}} \text{v=(}\!|\text{node-data=d, node-next=n}|\!)\text{)}\end{array}}$$

All such unfoldings keep assertions in the form of *symbolic heaps* $\bar{\exists} x_1 \ldots x_n. \Sigma \mathbin{\bar{\wedge}} \Pi$ [BCO05b], where Σ is an iterated spatial conjunction of atomic predicates and Π is a conjunction of pure side-conditions on addresses and the values extracted from memory in Σ.

Automatic unfolding (e.g. [BCO05b, §3.2,§4]; [BPS09, §6.1]) enables programs to shift memory content between different atomic predicates in the spatial part of the symbolic heap. The loop invariant of the standard list reversal example, e.g., contains

list p xs NULL $\star$ list q ys NULL

where the list at p contains the remaining input and the list at q is the current partial result. The loop body sets the next field of node p to q, which leads to:

p+off-data$\mapsto$d $\star$ p+off-next$\mapsto$q $\star$ list q ys NULL $\star$ list (node-next v) xs' NULL

To re-establish the invariant, the node p is folded into the list at q, by applying the above unfolding backwards to the target assertion; all side-conditions of unfolding rules are solved by equational reasoning. The proof thus follows the program and shifts a node from the input to the output list.

This paper therefore provides a collection of predicates and unfolding rules that enable the effective specification and verification of garbage collection algorithms. The algorithms are perceived as working on object collections $F \star R \star Q \star U$, where F are the finished objects, R the remaining objects and Q the objects currently being processed; the input objects U are unreachable from the given root set. The collector repeatedly extracts some object from Q, examines it, possibly moves further objects from R to Q, and moves it to F. The central insight is that the notion of reachability, which is necessarily at the core of the loop invariants (e.g. [MN05]), can be hidden in the predicates' definitions and need not be considered during verification. We demonstrate the effectiveness of the approach by two examples, Schorr-Waite graph marking and Cheney's copying collector. The development is available from the author's homepage [Gas12].

Contributions This paper makes two contributions: first, we mechanize in Isabelle/HOL a theory that enables a semi-automatic verification of garbage collection algorithms by symbolic execution in separation logic. While the definitions and meta-theory are higher-order, the predicates themselves could also be axiomatized in first-order logic, thus being in principle applicable in existing tools. Second, the construction of our verification environment pioneers the re-use of a verification condition generator for a classical Hoare logic [Sch05]. We show that the generated proof obligations can be pre-processed to enable symbolic execution in separation logic. The transfer of the technique to other verification tools (e.g. [NIC12]) is left as future work.

Structure of the Paper Section 2 briefly summarizes the basic definitions of separation logic and a rewriting setup for reasoning about maps (partial functions). Section 3 contains the main contribution, the meta-theory of object graphs suitable for symbolic execution. Section 4 describes the verification environment. Section 5 gives the application to the two benchmark examples. Section 6 discusses related work. Section 7 concludes.

2 Basic Definitions

2.1 Heaps and Heap Assertions

We model the heap as a map (a partial function in Isabelle/HOL) addr $\rightharpoonup$ byte, where partiality indicates allocatedness and addr is a 32-bit machine word [Daw09]. Typed access uses a shallow representation of C types in HOL: for a type with abstract HOL values of type 'a, we define a record constant of type 'a cty with three fields ty-rep, ty-val, and ty-size. The function ty-rep yields the byte representation of the HOL value, whose length must match ty-size and must be representable as a machine word. The function ty-val interprets the byte representation as a HOL value of type 'a and must be the left inverse of ty-rep. The consistency conditions are expressed by predicate ty-ok. We provide the usual primitive types and a command `structdef` that derives the necessary definitions, theorems, and unfolding rules (§2.2) from a C struct definition.

Subsequently, all heap accesses are expressed by three constants: heap-get ct p h interprets the content of heap h at p as a representation of a value of type ct. Conversely, heap-put ct p v h yields an updated heap where the byte representation of v is stored at p. Finally, heap-acc-cond ct p h states that in heap h a contiguous region of memory large enough for values of type ct is allocated at p.

Our definitions for heap assertions follow [Web04]. As usual in higher-order settings, heap assertions are functions from heaps to bool, which we abbreviate as the type hassn. P h then states that P holds on h. $p \mapsto_{ct} v$ denotes that p points to the memory representation of a HOL value v of type ct. The definitions of $P \star Q$ and emp are standard. To capture symbolic heaps [BCO05b], we introduce a conjunction $P \mathbin{\bar{\wedge}} Q$ of a spatial assertion P with a pure assertion Q, and existential quantification $\bar{\exists} x.\ P\ x$, where P is a spatial assertion. Rewrite rules normalize symbolic heaps into the form $\bar{\exists} x_1 \ldots x_n.\ P_1 \star \ldots \star P_m \mathbin{\bar{\wedge}} Q_1 \wedge \ldots \wedge Q_k$.

2.2 Tactics for Heap Assertions

Several recent studies [McC09, Tue09, Chl11] have explored the combination of automatic symbolic execution and interactive proofs for pure side-conditions. Our development follows the general setup of [McC09] (and [LW09]), which also underlies [Chl11, §3]: the current state of the execution is represented as an Isabelle goal $[\![P\ h; Q_1; \ldots; Q_m]\!] \Longrightarrow R$, where P is the spatial part of the assertion about heap h and $Q_1 \ldots Q_m$ are the currently valid side-conditions. Under these hypotheses, R is to be proven.

If R is a symbolic heap, then the tactic `heap` attempts to cancel the components of R against those of P, leaving behind any remainder to enable further user interaction (see [McC09, Chl11]). During the process, it takes into account unfolding rules of the form (following [Gas10]):

$$\frac{Q_1 \ldots Q_m}{p\ x_1 \ldots x_n = (\bar{\exists}\ y_1 \ldots y_m.\ P'_1 \star \ldots \star P'_k \mathbin{\bar{\wedge}} R)}$$

If the pure pre-conditions $Q_1 \ldots Q_m$ can be proven, then a component with predicate p is unfolded into $P'_1 \star \ldots \star P'_k$. The rules are applied both to the hypothesis P and the conclusion R (where the existentially bound variables become unification variables, as usual). Symbolic execution unfolds in P to expose the heap location accessed by the program. New unfolding rules can proven as lemmas and declared to `heap` by the attribute `sep_unfold`.

Heap assertions give rise to implicit inequalities and disjointness statements (e.g. [BCO05b, Table 1]). We have implemented an extension to Isabelle's rewriting engine (a *simproc*) that derives such implicit assertions. For any predicate p, the user can declare (attribute `heap_addrs`) a theorem of the following form, stating that the sets Xs are disjoint if p $x_1 \ldots x_n$ holds on a heap h:

$$p\ x_1 \ldots x_n\ h \Longrightarrow \text{disjoint-list Xs h}$$

A separate tactic `heap_sc` applies explicitly given theorems to derive information about the $x_1 \ldots x_n$ from a component p $x_1 \ldots x_n$ h, and makes this information available as new hypotheses. It is used, e.g., to make explicit the side-conditions of object collections and graphs (§3.1, §3.3).

2.3 Isabelle/HOL Maps

We introduce the following auxiliary all-quantifier over the content of maps:

$$\forall x{\mapsto}y \in f.\ P \equiv \forall x\ y.\ f\ x = \text{Some}\ y \longrightarrow P\ x\ y$$

Apart from yielding a more direct expression of properties of maps, the new constant prevents Isabelle's simplifier from using premises about maps as rewrite rules, and thus makes its behaviour more predictable during verification. Furthermore, the rewrite rules (1) split the assertion about the map content according to the construction of the map. The premises of the rules will be solved by the implicit disjointness constraints derived from the current symbolic heap (§2.2).

$$\frac{}{(\forall x{\mapsto}y \in \text{Map.empty}.\ P\ x\ y) = \text{True}} \qquad \frac{}{(\forall x{\mapsto}y{\in}[x{\mapsto}y].\ P\ x\ y) = P\ x\ y}$$

$$\frac{x \notin \text{dom}\ f}{(\forall x{\mapsto}y \in (f(x{\mapsto}y)).\ P\ x\ y) = (P\ x\ y \wedge (\forall x{\mapsto}y{\in}f.\ P\ x\ y))}$$

$$\frac{\text{dom}\ f \cap \text{dom}\ g = \{\}}{(\forall x{\mapsto}y{\in}(f\ ++\ g).\ P\ x\ y) = ((\forall x{\mapsto}y{\in}\ g.\ P\ x\ y) \wedge (\forall x{\mapsto}y{\in}f.\ P\ x\ y))} \tag{1}$$

These rules then lead to a direct reflection of the program's heap manipulations at the abstract level of sets of objects (§5.2): when the program accesses an object from a set, symbolic execution splits the set, hence the abstract map value (§3.1). The simplifier then splits the assertion about the parts by (1); in particular, assertions about single objects will become available.

3 A Theory of Object Graphs

The specification of garbage collection algorithms focuses on the notion of reachability: given a set of root locations, retain those objects, called the *live* objects, that are reachable by iterated pointer dereferencing, and discard the remainder of the heap as free memory. The central conjunct of the loop invariants (e.g. [MN05, HM05, Gas11]) consequently states that all unprocessed live objects are reachable from the working queue (or stack) and will thus be processed in due course. Reasoning about reachability for verification is, however, subtle and requires much manual interaction.[1]

[1] Mehta and Nipkow [MN05] give a human-readable Isar proof that exhibits the intricacy. Hawblitzel et al. [HP09, §4.1.1.] substitute reachability with the weaker assertion that all reached objects have been handled correctly. Leino [Lei10] uses a closure property instead of reachability in the specification; although the property is equivalent, the equivalence is not verified mechanically.

This section therefore develops a meta-theory for reasoning about object graphs given by reachability through pointer dereferencing. In a first step, §3.1 defines sets of objects on the heap, a restricted form of the higher-order spatial quantifier $\bar{\forall} x \in A.\, P$ [VB08, McC08, Myr10]. Based on a suitable definition of reachability (§3.2), §3.3 defines a predicate for the graph of objects reachable from a given root set and proves unfolding rules suitable for symbolic execution.

3.1 Collections of Objects

The theory of object graphs is formulated as an Isabelle *locale* [KWP99] and is thus re-usable for different types of objects. A locale is a theory parameterized over constants and types, about which assumptions can be stated. The locale object-graph has two parameters: obj p r is the atomic spatial predicate for a single object at address p storing abstract value r. The function succs r will retrieve from the value r the contained object pointers. The type 'rep of the HOL values is also a parameter (indicated by the apostrophe in Isabelle).

```
obj   :: "addr ⇒ 'rep ⇒ hassn"
succs :: "'rep ⇒ addr set"
```

The locale makes two straightforward assumptions: that the memory footprint of an object contains its base address and that the null pointer is not contained in the footprint.

```
obj p r h ⟹ p ∈ hdom h
obj p r h ⟹ NULL ∉ hdom h
```

We now define a predicate objs P A for a set of objects on the heap that is suitable for symbolic execution. The set P contains the base addresses, A is the abstract HOL value, a map from addresses to object values. The pure side-condition relates the parameters (fold denotes the fold functional for finite sets; note that address sets, as sets of machine words, are implicitly finite).

```
objs P A ≡ fold add-obj emp (map-graph A) ∧̄ (P = dom A)
  where  add-obj (a,r) Q ≡ obj a r ⋆ Q
```

The definition may be simplified by omitting the set P and replacing any references by dom A. We have found, however, that verification is complicated by having to reason about the maps, rather than sets, before accessing objects. Note also that the above definition parallels the standard predicate list p xs q, which delineates the list by the pointers p and q. Since these are usually stored in program variables, the value xs is determined by the endpoints. For objects, we will store the set P in a ghost variable to achieve the same reasoning pattern.

The chosen definition then yields straightforward unfolding rules to be applied automatically during verification. The base cases of no objects and a single object simplify as expected:

```
objs {} A = (emp ∧̄ (A = empty))
objs {p} A = (∃̄r. obj p r ∧̄ (A = [ p↦r ]))
```

For accessing part of the objects separately, we need to split the object set by extracting a given subset. This is accomplished by the following general rule.

$$\frac{Q \subseteq P}{\text{objs } P\ A = (\bar{\exists} B\ C.\ \text{objs } Q\ B \star \text{objs } (P - Q)\ C \mathbin{\bar{\wedge}} (A = B +\!\!+ C))}$$

Symbolic execution uses only the derived case (2): when the program accesses an object at p, that object must be exposed in the symbolic heap, to be decomposed further into its fields (§2.1).

The side-condition of (2) is simple enough to be solved automatically.

$$\frac{p \in P}{objs\ P\ A = (\bar{\exists} r\ P'\ A'.\ obj\ p\ r \star objs\ P'\ A' \bar{\wedge} (A = A'(p \mapsto r) \wedge insert\ p\ P' = P))} \quad (2)$$

It is worth noting that the pure equality on A in (2) replaces A by a form suitable for simplification with (1) (§2.3), which makes the knowledge about the extracted object available. The premise of (1) is proven from the implicit assertion (§2.2) $p \notin P'$ derived from the symbolic heap left by (2).

3.2 Reachability

Standard definitions of reachability (e.g. [MN05][McC08, Fig. 6.8]) capture the transitive closure of the successor relation between objects in memory. Since we will use reachability as a pure side-condition, we state it in terms of the map value of object sets. If G, for "graph", is such a representation, then a path between p and q in G is defined in the standard way (e.g. [MN05]).

path G p [] q = (p = q)
path G p (a # ps) q = (p = a ∧ (∃s r. G p = Some r ∧ s ∈ succs r ∧ path G s ps q)

The goal is to define a predicate for object graphs that is analogous to the standard list predicate (§1). We employ our earlier idea of a *boundary set* [Gas10, Gas11], which replaces the single end pointer of the list predicate. The set of objects reachable in graph G from a given set P of pointers without touching the boundary Q is then defined using paths:

reachable G P Q ≡ {r. ∃p ps. p∈P ∧ path G p ps r ∧ set ps ∩ Q = {} ∧ r ∉ Q }

The set of reachable objects can be split into three subsets (3) (by splitting the paths at the set D): the objects D themselves, the objects reachable from P without touching Q or D, and those reachable from the successors of D. Note, however, that the theorem is insufficient for the direct use in separation logic, because the latter two sets might overlap.

$$\frac{D \subseteq reachable\ G\ P\ Q}{reachable\ G\ P\ Q = D \cup (reachable\ (G \restriction (-D))\ P\ (Q \cup D) \cup reachable\ (G \restriction (-D))\ (Succs\ (G \restriction D))\ (Q \cup D))} \quad (3)$$

3.3 The Predicate for Object Graphs

Based on these preliminaries, we can now formulate the predicate graph P R G Q. It states that the objects R, with HOL representation G, are reachable from P without crossing Q.

graph P R G Q ≡ objs R G $\bar{\wedge}$ R = reachable G P Q

We now give the unfolding rules that enable automatic symbolic execution. Lemma (4) states the general case: for any set of reachable objects, the object graph can be split at these objects, i.e. these objects can be extracted for manipulation. The remainder of the graph can be reached from either the old root set or the successors of the extracted objects, without touching the extracted objects. The side-condition on the reachability of D in the original graph is necessary to prove the reverse direction of the equality. In systems that distinguish between folding and unfolding rules (e.g. [BCO05a]), it can be omitted, since (4) is used only for unfolding.

$$\frac{D \subseteq R}{\begin{array}{l} graph\ P\ R\ G\ Q = (\bar{\exists} G'\ G''.\ objs\ D\ G' \star graph\ (P \cup Succs\ (G \restriction D) - (Q \cup D))\ (R - D)\ G''\ (Q \cup D) \\ \bar{\wedge}\ (G = G' +\!\!+ G'' \wedge D \subseteq reachable\ G\ P\ Q)) \end{array}} \quad (4)$$

The special case (5) is used in for symbolic execution: the program accesses some node p which is known, from the collector's invariants, to be in the set of reachable objects. That object is exposed for manipulation, and the remainder of the graph is reachable from the original root set and the object's successors.

$$\frac{\mathsf{p} \in \mathsf{R}}{\begin{array}{l}\mathsf{graph\ P\ R\ G\ Q} = (\bar{\exists}\mathsf{r\ G'}.\ \mathsf{obj\ p\ r} \star \mathsf{graph}\ ((\mathsf{P} \cup \mathsf{succs\ r}) - (\mathsf{Q} \cup \{\mathsf{p}\}))\ (\mathsf{R} - \{\mathsf{p}\})\ \mathsf{G'}\ (\mathsf{Q} \cup \{\mathsf{p}\}) \\ \qquad \bar{\wedge}\ (\mathsf{G} = \mathsf{G'}(\mathsf{p} \mapsto \mathsf{r}) \wedge \mathsf{p} \in \mathsf{reachable\ G\ P\ Q}))\end{array}} \tag{5}$$

We will see in §5 that P is derived from the working queue of the collector, such that the addition of these successors corresponds precisely to adding the object p (or its copy) to the queue. Furthermore, as in the case of (2), the pure equality on G will cause the simplification rules from §2.3 to expose the knowledge about the manipulated object from the invariants.

The unfolding (6) is used to establish invariants initially. The input to collectors is a set of objects (§3.1), some of which are reachable from the root set. In order to use the predicate graph in invariants as intended, one has to extract the reachable part of the input (where closed G P Q denotes that the successors of objects in G are in G ∪ Q, i.e. the absence of dangling pointers).

$$\frac{\mathsf{closed\ G\ P\ Q}}{\begin{array}{l}\mathsf{objs\ A\ G} = (\bar{\exists}\mathsf{G'\ G''}.\ \mathsf{graph\ P\ (reachable\ G\ P\ Q)\ G'\ Q} \star \mathsf{objs\ (dom\ G - reachable\ G\ P\ Q\ )\ G''} \\ \qquad \bar{\wedge}\ (\mathsf{G'} +\!\!+ \mathsf{G''} = \mathsf{G} \wedge \mathsf{A} = \mathsf{dom\ G} \wedge \mathsf{reachable\ G\ P\ Q} = \mathsf{dom\ G'} \wedge\ \mathsf{closed\ G'\ P\ Q}))\end{array}} \tag{6}$$

4 Verification Environment

Previous studies on separation logic (e.g. [AB07, McC09, Chl11, JSP10, Myr10, MAY06, Web04]) have constructed completely new verification environments. The derivation of a sound Hoare logic and the implementation of the verification condition generator (VCG) are, however, substantial tasks and involve subtle considerations, e.g. on auxiliary (or logical) variables and recursive procedures (e.g. [ON02]). In the case of classical Hoare logics, the effort can be re-used by implementing the VCG for an intermediate language [BCD+06, CDH+09, Sch05].

We now propose an effective strategy for obtaining a verification environment for separation logic by embedding the source language, here the C dialect [Gas10], into an intermediate language with a classical VCG, here Simpl [Sch05]. The treatment, in particular for side-effecting expressions and functions, closely follows [Sch05, §2.4.3,2.4.5,4.2–4.5,4.9]. The heap is represented by a global variable of type heap (§2.1); a heap-acc-cond guard [Sch05, §2.2] is added for every heap-get and heap-put (§2.1). Ghost variables can contain any HOL values, and we allow Isabelle term syntax on the right-hand sides of ghost assignments (marked as comments by `//@`).

Simpl's VCG then generates classical verification conditions for partial correctness. However, these are not suitable for symbolic execution: heap writes modify the global variable heap, such that (a variant of) the assignment rule $\{Q[e/x]\}\ x := e\ \{Q\}$ is applied, which substitutes references to heap h in Q by heap-put ty p v h', where possibly h' is further replaced by earlier writes in the program. To solve this, we extend an observation employed in several studies [AB07, McC09, LW09]: if the syntactic substitution is replaced by a let-binding, then the sequence of variable updates is reflected correctly in the nesting of let-expressions. Differing from previous studies, however, we cannot introduce the let-bindings into Simpl's rules [Sch05, §3.1], because the VCG is carefully tuned to exactly the present formulation.

We therefore introduce a two-phase algorithm to re-construct (an approximation of) the execution sequence of heap accesses. The first bottom-up phase applies common subexpression elimination to the verification conditions to name all heap accesses, i.e. occurrences of heap-get and heap-put. The second top-down phase decides on appropriate insertion points for the let-bindings. With a view to the application, this process is called *scheduling* the heap operations. The tactic `prep_vc` combines the two phases and iterates appropriately for quantified and other bound variables in the VCs. The result is an approximation of the original sequence of operations only in so far as the order of different heap-gets cannot be recovered.

The scheduling process constitutes the core of the approach. Its decisions are governed by four principles. (1) Operations are scheduled at logical junctors, i.e. immediately above or below the conjunctions and implications introduced by the VCG. This associates operations with natural branching points in the program. (2) Operations are scheduled as high in the term as possible to avoid duplicate heap matchings (§2.2) for the same heap access substituted into different branches of the program. (3) An operation is not inserted by (1) and (2) if its arguments are guarded, i.e. if these arguments transitively depend on variables occurring on the left-hand side of an implication further below in the term structure. This avoids scheduling operations whose side-conditions are not provable without the further hypothesis. A typical case is an `if` statement testing a pointer for `null` before performing an access. (4) An operation heap-put can be scheduled only after all other operations referring to its input heap. If this rule is disobeyed, the current heap layout is modified by symbolic execution and cannot be recovered later on.

The tactic `run` then performs symbolic execution, using heap matching (§2.2) to retrieve values from the symbolic heap and updating the symbolic heap (cf. e.g. [McC09, Chl11]). We have thus constructed a verification environment for symbolic execution by re-using an existing VCG for a classical Hoare logic. The development took about 4 weeks, such that the approach is an effective alternative to a standalone derivation such as [Tue09].

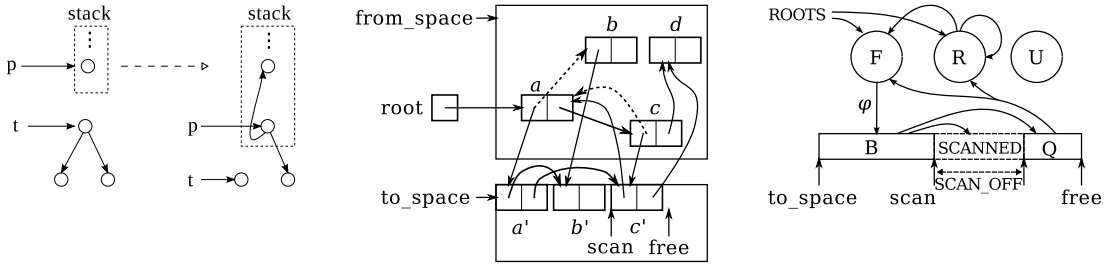
5 Applications

This section demonstrates the effectiveness of our approach using two standard benchmark examples, the Schorr-Waite graph marking algorithm [Bor00, MN05, HM05] and Cheney's copying collector [McC08, VB08, MSLL07, Myr10]. Since the algorithms, their specifications, and their invariants have been presented in detail in the literature, we focus on the application of our library (§3) in the loop invariants (§5.1) and during symbolic execution (§5.2). The structure of the proofs is summarized in §5.3, the proofs scripts are available from [Gas12]. To simplify comparisons, our implementations closely follow [MN05] and [McC08, Ch.6], respectively.

5.1 Loop Invariants

The litmus test for our approach are obviously the algorithms' loop invariants, and the subsequent proofs about them. The pre- and post-conditions [Gas12] follow [MN05, VB08]. They only use objects sets and the definition of reachability (§3.1, §3.2). Rule (6) then establishes the invariant from the pre-condition that the objects graph is closed.

Consider first the Schorr-Waite algorithm, which traverses an object graph by maintaining



(a) *Push* in Schorr-Waite (b) Details Cheney (c)

Figure 1: Main Ideas of the Example Algorithms

the backtracking stack within the objects themselves (Fig. 1 (a)). A pointer p marks the top of the stack; each stack object has a (boolean) *color* c, which specifies the field holding the next stack object. The *tip* pointer t points to the currently investigated object. Since t is always the pointer that is overwritten in object p, the original pointer structure can be restored [MN05, §8.2]. Following [HP09], ghost variables hold sets of objects: M for the marked objects, U for the unreachable input objects, R for the remaining unprocessed objects, and S for the stack objects.

Below, we now formulate the invariant as a symbolic heap [BCO05b]. (g is the instance of the object graph locale (§3), s describes the stack as an instance of a similar locale for linked lists). Lines 1–3 partition the objects into the sets postulated in §1. The graph (§3.3) captures the unprocessed objects as those objects that are reachable from (the non-null r fields of) the backtracking stack without crossing the stack or marked nodes.[2] The pure part of the symbolic heap in lines 4–10 follows [MN05], but uses our specialized map notation (§2.3). Note that reachability is not specified here, but is abstracted over in the graph predicate (line 10 merely keeps the knowledge about the original graph from the pre-condition for proving the post-condition).

```
1  ∃̄B C D. g.objs M B ⋆ g.objs U D ⋆ s.list p S NULL ⋆
2            g.graph  (({ t } ∪ obj-r ‘ (snd ‘ set S - is-C)) - is-NULL - M - s.nodes S)
3                     R  C (M ∪ s.nodes S)
4   ∧̄ (t∈{NULL} ∪ s.nodes S ∪ R ∪ M ∧
5      (∀p↦y∈C ++ map-of S. ∀q∈obj-succs y. q∈s.nodes S ∪ R ∪ M) ∧
6      (∀p↦y∈B++C++D. ∃n. A p=Some n∧obj-r y=obj-r n∧obj-l y=o bj-l n) ∧
7      stack-reco A t S ∧
8      (∀p↦y∈B ++ map-of S. obj-m y) ∧(∀p↦y∈C. ¬ obj-m y) ∧
9      N = s.nodes S ∪ R ∪ M ∪ U ∧
10     g.reachable A ({root} - is-NULL) {} = R ∪ M ∪ s.nodes S)
```

Cheney's collector has a more involved invariant, whose core is illustrated in Fig. 1 (b, c). The task is to copy objects reachable from the root set (restricted to one pointer, as in previous studies) out of the from-space to an empty to-space. At the level of individual objects (Fig. 1 (b)), the original objects are copied one-by-one into the free part of the to-space, their first fields are overwritten with a *forwarding pointer* to handle aliasing and cycles (the original pointers are shown dashed). The pointer `scan` splits the copied objects: those before `scan` (a', b') are complete, i.e. their original successors have also been copied; those after `scan` (c') form the working queue, and their fields point to their original successors. At a more abstract level,

[2] The formulation is slightly more precise than [Bor00, MN05] in also taking into account the color of the objects.

Fig. 1 (c) exhibits the partitioning postulated in §1. The collection of all forwarding pointers in the *forwarded* objects F constitutes an injective graph morphism φ, which will finally be a graph isomorphism between the input and result [MSLL07, VB08]. In the to-space, the completed objects B are called *black*. The ghost variable SCAN_OFF delineates the currently investigated objects SCANNED. The remaining objects R in the from-space are reachable from the queue Q and a set ROOTS of additional roots, which contains the original successors of the SCANNED objects.

The collector's invariant is, again, a symbolic heap. For brevity, we only describe its spatial part, shown below. The first line captures the to-space with the linear lists (instance q of the list locale) with black and queue objects, and a continuous free memory block. The from-space in the second line contains the forwarded objects F, the unreachable objects U and the remaining objects R. The latter are specified as an object graph, using as entry points the working queue and auxiliary ROOTS, and never crossing a forwarded object F. The *hole* H allows the invariant to be used to specify intermediate states during processing the currently scanned object [Gas12], where it contains the remaining fields of the current object and the input root reference.

q.list to-space B scan ⋆ q.list (scan+SCAN-OFF) Q free ⋆mem-block free (space-sz - (free - to-space)) ⋆
g.objs F A ⋆ g.objs U C ⋆ g.graph ((($\bigcup$ r∈q.vals Q. obj-succs r) ∪ ROOTS) - F) R G F ⋆ H

The pure part of the invariant mainly relates the pointers between the objects of the different partitions according to Fig. 1 (c). The forwarding morphism φ = forw-morph A merely examines the objects' car fields. Furthermore, the to-space after free is sufficiently large for the objects R.

The similarity between the Schorr-Waite and Cheney algorithms, which reflects the fundamental structure of collectors, is now explicit: in each case, the remaining objects can be specified concisely using the graph predicate from §3.3, and the entry points are successors of the current working queue/stack and the boundary set are the already processed objects.

5.2 Symbolic Execution

We now demonstrate how the automatic unfoldings from §3, together with the rewrite rules from §2.3, enable verification by symbolic execution [BCO05b], and thus reduce the need for manual interaction (§5.3, §6) in the style of [Tue09, Chl11].

The Schorr-Waite algorithm repeatedly performs steps *Push*, *Pop*, and *Swing* at the top the stack [Bor00, MN05]. We concentrate here on the *Push* step (Fig. 1 (a)), since its access to the unprocessed graph of objects can break the reachability relation in R, which makes the step the most complex for verification (e.g. [MN05, §8.3]). The code of the step below (lines 5–9) moves object t from R to S, and adapts the pointers as suggested in Fig. 1 (a).

```
1  while (p != null || (t != null && !t→m)) {
2   if  (t == null || t→m) {
3     ...
4   } else {
5     //@ S = ((t, (| obj-m = True, obj-c = False, obj-l = p, obj-r = t→r |)) # S);
6     //@ R = R - {t};
7     q = p;     p = t;
8     t = t→l;
9     p→m = true;     p→l = q;     p→c = false;
10  }
11 }
```

During symbolic execution, the analysis of t in lines 1 and 2 automatically extracts that object from the possible partitions (line 4 of the invariant) after a case distinction. For t ∈ R, an au-

tomatic application of (5) exposes an object r at t, which in turn enables access to the object's fields (§2.1); no reasoning about reachability is necessary. When line 5 is reached, the object is thus ready for standard manipulation [Tue09, Chl11]. Line 5 enables automatic folding (§2.1) to incorporate t into the stack to re-establish the invariant. Symbolic execution has thus moved an unmarked node from R to the stack S, in direct correspondence with the code.

In Cheney's algorithm, the crucial point is found in function `copy_ref` [McC08], which processes a machine word at p. The word can be either an atomic value or an object reference, as distinguished by its least significant bit. Atomic values are left unchanged; for already forwarded objects the reference is replaced by the forwarding pointer. None of these operations manipulates the remaining objects. The last case concerns the copying of a newly discovered remaining object. Setting the forwarding point in that object, again, breaks the reachability relation.

The function `copy_ref` here performs a case distinction, by reading the value at p and the car field of any object reference found there. As before, the proof proceeds by a case distinction between R and F as the possible targets of obj (Fig. 1 (c)), then symbolic execution exposes the object automatically by either (2) or (5).

```
int obj = *p;
if ( obj & 1 == 0 && (struct obj*)obj != null) {
  int fwd = ((struct obj*)obj)→car;
  ... case distinction on fwd being a to-space pointer ...
}
```

The code for actually copying an object thus finds the object exposed in the symbolic heap. Lines 1–4 below allocate an initialize the copy, line 5 sets the forwarding pointer; line 6 updates the input reference from the original to the copy. Line 7 transfers the object from R to F.

```
1 struct obj *n = (struct obj*)free;
2 free = free + 8;
3 n→car = ((struct obj*)obj)→car;
4 n→cdr = ((struct obj*)obj)→cdr;
5 ((struct obj*)obj)→car = (int)n;
6 *p = (int)n;
7 //@ F = {obj} ∪ F; R = R - {obj};
```

Throughout, symbolic execution thus exposes automatically the objects under consideration, such that the manipulations can be performed directly. In particular, no reasoning about reachability is necessary, and the intuition of "moving" objects between different heap partitions (§1) is maintained. The main difference to list and tree examples is the necessity of case-distinctions in proofs, because pointers can reference different object sets, rather than a single data structure.

5.3 Structure of Proofs

The overall structure of the proofs [Gas12] for both examples shows that we have achieved a semi-automatic symbolic execution [Tue09, Chl11]. The tactics `run` and `heap` from §2.2 perform the necessary manipulations on the symbolic heap, using the unfoldings from §3. The resulting behaviour is similar to the cases of lists and trees (§1; [Tue09, Chl11, McC09]).

Manual intervention is necessary only at two points: first, we prefer the user to introduce case-distinctions (by a tactic `split_or`) where pointers reference several alternative sets of objects; second, the user has to solve the more intricate set-theoretic pure side-conditions and inequalities on finite machine words. The rewrite rules from §2.3 otherwise contribute to the solution of pure

conditions by splitting assertions about unions of sets/maps into separate conjuncts, most of which are then already present in the pre-condition.

The proof scripts are found to be very short, compared to previous results (§6). The proof of the Schorr-Waite algorithm has about 140 lines (all lines are non-empty, non-comment), the length being due to the number of cases induced by the code's boolean short-circuit operators. The overall theory has 250 lines. The function `copy_ref` of Cheney's collector takes 150 lines, of which 60 lines concern side-conditions on word inequalities and further 30 concern set equalities. The driver loop that repeatedly calls `copy_ref` requires 180 lines of proof. The overall theory has 710 lines. The work on Cheney's algorithm took the author roughly 5 days, building on the library from §3 and the experience from the Schorr-Waite algorithm and [Gas11]. The library from §3 has 630 lines and has been re-used without modification between the algorithms.

6 Related Work

Our implementation of separation logic and symbolic execution (§2.1, §2.2) follows previous studies [Tue09, McC09, BCO05a, Chl11, Web04]. We have found the proposed strategies, especially those for heap matching, sufficient for reasoning about object graphs, based on the library from §3. We therefore expect that our development benefits other tools as well.

To the best of the author's knowledge, the re-use of a classical VCG for verification by symbolic execution from §4 has not been proposed before.

Previous proofs of garbage collectors in separation logic [McC08, McC09, MSLL07, Myr10, VB08] are based on a higher-order spatial quantifier $\bar{\forall} x \in A.\ P$. McCreight [McC08, §7] uses a layered specification with special-purpose predicates about objects. While the specification is readable, the proofs require detailed manipulations of spatial formulae [McC08, §6.3.3; §6.4.3], even when using specialized tactics [McC09]. Varming and Birkedal [VB08] likewise specify different parts of the heap independently, but do not introduce further abstractions; their proof scripts comprise 7500 lines. Myreen [Myr10] uses only the basic quantifier and states invariants as pure assertions. He proceeds by refinement, starting with a general specification of copying collection and ending in a low-level implementation. Despite the abstraction, the proofs have 1800 lines. In contrast, our predicates capture invariants succinctly (§5.1) and the proofs consist mostly of automatic symbolic execution (§5.3), without direct manipulations of spatial formulae.

Hawblitzel and Petrank [HP09] verify several practical collectors based on Boogie [BCD+06]. We adapt their technique of capturing the abstract data structures conceptually manipulated by the collector in ghost variables (§5). To enable automatic proofs, substantial manual annotations are necessary. Symbolic execution separation logic, in contrast, computes updates to a symbolic heap automatically, and the user only needs to prove that the final state matches the specified post-condition. Furthermore, [HP09] omits the central aspect of reachability [HP09, §4.1.1], and proves only that the reached objects have been handled correctly. We have shown that reachability can be factored out into the meta-theory and need not be considered during verification.

Leino [Lei10] verifies the Schorr-Waite algorithm in a Boogie-based language Dafny. For automatic proofs, he requires a fine-grained, 31-line loop invariant and replaces reachability in the specification by an (equivalent) closure property. The equivalence is, however, not mechanically established. Furthermore, careful formulation is necessary to guide the SMT solver.

7 Conclusion

We have shown that the established approach of verification by symbolic execution in separation logic can be extended to cover garbage collection algorithms. Previously, verification of this class of algorithms in separation logic required substantial interactive manipulations of spatial formula [McC08, §6.3.3; §6.4.3][McC09, VB08], or the formulation of invariants in pure assertions alone [Myr10]. We have shown that the application of symbolic execution yields straightforward and considerably shorter proofs (§5.3, §6).

The cornerstone of our solution is a spatial predicate graph that encapsulates reasoning about the reachability of objects (§3.3). The predicate has unfolding rules suitable for automatic symbolic execution, such that proofs become as straightforward as for list or tree nodes in existing tools (e.g. [JSP10, Tue09, Chl11, DJ08]). In particular, interactive manipulations of spatial formulae were unnecessary.

The library for graphs is developed as an Isabelle locale to make it re-usable for different types of objects. Apart from locales, we do not require special properties of Isabelle/HOL and it is expected that the results can be re-proven in other higher-order systems [Tue09, Chl11, McC09].

We have verified two central case studies, the Schorr-Waite graph marking algorithm and Cheney's copying garbage collector. In both, the proofs consist of automatic symbolic execution, followed by automatic heap matching (§2.2), which leaves only pure side-conditions. Due to a specialized quantifier for maps (§2.3), most of these were solved by rewriting; the remainder was solved by Isabelle's built-in provers with little user assistance.

The construction of our verification environment for separation logic (§4) pioneers the re-use of an existing VCG for classical Hoare logic, similar to the re-use of Boogie [BCD+06] as a target for different languages (e.g. [CDH+09]).

The presented results suggest two directions for future work: first, it seems feasible to take the proven unfolding rules from §3.3 as an axiomatization of a first-order constant graph, which could be added to tools like [JSP10, BPS09, DJ08]. The step would require suitable representations of sets and object collections, both of which have been studied before [HP09]. Second, following the approach from §4, it would be worth investigating whether existing more detailed models of C, e.g. [NIC12], can be combined with proofs by symbolic execution.

Bibliography

[AB07] A. W. Appel, S. Blazy. Separation Logic for Small-Step C minor. In Schneider and Brandt (eds.), *Theorem Proving in Higher Order Logics, 20th International Conference*. LNCS 4732. Springer, 2007.

[BCD+06] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, K. R. M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *4th International Symposium on Formal Methods for Components and Objects*. LNCS 4111. Springer, 2005.

[BCO05a] J. Berdine, C. Calcagno, P. W. O'Hearn. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In Boer et al. (eds.), *4th International Symposium on Formal Methods for Components and Objects*. LNCS 4111. Springer, 2005.

[BCO05b] J. Berdine, C. Calcagno, P. W. O'Hearn. Symbolic Execution with Separation Logic. In Yi (ed.), *Programming Languages and Systems, Third Asian Symposium (APLAS)*. LNCS 3780. Springer, 2005.

[Bor00] R. Bornat. Proving Pointer Programs in Hoare Logic. In *Mathematics of Program Construction*. 2000.

[BPS09] M. Botincan, M. Parkinson, W. Schulte. Separation Logic Verification of C Programs with an SMT Solver. In *4th International Workshop on Systems Software Verification (SSV)*. Volume 254. Elsevier, 2009.

[CDH+09] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, S. Tobies. VCC: A Practical System for Verifying Concurrent C. In *Theorem Proving in Higher Order Logics, 22nd International Conference (TPHOLs)*. LNCS 5674. Springer, 2009.

[Chl11] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *32nd ACM SIGPLAN conference on Programming language design and implementation (PLDI)*. ACM, 2011.

[Daw09] J. E. Dawson. Isabelle Theories for Machine Words. In *7th International Workshop on Automated Verification of Critical Systems (AVOCS)*. ENTCS 250. Elsevier, 2009.

[DJ08] D. Distefano, M. J. P. J. jStar: towards practical verification for Java. *SIGPLAN Not.* 43(10):213–226, 2008.

[Gas10] H. Gast. Reasoning about memory layouts. *Formal Methods in System Design* 37(2-3):141–170, 2010.

[Gas11] H. Gast. Developer-oriented Correctness Proofs: a Case Study of Cheney's Algorithm. In Qin and Qiu (eds.), *Proceedings of 13th International Conference on Formal Engineering Methods (ICFEM 2011)*. LNCS 6991. Springer, 2011.

[Gas12] H. Gast. Mechanized Development of Object Graphs for Symbolic Execution. http://www-pu.informatik.uni-tuebingen.de/users/gast/files/sepgraph.zip, 2012.

[HM05] T. Hubert, C. Marché. A case study of C source code verification: the Schorr-Waite algorithm. In Aichernig and Beckert (eds.), *3rd International Conference on Software Engineering and Formal Methods (SEFM)*. IEEE, 2005.

[HP09] C. Hawblitzel, E. Petrank. Automated verification of practical garbage collectors. *SIGPLAN Not.* 44(1):441–453, 2009.

[JSP10] B. Jacobs, J. Smans, F. Piessens. VeriFast: Imperative Programs as Proofs. In *VS-Tools workshop at VSTTE 2010*. (no formal proceedings), 2010.

[KWP99] F. Kammüller, M. Wenzel, L. C. Paulson. Locales: a Sectioning Concept for Isabelle. In Bertot et al. (eds.), *Theorem Proving in Higher Order Logics, 12th International Conference (TPHOLs)*. LNCS 1690. Springer, 1999.

[Lei10] K. R. M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *16th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR-16)*. 2010.

[LW09] C. Lüth, D. Walter. Certifiable specification and verification of C programs. In Cavalcanti and Dams (eds.), *FM 2009: Formal Methods, Second World Congress*. LNCS 5850. Springer, 2009.

[MAY06] N. Marti, R. Affeldt, A. Yonezawa. Formal Verification of the Heap Manager of an Operating System Using Separation Logic. In Liu and He (eds.), *8th International Conference on Formal Engineering Methods (ICFEM)*. LNCS 4260. Springer, 2006.

[McC08] A. McCreight. *The Mechanized Verification of Garbage Collector Implementations*. PhD thesis, Department of Computer Science, Yale University, Dec. 2008.

[McC09] A. McCreight. Practical Tactics for Separation Logic. In Berghofer et al. (eds.), *Theorem Proving in Higher Order Logics, 22nd International Conference (TPHOLs)*. LNCS 5674. Springer, 2009.

[MN05] F. Mehta, T. Nipkow. Proving pointer programs in higher-order logic. *Inf. Comput.* 199(1–2):200–227, 2005.

[MSLL07] A. McCreight, Z. Shao, C. Lin, L. Li. A general framework for certifying garbage collectors and their mutators. *SIGPLAN Not.* 42(6):468–479, 2007.

[Myr10] M. O. Myreen. Reusable verification of a copying collector. In *Proceedings of the Third international conference on verified software: theories, tools, experiments (VSTTE '10)*. LNCS 6217. Springer, 2010.

[NIC12] C-Parser of L4.verified. http://www.ertos.nicta.com.au/software/c-parser, 2012.

[ON02] D. v. Oheimb, T. Nipkow. Hoare Logic for NanoJava: Auxiliary Variables, Side Effects and Virtual Methods revisited. In Eriksson and Lindsay (eds.), *Formal Methods Europe (FME'02)*. LNCS 2391. Springer, 2002.

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

[Tue09] T. Tuerk. A Formalisation of Smallfoot in HOL. In Berghofer et al. (eds.), *Theorem Proving in Higher Order Logics, 22nd International Conference (TPHOLs)*. LNCS 5674. Springer, 2009.

[VB08] C. Varming, L. Birkedal. Higher-Order Separation Logic in Isabelle/HOLCF. *ENTCS* 218:371–389, October 2008.

[Web04] T. Weber. Towards Mechanized Program Verification with Separation Logic. In Marcinkowski and Tarlecki (eds.), *Computer Science Logic – 18th International Workshop, CSL 2004*. LNCS 3210, Springer, 2004.