Separation Logic in the Presence of Garbage Collection

Chung-Kil Hur Derek Dreyer Viktor Vafeiadis

Max Planck Institute for Software Systems (MPI-SWS)
Kaiserslautern and Saarbrücken, Germany
E-mail: {gil, dreyer, viktor}@mpi-sws.org

***Abstract*—Separation logic has proven to be a highly effective tool for the verification of heap-manipulating programs. However, it has been applied almost exclusively in language settings where either memory is managed manually or the issue of memory management is ignored altogether. In this paper, we present a variant of separation logic, GCSL, for reasoning about low-level programs that interface to a garbage collector. In contrast to prior work by Calcagno et al., our model of GCSL (1) permits reasoning about programs that use internal pointers and address arithmetic, (2) supports logical variables that range over pointers, and (3) validates the "frame" rule, as well as a standard interpretation of separation-logic assertions, without requiring any restrictions on existentially-quantified formulae. Essential to our approach is the technique (due originally to McCreight et al.) of distinguishing between "logical" and "physical" states, which enables us to insulate the logic from the physical reality that pointer "values" may be moved and/or deallocated by the garbage collector.**

I. Introduction

Separation logic [13, 15] has proven to be a highly effective tool for the verification of heap-manipulating programs. The key advance of separation logic over traditional Hoare logic is its support for *local reasoning* about shared mutable data structures, which is achieved via the notion of *spatial (or separating) conjunction* $P_1 * P_2$, together with the so-called *frame rule*. Briefly, $P_1 * P_2$ is satisfied by a heap h iff h can be split into disjoint subheaps h_1 and h_2 satisfying P_1 and P_2, respectively. The frame rule says that if a command C satisfies the Hoare triple $\{P\}\ C\ \{Q\}$, then C also satisfies $\{P * R\}\ C\ \{Q * R\}$, for any framing assertion R whose free variables are not modified by C. In other words, C is guaranteed to preserve any invariant R concerning the memory outside of C's *footprint* (*i.e.,* the piece of the heap that it accesses). The frame rule thus allows one to reason locally about C's effect on its own footprint, and "frame in" knowledge about the rest of the heap after the fact.

Thus far, separation logic has mainly been applied to two general classes of programming languages: low-level languages in which programs manage their memory manually, and high-level languages in which programs assume the presence of a garbage collector (but where the garbage collector is ignored in the context of verification).

The goal of the present work is to explore the use of separation logic for reasoning about a third class of languages: low-level languages in which programs interface to a garbage collector. This third class arises naturally when one considers the interoperation of C or assembly code fragments from different sources (*e.g.,* from different compilers or coded by hand) in tandem with a single garbage collector. The correctness of linking such program fragments depends on their preserving the invariants of the garbage collector (GC), such as that there are no dangling pointers reachable from the "roots". Internally, however, in between calls to the memory allocator or to one another, the fragments should be free to violate the GC's invariants by employing internal pointers, address arithmetic, etc.

Adapting separation logic to account for the presence of a garbage collector is challenging because separation logic is geared toward local reasoning, whilst garbage collection requires a global view of the program state. (Note that this is quite different from using separation logic to reason about the correctness of the garbage collector itself [16, 11].) To our knowledge, there have only been two significant attempts to grapple with this topic. We discuss them both here, as they are directly relevant in understanding the merits of the present work.

A. Prior Work

Adapting Hoare Logic to the Presence of a Garbage Collector: The work of Calcagno, O'Hearn and Bornat [3] is primarily focused on addressing a "conundrum" of Hoare logic, namely that it is seemingly incompatible with garbage collection. The canonical example of this, due to Reynolds [14], is the following triple, where $\mathsf{cons}(1,2)$ allocates a pair of heap cells containing 1 and 2 and returns a pointer to the first:

$$\begin{array}{l}\{\mathsf{true}\}\\ \mathtt{x} := \mathsf{cons}(1,2);\quad \mathtt{x} := 3\\ \{\mathtt{x} = 3 \wedge \exists y.\ y \hookrightarrow 1,2\}\end{array}$$

This triple is valid in Hoare logic, but in the presence of a garbage collector the cell that x initially points to may be reclaimed after x is set to 3, in which case $\exists y.\ y \hookrightarrow 1,2$ may cease to hold. This is a problem if we want to treat garbage collection as being equivalent logically to skip.

Calcagno *et al.* present two alternative approaches to resolving the conundrum. The first employs a model based

on "total states"—*i.e.,* where the machine states under which assertions are interpreted have no dangling pointers. Under this model, they give a non-standard interpretation of assertions (in particular, existential quantification), which renders all assertions insensitive to the effects of garbage collection.[1] Specifically, they consider a heap h to satisfy $\exists x.P$ if (roughly) there exists some potentially *larger* heap h' satisfying P for some instantiation of x. This interpretation renders the assertion $\exists y.\ y \hookrightarrow 1, 2$ from the above example equivalent to true, since (assuming, as usual, an infinite memory model) one may extend any heap with a fresh pair of cells storing 1 and 2.

A key limitation of the total-states model is that it fails to validate the frame rule. Calcagno *et al.* subsequently present a second model based on "partial states", which *does* validate the frame rule.[2] Unlike the total-states model, however, the partial-states model employs a standard intuitionistic interpretation of separation-logic assertions and, as a result, runs afoul of the original conundrum in the sense that existentially-quantified formulae like $\exists y.\ y \hookrightarrow 1, 2$ are once again garbage-sensitive. To rule out such problematic (and arguably useless) formulae as this, Calcagno *et al.* propose a syntactic "guarded" restriction on existentials. Guardedness conservatively ensures the garbage-insensitivity of $\exists y.\, P$ by requiring P to include explicit evidence that y is reachable (by pointer dereference) from some free program variable x. Thus, it avoids Reynolds's conundrum by—to use Lakatos's evocative term [8]—"monster-barring".

In addition to the treatment of Reynolds's conundrum, there are two key aspects of Calcagno *et al.*'s account that we seek to improve on in the present work.

First, the programming language they consider treats pointers abstractly, assumes all allocated blocks are of size two, and does not permit pointer arithmetic or internal pointers. This effectively prevents one from writing code that violates the invariants of the garbage collector, thus simplifying verification effort at the cost of expressiveness. We are interested in reasoning about "dirtier", lower-level programs than one can express in their language.

Second, their logic is limited in its support for *logical variables*, *i.e.,* non-assignable variables representing values of the language. Logical variables are useful in giving clean and easily composable specifications for low-level programs, such as the following Hoare triple for a $\mathsf{swap}(\mathtt{x})$ routine, which swaps the two values stored in the heap at the cons cell pointed to by x:

$$\{\mathtt{x} \hookrightarrow v_1, v_2\}\ \mathsf{swap}(\mathtt{x})\ \{\mathtt{x} \hookrightarrow v_2, v_1\}$$

Although Calcagno *et al.* do not explicitly consider logical variables, their logic is incompatible with logical variables that range over pointers. In particular, since they model garbage collection as an operation that may α-rename pointers in the heap at any time, they have no way of talking about a permanent pointer "value", such as the v_i above, that would have the same meaning before and after garbage collection. The best specification they can give for a routine like $\mathsf{swap}(\mathtt{x})$ is one that either restricts v_1 and v_2 to range over integers, or else makes use of auxiliary program variables y and z as follows:

$$\{\mathtt{x} \hookrightarrow \mathtt{y}, \mathtt{z}\}\ \mathsf{swap}(\mathtt{x})\ \{\mathtt{x} \hookrightarrow \mathtt{z}, \mathtt{y}\}$$

This specification is inferior for three reasons: (1) to be useful, it requires an additional "modifies" clause specifying that $\mathsf{swap}(\mathtt{x})$ does not modify y and z; (2) it requires one to (pointlessly) assign y and z the contents of x's cons cell before calling $\mathsf{swap}(\mathtt{x})$; and (3) the question of whether such auxiliary variable assignments are safely erasable is complicated by the fact that they may artificially extend the lifetime of certain data. Clearly, allowing logical variables to range over pointers would be vastly preferable.

Certifying Garbage-Collected Programs: McCreight, Shao, Lin and Li [10] develop a general framework for certifying garbage-collected programs, in which the interface between mutator and collector is abstract enough that multiple collectors can be safely linked with the same mutator, and the mutator can be verified without knowing the gory implementation details of the collector. A major focus of their work is on proving various collectors correct w.r.t. such an interface, including mark-and-sweep, copying and incremental collectors; this is orthogonal to the problem we are concerned with.

For reasoning about the mutator, they propose a very useful idea that we adopt in this paper. We have already seen above—in the discussion of Calcagno *et al.*'s inability to support logical variables ranging over pointers—that it is problematic for the mutator to have too concrete a view of pointers, since a moving collector may very well fail to preserve it. To address this problem, McCreight *et al.* employ a distinction between "abstract" and "concrete" states. The mutator reasons solely about an abstract representation of the program state; the concrete locations used to represent abstract pointers are immaterial to the mutator's reasoning and may change during a GC. This approach allows the mutator to view pointers as having a persistent (abstract) identity, even if their concrete identity may change over time. (We present this idea in more detail in Section I-B.)

What is missing from McCreight *et al.*'s work, and what we aim to develop in this paper, is a clear account of how the GC affects local, separation-logic-style reasoning about the mutator. While McCreight *et al.* provide two worked examples of mutator verification in Coq using a variant of Feng *et al.*'s SCAP system [5], their specifications for both

[1] Although this interpretation of existentials seems somewhat unusual, Calcagno *et al.* show how it can be recast as a specialization of the well-known "dense topology" semantics of classical logic [4].

[2] For details about the issues surrounding the frame rule, see Calcagno's thesis [2].

examples are so permissive that they are not composable: in particular, they are defined over the global (abstract) state and say nothing to prevent the mutators from overwriting the whole state. While we believe that their framework is also capable of expressing local, composable specifications by explicitly parameterizing over frame heaps, they do not develop any general proof principles to explain what additional reasoning is required to verify such specifications in the presence of a garbage collector.

In more recent work, McCreight, Chevalier and Tolmach [9] and Hur and Dreyer [6] fruitfully apply the idea of abstract states to the verification of compiler correctness for garbage-collected languages. Like the aforementioned work, however, neither of these papers studies how the principles of separation logic are affected by GC.

B. Contributions and Main Ideas

In this paper, we present a novel variant of intuitionistic separation logic, GCSL, which supports local reasoning about low-level programs in the presence of a garbage collector.[3] GCSL addresses the two main limitations of Calcagno *et al.*'s work discussed above: (1) the language we consider supports the use of internal pointers and address arithmetic; and (2) GCSL supports logical variables ranging over arbitrary values (including pointers). Furthermore, our model of GCSL validates the frame rule, as well as a standard interpretation of separation-logic assertions (including existentials), without requiring—as Calcagno *et al.*'s partial-states model does—any monster-barring restrictions on existentials in order to avoid Reynolds's conundrum.

GCSL has two main limitations of its own. First, it only accounts for the presence of a stop-the-world collector, not an incremental or concurrent one. We believe it should be possible to adapt the basic framework of GCSL to these other kinds of collectors, but we leave that to future work. Second, while we support reasoning about programs that store internal pointers in program variables, we require that the user heap only contain *GC-safe* values—*i.e.,* values that the GC considers valid (non-pointer values or pointers to the heads of allocated blocks). Since in practice internal pointers are typically employed as a way of optimizing register usage, we believe that confining them to program variables is a reasonable simplifying assumption for a large class of real-world code.

The main contribution of our paper is to clarify the extent to which garbage collection affects the standard principles of local reasoning that are codified by separation logic. At least for programs that satisfy the aforementioned restrictions of our logic, we demonstrate that the additional reasoning required to account for GC is relatively minimal.

The success of GCSL relies on two key technical novelties. We consider each in turn.

[3]We work with intuitionistic rather than classical separation logic because we are not reasoning about manual deallocation.

Two-Level Logic: In order to support internal pointers and address arithmetic, while ensuring that programs obey the invariants of the garbage collector whenever it is invoked, we split the logic into two levels.

At the *outer* level of the logic, where Hoare triples are written $\{\{P\}\}\ C\ \{\{Q\}\}$, we follow Calcagno *et al.* in only permitting reasoning with *GC-insensitive* assertions, *i.e.,* assertions that the GC is guaranteed to preserve. In other words, the specification of garbage collection in this logic is equivalent to that of skip—namely, $\{\{P\}\}\ \mathsf{gc}()\ \{\{P\}\}$ for any P—and so the GC may be invoked before or after executing any well-specified command C. In order to guarantee the soundness of this specification for the GC, we assume and maintain the invariant that *both* the heap *and* all program variables only store GC-safe values.

At the *inner* level of the logic, where Hoare triples are written $\{\mathbf{P}\}\ C\ \{\mathbf{Q}\}$, memory allocation and garbage collection are not permitted, but GC-unsafe operations are. In particular, the inner level enables one to reason about commands C that store unsafe values (such as internal pointers) in program variables, although the heap is still restricted to only contain GC-safe values. This inner level is useful for reasoning about small program fragments that are executed in between calls to the memory manager and that may therefore break the GC's invariants during their execution (so long as they restore them in the end).

In essence, the key formal difference between the two levels of the logic is that at the outer level we assume and maintain the invariant that all program variables contain GC-safe values—critical for ensuring that the GC can be safely invoked—whereas at the inner level this safety invariant is neither assumed nor maintained. Thus, in order to connect the two levels, we must make the preservation of the invariant into an explicit proof burden at the inner level. This is achieved by the following "inclusion" rule:

$$\frac{\{P \wedge \mathsf{safe}(V)\}\ C\ \{Q \wedge \mathsf{safe}(\mathrm{Mod}(C))\}}{\{\{P\}\}\ C\ \{\{Q\}\}}$$

The rule says that we may prove an outer-level triple using an inner-level one, so long as the inner-level one preserves the GC-safety of any variables it modifies. As a precondition, it may assume that any finite set of program variables V are GC-safe, since that is a valid outer-level assumption.

Ironically, while the GC-safety predicate safe must implicitly hold of all program variables in the outer-level logic, it is only safe to mention safe itself in the assertions of the inner-level logic! This is because GC-safety is a *GC-sensitive* property: the GC may take a pointer to the head of an allocated block—a safe pointer—and reclaim it if it is inaccessible, after which it will be a pointer to the head of a *deallocated* block—an unsafe pointer. Thus, outer-level assertions P and Q may not mention safe since the outer-level logic requires them to be GC-insensitive. (We write

P and **Q** to denote inner-level assertions, which extend the language of outer-level assertions with the safe predicate.)

Logical vs. Physical States: Our second technical novelty is to use McCreight *et al.*'s idea of abstract vs. concrete states—which, following Hur and Dreyer [6], we prefer to call *logical* vs. *physical* states, respectively—in order to give a model of GC-aware separation logic in which logical variables can range over pointers. *En passant*, it also enables us to avoid the need for Calcagno *et al.*'s "guarded" restriction on existentials.

The essential components of our model are the following: First, we interpret assertions as predicates on logical, not physical, states. Second, we say that a logical state LS represents a physical state PS if the piece of LS that is reachable from the roots (which for us means program variables) is isomorphic to the piece of PS that is reachable from the roots. (It is perfectly fine for both LS and PS to have extra junk in them so long as it is unreachable from the roots.) Last but not least, we interpret outer-level triples $\{\{P\}\}\ C\ \{\{Q\}\}$ roughly as follows. Assume we are given some initial physical state PS, represented by some initial GC-safe logical state LS that satisfies the precondition P. Then: (1) the execution of C in PS will not get stuck, and (2) if it terminates in a final physical state PS', then there must exist *some* final GC-safe logical state LS' that represents PS' and that satisfies the postcondition Q. (The interpretation of inner-level triples is similar, minus the assumption and preservation of GC-safety.)

Our model of Hoare triples addresses Calcagno *et al.*'s problem concerning logical variables because it allows such variables to range over *logical pointers*, abstract entities that comprise the domain of logical heaps. Even if the physical concretization of a logical pointer is moved or deallocated by the GC, the logical identity of the pointer may safely persist across calls to the GC because our model of Hoare triples gives us the freedom to choose whatever logical state we please to represent the post-GC physical state. In particular, we can always choose the post-GC logical state to be the same as the pre-GC logical state because the logical/physical representation relation is invariant under garbage collection. As a result, it is easy to see that $\{\{P\}\}\ \mathsf{gc}()\ \{\{P\}\}$ holds for any outer-level P, as desired.

A corollary of this observation is that our model easily resolves Reynolds's conundrum without requiring any non-standard interpretation of (or syntactic restriction on) existential quantification. Specifically, just like every other outer-level assertion, $\exists y.\ y \hookrightarrow 1,2$ is guaranteed to be preserved under garbage collection: even if the witness of the existential is physically reclaimed, we can ensure that it persists logically. Note that our solution to the conundrum is subtly different from the one given by Calcagno *et al.*'s total-states model, because while we do validate the triple $\{\{\mathsf{true}\}\}\ \mathsf{skip}\ \{\{\exists y.\ y \hookrightarrow 1,2\}\}$, we do not consider the assertion $\exists y.\ y \hookrightarrow 1,2$ to be *equivalent* to true (as they do).

$$
\begin{array}{ll}
\mathrm{ProgVars} & \stackrel{\text{def}}{=} \{\, \mathtt{x}, \mathtt{y}, \ldots \,\} \\
\mathrm{Words} & \stackrel{\text{def}}{=} \{\, w \in \mathbb{Z} \,\} \\
\mathrm{Ptrs} & \stackrel{\text{def}}{=} \{\, p \in \mathrm{Words} \mid p > 0 \wedge p \text{ is a multiple of } 4 \,\} \\
\mathrm{NonPtrs} & \stackrel{\text{def}}{=} \{\, a \in \mathrm{Words} \setminus \mathrm{Ptrs} \,\} \\
\mathrm{Stores} & \stackrel{\text{def}}{=} \{\, s \in \mathrm{ProgVars} \to \mathrm{Words} \,\} \\
\mathrm{Heaps} & \stackrel{\text{def}}{=} \{\, h \in \mathrm{Ptrs} \rightharpoonup_{\mathrm{fin}} \mathrm{Words} \,\}
\end{array}
$$

$$
\begin{array}{rcl}
\star & ::= & + \mid - \mid \times \mid \div \mid < \mid = \mid \mathsf{and} \\
E & ::= & \mathtt{x} \mid w \mid \mathsf{not}\ E \mid E \star E \\
C & ::= & \mathsf{skip} \mid \mathtt{x} := E \mid \mathtt{x} := [E] \mid [E] := E \mid \mathsf{alloc}\ \mathtt{x} \\
 & & \mid\ C; C \mid \mathsf{if}\ E\ \mathsf{then}\ C\ \mathsf{else}\ C\ \mathsf{fi} \mid \mathsf{while}\ E\ \mathsf{do}\ C\ \mathsf{od}
\end{array}
$$

Figure 1. Programming language.

Finally, in order to validate the frame rule, our actual model of $\{\{P\}\}\ C\ \{\{Q\}\}$ is slightly more complicated than the rough description given above. Following Birkedal *et al.* [1], we bake the frame property into our model by explicitly quantifying over a logical heap frame $\mathbf{h}_{\mathrm{F}}$ (*i.e.,* a logical heap that is disjoint from the logical state satisfying P) and requiring that C leave $\mathbf{h}_{\mathrm{F}}$ alone. (See Section VI for details.) This approach is necessitated by the fact that "heap locality", a property that is key to proving the frame rule for simpler separation-logic models, does not hold in the presence of GC. Note that it is crucial to express the immutable frame as a *logical* heap, since the GC may very well modify the entire physical heap.

Overview: The rest of the paper is structured as follows. In Section II, we describe the programming language under consideration. In Section III, we present our language of assertions and some basic laws of entailment. In Section IV, we give the key inference rules for our Hoare triples. In Section V, we work through some illustrative examples. In Section VI, we present our model of GCSL, and formalize our assumptions about the operational semantics of the GC. Finally, in Section VII, we conclude with further discussion of several technical issues.

II. Programming Language

We consider a simple programming language with a built-in garbage collector. In order to support non-conservative collectors, we use the least significant bit of each word to distinguish between pointer and non-pointer values. We assume all pointers are 32-bit aligned (*i.e.,* positive multiples of 4), and encode the integer n as $2n + 1$.

Program expressions, E, consist of program variables, constant words, unary and binary operators. Commands, C, consist of the empty command, assignments, memory reads and writes, memory allocation, sequential composition, conditionals and loops.

The memory allocation command, alloc x, expects x to contain the size n of the memory block (in terms of number of words) to be allocated, tag-encoded as $2n + 1$ (as de-

scribed above). After possibly collecting some unreachable memory blocks, it allocates a new block of the right size, fills it with zeros, and makes $\mathtt{x}$ point to the newly allocated block. We use $\mathtt{x}$ both as the input and the output of our basic allocation command for semantic convenience; a more standard allocation command can be encoded as follows:

$$\mathtt{x} := \mathsf{ALLOC}(E) \stackrel{\text{def}}{=} \mathtt{x} := E;\ \mathsf{alloc}\ \mathtt{x}$$

The formal semantics of the memory allocation command is quite involved because $\mathsf{alloc}\ \mathtt{x}$ can invoke the garbage collector, so we defer its presentation to Section VI.

Besides the memory allocation command, the semantics of our programming language is standard. The semantics of expressions is given denotationally as a partial function from stores to words: undefinedness arises from errors such as division by zero. Commands are given a standard small-step semantics with a six-place reduction relation of the form: $C, s, h \rightsquigarrow C', s', h'$. Terminal configurations are of the form skip, s, h. Any other configurations that cannot reduce further are considered erroneous, for example reading from unallocated memory.

III. Assertions

As explained in Section I-B, the assertions of GCSL describe *logical* machine states. We take logical values, $\mathbf{v}$, to be either physical words, w, or logical pointers, $\boldsymbol{\ell}\widehat{+}i$. The latter signifies offset i from an abstract location $\boldsymbol{\ell}$, which is assumed to represent the head of some memory block. (Note that a physical word and a logical pointer may in fact represent the same physical location, but within the logic they are considered distinct entities.) Logical stores, $\mathbf{s}$, map program variables to logical values, while logical heaps, $\mathbf{h}$, map abstract locations and offsets to logical values.

Logical expressions, $\mathbf{E}$, extend the grammar of program expressions with logical values, $\mathbf{v}$, and logical variables, v; the latter may be instantiated with arbitrary logical expressions. The meaning of logical expressions with no free logical variables is defined by a mostly standard partial function from logical stores to logical values. In addition to being undefined in case of division by zero, evaluation of $\mathbf{E}$ is also undefined if $\mathbf{E}$ involves operations such as multiplication of pointers (which have no logical meaning) or comparisons of logical pointers with different heads (whose result is GC-sensitive). Safe pointer arithmetic, such as adding a pointer and a word, is allowed and has a defined semantics. See Section VI-B for details.

As explained in Section I-B, GCSL is divided into two levels. The syntaxes of outer-level assertions, P, and of inner-level assertions, $\mathbf{P}$, are almost the same, except that the latter additionally includes the safety predicate, $\mathsf{safe}(\mathbf{E})$, which asserts that $\mathbf{E}$ denotes a *safe* logical value. A logical value is safe if it is either a physical non-pointer word or a logical head pointer, $\boldsymbol{\ell}\widehat{+}0$, representing the beginning of a *physically allocated* memory block. The model of $\mathbf{P}$

$$\begin{aligned}
\mathrm{Locs} &\stackrel{\text{def}}{=} \{\, \boldsymbol{\ell}_1, \boldsymbol{\ell}_2, \dots \,\} \\
\mathrm{LogPtrs} &\stackrel{\text{def}}{=} \{\, \boldsymbol{\ell}\widehat{+}i \mid \boldsymbol{\ell} \in \mathrm{Locs} \wedge i \in \mathbb{Z} \,\} \\
\mathrm{LogVals} &\stackrel{\text{def}}{=} \{\, \mathbf{v} \in \mathrm{Words} \uplus \mathrm{LogPtrs} \,\} \\
\mathrm{LStores} &\stackrel{\text{def}}{=} \{\, \mathbf{s} \in \mathrm{ProgVars} \to \mathrm{LogVals} \,\} \\
\mathrm{Span}(\mathbf{h}) &\stackrel{\text{def}}{=} \{\, (\boldsymbol{\ell}, i) \in \mathrm{Locs} \times \mathbb{N} \mid i \in \mathrm{dom}(\mathbf{h}(\boldsymbol{\ell})) \,\} \\
\mathrm{LHeaps} &\stackrel{\text{def}}{=} \{\, \mathbf{h} \in \mathrm{Locs} \to \mathbb{N} \rightharpoonup_{\mathrm{fin}} \mathrm{LogVals} \\
&\qquad \mid \mathrm{Span}(\mathbf{h}) \text{ is finite} \,\} \\
\mathrm{LogVars} &\stackrel{\text{def}}{=} \{\, u, v, \dots \,\}
\end{aligned}$$

$$\mathbf{E} \in \mathrm{LExps} ::= v \mid \mathtt{x} \mid \mathbf{v} \mid \mathsf{not}\ \mathbf{E} \mid \mathbf{E} \star \mathbf{E}$$

$$\begin{aligned}
P ::= {} & \mathbf{E} \mid \mathsf{logptr}(\mathbf{E}) \mid \mathsf{word}(\mathbf{E}) \\
& \mid \mathbf{E} \hookrightarrow \mathbf{E} \mid P * P \mid P \mathrel{-\!\!*} P \\
& \mid P \Rightarrow P \mid P \wedge P \mid P \vee P \mid \forall v.\, P \mid \exists v.\, P
\end{aligned}$$

$$\begin{aligned}
\mathbf{P} ::= {} & \mathsf{safe}(\mathbf{E}) \\
& \mid \mathbf{E} \mid \mathsf{logptr}(\mathbf{E}) \mid \mathsf{word}(\mathbf{E}) \\
& \mid \mathbf{E} \hookrightarrow \mathbf{E} \mid \mathbf{P} * \mathbf{P} \mid \mathbf{P} \mathrel{-\!\!*} \mathbf{P} \\
& \mid \mathbf{P} \Rightarrow \mathbf{P} \mid \mathbf{P} \wedge \mathbf{P} \mid \mathbf{P} \vee \mathbf{P} \mid \forall v.\, \mathbf{P} \mid \exists v.\, \mathbf{P}
\end{aligned}$$

$$\mathsf{false} \stackrel{\text{def}}{=} 0; \qquad \mathsf{true} \stackrel{\text{def}}{=} 1; \qquad \neg\mathbf{P} \stackrel{\text{def}}{=} \mathbf{P} \Rightarrow \mathsf{false}$$

$$\begin{aligned}
\mathsf{defined}(\mathbf{E}) &\stackrel{\text{def}}{=} \mathbf{E} = \mathbf{E} \\
\mathsf{nonptr}(\mathbf{E}) &\stackrel{\text{def}}{=} \mathbf{E} = 0 \vee \exists v.\, \mathbf{E} = 2 \times v + 1 \\
\mathsf{offsafe}(\mathbf{E}) &\stackrel{\text{def}}{=} \mathsf{word}(\mathbf{E}) \vee \exists i.\, \mathsf{safe}(\mathbf{E} + i) \\
\mathsf{p}(\{\, \mathbf{E}_1, \dots, \mathbf{E}_n \,\}) &\stackrel{\text{def}}{=} \mathsf{p}(\mathbf{E}_1) \wedge \dots \wedge \mathsf{p}(\mathbf{E}_n) \\
&\quad \text{for } \mathsf{p} \in \{\mathsf{safe}, \mathsf{logptr}, \mathsf{word}, \mathsf{defined}, \mathsf{nonptr}, \mathsf{offsafe}\} \\
\mathbf{E} \hookrightarrow - &\stackrel{\text{def}}{=} \exists v.\, \mathbf{E} \hookrightarrow v \\
\mathbf{E} \hookrightarrow_n \mathbf{E}_0, \dots, \mathbf{E}_{n-1} &\stackrel{\text{def}}{=} \mathbf{E} + 4 \cdot 0 \hookrightarrow \mathbf{E}_0 * \dots * \\
&\qquad \mathbf{E} + 4(n-1) \hookrightarrow \mathbf{E}_{n-1}
\end{aligned}$$

Figure 2. Assertions.

(see Section VI-B below) is thus indexed by a set of such safe locations, $\mathbf{L}$. The remaining assertions include: logical expressions $\mathbf{E}$ themselves (viewed as true iff $\mathbf{E}$ denotes a nonzero physical word), predicates describing whether $\mathbf{E}$ is a logical pointer or a physical word, the points-to predicate, separating conjunction and implication, as well as normal conjunction, disjunction, implication and first-order quantification.

Among the derived assertion forms, of particular note is the assertion $\mathsf{offsafe}(\mathbf{E})$, which stipulates that $\mathbf{E}$ either denotes a physical word or a logical pointer that is equal to some offset from a safe pointer. Whereas our outer-level logic only permits program variables to store safe values, our inner-level logic also permits them to store "offsafe" values.

Entailment of assertions, $\mathbf{P} \models \mathbf{Q}$, states that every valid logical state satisfying $\mathbf{P}$ also satisfies $\mathbf{Q}$, where a valid logical state is one that properly represents some physical one (see Section VI-B for a formal specification of this).

Besides all the standard entailments from intuitionistic BI,

$$\begin{aligned}
\mathsf{nonptr}(\mathbf{E}) &\models \mathsf{safe}(\mathbf{E}) \\
\mathsf{defined}(E) &\models \mathsf{offsafe}(E) \\
\mathbf{E} \hookrightarrow \mathbf{E}' \wedge \mathsf{offsafe}(\mathbf{E}) &\models \mathsf{safe}(\mathbf{E}') \\
E \hookrightarrow \mathbf{E}' &\models \mathsf{safe}(\mathbf{E}') \\
\mathsf{safe}(\mathbf{E}, \mathbf{E}') &\models \mathsf{defined}(\mathbf{E} = \mathbf{E}')
\end{aligned}$$

Figure 3. Assertion entailments.

$$\frac{}{\{\mathtt{x} = v \wedge \mathsf{defined}(E)\}\ \mathtt{x} := E\ \{\mathtt{x} = E[v/\mathtt{x}]\}} \quad \text{(Assign)}$$

$$\frac{}{\{\mathtt{x} = u \wedge E \hookrightarrow v\}\ \mathtt{x} := [E]\ \{\mathtt{x} = v \wedge E[u/\mathtt{x}] \hookrightarrow v\}} \quad \text{(Read)}$$

$$\frac{}{\{E \hookrightarrow - \wedge \mathsf{safe}(E')\}\ [E] := E'\ \{E \hookrightarrow E'\}} \quad \text{(Write)}$$

$$\frac{\{\mathbf{P} \wedge E\}\ C\ \{\mathbf{P} \wedge \mathsf{word}(E)\}}{\{\mathbf{P} \wedge \mathsf{word}(E)\}\ \mathsf{while}\ E\ \mathsf{do}\ C\ \mathsf{od}\ \{\mathbf{P} \wedge \mathsf{not}\ E\}} \quad \text{(While)}$$

$$\frac{\{\mathbf{P}\}\ C\ \{\mathbf{Q}\} \qquad \mathrm{FPV}(\mathbf{R}) \cap \mathrm{Mod}(C) = \emptyset}{\{\mathbf{P} * \mathbf{R}\}\ C\ \{\mathbf{Q} * \mathbf{R}\}} \quad \text{(Frame)}$$

$$\frac{\mathbf{P} \models \mathbf{P}' \qquad \{\mathbf{P}'\}\ C\ \{\mathbf{Q}'\} \qquad \mathbf{Q}' \models \mathbf{Q}}{\{\mathbf{P}\}\ C\ \{\mathbf{Q}\}} \quad \text{(Conseq)}$$

$$\frac{n \geq 0}{\{\mathtt{x} = 2n + 1\}\ \mathsf{alloc}\ \mathtt{x}\ \{\mathtt{x} \hookrightarrow_n 0, \ldots, 0\}} \quad \text{(Alloc)}$$

$$\frac{V \subseteq_{\mathrm{fin}} \mathrm{ProgVars} \qquad \{P \wedge \mathsf{safe}(V)\}\ C\ \{Q \wedge \mathsf{safe}(\mathrm{Mod}(C))\}}{\{\{P\}\}\ C\ \{\{Q\}\}} \quad \text{(Incl)}$$

Figure 4. Selected proof rules.

the five rules in Figure 3 are particularly useful. The first rule says that any physical *non-pointer* word is safe. The second rule says that any well-defined *program* expression E is guaranteed to be offsafe. This somewhat subtle property relies intuitively on the fact that E either denotes a physical word, in which case it is trivially offsafe, or else it must be equivalent to $\mathtt{x}+i$ for some program variable $\mathtt{x}$ and offset i. In that latter case, the outer-level logic dictates that $\mathtt{x}$ must store a safe pointer, and the inner-level logic dictates that $\mathtt{x}$ must store an offsafe pointer, but either way, $\mathtt{x} + i$ will be offsafe. The third rule captures the invariant of our logic that the portion of the logical heap that is actually physically allocated can store only safe values.

The fourth rule, which is a corollary of the previous two, says that any value pointed to by a program expression is safe. The last rule says that equality between two safe expressions is always defined. This follows from the fact that safe logical expressions are logically equivalent if and only if they represent the same physical word, thanks to the semantics of expressions that we will give in Section VI-B.

IV. Key Proof Rules for Partial Correctness

GCSL has two levels of Hoare triples: the outer-level $\{\{P\}\}\ C\ \{\{Q\}\}$, and the inner-level $\{\mathbf{P}\}\ C\ \{\mathbf{Q}\}$. Garbage collection is well-specified at the outer level, while the inner level allows GC-unsafe values (specifically, "offsafe" values) to be stored in program variables. For the most part, the proof rules for both kinds of triples are the usual ones from separation logic, as one would hope. Figure 4 displays some of the more interesting rules. A number of other rules appear in the online appendix [7], including analogous rules for reasoning about total correctness.

$$\frac{}{\{\{E \hookrightarrow -\}\}\ [E] := \mathtt{x}\ \{\{E \hookrightarrow \mathtt{x}\}\}}$$

$$\frac{}{\{\{E \hookrightarrow - \wedge \mathsf{nonptr}(E')\}\}\ [E] := E'\ \{\{E \hookrightarrow E'\}\}}$$

$$\frac{\mathtt{x} \notin \mathrm{FPV}(E) \cup \mathrm{FPV}(\mathbf{E}')}{\{\{E \hookrightarrow \mathbf{E}'\}\}\ \mathtt{x} := [E]\ \{\{\mathtt{x} = \mathbf{E}' \wedge E \hookrightarrow \mathbf{E}'\}\}}$$

$$\frac{n \geq 0}{\{\{E = 2n + 1\}\}\ \mathtt{x} := \mathsf{ALLOC}(E)\ \{\{\mathtt{x} \hookrightarrow_n 0, \ldots, 0\}\}}$$

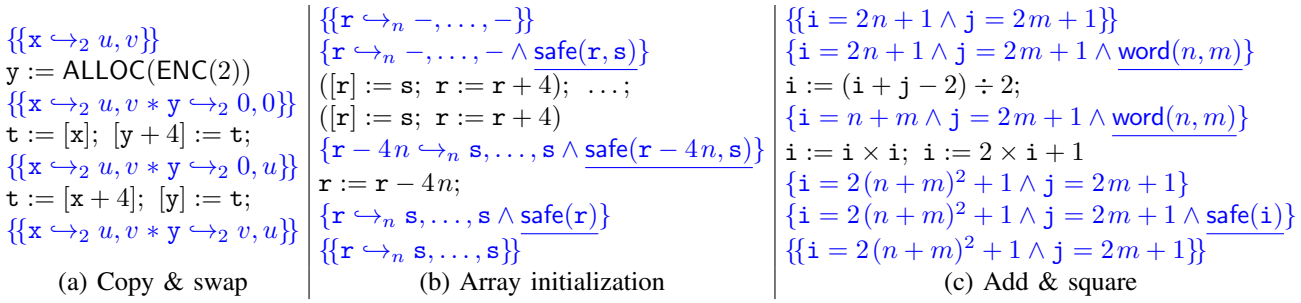
Figure 5. Selected derived outer-level rules.

The inner-level axiom for memory writes (Write) requires in its precondition that the value being written to the heap is safe. This ensures that the heap contains only safe values. In contrast, the assignment axiom (Assign) does not require such a check because the store can contain offsafe values, and by the second entailment axiom from Figure 3, E must be offsafe if it is well-defined.

In the frame rule, $\mathrm{FPV}(\mathbf{R})$ is the set of program variables appearing free in $\mathbf{R}$, while $\mathrm{Mod}(C)$ is the set of program variables appearing on the l.h.s. of assignments in C. The (While) rule requires the expression, E, to denote some physical word (and not be undefined or a logical pointer); this condition has to hold in the precondition and be re-established after each loop iteration. We also have the standard rules for skip, sequential composition, and conditionals, and the standard structural rules: consequence, disjunction, existential (*a.k.a.*, auxiliary variable elimination), substitution. Notably, we do not support the conjunction rule, for reasons we will discuss in Section VII. There are analogous outer-level versions of these rules as well.

The two most interesting rules for outer-level triples are (Incl) and (Alloc). We have already discussed the "inclusion" rule (Incl) in Section I-B: it enables one to prove an outer-level triple using an inner-level one. The (Alloc) rule specifies the behavior of the memory allocator, which is only well-specified at the outer level since it may invoke the garbage collector. The precondition $\mathtt{x} = 2n+1$ accounts for the fact that the alloc routine expects its integer argument to be encoded with a bit tag. Our language does not have an explicit gc() call, but it can be mimicked by $\mathtt{x} := \mathsf{ALLOC}(1)$ for an arbitrary $\mathtt{x}$. This command can be given pre and post P for any P whose free variables do not include $\mathtt{x}$.

Figure 5 contains some *derived* rules for reasoning directly in the outer-level logic without having to use (Incl)



(a) Copy & swap (b) Array initialization (c) Add & square

Figure 6. Simple GCSL examples.

around each basic command. Since the outer-level logic cannot mention the safe predicate, we have two outer-level axioms for memory writes. One requires E' to be a non-pointer word, while the other requires it to be a program variable; both are guaranteed to be safe values.

V. Examples

Figure 6 contains GCSL proof outlines for three small example programs: copy&swap, array initialization, and add&square. The first example demonstrates reasoning in the outer-level logic using derived rules for the basic commands. The second and third examples demonstrate the inner-level logic and how internal pointers and GC-unsafe intermediate values can be used to optimize a simple pointer program and an arithmetic program, respectively. See the online appendix [7] for more examples presented in more detail.

In all cases, accounting for the GC requires relatively little reasoning (if any) beyond what would be required in a traditional separation-logic proof. What little additional reasoning is required primarily concerns the safe predicate, which the proof rules from Sections III and IV make it very easy to manipulate.

Copy & Swap: The first program assumes that the program variable x stores a pointer to a block of size 2 containing values u, v; it then allocates a new block of the same size, fills it with v, u (*i.e.,* the input in reverse order), and stores a pointer to it in y. As the program does not store any GC-unsafe values, it is possible for us to prove it correct using only the outer-level logic, and the proof looks exactly as it would in standard separation logic.

What is going on under the hood is, however, quite different. In the case where u and v themselves represent pointer values, the ALLOC command might perform some garbage collection and relocate the physical pointers that they abstractly represent. What we have proven in this case is that, whatever new physical pointers are stored in this block, they logically correspond to the old pointers stored there, and the new block pointed to by y contains the same new values in reverse order. Furthermore, our reasoning guarantees that the values stored in the memory outside the program's footprint are logically unchanged, even though they might have been physically changed by ALLOC.

Array Initialization: This is a simple program that initializes the array of size n pointed to by r with the value stored in s. The reasoning is completely standard except for the use of the (Incl) rule and safe predicate. In more detail:

1) As r is used to store an internal pointer, and there may be no other reachable pointer to the head of the memory block being initialized, we have to make sure that there is no GC call during the computation. This is guaranteed by (Incl).
2) When we write the value s into the heap, we have to make sure that we store a GC-safe value. This is guaranteed by the safe(s) assumption, which again comes from (Incl).
3) At the end, we have to make sure that the program variables we have modified are GC-safe, which in this case means verifying safe(r).

Add & Square: After the first assignment, the value of i is temporarily unsafe (in fact, offsafe) because at that point it represents an unencoded integer. Therefore, we use (Incl) and then the inner-level rules to verify this example. In this case, it is straightforward to re-establish safe(i) using the first assertion entailment rule from Figure 3.

VI. Semantics & Soundness

In this section, we give a formal specification of the behavior we expect from the GC, and develop a sound model of GCSL connecting logical and physical machine states.

A. Garbage Collector Specification

To specify the garbage collector, we assume that it has an internal invariant, I_{gc}, which is a partial function that, given a set of roots, R, and a heap, h, returns the shape, σ, of the heap if the heap is valid; otherwise, $I_{\text{gc}}(R, h)$ is undefined. Shapes (see Figure 7) are finite partial maps saying which pointers point to the heads of allocated blocks and, if so, how large are the blocks they point to. We call a state (s, h) GC-safe whenever $I_{\text{gc}}(\text{roots}(s), h)$ is defined.

We say that a shape, σ, is valid with respect to a heap, h, and a root set, R, if (1) all pointers deemed allocated by σ are actually allocated in h and (2) all the pointers reachable from R and h according to shape σ are actually

$$
\begin{array}{ll}
\text{Shapes} & \stackrel{\text{def}}{=} \{\, \sigma \in \text{Ptrs} \rightharpoonup_{\text{fin}} \mathbb{N}^+ \,\} \\
\overline{\text{dom}}(\sigma) & \stackrel{\text{def}}{=} \biguplus_{p \in \text{dom}(\sigma), 0 \le i < \sigma(p)} \{\, p + 4i \,\} \\
\text{roots}(s) & \stackrel{\text{def}}{=} \{\, p \in \text{Ptrs} \mid \exists \mathtt{x}.\ p = s(\mathtt{x}) \,\} \\
\text{reach}_0(R, h, \sigma) & \stackrel{\text{def}}{=} R \\
\text{reach}_{n+1}(R, h, \sigma) & \stackrel{\text{def}}{=} \{\, h(p' + 4i) \in \text{Ptrs} \mid \\
 & \qquad p' \in \text{reach}_n(R, h, \sigma) \wedge 0 \le i < \sigma(p') \,\} \\
\text{reach}(R, h, \sigma) & \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}} \text{reach}_n(R, h, \sigma)
\end{array}
$$

$$
\begin{array}{l}
(s, h, \sigma) \cong (s', h', \sigma') \stackrel{\text{def}}{=} \\
\quad \exists r \in \text{Bij}(\text{reach}(\text{roots}(s), h, \sigma), \text{reach}(\text{roots}(s'), h', \sigma')). \\
\qquad \big(\forall \mathtt{x}.\ (s(\mathtt{x}), s'(\mathtt{x})) \in \overline{r}\,\big) \wedge \\
\qquad \big(\forall (p, p') \in r.\ \exists n.\ \sigma(p) = \sigma'(p') = n \wedge \\
\qquad\quad \forall i.\ 0 \le i < n \implies (h(p + 4i), h'(p' + 4i)) \in \overline{r}\big) \\
\quad \text{where } \overline{r} \stackrel{\text{def}}{=} r \cup \{\, (a, a) \mid a \in \text{NonPtrs} \,\}
\end{array}
$$

$$
\begin{array}{l}
[p \mapsto_n w_0, \ldots, w_{n-1}] \stackrel{\text{def}}{=} (\emptyset \mid p + 4 \times 0 \mapsto w_0 \mid \ldots \\
\qquad\qquad\qquad\qquad \mid p + 4(n-1) \mapsto w_{n-1}) \\
[p \mapsto n] \stackrel{\text{def}}{=} \begin{cases} (\emptyset \mid p \mapsto n) & \text{if } n > 0 \wedge p \in \text{Ptrs} \\ \emptyset & \text{if } n = 0 \wedge p = 0 \\ \text{undef} & \text{otherwise} \end{cases}
\end{array}
$$

Figure 7. Auxiliary definitions for GC semantics.

$$
\begin{array}{l}
\forall s, h, \sigma, \mathtt{x}, n \ge 0. \\
\sigma = I_{\text{gc}}(\text{roots}(s), h) \wedge s(\mathtt{x}) = 2n + 1 \implies \\
(\exists C', s', h'.\ \mathsf{alloc}\ \mathtt{x}, s, h \rightsquigarrow C', s', h') \wedge \\
(\forall C', s', h'.\ \mathsf{alloc}\ \mathtt{x}, s, h \rightsquigarrow C', s', h' \implies \\
\quad \exists p, h'', \sigma''.\ C' = \mathsf{skip} \wedge \\
\qquad\qquad \sigma'' \uplus [p \mapsto n] = I_{\text{gc}}(\text{roots}(s'), h') \wedge \\
\qquad\qquad s'(\mathtt{x}) = p \wedge h' = h'' \uplus [p \mapsto_n 0, \ldots, 0] \wedge \\
\qquad\qquad (s, h, \sigma) \cong ((s' \mid \mathtt{x} \mapsto 2n + 1), h'', \sigma''))
\end{array}
$$

Figure 8. Specification of alloc x.

in the domain of σ—*i.e.,* they are head pointers to allocated blocks.

Definition 1. *ValidShape*(R, h, σ) iff $\overline{\text{dom}}(\sigma) \subseteq \text{dom}(h)$ and $\text{reach}(R, h, \sigma) \subseteq \text{dom}(\sigma)$.

Our expectation of the GC is that if $I_{\text{gc}}(R, h) = \sigma$, then σ is valid with respect to R and h and, further, if the user-portion of the heap and the root set are mutated in a safe way (in a way for which σ remains valid), then I_{gc} is still defined on the updated root set and heap, and returns a possibly smaller, but compatible, shape. Formally,

$$
\begin{array}{l}
\quad (\forall R, h, \sigma = I_{\text{gc}}(R, h).\ \textit{ValidShape}(R, h, \sigma)) \\
\wedge\ (\forall R, h, R', h'.\quad \textit{ValidShape}(R', h', I_{\text{gc}}(R, h)) \\
\qquad\qquad \wedge\ (\forall p \notin \overline{\text{dom}}(I_{\text{gc}}(R, h)).\ h'(p) = h(p)) \\
\qquad \implies I_{\text{gc}}(R', h') \subseteq I_{\text{gc}}(R, h))
\end{array}
$$

These conditions are formulated only in terms of physical states (not logical states) and can be implemented both by mark&sweep and by Cheney-style copying collectors.

Figure 8 contains our specification of the alloc x command. Given an initial GC-safe state and x containing the (encoded) size of the block to be allocated, then the configuration $\mathsf{alloc}\ \mathtt{x}, s, h$ is not stuck and can reduce only to terminal configurations. The state that it reduces to is GC-safe; the value of x in the new store contains p, a pointer to the newly allocated block; and the new heap consists of the newly allocated block, $[p \mapsto_n 0, \ldots, 0]$, and the remainder, h'', that is isomorphic to the initial heap. In the special case that $n = 0$, the specification of alloc simplifies to selecting a new triple (s', h', σ') that is isomorphic to the original one.

$$
\begin{array}{ll}
\text{Table} & \stackrel{\text{def}}{=} \{\, \mathbf{T} \in \text{Locs} \rightharpoonup_{\text{fin}} \text{Ptrs} \times \mathbb{N}^+ \,\} \\
\text{phyv}_{\mathbf{T}}(\mathbf{v}) & \stackrel{\text{def}}{=} \begin{cases} w & \text{if } \mathbf{v} = w \in \text{Words} \\ p + i & \text{if } \mathbf{v} = \boldsymbol{\ell} \widehat{+} i \wedge \mathbf{T}(\boldsymbol{\ell}) = (p, n) \\ \text{undef} & \text{otherwise} \end{cases} \\
\text{phyh}_{\mathbf{T}}(\mathbf{h}) & \stackrel{\text{def}}{=} \biguplus_{(p,n) = \mathbf{T}(\boldsymbol{\ell})} [p \mapsto_n \text{phyv}_{\mathbf{T}}(\mathbf{h}(\boldsymbol{\ell})(0)), \ldots, \\
 & \qquad\qquad \text{phyv}_{\mathbf{T}}(\mathbf{h}(\boldsymbol{\ell})(n-1))] \\
\text{shape}(\mathbf{T}) & \stackrel{\text{def}}{=} \biguplus_{(p,n) = \mathbf{T}(\boldsymbol{\ell})} [p \mapsto n] \\
\text{Safe}(\mathbf{L}) & \stackrel{\text{def}}{=} \{\, \boldsymbol{\ell} \widehat{+} 0 \mid \boldsymbol{\ell} \in \mathbf{L} \,\} \cup \text{NonPtrs} \quad \text{for } \mathbf{L} \subseteq \text{Locs} \\
\mathbf{s} \sim_{\mathbf{T}} s & \text{iff } \forall \mathtt{x}.\ s(\mathtt{x}) = \text{phyv}_{\mathbf{T}}(\mathbf{s}(\mathtt{x})) \\
\mathbf{s} \approx_{\mathbf{T}} s & \text{iff } \mathbf{s} \sim_{\mathbf{T}} s \wedge \forall \mathtt{x}.\ \mathbf{s}(\mathtt{x}) \in \text{Safe}(\text{dom}(\mathbf{T})) \\
\mathbf{h} \approx_{\mathbf{T}} h & \text{iff } \big(\forall \boldsymbol{\ell}.\ \forall (p, n) = \mathbf{T}(\boldsymbol{\ell}). \\
 & \qquad \text{dom}(\mathbf{h}(\boldsymbol{\ell})) = \{\, 0, \ldots, n-1 \,\} \wedge \\
 & \qquad \forall i < n.\ \mathbf{h}(\boldsymbol{\ell})(i) \in \text{Safe}(\text{dom}(\mathbf{T}))\big) \wedge \\
 & \quad \text{phyh}_{\mathbf{T}}(\mathbf{h}) \subseteq h \wedge \\
 & \quad \text{shape}(\mathbf{T}) \subseteq I_{\text{gc}}(\text{dom}(\text{shape}(\mathbf{T})), h) \\
\mathbf{h}_1 \# \mathbf{h}_2 & \stackrel{\text{def}}{=} \text{Span}(\mathbf{h}_1) \cap \text{Span}(\mathbf{h}_2) = \emptyset \\
\mathbf{h}_1 \uplus \mathbf{h}_2 & \stackrel{\text{def}}{=} \begin{cases} \lambda \boldsymbol{\ell}.\ \mathbf{h}_1(\boldsymbol{\ell}) \uplus \mathbf{h}_2(\boldsymbol{\ell}) & \text{if } \mathbf{h}_1 \# \mathbf{h}_2 \\ \text{undef} & \text{otherwise} \end{cases}
\end{array}
$$

Figure 9. Relating logical and physical states.

B. Semantics of Assertions and Triples

In order to define the meaning of program logic judgments, we need several auxiliary definitions shown in Figure 9. A translation table, $\mathbf{T}$, is a finite partial function mapping safe logical pointers to their corresponding physical pointers and block sizes. The function $\text{phyv}_{\mathbf{T}}(\mathbf{v})$ translates logical values to physical words by looking up locations in the table, $\mathbf{T}$. It is defined if and only if $\mathbf{v}$ is offsafe. The next definition, $\text{phyh}_{\mathbf{T}}(\mathbf{h})$, lifts this function to translate logical heaps to physical heaps. Note that this translation covers only locations and offsets in $\mathbf{T}$; all other locations are not translated and, hence, the logical heap may contain extra junk in it that is not governed by $\mathbf{T}$.

A logical store *represents* a physical store with respect to a translation table ($\mathbf{s} \sim_{\mathbf{T}} s$) if the physical store contents are the translation of the logical store contents. Implicitly, this means that the logical store contains only offsafe values. A logical store *safely represents* a physical store ($\mathbf{s} \approx_{\mathbf{T}} s$) if, in addition, it contains only safe values—*i.e.,* either non-pointer words or logical pointers in the domain of $\mathbf{T}$.

A logical heap, $\mathbf{h}$, *safely represents* a physical heap, h, with respect to a table, $\mathbf{T}$, written $\mathbf{h} \approx_{\mathbf{T}} h$, if the following

$$[\![v]\!]_{\mathbf{s}} \stackrel{\text{def}}{=} \text{undef}$$
$$[\![\mathtt{x}]\!]_{\mathbf{s}} \stackrel{\text{def}}{=} \mathbf{s}(\mathtt{x})$$
$$[\![\mathbf{v}]\!]_{\mathbf{s}} \stackrel{\text{def}}{=} \mathbf{v}$$

$$[\![\mathsf{not}\ \mathbf{E}]\!]_{\mathbf{s}} \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } [\![\mathbf{E}]\!]_{\mathbf{s}} = 0 \\ 0 & \text{if } [\![\mathbf{E}]\!]_{\mathbf{s}} \in \text{NonPtrs}\backslash\{0\} \\ \text{undef} & \text{otherwise} \end{cases}$$

$$[\![\mathbf{E}_1 \star \mathbf{E}_2]\!]_{\mathbf{s}} \stackrel{\text{def}}{=} \begin{cases} w_1 \star w_2 & \text{if } [\![\mathbf{E}_1]\!]_{\mathbf{s}} = w_1 \in \text{Words} \wedge [\![\mathbf{E}_2]\!]_{\mathbf{s}} = w_2 \in \text{Words} \\ \boldsymbol{\ell}\widehat{+}(i+w) & \text{if } \star = + \wedge [\![\mathbf{E}_k]\!]_{\mathbf{s}} = \boldsymbol{\ell}\widehat{+}i \wedge [\![\mathbf{E}_{3-k}]\!]_{\mathbf{s}} = w \text{ for } k = 1,2 \\ \boldsymbol{\ell}\widehat{+}(i-w) & \text{if } \star = - \wedge [\![\mathbf{E}_1]\!]_{\mathbf{s}} = \boldsymbol{\ell}\widehat{+}i \wedge [\![\mathbf{E}_2]\!]_{\mathbf{s}} = w \\ i - j & \text{if } \star = - \wedge [\![\mathbf{E}_1]\!]_{\mathbf{s}} = \boldsymbol{\ell}\widehat{+}i \wedge [\![\mathbf{E}_2]\!]_{\mathbf{s}} = \boldsymbol{\ell}\widehat{+}j \\ i < j & \text{if } \star = < \wedge [\![\mathbf{E}_1]\!]_{\mathbf{s}} = \boldsymbol{\ell}\widehat{+}i \wedge [\![\mathbf{E}_2]\!]_{\mathbf{s}} = \boldsymbol{\ell}\widehat{+}j \\ i = j & \text{if } \star = = \wedge [\![\mathbf{E}_1]\!]_{\mathbf{s}} = \boldsymbol{\ell}\widehat{+}i \wedge [\![\mathbf{E}_2]\!]_{\mathbf{s}} = \boldsymbol{\ell}\widehat{+}j \\ \boldsymbol{\ell} = \boldsymbol{\ell}' & \text{if } \star = = \wedge [\![\mathbf{E}_1]\!]_{\mathbf{s}} = \boldsymbol{\ell}\widehat{+}0 \wedge [\![\mathbf{E}_2]\!]_{\mathbf{s}} = \boldsymbol{\ell}'\widehat{+}0 \\ 0 & \text{if } \star = = \wedge [\![\mathbf{E}_k]\!]_{\mathbf{s}} = \boldsymbol{\ell}\widehat{+}4i \wedge i \geq 0 \wedge [\![\mathbf{E}_{3-k}]\!]_{\mathbf{s}} \in \text{NonPtrs for } k = 1,2 \\ \text{undef} & \text{otherwise} \end{cases}$$

Figure 10. Semantics of assertion expressions, $[\![\mathbf{E}]\!] \in \text{LStores} \rightharpoonup \text{LogVals}$.

$$\{\mathbf{P}\}\ C\ \{\mathbf{Q}\} \text{ iff } \forall \mathbf{s}, \mathbf{h}, \mathbf{h}_{\text{F}}, \mathbf{T}, s, h, C', s', h'.$$
$$\mathbf{s}, \mathbf{h} \models_{\text{dom}(\mathbf{T})} \mathbf{P} \wedge \mathbf{s} \sim_{\mathbf{T}} s \wedge \mathbf{h} \uplus \mathbf{h}_{\text{F}} \approx_{\mathbf{T}} h \wedge C, s, h \rightsquigarrow^* C', s', h' \implies$$
$$(\exists C'', s'', h''.\ C', s', h' \rightsquigarrow C'', s'', h'') \vee$$
$$(\exists \mathbf{s}', \mathbf{h}'.\ C' = \mathsf{skip} \wedge \mathbf{s}', \mathbf{h}' \models_{\text{dom}(\mathbf{T})} \mathbf{Q} \wedge (\forall \mathtt{x} \notin \text{Mod}(C).\ \mathbf{s}'(\mathtt{x}) = \mathbf{s}(\mathtt{x})) \wedge \mathbf{s}' \sim_{\mathbf{T}} s' \wedge \mathbf{h}' \uplus \mathbf{h}_{\text{F}} \approx_{\mathbf{T}} h')$$

$$\{\{P\}\}\ C\ \{\{Q\}\} \text{ iff } \forall \mathbf{s}, \mathbf{h}, \mathbf{h}_{\text{F}}, \mathbf{T}, s, h, C', s', h'.$$
$$\mathbf{s}, \mathbf{h} \models P \wedge \mathbf{s} \approx_{\mathbf{T}} s \wedge \mathbf{h} \uplus \mathbf{h}_{\text{F}} \approx_{\mathbf{T}} h \wedge C, s, h \rightsquigarrow^* C', s', h' \implies$$
$$(\exists C'', s'', h''.\ C', s', h' \rightsquigarrow C'', s'', h'') \vee$$
$$(\exists \mathbf{s}', \mathbf{h}', \mathbf{T}'.\ C' = \mathsf{skip} \wedge \mathbf{s}', \mathbf{h}' \models Q \wedge (\forall \mathtt{x} \notin \text{Mod}(C).\ \mathbf{s}'(\mathtt{x}) = \mathbf{s}(\mathtt{x})) \wedge \mathbf{s}' \approx_{\mathbf{T}'} s' \wedge \mathbf{h}' \uplus \mathbf{h}_{\text{F}} \approx_{\mathbf{T}'} h')$$

Figure 11. Semantics of the partial correctness judgments.

three conditions hold: (1) the logical heap's blocks are of the same size as the table says they are supposed to be and contain only safe values; (2) part of the physical heap is the translation of the logical heap (this corresponds to the "user" part of the heap—the physical heap may have additional cells for internal use by the GC, *e.g.,* for storing the sizes of the allocated blocks); and (3) the physical heap is GC-safe.

Logical Expressions: Figure 10 presents the semantics of logical expressions. Particularly notable is our semantics for equality comparison involving logical pointers: (1) two logical pointers that are offsets from the same abstract location are equal iff their offsets are equal, (2) head pointers are equal iff they point to identical abstract locations, and (3) "possibly allocated" pointers (*i.e.,* those that are offset $4i$ from some abstract location, where $i \geq 0$) are not equal to any non-pointer. These cases ensure that equality between two safe expressions is always defined (*cf.* the fifth entailment in Figure 3).

Assertions: The semantics of assertions is given by a judgment of the form $\mathbf{s}, \mathbf{h} \models_{\mathbf{L}} \mathbf{P}$, where $\mathbf{L}$ is the set of safe locations. The definition of this judgment is straightforward: $\mathbf{s}, \mathbf{h} \models_{\mathbf{L}} \mathsf{safe}(\mathbf{E})$ holds iff $[\![\mathbf{E}]\!]_{\mathbf{s}} \in \text{Safe}(\mathbf{L})$, and we give the intuitionistic BI semantics to the logical connectives; see the online appendix [7] for the full details. The meaning of outer-level assertions, P, is independent of $\mathbf{L}$, since they do not contain safety predicates. Therefore, for convenience we often drop the $\mathbf{L}$ subscript when referring to outer-level assertions. Assertion entailment is defined as follows:

Definition 2 (Entailment)**.** $\mathbf{P} \models \mathbf{Q}$ iff for all $\mathbf{s}$, $\mathbf{h}$, $\mathbf{h}_{\text{F}}$, $\mathbf{T}$, s, h, if $\mathbf{s} \sim_{\mathbf{T}} s$ and $\mathbf{h} \uplus \mathbf{h}_{\text{F}} \approx_{\mathbf{T}} h$ and $\mathbf{s}, \mathbf{h} \models_{\text{dom}(\mathbf{T})} \mathbf{P}$, then $\mathbf{s}, \mathbf{h} \models_{\text{dom}(\mathbf{T})} \mathbf{Q}$.

The definition quantifies only over valid logical states, namely ones that represent some physical states: the logical store, $\mathbf{s}$, represents some physical store, s, and the logical heap, $\mathbf{h}$, can be extended to a bigger heap, $\mathbf{h} \uplus \mathbf{h}_{\text{F}}$, representing some physical heap, h. Restricting our attention to valid logical states is key in proving the assertion entailments shown in Figure 3.

Program Judgments: We proceed to the semantics of the inner- and outer-level judgments. Figure 11 displays the meaning of the triples for closed judgments, *i.e.,* ones with no free logical variables. (See the online appendix [7] for the meaning of open judgments.) Both judgments require that whenever the command C is executed in a physical state (s, h) related to a logical state $(\mathbf{s}, \mathbf{h})$ satisfying the precondition together with a remaining frame logical heap $\mathbf{h}_{\text{F}}$, then the execution never gets erroneously stuck: after any number of steps $(C, s, h \rightsquigarrow^* C', s', h')$, either C', s', h' can further reduce or it is a terminal configuration that is related to a logical configuration $(\mathbf{s}', \mathbf{h}')$ satisfying the postcondition together with the same frame logical heap, $\mathbf{h}_{\text{F}}$. As explained in Section I-B, this quantification over the frame heap is crucial in order to support the frame rule.

Logical and physical states are related with respect to a

translation table, $\mathbf{T}$. Note that the inner-level judgment uses $\mathbf{s} \sim_{\mathbf{T}} s$, whereas the outer-level judgment uses $\mathbf{s} \approx_{\mathbf{T}} s$. This is because the store at the inner level can contain offsafe values, whereas the store at the outer level can contain only safe values. Furthermore, at the inner level, the final states are related with respect to the same translation table, $\mathbf{T}$, whereas the outer-level judgment permits a different table, $\mathbf{T}'$. This captures our intention that no garbage collection occurs during execution of commands at the inner level, while it can occur at the outer level.

Finally, both judgments require that the initial and final logical stores agree on variables not modified by C. (Since we get to *choose* whatever final logical store we want to represent the physical one, this requirement is not trivially satisfied.) One can view the requirement as the store analogue of the frame heap $\mathbf{h}_{\mathrm{F}}$, and indeed it is needed for the soundness of the frame rule.

VII. Discussion

Conjunction Rule: Our semantics does not validate the conjunction rule. The reason is that physical words can be represented as either one of two different logical words: either as the physical word itself or as an offset from an abstract location. For example, the program $\mathtt{x} := \mathtt{x}$ with precondition $\mathtt{x} = 0 \wedge \mathtt{y} \hookrightarrow -$ can be given two incompatible postconditions $\mathtt{x} = 0$ and $\mathsf{logptr}(\mathtt{x})$ (the latter holds only semantically and only because we know that $\mathtt{x} = \mathtt{y} - n$ for some n; it cannot be derived using the assignment axiom) and so by the conjunction rule, we would get a contradiction. We do not think that the conjunction rule is inherently unsound, but our semantics does not support it. That said, we do not believe this to be of great practical concern, since the conjunction rule is often unsound in concurrent program logics [12] and is not frequently used in practice.

Full Abstraction: Calcagno *et al.* [3, Theorem 8] prove that the logic based on their total-states model is fully abstract with respect to their operational semantics, namely that two programs are observationally equivalent if and only if they satisfy the same correctness assertions. Since we are dealing with a much lower-level language, we cannot expect such a result to hold.

Xor-Swap Example: We conclude with a short example that illustrates a limitation of our program logic. Consider the following implementation for a variable swap operation:

$$\{\mathtt{x} = v \wedge \mathtt{y} = w\}$$
$$\mathtt{x} := \mathtt{x} \mathsf{\ xor\ } \mathtt{y};\quad \mathtt{y} := \mathtt{x} \mathsf{\ xor\ } \mathtt{y};\quad \mathtt{x} := \mathtt{x} \mathsf{\ xor\ } \mathtt{y}$$
$$\{\mathtt{x} = w \wedge \mathtt{y} = v\}$$

While this triple holds semantically for every $v, w \in \mathrm{LogVals}$, we can prove it only for $v, w \in \mathrm{Words}$. The issue is that $\mathrm{LogVals}$ does not contain a useful value to represent the xor of logical pointers. A possible solution would be to change the semantics of inner-level expressions to take the translation table as an argument and return a physical word. We leave this direction to be explored in future work.

Acknowledgments

We would like to thank Peter O'Hearn, Deepak Garg, and Jacob Thamsborg for helpful feedback on an earlier draft.

References

[1] L. Birkedal, N. Torp-Smith, and H. Yang, "Semantics of separation-logic typing and higher-order frame rules," *LMCS*, vol. 2, no. 5:1, 2006.

[2] C. Calcagno, "Semantic and logical properties of stateful programming," Ph.D. dissertation, University of Genova, 2002.

[3] C. Calcagno, P. W. O'Hearn, and R. Bornat, "Program logic and equivalence in the presence of garbage collection," *TCS*, vol. 298, no. 3, pp. 557–581, 2003.

[4] P. J. Cohen, *Set Theory and the Continuum Hypothesis*. W. A. Benjamin, 1966.

[5] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni, "Modular verification of assembly code with stack-based control abstractions," in *PLDI*, 2006, pp. 401–414.

[6] C.-K. Hur and D. Dreyer, "A Kripke logical relation between ML and assembly," in *POPL*, 2011.

[7] C.-K. Hur, D. Dreyer, and V. Vafeiadis, "Separation logic in the presence of garbage collection (Technical appendix)," 2011, Available from the authors' website: `http://www.mpi-sws.org/~dreyer/papers/gcsl/`.

[8] I. Lakatos, *Proofs and Refutations*. Cambridge University Press, 1976.

[9] A. McCreight, T. Chevalier, and A. P. Tolmach, "A certified framework for compiling and executing garbage-collected languages," in *ICFP*, 2010, pp. 273–284.

[10] A. McCreight, Z. Shao, C. Lin, and L. Li, "A general framework for certifying garbage collectors and their mutators," in *PLDI*, 2007, pp. 468–479.

[11] M. O. Myreen, "Reusable verification of a copying collector," in *VSTTE*, 2010, pp. 142–156.

[12] P. W. O'Hearn, "Resources, concurrency and local reasoning," *TCS*, vol. 375, no. 1–3, pp. 271–307, May 2007.

[13] P. W. O'Hearn, J. Reynolds, and H. Yang, "Local reasoning about programs that alter data structures," in *CSL*, 2001, pp. 1–19.

[14] J. C. Reynolds, "Intuitionistic reasoning about shared mutable data structure," in *Millennial Perspectives in Computer Science*, J. Davies, B. Roscoe, and J. Woodcock, Eds. Houndsmill, Hampshire: Palgrave, 2000, pp. 303–321.

[15] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *LICS*, 2002, pp. 55–74.

[16] N. Torp-Smith, L. Birkedal, and J. C. Reynolds, "Local reasoning about a copying garbage collector," *TOPLAS*, vol. 30, no. 4, 2008.