A General Framework for Certifying Garbage Collectors and Their Mutators

Andrew McCreight[†] Zhong Shao[†] Chunxiao Lin[‡] Long Li[‡]

[†]Department of Computer Science
Yale University
New Haven, CT 06520-8285, U.S.A.
{aem, shao}@cs.yale.edu

[‡]Department of Computer Science and Technology
University of Science and Technology of China
Hefei, Anhui 230026, China
{cxlin3, liwis}@mail.ustc.edu.cn

Abstract

Garbage-collected languages such as Java and C# are becoming more and more widely used in both high-end software and real-time embedded applications. The correctness of the GC implementation is essential to the reliability and security of a large portion of the world's mission-critical software. Unfortunately, garbage collectors—especially incremental and concurrent ones—are extremely hard to implement correctly. In this paper, we present a new uniform approach to verifying the safety of both a mutator and its garbage collector in Hoare-style logic. We define a formal garbage collector interface general enough to reason about a variety of algorithms while allowing the mutator to ignore implementation-specific details of the collector. Our approach supports collectors that require read and write barriers. We have used our approach to mechanically verify assembly implementations of mark-sweep, copying and incremental copying GCs in Coq, as well as sample mutator programs that can be linked with any of the GCs to produce a fully-verified garbage-collected program. Our work provides a foundation for reasoning about complex mutator-collector interaction and makes an important advance toward building fully certified production-quality GCs.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.2.4 [*Software Engineering*]: Software/Program Verification — correctness proofs, formal methods

General Terms Languages, Verification

Keywords Garbage Collection, Abstract Data Type, Assembly Code Verification, Separation Logic, Proof-Carrying Code

1. Introduction

Type-safe programming languages with automatic memory management such as Java [15] and C# [21] allow the automatic verification of basic program properties, improving software security and reliability. However, implementations of these languages are not truly safe unless their entire runtime systems, including their garbage collectors, are safe. Because of this, errors in the GC can lead to security problems. This is especially bad because garbage collectors are often used when running untrusted code.

Unfortunately, garbage collectors—especially incremental and concurrent ones—are extremely hard to implement correctly. Bugs can be caused by incorrect interaction of the collector and the mutator or the violation of complex unstated invariants. These bugs are also often difficult to find and fix. This is not merely a theoretical concern—last year, GC-related bugs were fixed in the Mozilla and Internet Explorer web browsers which had the possibility of allowing remote attackers to run arbitrary code [34, 36], due to either the violation of a mutator invariant or an incorrect garbage collector implementation.

The importance of correct garbage collector implementation will only grow as more system critical code is written in garbage collected languages. For instance, the Singularity Project [22] is working on building an operating system using as much "managed code" as possible. As efforts like this and the Verisoft Project [38] check more of the operating system kernel, garbage collectors will make up an increasing portion of possibly unsafe code, if left unverified.

Formally verifying the safety of the mutator, the garbage collector, and the interaction between them in a single system will improve the reliability of systems with automated memory management. This is because we will only need to trust a proof checker and definitions of the machine's behavior and safety, not the implementation of the mutator or garbage collector.

Java and C# programs run on a wide variety of computing platforms ranging from small embedded devices, to multi-core machines, to large-scale parallel computers. To support these programs efficiently and reliably on different platforms, the underlying run time system will likely provide a variety of garbage collectors, and we must be able to reason about all of them.

In this paper, we present a powerful new framework for reasoning about general mutator-collector interfaces and building certified garbage collectors. We have used our framework to mechanically verify the safety of assembly language implementations of mark-sweep, Cheney copying and Baker incremental copying collectors. While there has been much previous work on garbage collector verification (such as [11, 3, 41, 19, 4, 44, 30, 20]), our system formalizes GC safety in a way that allows separate verification of both the mutator and the collector by abstracting away implementation-specific details of the collector from the mutator, while still allowing verification of partial correctness. We show how this approach is used in a Hoare-style logic, using a MIPS-like abstract machine. After we present the general methodology, we discuss how to apply it to verify a mutator program and the assembly-code implementation of a few standard GC algorithms—including an incremental copying collector that requires a read barrier. We view the verifi-

PLDI'07 June 11–13, 2007, San Diego, California, USA.

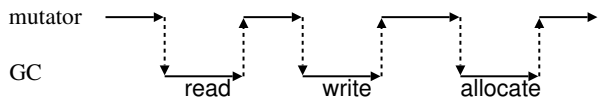

Figure 1. High level view of mutator and GC as threads

```
listSum (word* list) {
  int sum = 0;
  while (list != NULL) {
    sum += first(list);
    list = second(list);
  }
  return sum;
}
```

Figure 2. Example mutator

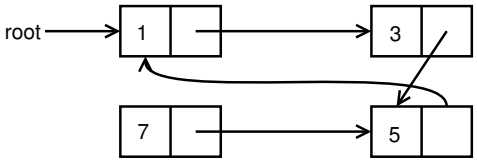

Figure 3. Abstract heap

cation of incremental collectors, which are finer grained than stop-the-world collectors, as a stepping stone towards our ultimate goal of verifying concurrent collectors, which are notoriously difficult to implement correctly [11]. Our paper builds upon previous work on program verification but makes the following new contributions:

- As far as we know, our work is the first to successfully certify the real machine-level implementation of mutator programs together with a variety of standard garbage collectors. Doing our verification at the assembly level—on real code—forces us to address every aspect of the mutator-collector interaction including read and write barriers and other low-level operations which are often abstracted away at the source level. Our system is fully mechanized in Coq [10] and can be directly used to construct foundational proof-carrying code [1]. Our implementation is available at [29].
- In fact, as we'll show later in the paper, our framework can certify the mutator-collector interface for many different GC algorithms. The properties verified about the GC can range from simple type safety to partial correctness, allowing the GC to be used with mutator programs with different correctness requirements. Our thread-centric view of GC safety also extends naturally to the concurrent setting.
- More specifically, we introduce two novel concepts—an abstract state representation predicate *repr* and a garbage collector behavior relation *gcStep*—to help define a uniform and general mutator-collector interface. The *repr* predicate relates an abstract state seen by the mutator to its concrete binary representation seen by the collector. The *gcStep* relation specifies the high-level behavior of the garbage collector. We show that these concepts capture the essence of mutator-collector interaction even in the presence of read and write barriers.
- As far as we know, our work is the first mechanized verification of the Baker incremental copying garbage collector.

Our intention is to support languages like Java and C# as mutators using a GC-aware certifying compiler that emits typed assembly language (TAL). Prior work describes how to translate Java-like languages into TAL [9] and how to represent TAL using our style of program logic [12]. In addition, Lin *et al.* have demonstrated how to represent TAL within our garbage collection framework [27].

The rest of this paper is organized as follows: we first discuss the generic interface using examples and then describe the abstract machine and its program logic. We then present the generic interface in more detail, giving specifications for the interface of the collector, and some abstract guarantees for GCs. In Section 5, we discuss the specification and verification of an example. Section 6 discusses the specifications of the heaps for some actual GCs. In Section 7 we discuss our Coq implementation. Finally, we discuss related work and conclude.

2. A general garbage collector interface

There are a wide variety of garbage collection algorithms [26], which interact in a variety of ways with mutators. At the same time, there are a range of properties about the mutators we may wish to verify, from type safety to partial correctness. Informally, we can think of the mutator and the garbage collector as two separate threads, interacting via a fixed number of operations, as illustrated in Figure 1. Each of these "threads" has its own view of the machine, which we want to formalize.

2.1 Basic operations

Three basic operations comprise the mutator's interface with the object heap, and thus the garbage collector: the mutator can *read* the field of an object, *write* a new value to a field of an object, or *allocate* a fresh object. We also refer to reads and writes as *read barriers* and *write barriers*. Collection may occur during any of these operations, depending on the algorithm.

Consider the pseudo code program `listSum` in Figure 2 that sums up a linked list. We will use this as a running example. The procedure takes a single argument `list` that is a linked list. It loops through the list, adding the value stored in each element, until it reaches the end. The fourth and fifth lines of this procedure use read barriers to retrieve the first and second fields of `list`. For the purposes of this paper, we will be attempting to verify that this procedure correctly computes the sum of the list in the presence of garbage collection.

2.2 Abstract and concrete states

By looking at the implementation of one of these operations in a particular collector we can develop a specification given in terms of the gritty garbage collector machinery, but the entire purpose of read and write barriers is to allow the mutator to go about its business without concern for the cruel realities of the world. It would be a shame if we had to give that up simply because we want to mechanically verify our programs. To avoid this, we make explicit the white lies the collector tells the mutator. Instead of having the mutator reason about the harsh *concrete state* that is the actual heap and registers of the machine our program is running on, we permit it to primarily reason about a higher level *abstract state*. Because different collectors can have the same abstract interface, we can verify a mutator once and combine it with different collectors. The abstract state corresponds to what can be observed of the concrete state using only the basic operations given in Sec 2.3.

In Figure 3, we give an example of an abstract state. For simplicity, in this section we will refer to an object containing a value *n* in the first field as *object n*. The root points to a cyclic linked list containing the objects 1, 3 and 5. Object 7 is unreachable and points

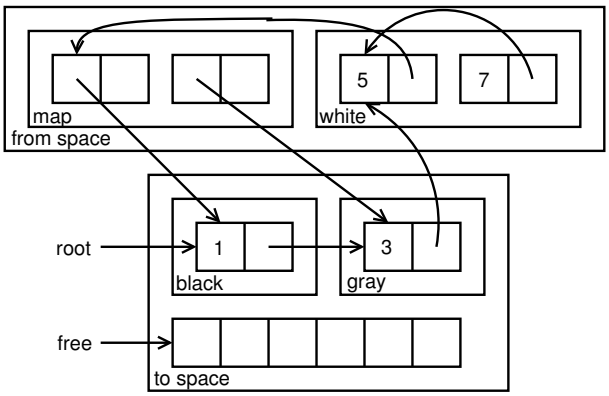

Figure 4. Baker collector concrete heap

to object 5. This is the sort of view we want the mutator to have, where, say, dereferencing the second field of object 5 will give us a pointer to object 1. If we allocate a new object, a fresh object is produced, but the mutator doesn't care where it came from.

Next, consider Figure 4, where we give a possible concrete representation of this abstract heap in a Baker collector [2]. One difference from the abstract heap is that we have a block of memory that *free* points to, where new objects are allocated from. But there are more differences than that. The Baker collector is an incremental copying collector (discussed in more detail in Section 6.3), which means that collection is done by copying all reachable objects from one area (the *from-space*) to another (the *to-space*), and further that this copying is done a little bit at a time, so the mutator may run in a state where the heap has been only partially copied.

In this example, object 1 has been copied from the from-space to the to-space, and its fields have been updated (it is a *black object*), and object 3 has been copied to the to-space, but its fields have not been updated (it is a *gray object*). Finally, objects 5 and 7 have not been copied (and are thus *white objects*). Assigning these classes of objects these 3 colors is known as the tricolor abstraction [11]. In addition to these objects, we also have the original copies of objects 1 and 3, which contain, in their first fields, pointers to the new locations of these objects. We call these *map* objects.

Object 1 points to a to-space object, while objects 3, 5 and 7 still point to from-space objects, Object 5 points to the old copy of object 1. If the mutator could suddenly load the value of the second field of object 5, then load the first field of that object, it would be able to discover the GC is lying because the value would be a pointer, and not 1 as expected. We must carefully construct our interface to prevent this, and require that all interaction with the object heap occurs through read or write barriers.

In our example, we have seen three types of abstraction. First, there is the simple hiding of heap data: the collector knows about the free block (and will update it as objects are allocated and collected), but the mutator does not. Second, constraints on the object heap are hidden: the collector knows that objects 5 and 7 are in the from-space and objects 1 and 3 are in the to-space, but the mutator does not. Finally, and perhaps most dramatically, the state of the objects or even the roots may differ between views: in our example, the mutator and collector do not agree, for instance, on the value of the second field of object 5. This last type of abstraction is not needed for stop-the-world or non-copying incremental collectors.

2.3 The collector interface

To allow the mutator to reason about the abstract level, we must create an abstract garbage collector interface. This interface has three parts:

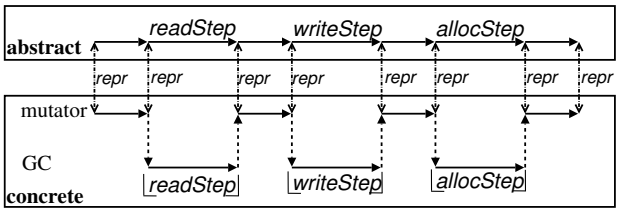

Figure 5. High level view of GC interface

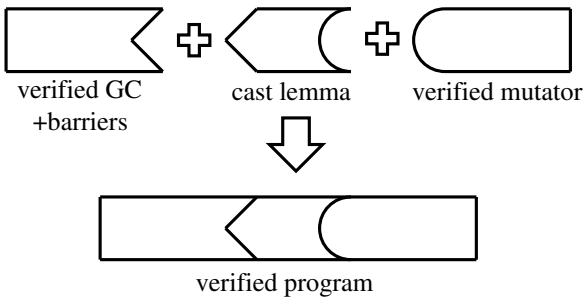

Figure 6. Connecting collector and mutator

1. the abstract specification of each operation, including the behavior (given by *readStep*, *writeStep* and *allocStep*), along with the specification of the actual collection, again at the abstract level, which is given by a binary state predicate *gcStep*
2. a representation predicate *repr* that defines the representation of an abstract state within a concrete state, along with rules to allow some manipulation of the state
3. an embedding function that transforms an abstract specification to a concrete specification using *repr*

We show how the interface is related to the concrete behavior of the machine in Figure 5. The top box contains the series of abstract states seen by the mutator. Edges corresponding to GC activity are labeled with the abstract specification of that action, while unlabeled edges represent mutator activity. The abstract behavior of the allocator (*allocStep*) is given in terms of the abstract behavior of the collector (*gcStep*). We can change *gcStep* depending on the property of the mutator, such as type safety or partial correctness, that we wish to verify.

The lower box contains the series of concrete states of the actual execution of the program, and corresponds to the diagram shown in Figure 1. The specification of the concrete behavior of the GC is derived from the abstract behavior by the function $\lfloor\ \rfloor$, defined in Section 4.3. For example, the concrete behavior of the read barrier is $\lfloor readStep \rfloor$. Each abstract state in the upper box is related to a concrete state in the lower box by the relation *repr*. If at a given point in the execution the abstract state is $\mathbb{A}$ and the concrete state is $\mathbb{S}$, then the relation $repr\ \mathbb{S}\ \mathbb{A}$ holds. A key point is that the mutator only needs to know a few basic properties of *repr*, not its definition.

2.4 Connecting the collector and the mutator

We want to be able to separately verify the mutator and the collector. This goal affects how each part is verified and how they are combined. Figure 6 gives an overview of this process. The collector and the mutator are each verified separately. The collector is shown to match some interface, while the mutator assumes there is a collector with some other interface. The two interfaces are then

(*Program*)	$\mathbb{P}$	$::= (\mathbb{C}, \mathbb{S}, \mathbb{I})$
(*CodeHeap*)	$\mathbb{C}$	$::= \{f \rightsquigarrow \mathbb{I}\}^*$
(*State*)	$\mathbb{S}$	$::= (\mathbb{H}, \mathbb{R})$
(*Heap*)	$\mathbb{H}$	$::= \{l \rightsquigarrow w\}^*$
(*RegFile*)	$\mathbb{R}$	$::= \{r \rightsquigarrow w\}^*$
(*Reg*)	r	$::= \{\mathsf{rk}\}^{k \in \{0 \ldots 31\}}$
(*Nat*)	w, f	$::= 0 \mid 1 \mid 2 \mid \ldots$
(*Addr*)	l	$::= 0 \mid 4 \mid 8 \mid \ldots$
(*InstrSeq*)	$\mathbb{I}$	$::= c;\mathbb{I} \mid \mathsf{beq}\ r_s, r_t, f; \mathbb{I} \mid \mathsf{bne}\ r_s, r_t, f; \mathbb{I}$ $\mid \mathsf{j}\ f \mid \mathsf{jal}\ f, f_{ret} \mid \mathsf{jr}\ r_s$
(*Command*)	c	$::= \mathsf{addu}\ r_d, r_s, r_t \mid \mathsf{addiu}\ r_d, r_s, w$ $\mid \mathsf{subu}\ r_d, r_s, r_t \mid \mathsf{sltu}\ r_d, r_s, r_t \mid \mathsf{andi}\ r_d, r_s, 1$ $\mid \mathsf{lw}\ r_d, w(r_s) \mid \mathsf{sw}\ r_s, w(r_d)$

Figure 7. Machine Syntax

shown to be compatible using a lemma, which allows the two verified components to be combined into a verified program.

In more detail, to verify the mutator, we select a *gcStep* that suits the needs of the mutator, which will simplify reasoning and allow the mutator to be combined with more collectors. We also do not pick a particular heap representation *repr*. Instead, we parameterize our verification of the mutator by a representation, a few standard properties of that representation, and a verified collector implementation (including the read and write barriers) that uses that representation. In Coq, this can be done using a functor. Later, we can instantiate a verified mutator with any verified collector that matches the specification to produce a fully verified implementation.

The mutator can only access the garbage collected heap using the barriers because the actual representation is hidden from the mutator, but the soundness of our approach does not rely on this: if a mutator is verified in terms of a specific *repr*, the soundness of the underlying program logic keeps the mutator from calling the collector unless the heap is well-formed.

Verifying the collector is more straightforward: we select a *gcStep* specific to the style of GC, and verify it as you would any program. The abstract specifications of each collector must be the same as that of the mutator, modulo the choice of *gcStep*.

When it comes time to link the collector and the mutator, we must first coerce the collector to match the specification expected by the mutator. We can show that if a GC implements the interface with some *gcStep*, and that any time we take a *gcStep* we also take a *gcStep'*, then the GC also implements the interface with *gcStep'*. In the Coq implementation, this fact is embodied by a functor that takes a collector with one interface along with a proof of compatibility with another interface and produces a collector with the second interface. Using this, we convert the GC to use the mutator's interface, without having to reverify the entire collector. After the collector has been coerced, it can be combined with the mutator to produce an entire verified program.

We will discuss more details of this approach after we have described the system we use to reason about programs.

3. Preliminaries

Before we can describe our approach in detail, we must present the formal setting, based on existing work, that we are using. This has three parts: the abstract machine, the program logic SCAP, and separation logic, used to reason about heaps.

3.1 Machine

The abstract machine we use is a MIPS-like architecture. The syntax is given in Figure 7. A program $\mathbb{P}$ is a code heap, a state, and an extended basic block. The basic block takes the place of a program counter, which simplifies the operational semantics.

$(\mathbb{C}, \mathbb{S}, \mathbb{I}) \longmapsto \mathbb{P}$ where	
if $\mathbb{I} =$	then $\mathbb{P} =$
$c;\mathbb{I}'$	$(\mathbb{C}, \mathbb{S}', \mathbb{I}')$ if $\text{Next}_c(\mathbb{S}) = \mathbb{S}'$
$\mathsf{beq}\ r_s, r_t, f; \mathbb{I}'$	$\begin{cases} (\mathbb{C}, \mathbb{S}, \mathbb{C}(f)) & \text{if } \mathbb{S}(r_s) = \mathbb{S}(r_t) \\ (\mathbb{C}, \mathbb{S}, \mathbb{I}') & \text{otherwise} \end{cases}$
$\mathsf{bne}\ r_s, r_t, f; \mathbb{I}'$	$\begin{cases} (\mathbb{C}, \mathbb{S}, \mathbb{C}(f)) & \text{if } \mathbb{S}(r_s) \neq \mathbb{S}(r_t) \\ (\mathbb{C}, \mathbb{S}, \mathbb{I}') & \text{otherwise} \end{cases}$
$\mathsf{j}\ f$	$(\mathbb{C}, \mathbb{S}, \mathbb{C}(f))$
$\mathsf{jal}\ f, f_{ret}$	$(\mathbb{C}, \mathbb{S}\{\mathsf{r31} \rightsquigarrow f_{ret}\}, \mathbb{C}(f))$
$\mathsf{jr}\ r_s$	$(\mathbb{C}, \mathbb{S}, \mathbb{C}(\mathbb{S}(r_s)))$

where

if $c =$	then $\text{Next}_c(\mathbb{S}) =$
$\mathsf{addu}\ r_d, r_s, r_t$	$\mathbb{S}\{r_d \rightsquigarrow \mathbb{S}(r_s) + \mathbb{S}(r_t)\}$
$\mathsf{addiu}\ r_d, r_s, w$	$\mathbb{S}\{r_d \rightsquigarrow \mathbb{S}(r_s) + w\}$
$\mathsf{subu}\ r_d, r_s, r_t$	$\mathbb{S}\{r_d \rightsquigarrow \mathbb{S}(r_s) - \mathbb{S}(r_t)\}$
$\mathsf{sltu}\ r_d, r_s, r_t$	$\mathbb{S}\{r_d \rightsquigarrow k\}$ if $\mathbb{S}(r_s) < \mathbb{S}(r_t)$, $k = 1$, else $k = 0$
$\mathsf{andi}\ r_d, r_s, 1$	$\mathbb{S}\{r_d \rightsquigarrow k\}$ if $\mathbb{S}(r_s)$ is odd, $k = 1$, else $k = 0$
$\mathsf{lw}\ r_d, w(r_s)$	$\mathbb{S}\{r_d \rightsquigarrow \mathbb{S}(\mathbb{S}(r_s) + w)\}$
$\mathsf{sw}\ r_s, w(r_d)$	$\mathbb{S}\{(\mathbb{S}(r_d) + w) \rightsquigarrow \mathbb{S}(r_s)\}$ if $(\mathbb{S}(r_d) + w) \in dom(\mathbb{S})$

Figure 8. Machine step

(*Prop*)	$P ::= \ldots$
(*SPred*)	$p \in State \to Prop$
(*Guar*)	$g \in State \to State \to Prop$
(*BlockSpec*)	$\sigma ::= (p, g)$
(*CodeHeapSpec*)	$\Psi ::= \{f : \sigma\}^*$
	$\mathbb{A} ::= \mathbb{S}$

Figure 9. Specification syntax

A code heap $\mathbb{C}$ maps natural numbers to instruction sequences. A state is a heap and a register file. A heap is a partial mapping of addresses to words, while a register file is a partial mapping of registers to words. There are 31 registers. Register $\mathsf{r0}$ always has the value 0. A nat or word is any natural number, while an address is any natural number that is a multiple of 4. An instruction sequence $\mathbb{I}$ is an extended basic block, and is either a command followed by another instruction sequence, a branch to f if r_s is equal to r_t, a branch to f if r_s is *not* equal to r_t, a jump to block f, a call to the function at f (returning to f_{ret}), or an indirect jump to the address stored in register r_s. A command c is either an unsigned addition of a register and a register or a constant, the subtraction of two registers, the less-than comparison of registers r_s and r_t, the bitwise-and of a register with 1, a load, or a store.

The machine has a standard small step dynamic semantics, given in Figure 8. The step relation $\mathbb{P} \longmapsto \mathbb{P}'$ says that $\mathbb{P}$ steps to $\mathbb{P}'$. $\text{Next}_c(\mathbb{S})$ is the state that results from executing a non-control flow command c in state $\mathbb{S}$, if one exists. We write $X(z)$ for the binding of z in any partial map X when z is in the domain of X, and $X\{z \rightsquigarrow v\}$ for the addition of a binding of z to v (where an old binding of z, if any, is removed), for various X and z. We write $(\mathbb{H}, \mathbb{R})(r)$ for $\mathbb{R}(r)$ and $(\mathbb{H}, \mathbb{R})(l)$ for $\mathbb{H}(l)$. We write $(\mathbb{H}, \mathbb{R})\{l \rightsquigarrow w\}$ for $(\mathbb{H}\{l \rightsquigarrow w\}, \mathbb{R})$ and $(\mathbb{H}, \mathbb{R})\{r \rightsquigarrow w\}$ for $(\mathbb{H}, \mathbb{R}\{r \rightsquigarrow w\})$. We write $(\mathbb{H}, \mathbb{R})$ for $\mathbb{H}$ or $\mathbb{R}$, when it is obvious from context. We define $dom(\mathbb{H}, \mathbb{R}) = dom(\mathbb{H}) \cup dom(\mathbb{R})$. $\text{dom}(\mathbb{H})$ is the domain of $\mathbb{H}$, while $\text{dom}(\mathbb{R})$ is the domain of $\mathbb{R}$.

3.2 Program logic

To reason about programs, we use the Hoare-logic-based language SCAP (for Stack-based Certified Assembly Programming) [13]. An assembly program that is valid in SCAP will run forever without,

$\boxed{\vdash \mathbb{P}\ \text{ok}}$ (Well-formed program)

$$\frac{\Psi \vdash \mathbb{C} : \Psi \quad p\ \mathbb{S} \quad \Psi; (p,g) \vdash \mathbb{I}\ \text{ok} \quad \Psi \vdash (g\ \mathbb{S})\ \text{WFST}}{\vdash (\mathbb{C}, \mathbb{S}, \mathbb{I})\ \text{ok}} \quad \text{(PROG)}$$

$\boxed{\Psi' \vdash \mathbb{C} : \Psi}$ (Well-formed code heap)

$$\frac{\forall f \in \text{dom}(\Psi).\ \Psi'; \Psi(f) \vdash \mathbb{C}(f)\ \text{ok}}{\Psi' \vdash \mathbb{C} : \Psi} \quad \text{(CDHP)}$$

$\boxed{\Psi \vdash p\ \text{WFST}}$ (Well-formed call stack)

$$\frac{\forall \mathbb{S}.\ p\ \mathbb{S} \rightarrow \mathit{False}}{\Psi \vdash p\ \text{WFST}} \quad \text{(BASE)}$$

$$\frac{\Psi(f) = (p,g) \quad \forall \mathbb{S}.\ p_0\ \mathbb{S} \rightarrow \mathbb{S}(\mathsf{r32}) = f \wedge p\ \mathbb{S} \quad \Psi \vdash (\lambda \mathbb{S}'.\ \exists \mathbb{S}.\ p_0\ \mathbb{S} \wedge g\ \mathbb{S}\ \mathbb{S}')\ \text{WFST}}{\Psi \vdash p_0\ \text{WFST}} \quad \text{(FRAME)}$$

$\boxed{\Psi; \sigma \vdash \mathbb{I}\ \text{ok}}$ (Well-formed instruction sequence)

$$\frac{\Psi; (p',g') \vdash \mathbb{I}\ \text{ok} \quad \forall \mathbb{S}.\ p\ \mathbb{S} \rightarrow \exists \mathbb{S}'.\text{Next}_c \mathbb{S} = \mathbb{S}' \wedge p'\ \mathbb{S}' \wedge \forall \mathbb{S}''.\ g'\ \mathbb{S}'\ \mathbb{S}'' \rightarrow g\ \mathbb{S}\ \mathbb{S}''}{\Psi; (p,g) \vdash c; \mathbb{I}\ \text{ok}} \quad \text{(SEQOK)}$$

$$\frac{\begin{array}{l}\Psi(f) = (p',g') \quad \Psi; (p'',g'') \vdash \mathbb{I}\ \text{ok} \\ (iop, op) \in \{(\mathsf{beq}, =), (\mathsf{bne}, \neq)\} \\ \forall \mathbb{S}.\ p\ \mathbb{S} \rightarrow \\ \quad (\mathbb{S}(r_s)\ op\ \mathbb{S}(r_t) \rightarrow p'\ \mathbb{S} \wedge \forall \mathbb{S}'.\ g'\ \mathbb{S}\ \mathbb{S}' \rightarrow g\ \mathbb{S}\ \mathbb{S}') \wedge \\ \quad (\neg \mathbb{S}(r_s)\ op\ \mathbb{S}(r_t) \rightarrow p''\ \mathbb{S} \wedge \forall \mathbb{S}'.\ g''\ \mathbb{S}\ \mathbb{S}' \rightarrow g\ \mathbb{S}\ \mathbb{S}')\end{array}}{\Psi; (p,g) \vdash iop\ r_s, r_t, f; \mathbb{I}\ \text{ok}} \quad \text{(BROK)}$$

$$\frac{\Psi(f) = (p',g') \quad \forall \mathbb{S}.\ p\ \mathbb{S} \rightarrow p'\ \mathbb{S} \wedge \forall \mathbb{S}'.\ g'\ \mathbb{S}\ \mathbb{S}' \rightarrow g\ \mathbb{S}\ \mathbb{S}'}{\Psi; (p,g) \vdash \mathsf{j}\ f\ \text{ok}} \quad \text{(J)}$$

$$\frac{\begin{array}{l}\Psi(f) = (p',g') \quad \Psi(f_{ret}) = (p'',g'') \\ \forall \mathbb{S}.\ p\ \mathbb{S} \rightarrow\ p'\ \mathbb{S}\{\mathsf{r31} \rightsquigarrow f_{ret}\} \wedge \forall \mathbb{S}'.\ g'\ \mathbb{S}\{\mathsf{r31} \rightsquigarrow f_{ret}\}\ \ \mathbb{S}' \rightarrow \\ \quad p''\ \mathbb{S}' \wedge \forall \mathbb{S}''.\ g''\ \mathbb{S}'\ \mathbb{S}'' \rightarrow g\ \mathbb{S}\ \mathbb{S}'' \\ \forall \mathbb{S}, \mathbb{S}'.\ g\ \mathbb{S}\ \mathbb{S}' \rightarrow \mathbb{S}(\mathsf{r31}) = \mathbb{S}'(\mathsf{r31})\end{array}}{\Psi; (p,g) \vdash \mathsf{jal}\ f, f_{ret}\ \text{ok}} \quad \text{(CALL)}$$

$$\frac{\forall \mathbb{S}.\ p\ \mathbb{S} \rightarrow g\ \mathbb{S}\ \mathbb{S}}{\Psi; (p,g) \vdash \mathsf{jr}\ \mathsf{ra}\ \text{ok}} \quad \text{(RETURN)}$$

Figure 10. Typing rules for SCAP

for instance, trying to load from an invalid memory address. We use SCAP to verify our mutators and collectors. With the power of SCAP and other CAP-style systems [45, 12] (which can be integrated with SCAP), our framework can be directly applied to support a wide range of mutator languages.

The specification syntax is given in Figure 9. Propositions P are propositions in the Calculus of Inductive Constructions (CIC) [39], a higher order predicate logic extended with inductive definitions. A state predicate p describes a single state, while a guarantee g relates a pair of states. State predicates are used to specify the precondition of an instruction block, which ensures that executing the instruction block is safe. Guarantees describe the behavior of an instruction block by relating the current state to the state in which the current procedure returns. An instruction block specification σ is a precondition and a guarantee. A code heap specification Ψ gives a specification for each instruction block in the code heap.

The inference rules for SCAP are given in Figure 10. There are four judgments for static checking in SCAP. $\vdash \mathbb{P}$ ok holds if $\mathbb{P}$ is well-formed. For this to hold, the code heap must be closed, a precondition p must hold on the current state, the current instruction sequence must be safe to execute assuming p and have a guarantee g. Finally, the stack must be well formed assuming g holds on the current state. $\Psi' \vdash \mathbb{C} : \Psi$ holds if $\mathbb{C}$ implements the specifications in Ψ, assuming code with specs Ψ is available. The rule for this judgment simply checks that each spec in Ψ is properly implemented by the corresponding code block.

$\Psi \vdash p$ WFST holds if any state that satisfies p implies a well-formed call stack in which is okay to immediately return. If this precondition cannot be satisfied by any state, then this is a top-level function that cannot be returned from. Otherwise, the precondition must imply that the return register contains some return code pointer f of a function that has a specification (p,g). The current state must satisfy the p, to allow the function to return, and when f has finished executing, it is okay to return once more.

Finally, $\Psi; \sigma \vdash \mathbb{I}$ ok holds if, assuming code with specs Ψ is available, sequence $\mathbb{I}$ matches specification σ. These rules each ensure that it is safe to take a step and that the guarantee correctly describes the behavior of the function until a return is executed. In the RET rule, p must imply that g holds with the current state as both arguments. A typical g for this rule specifies that the two states are equal. For the call rule, we must ensure that it is safe to call the function f, and that it will be safe to call f_{ret} after we return from f. In addition, we require that the behavior of the call instruction combines the behavior of the functions f and f_{ret}.

The soundness theorem for SCAP states that a well-formed program can always take a step to another well-formed program.

3.3 Separation logic

We describe heaps using separation logic [40]. We define separation logic predicates, written A and B, as state predicates, based on Reynolds's semantics [40]. $A\ \mathbb{S}$ holds if state $\mathbb{S}$ satisfies the state predicate A. We define this judgment in terms of standard CIC predicates, allowing us to use separation logic predicates in our specifications. The syntax of the fragment of separation logic we use in this paper is as follows:

$A, B ::= n \mapsto m \mid \texttt{true} \mid A * B \mid A \wedge B \mid \exists x.\ A \mid \ldots$

In the verification itself we use other separation logic connectives, such as iterated separating conjunction [4]. $n \mapsto m$ holds on a state if and only if the state's heap is $\{n \rightsquigarrow m\}$. `true` holds on all states. $A * B$ holds if the state's heap can be split into two disjoint parts such that A holds on one and B on the other. $A \wedge B$ holds if both A and B hold on the entire state. $\exists x.\ A$ holds if there exists an x such that $A\ x$ holds on the state.

4. The garbage collector-mutator interface

The garbage collector may require that various exotic invariants hold on the object heap, but the mutator does not have to worry about these details as long as it interacts with the object heap only through the appropriate barriers. We hide these details away by having the mutator reason about an *abstract state*, which presents a high-level view of the *concrete state* which represents the actual machine state. For instance, in a mark-sweep garbage collector space is set aside for the collector to track which objects are marked, but the mutator never directly interacts with this mark data, so it shouldn't have to keep track of it. Or, as we described earlier, in an incremental copying collector the object heap might be in a partially copied state, with old and new copies of objects coexisting. We can hide this mess from the mutator, and as long as the mutator only uses the operations we provide, it will never discover the deception.

We write $\mathbb{A}$ for a state that is intended to be an abstract state. It is represented in the same way as a concrete state $\mathbb{S}$, but for a particular state will not be the same. In this section, we give specifications for the three garbage collector operations we discussed previously, then discuss the specification of the behavior of the garbage col-

$$\mathit{readPre}_k\ \mathbb{A} ::= \{\mathsf{rroot}, \mathbb{A}(\mathsf{rroot}) + 4k\} \subseteq \mathsf{dom}(\mathbb{A})$$
$$\mathit{readStep}_k\ \mathbb{A}\ \mathbb{A}' ::= \mathbb{A}' = \mathbb{A}\{\mathsf{v0} \rightsquigarrow \mathbb{A}(\mathbb{A}(\mathsf{rroot}) + 4k)\}$$

$$\mathit{writePre}_k\ \mathbb{A} ::= \{\mathsf{rroot}, \mathsf{a0}, \mathbb{A}(\mathsf{rroot}) + 4k\} \subseteq \mathsf{dom}(\mathbb{A})$$
$$\mathit{writeStep}_k\ \mathbb{A}\ \mathbb{A}' ::= \mathbb{A}' = \mathbb{A}\{\mathbb{A}(\mathsf{rroot}) + 4k \rightsquigarrow \mathbb{A}(\mathsf{a0})\}$$

$$\mathit{allocPre}\ \mathbb{A} ::= \mathsf{rroot} \in \mathsf{dom}(\mathbb{A})$$
$$\mathit{allocStep}\ \mathbb{A}\ \mathbb{A}'' ::=$$
$$\quad \exists \mathbb{A}'.\ \mathit{gcStep}\ \mathbb{A}\ \mathbb{A}' \wedge$$
$$\quad\quad (\forall p.\ p\ \mathbb{A}' \rightarrow (p * \mathbb{A}''(\mathsf{v0}) \mapsto \mathit{NULL}, \mathit{NULL})\ \mathbb{A}'') \wedge$$
$$\quad\quad \mathbb{A}'(\mathsf{rroot}) = \mathbb{A}''(\mathsf{rroot})$$

Figure 11. Abstract specifications for basic operations

lector that the mutator can observe. Finally, we discuss the general mechanism for relating the abstract and concrete states.

4.1 The operations

The three garbage collector operations are reading from an object, writing to an object, and allocating a new object. Here we define their basic specifications, in Figure 11, ignoring the effects of any garbage collection that may occur. To simplify the presentation (and our implementation), we only allow registers to be roots. The set of roots is the set of registers in the domain of $\mathbb{A}$. For example, for a read operation, initially the register rroot must be a root. Afterwards, all of the registers that were roots are still roots, plus register v0.

The specifications for the read and write are parameterized by the field k being operated on. For the read to be executed, rroot must be in the domain of the abstract state, and must contain a valid pointer to the field of interest. After the read is performed, the return register v0 contains the value of the root's kth field, but the state is otherwise unchanged. For a write to be executed, rroot must again be in the domain of the heap, and contain a valid pointer to the kth field. In addition, register a0 must be in the domain of the abstract state, indicating it contains a valid abstract value. After the write is done, the kth field of the root is updated, but the state otherwise remains the same. To safely perform an allocation, all roots must be in the domain of the abstract state to ensure they are valid abstract values, which is needed to ensure a collection will work. Afterwards, the collector has taken a step, as defined by *gcStep*, and a fresh object has been allocated, initialized, and placed in register v0. We can, if we want, similarly add a *gcStep* to the read or write specifications. This can be useful if the read or write barrier performs some collection work that is not hidden from the mutator, or if, as in a collector that uses reference counting, objects are actually collected during reading or writing.

4.2 Abstract collection

During any of these operations, in particular allocation, the GC might do some collection work, which will affect the abstract state. We specify this behavior with a binary state predicate *gcStep* that relates the abstract state before the GC executes with the abstract state after it executes. The definition of *gcStep* is fixed for a particular mutator, as it describes the behavior of the GC that the mutator can see.

In Figure 12 we define two basic GC step relations. The first, *basicGcStep*, says that the roots are the same after the GC runs, and that all objects in the heap that are reachable from the roots in the initial state are exactly the same in the final state. The definition requires the definition of two other heap predicates. The first, $\mathit{minObjHp}(x)$ holds on any heap that contains exactly the objects reachable from x. The second, $\mathit{heq}\ \mathbb{H}$, simply holds on any state where the heap is $\mathbb{H}$. Essentially what we are doing is scooping out a heap $\mathbb{H}$ from the initial state that contains the objects

$$\mathit{basicGcStep}\ \mathbb{A}\ \mathbb{A}' ::=$$
$$\quad (\forall \mathbb{H}.\ ((\mathit{minObjHp}(\mathbb{A}(\mathsf{rroot})) \wedge \mathit{heq}\ \mathbb{H}) * \mathtt{true})\ \mathbb{A} \rightarrow$$
$$\quad\quad (\mathit{heq}\ \mathbb{H} * \mathtt{true})\ \mathbb{A}') \wedge$$
$$\quad \mathbb{A}(\mathsf{rroot}) = \mathbb{A}'(\mathsf{rroot})$$

$$\mathit{typesafeGcStep}\ \mathbb{A}\ \mathbb{A}' ::=$$
$$\quad \forall \Gamma.\ \mathit{stateHasType}(\Gamma)\ \mathbb{A} \rightarrow \mathit{stateHasType}(\Gamma)\ \mathbb{A}'$$

Figure 12. Basic GC step relations

$$\lfloor (p, g) \rfloor ::= (\lfloor p \rfloor, \lfloor g \rfloor)$$

$$\lfloor p \rfloor ::= \lambda \mathbb{S}.\ \exists \mathbb{A}.\ \mathit{repr}\ \mathbb{A}\ \mathbb{S} \wedge p\ \mathbb{A}$$

$$\lfloor g \rfloor ::= \lambda \mathbb{S}, \mathbb{S}'.\ \forall \mathbb{A}.\ \mathit{repr}\ \mathbb{A}\ \mathbb{S} \rightarrow \exists \mathbb{A}'.\ \mathit{repr}\ \mathbb{A}'\ \mathbb{S}' \wedge g\ \mathbb{A}\ \mathbb{A}'$$

Figure 13. Specification lifting functions

reachable from the root, and saying that the final state also contains this heap $\mathbb{H}$. Therefore, if you can show that an object is reachable in the initial state $\mathbb{A}$ you can show that it is part of $\mathbb{S}$, and thus has the same value in the final state $\mathbb{A}'$. Secondly, this step relation guarantees that the roots remain the same.

The second GC step relation we define, *typesafeGcStep*, is an example of how a step relation for type safety could be defined. We assume that Γ is a map from registers to types and predicate $\mathit{stateHasType}(\Gamma)$ holds on a state if the registers can be given the types specified in Γ. The step relation simply says that the final state has the same type as the initial state.

We can also define more application-specific GC steps. Basic type safety is too weak to verify the partial correctness of our list summation example, while *basicGcStep* is stronger than is necessary. Instead, we can use a GC step that specifies that the GC is *list sum preserving*, so that the the initial and final states contain linked lists with the same sum and heads.

4.3 From abstract to concrete

While we want the mutator to reason abstractly as much as possible, at some point we must connect the abstract and concrete behaviors. This is done using a binary state predicate $\mathit{repr}\ \mathbb{A}\ \mathbb{S}$ which holds if the state $\mathbb{S}$ is a concrete representation of the abstract state $\mathbb{A}$. Each garbage collector will have its own *repr*.

The verification of a mutator is parameterized over a verified implementation of a GC, which includes the definition of *repr*, keeping it abstract. This allows us to instantiate the verified mutator with different collectors as needed. While it is mostly abstract, we need to give some additional rules to reason about the representation. The most important rules are the following: if an abstract or concrete root is *atomic* (*i.e.,* not an object pointer), the abstract and concrete values of the root are equal; a value can be copied from one root to another; a root can always be assigned an atomic value. These rules allows the mutator to reason about and manipulate the roots in a few specific circumstances.

Given a predicate *repr*, we can define a function $\lfloor\ \rfloor$ in Figure 13 that lifts an abstract specification to a concrete specification. We use the lifted forms of the specifications to verify the implementations of the three GC operations. A specification is lifted by lifting the precondition and the guarantee. A precondition p is lifted by requiring that there exists an abstract state $\mathbb{A}$ that is represented in the current state $\mathbb{S}$, such that p holds on $\mathbb{A}$. A guarantee is lifted by requiring that for all abstract states $\mathbb{A}$ represented in the initial state $\mathbb{S}$, there exists some other abstract state $\mathbb{A}'$ represented in the final concrete state $\mathbb{S}'$, such that $\mathbb{A}$ and $\mathbb{A}'$ are related by g.

```
menter:                     mloop2:
addiu rsum,r0,1             addiu rtemp,rzero,1
j mloop                     subu v0,v0,rtemp
                            addu rsum,rsum,v0
mloop:                      jal read2, mloop3
addiu rtemp,r0,NULL
beq rroot, rtemp, mret      mloop3:
jal read1, mloop2           addiu rroot,v0,0
                            j mloop
mret:
jr ra
```

Figure 14. List sum implementation

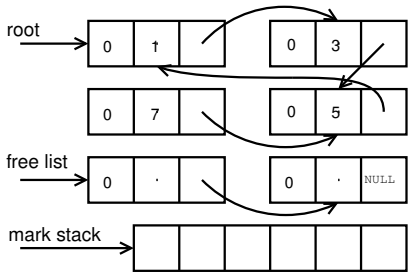

Figure 15. Mark-sweep heap

$$idRepr\ (\mathbb{H},\mathbb{R})\ (\mathbb{H}',\mathbb{R}') ::= (\mathbb{H}=\mathbb{H}') \wedge (\mathbb{R}\subseteq\mathbb{R}')$$

$$\begin{array}{l} msRepr\ \mathbb{A}\ \mathbb{S} ::= \\ \quad (\exists objs, free, ohpSt, ohpEnd, freePtr, stBot, stTop. \\ \qquad (objHp(objs, objs) \wedge idRepr\ \mathbb{A}) * \\ \qquad objHdrs\ (objs \cup free, 0) * \\ \qquad freeList\ (freePtr, free) * buffer(stTop, stBot) * \\ \qquad msAuxOk(\mathbb{A}, objs, free, ohpSt, ohpEnd, stBot, stTop, freePtr))\ \mathbb{S} \end{array}$$

Figure 16. Mark-sweep predicate

5. Mutator specification and verification

Now that we have defined our formal system, we can return to our list summation example. Most mutator programs will be the result of compilation of a high-level language using a compiler that generates typed assembly language (TAL) [33]. We can then embed TAL into a SCAP-style system by representing the state typing judgment as a state predicate and showing that the typing rules for instructions are admissible. For more details on this embedding, please see the companion paper [27]. To avoid introducing another formal system, for our example we will instead verify the partial correctness of the list sum example using SCAP. We show the assembly implementation, then discuss the specification and, briefly, the verification. The assembly implementation (of the pseudo code from Figure 2) is given in Figure 14. Our goal is to verify that the mutator correctly sums a list.

For our example, all atomic values are odd, allowing us to easily distinguish them from object pointers. With this encoding, an integer n would be stored in the state as $2n+1$. The translation is straightforward, aside from the need to add in extra arithmetic to properly handle the integer encoding: 0 is encoded as 1, and we must subtract 1 from a value we load before using it to increment the sum. The calls to `first` and `second` in the pseudo code have been changed to calls to `read1` and `read2`. We use register rtemp to hold intermediate values. As before, register r0 is always 0.

For the specification, we must define a list predicate which holds on a state that contains a linked list starting at address x that has the sum n. This can be inductively defined in a straightforward manner and is defined as $\text{LListSumAt}(n,x)$.

Using this predicate, we can define the abstract specifications of the mutator entry point `menter` and the loop header `mloop`. These blocks have the same precondition, which requires that the register rroot contains a linked list, with some sum n. More formally, this is defined as

$$mpre\ \mathbb{A} ::= \exists n.\ \mathsf{rroot} \in \mathsf{dom}(\mathbb{A}) \wedge\ \text{LListSumAt}(n, \mathbb{A}(\mathsf{rroot}))\ \mathbb{A}$$

The loop header guarantees that if rsum initially contains the encoding of some n and rroot contains a linked list with the sum m, then in the state in which we return rsum contains the encoding of $n+m$. The guarantee for the entire function is very similar. For simplicity, we do not specify that the heap is unchanged in the final state. Formally, the abstract loop guarantee is:

$$\begin{array}{rl} loopGuar\ \mathbb{A}\ \mathbb{A}' ::= & \forall m, n.\ \mathbb{A}(\mathsf{rsum}) = 2n+1 \rightarrow \\ & \text{LListSumAt}(m, \mathbb{A}(\mathsf{rroot}))\ \mathbb{A} \rightarrow \\ & \mathbb{A}'(\mathsf{rsum}) = 2(n+m)+1 \end{array}$$

In order to actually verify this mutator, we need to specify what behavior we expect from the collector by defining *gcStep*. We cannot simply require that the GC preserves the sum of the list in the root, because this allows the head of the list to change whenever we perform a read, if *gcStep* is in the specification of the read. As we read the head of the list before the tail, the old value we loaded from the head might become invalid, preventing us from verifying the mutator. Instead, we require that the GC does not change rroot if it is null, or if it is not null, it will not change the head, but can change the tail to another list with the same length. Finally, the value of rsum cannot be changed.

With the specification of the mutator and the main loop determined, along with the behavior of the read barriers, the basic outline of the verification is complete. Verification is more difficult than it would be in the absence of garbage collection, but the abstraction limits the complexity.

6. Collector specification and verification

In this section, we discuss the collector's view of the state by discussing how various collectors represent the abstract state. Additional details about the collectors we verified can be found in the extended version of this paper [28].

6.1 Stop-the-world mark-sweep

A mark-sweep garbage collector, as you might infer from the name, has two phases: mark and sweep. In the mark phase, all reachable objects are marked by a depth-first traversal of the heap, using a *mark stack* to hold objects that remain to be examined. In our example collector, the mark is held in an object header. In the sweep phase, the object heap is traversed. Any object that remains unmarked must not be reachable from the root, and thus can be added safely to the free list, which is also done during this phase, along with the resetting of the marks. Figure 15 is diagram of a possible concrete mark-sweep heap for the abstract heap given in Figure 3.

We define the representation predicate for a mark-sweep collector in Figure 16. We need a few auxiliary heap predicates to describe the heap. $idRepr\ \mathbb{A}\ \mathbb{S}$ holds if $\mathbb{S}$ is a simple representation of $\mathbb{A}$, where $\mathbb{A}$ and $\mathbb{S}$ share the heap, and the register file of $\mathbb{A}$ is a subset of the register file of $\mathbb{S}$. $objHp(objs', objs)$ holds on a heap if it exactly contains the objects in *objs*, and all of the object pointers in the heap are elements of $objs'$. Thus if $objHp(objs, objs)$ holds on a heap, that heap is closed. $objHdrs\ (objs, n)$ holds on a heap that consists entirely of object headers for the objects in *objs*, where the headers all have the value n. We use 0 to indicate unmarked objects.

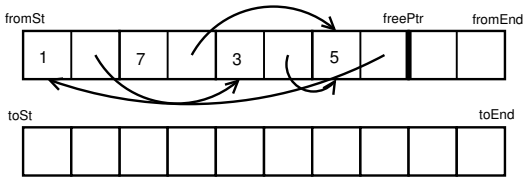

Figure 17. Cheney heap

$$
\begin{array}{l}
\mathit{isoState}\ (\mathbb{H},\mathbb{R})\ (\mathbb{H}',\mathbb{R}') ::= \\
\quad \exists \mathit{objs}, \mathit{objs}', \phi. \\
\qquad \mathit{objHp}(\mathit{objs},\mathit{objs})\ \mathbb{H} \wedge \mathit{objHp}(\mathit{objs}',\mathit{objs}')\ \mathbb{H}' \wedge \\
\qquad \mathit{objs}' \cong_\phi \mathit{objs} \wedge \\
\qquad \forall x \in \mathit{objs}'.\ \mathit{objIsFwded}_\phi\ \mathbb{H}'\ \mathbb{H}\ x \wedge \\
\qquad \forall r \in \mathrm{dom}(\mathbb{R}).\ \phi^*(\mathbb{R}'(r)) = \mathbb{R}(r)
\end{array}
$$

$$
\begin{array}{l}
\mathit{chRepr}\ \mathbb{A} ::= \\
\quad \exists \mathit{objs}, \mathit{fromSt}, \mathit{fromEnd}, \mathit{toSt}, \mathit{toEnd}, \mathit{freePtr}. \\
\qquad (\mathit{objHp}(\mathit{objs},\mathit{objs}) \wedge \mathit{isoState}\ \mathbb{A}) * \\
\qquad \mathit{buffer}(\mathit{freePtr},\mathit{fromEnd}) * \mathit{buffer}(\mathit{toSt},\mathit{toEnd}) * \\
\qquad \mathit{chAuxOk}(\mathit{objs},\mathit{fromSt},\mathit{fromEnd},\mathit{toSt},\mathit{toEnd},\mathit{freePtr})
\end{array}
$$

Figure 18. Cheney collector state predicate

freeList (n,S) holds on a heap that is a linked list of pairs starting at n, and contains the objects in the set S in some order. A heap satisfies $\mathit{buffer}(x,y)$ if it contains a buffer from address x to y, not including y. For the mark-sweep collector this is used to hold the mark stack. Finally, we have a predicate *msAuxOk* which ensures various constraints hold on the auxiliary variables, such as ensuring that the set of objects matches up with the heap pointers, and checking that the registers are well-typed.

To put these all together, we first require that one part of the heap both contains a closed set of objects *objs* (which *msAuxOk* will ensure includes all of the roots) and also contains the abstract heap. The identity representation is sufficient because the mutator and collector views of this part of the heap coincide. Other than that, we require the state contain headers for the objects, a free list containing objects *free*, space for a mark stack, and whatever is needed to ensure the auxiliary variables are valid.

6.2 Stop-the-world copying

In a copying collector, the heap is divided into two *semi-spaces*. One, the from-space, contains all of the allocated objects, all in a row, and all of the free objects, also all in a row. The other semi-space, the to-space, lays fallow. During collection, the reachable objects in the from-space are copied to the to-space, then their fields are forwarded and finally the roles of the semi-spaces are swapped. All of the unreachable objects will be automatically reused next cycle. In a Cheney collector [26], no extra space is needed for forwarding pointers, because they can be stored within the abandoned hulks of the copied objects.

The representation of objects within the concrete state depends on the abstract behavior of the GC, as defined by *gcStep*. If *gcStep* can be satisfied even if the objects are moved around, then we can use a simple object representation like *idRepr*. This will be the case if *gcStep* only guarantees type safety, because the type of an object does not depend on its location, or if we simply state in *gcStep* that the objects might be copied. On the other hand, if we want to use *basicGcStep*, we must hide any copying from the mutator. This can be done by maintaining an isomorphism ϕ from

```
fwdObj(word* obj) {
  if (free == alloc)
    while(1) {};  // no space left: give up
  x = obj[0];  // copy 1st field
  free[0] = x;
  x = obj[1];  // copy 2nd field
  free[1] = x;
  obj[0] = free;  // store forwarding pointer in obj
  x = free;
  free += 2;  // increment free by two words
  return x;  // return location of new object
}

fromSpacePtr (x) {
  return !(x & 1) && x >= fromStart && x < fromEnd;
}

toSpacePtr (x) {
  return !(x & 1) && x >= toStart && x < toEnd;
}
```

Figure 19. Baker pseudo code (auxiliary functions)

the concrete object addresses to the abstract addresses: whenever we move objects in the concrete state we can update ϕ instead of updating the abstract state. The interface for representations does not tell the mutator anything about the relationship between concrete and abstract pointers, so the collector can change this relationship (which in this case is precisely given by ϕ) at any time without the mutator being able to tell. The end result is that we can give an implementation of a copying collector the same interface as a mark-sweep collector.

We define a representation in Figure 18 that uses this approach to hide the action of the copying collector. The from-space ranges from *fromSt* to *fromEnd* and the to-space ranges from *toSt* to *toEnd*. The free pointer *freePtr* divides the from-space into allocated and unallocated parts. *objs* is the set of objects in the allocated portion. Instead of using *idRepr*, we define a new binary state predicate $\mathit{isoState}\ \mathbb{A}\ \mathbb{S}$, which holds if there exists an isomorphism ϕ between the objects in $\mathbb{A}$ and those in $\mathbb{S}$. We write ϕ^* for the extension of ϕ by identity on non-pointers. We require that every object in the heap of $\mathbb{S}$ is forwarded to $\mathbb{A}$ using ϕ, which can be defined following [4]. Roughly, if an object is located at x in $\mathbb{S}$, then it must be located at $\phi(x)$ in $\mathbb{A}$, with the fields mapped by ϕ^*. In addition, all of the roots must be forwarded with ϕ^*.

chAuxOk describes the various constraints on the auxiliary variables, such as that the allocated and free objects must cover the from-space, and, importantly, that the from- and to-spaces have the same size. This will let the collector run safely without a bounds check.

For the actual representation, the state must contain a closed well-formed object heap containing all of the objects in *objs*. This ensures that we can safely trace the heap. This part of the heap must be isomorphic to the abstract state $\mathbb{A}$. Next we have buffers containing the free space and the to-space. We use buffers because we don't care about their contents. Finally, we must ensure that all the auxiliary variables are properly accounted for.

6.3 Incremental copying

Baker's algorithm [2] is a variant of Cheney's algorithm that supports incremental collection. The basic layout of the Baker collector heap has already been shown in Figure 4, but we discuss here it in more detail. We give an excerpt from the pseudo code for this collector in Figures 19 and 20. See the extended version [28] for the full pseudocode and assembly implementations of this collector.

Figure 19 contains the pseudocode implementation of the auxiliary functions of the Baker collector. `fwdObj` copies the object

```
bakerScanField (word* fieldPtr) {
  fval = *fieldPtr;
  if (fromSpacePtr(fval)) {
    field1 = fval[0];
    if (toSpacePtr(field1))
      *fieldPtr = field1;
    else
      *fieldPtr = fwdObj(fval);
  }
}

bakerGC(word* root) {
  count = 0;
  if (free == alloc) {
    ...  // flip spaces, scan root
  }
  while (scan != free && count < scan_per_gc) {
    bakerScanField(scan);
    bakerScanField(scan + 1);
    scan = scan + 2;
    count = count + 1;
  }
}
```

Figure 20. Baker pseudo code (collector)

$$
\begin{array}{l}
\mathit{bakerRepr}\ \mathbb{A} ::= \\
\quad \exists B, B', G, W, M, \phi, \mathbb{S}. \\
\quad \exists \mathit{ohpSt}, \mathit{ohpEnd}, \mathit{toSt}, \mathit{toEnd}, \mathit{freePtr}, \mathit{allocPtr}, \mathit{scanPtr}. \\
\qquad ((\phi^* \cup \mathrm{id}_{to \cup W}) \circ \mathbb{S}) = \mathbb{A} \wedge \\
\qquad ((\mathit{objHp}(to, B \cup B') * \mathit{objHp}(to \cup fr, G) * \mathit{objHp}(fr, W)) \wedge \\
\qquad\quad \mathit{idRepr}\ \mathbb{S}) * \\
\qquad \mathit{mapHp}(M, \phi) * \mathit{buffer}(\mathit{freePtr}, \mathit{allocPtr}) * \\
\qquad \mathit{bAuxOk}(B, B', G, W, M, to, fr, \phi, \mathit{ohpSt}, \mathit{ohpEnd}, \\
\qquad\quad \mathit{toSt}, \mathit{toEnd}, \mathit{scanPtr}, \mathit{allocPtr}, \mathit{freePtr}) \\
\quad \text{where } to = B \cup B' \cup G \text{ and } fr = W \cup M
\end{array}
$$

Figure 21. Baker collector state predicate

located at `obj` to the start of the free space, which begins at `free`, stores the forwarding pointer in `obj`, and returns the location of the newly created object. `fromSpacePtr` returns true if the argument `x` is not atomic, and is between the bounds of the from-space, given by the variables `fromStart` and `fromEnd`. `toSpacePtr` is the analogous function for the to-space.

Figure 20 is the pseudocode implementation of the main functions of the Baker collector. `bakerScanField` is the most complex part of the collector. It forwards the value of the field of an object pointed to by the argument `fieldPtr`, if necessary. The value of this field, `fval`, is either atomic, a to-space space pointer, or a from-space pointer. In the first two cases, nothing needs to be done because the field value will still be valid after the collection is done. If the field contains a from-space pointer, then the collector must determine if the object has been forwarded already. It does this by loading the first field of the object into `field1`. `field1` will either be a to-space pointer, an atomic value, or a from-space pointer. In the first case, `field1` must be the forwarding pointer of `fval`, so the collector update the field of the original object to `fval`. Otherwise the object must be forwarded using `fwdObj`.

The function `bakerGC` is the main part of the collector. This function is invoked on every allocation and takes a single root value as an argument. If `free` is equal to `alloc`, then there is no more free space, so the collector must flip the semi-spaces and scan the root. After this check, the collector begins scanning gray objects, which lie between `scan` and `free`. This is an incremental collector, so the GC will not scan more than `scan_per_gc` objects per invocation. An object is scanned by scanning each object, then updating the `scan` pointer and the `count`.

While the implementation of the Baker collector is similar to that of the Cheney collector, the invariants are more complex, because invoking the Baker collector only partially processes the heap.

The most interesting part of the Baker state representation is how we reason about the partially-copied heap. For simplicity, we assume a *gcStep* which permits moving objects. (To avoid this, and allow the use of *basicGcStep*, we can use the technique described in Section 6.2 for hiding copying by introducing an isomorphism between concrete and abstract addresses, but having two isomorphisms would obscure the issues specifically related to incremental copying collection.) As we discussed before, reasoning about the heap of an incremental copying collector is difficult because object references may be out of date, pointing to the old copies of objects. We avoid this by making the abstract heap a forwarded version of the concrete object heap. In some sense, the abstract heap is what the object heap will look like after the collector has finished[1].

The specification for the Baker heap is given in Figure 21. First, we describe the different auxiliary variables. Black objects (B) have been copied to the to-space and their fields forwarded. The GC will not visit them again. New black objects (B') have been allocated during the current cycle, starting from the end of the to-space, and will not be visited by the GC. Gray objects (G) have been copied, but not forwarded. The GC will eventually forward these objects. Gray objects can contain to-space objects because the mutator may write to them before the collector scans them. White objects (W) have not been examined by the GC. Map objects (M) store the mapping ϕ from the old location of objects in M to their new location in the to-space: if an object at w has been forwarded to w', then $\phi(w) = w'$ and the heap contains w' at address w. This allows the collector to forward fields while ensuring that no object is copied more than once. *ohpSt* and *ohpEnd* are the bounds of from-space, while *toSt* and *toEnd* are the bounds of to-space. *freePtr* and *allocPtr* point to the start and end of the free space, respectively, which is located within the to-space. *scanPtr* marks the separation between black and gray objects in the to-space. The from-space objects (M and W) together cover the entire from-space. The collector relies on the fact that objects in the from-space can be dynamically distinguished from objects in the to-space.

As for the actual constraints, the first requirement relates the abstract state $\mathbb{A}$ to the part of the concrete state that contains the objects, given by $\mathbb{S}$. We transform the concrete state to the abstract state by mapping the range of $\mathbb{S}$, including its registers. To forward the fields, we must map everything in M using ϕ and all non-pointers, to-space pointers, and pointers in W using the identity function, because these field values do not need to be changed. We write id_S for the identity on set S, and as before, we write ϕ^* for the function ϕ extended by the identity function on non-pointers. We scoop $\mathbb{S}$ out of the entire concrete state in the same manner as in the mark-sweep collector representation predicate, except that we have three groups of objects instead of one. Black objects can only point to to-space objects, white objects can contain only point to from-space objects, and gray objects can point to either.

$\mathit{mapHp}(M, \phi)$ holds on a state that contains the mapping ϕ, along with the rest of the objects in M. The buffer contains the free space still left in the to-space. Finally, the predicate *bAuxOk* en-

[1] This is not strictly true, because the collector may copy additional objects later. It is impossible to predict how this copying will occur because it depends on mutator behavior, but this doesn't matter because as we said above our *gcStep* allows the collector to move objects freely.

```
read1 (word* root) {
  word* fieldVal = root[0];
  if (fromSpacePointer(fieldVal))
    fieldVal = fwdObj(fieldVal);
  return fieldVal;
}
```

Figure 22. Baker read barrier

forces the rest of the constraints on the auxiliary variables, including the requirement that ϕ is an isomorphism from M to $(B \cup G)$. This constraint is the reason we must keep B and B' separate. It also enforces the requirement that the roots must be in to-space. Note that while we must describe the entire heap at the boundaries of the GC, within the GC we use more "local reasoning" [4] that does not explicitly describe the entire state. We omit further details of *bAuxOk* in the interest of space.

6.3.1 Read barrier

Because the root set can only contain to-space objects, any write will preserve the color invariant, so the write barrier can be implemented using a single store instruction. However, we do need to use a read barrier because if a root object is gray loading a field may break the color invariant for the root set, because gray objects can contain pointers to from-space objects. The read barrier, seen in pseudo code form in Figure 22, forwards the value loaded (copying if needed) if it is a pointer to the from-space. This maintains the invariant, and thus the read barrier satisfies the specification discussed in Section 4, though we must use the variation discussed in that section that includes a *gcStep*, because an object may be copied. If we use the more complex representation that is able to hide copying, then we can use the simpler specification (because this read barrier does not perform any collection) and thus make the read barrier look like just a load operation.

6.4 Other collectors

There are a variety of other collectors that can be verified using our methods. There is the Brooks collector [6], which is a variation of the Baker collector that uses a write barrier instead of a read barrier. We believe that an alteration of the *repr* predicate for the Baker collector would serve that collector, as the relationship between the objects in the concrete state and the objects in the abstract state is the same: only the auxiliary data structures used by the collector change. There are also incremental and concurrent mark-sweep collectors, which must maintain complex color invariants [11, 46, 18, 26]. We have begun investigating this type of collector, and at least the incremental collectors seem to be a fairly natural extension of the stop-the-world mark-sweep collector, in the same way that the Baker collector is a fairly natural extension of the Cheney collector. Mark-sweep collectors do not move objects around, so the relationship between the concrete objects and the abstract objects is the same.

6.5 Conservative collectors

Up until now, we have only discussed precise collectors, where the collector is always able to tell if a value is an object pointer. Conservative collectors [5] relax this constraint, greatly reducing the impact of garbage collection on the mutator at a cost of decreased precision. In terms of the interface, the collector can no longer abstract object pointers, because it does not know exactly what is a pointer. The practical implication of this is that we must add a lemma to the collector interface that says that the representation of all heap values is the identity function. The interface still allows us to hide auxiliary data structures and constraints from the mutator. For instance, the collector may require mark bits, or segregate objects by size, but the particular details of this should not affect the mutator. Aside from these additional constraints the collector may use to improve precision, the actual representation of objects within the concrete state will likely use something like *idRepr*, as defined in Section 6.1. For details on how our approach can be used with a conservative collector, see [27].

component	# of lines
core machine definitions	433
SCAP definition and soundness	975
separation logic library	4326
finite set library	3858
map libraries	1112
basic GC definitions	2258
dummy GC	1106
mark-sweep GC	10744
incremental mark-sweep GC	11016
Cheney copying GC	7775
Baker incremental copy GC	17252
list sum and reverse examples	2832
misc. utility files	1310

Figure 23. Length of proof scripts

7. Implementation

The focus and key novelty of our paper is the development of a general framework for reasoning about mutator-garbage collector interaction. We discuss our mechanized proofs primarily to support our claim of generality. In our Coq implementation [29] we have defined the machine semantics, implemented and proved sound SCAP and our fragment of separation logic. We have also verified the list sum mutator example to match the specification described in the paper, along with a list reversal example not mentioned here. We have verified the safety of an assembly implementation of a minimal stub GC, a mark-sweep GC, a Cheney collector, a Baker collector, and shown that these collectors satisfy the abstract interface described in this paper. We have also undertaken a large part of the verification of an incremental mark-sweep GC. The mark-sweep collectors use a *gcStep* similar to *basicGcStep*, while the copying collectors use a *gcStep* similar to the heap isomorphism relation of [8]. Consequently, the Baker representation predicate is slightly different than that presented in this paper.

The Coq implementation is around 64,000 lines, including comments and whitespace, although most of this is proofs. Figure 23 gives a breakdown of number of lines in the proof scripts for various components. The length of the collectors includes their definitions and verification. The work has taken many man months. Our Coq proofs do not use any axioms and pass the Coq proof checker.

For Coq proofs, lines of code is probably a reasonable measure of effort, but a poor metric of complexity, as the length of a proof can vary greatly depending on how aggressively tactics are used. We found that the verification of more complex collectors was "denser" than those of simpler ones, as we had improved the tool set enough as we were working to make things easier later. Also, there is some redundancy in the proofs, because the incremental versions of stop-the-world collectors use altered versions of the original proofs. While we made some effort to reduce the size of the proof scripts, there is probably a lot of room for improvement. A reworking of the verification of the mark-sweep collector reduced the size by two-thirds.

The verification of a basic block can be broken into a few steps. First, we automatically infer a simple verification condition in terms of explicit machine steps. This may produce a few simple side conditions. After those have been shown, we compute, step-

by-step, the final machine state of the block. For instructions that are always safe this is trivial, but for instructions that operate on memory we must prove that the location being accessed is valid. In the event of a branch, we must consider both cases. Finally, we have a description of the final state in terms of the initial state. From this, we must show the final state satisfies the precondition of whatever code block we will end up in, and that the behavior of this block plus the behavior of the rest of the procedure satisfies the specification of this block. This final step constitutes the overwhelming portion of verifying a block, as it involves manipulating the high-level predicates involved in the specifications.

To simplify the verification work, we use a wide variety of tactics. We use Coq's tactic for automatically applying a set of rewriting rules to simplify the state where possible. For instance, $\mathbb{S}\{r' \rightsquigarrow n\}(r)$ is equal to n when $r' = r$ and equal to $\mathbb{S}(r)$ when $r' \neq r$. We have 8 such rules. This tactic greatly simplifies verifying blocks of code that are mostly sequences of arithmetic instructions. The Omega tactic, provided by Coq, does the bulk of the arithmetic reasoning we need. Another set of heavily used tactics manipulate proofs of separation logic predicates. One pair of tactics automatically performs simplification of assumptions or goals. These simplifications include combining together all instances of `true`, rearranging nested uses of $*$ into a right-normal form, introducing or eliminating existentials, and taking advantage of the fact that we can prove anything from a proof of $\mathbb{H} \vdash x \mapsto m * x \mapsto n * A$. Another set of tactics can be used to try to match up a separation logic goal with a hypothesis, with varying degrees of aggressiveness about reordering.

8. Related work and conclusion

There is a large body of work on verifying garbage collector algorithms (such as [16, 17, 11, 3, 14]), including mechanized verification in a variety of formal systems (such as [41, 24, 19, 35, 7]) This work mostly focuses on verifying more abstract algorithms, in contrast to our focus on the verification of actual machine level implementations being executed on a (more or less) realistic machine. We also want to verify mutator and collector separately, with a well-defined interface between them, instead verifying a model of the entire system at once. On the other hand, their work generally is able to verify liveness, while it is not clear how to handle it in our system. Their work also has verified concurrent algorithms, while we have not. Higher level verification is of course also very useful when doing lower level verification.

Vechev *et al.* [43] discuss a way to apply a series of correctness-preserving transformations to a very abstract concurrent mark-sweep algorithm, thereby deriving a more realistic (and still correct) algorithm. Their work focuses on explaining the behavior of variants of a single class of collectors (concurrent mark-sweep) within a single framework, while we are attempting to describe a broader class of algorithms, but we are focusing mostly on generalizing the interface, and only on correctness. Morrisett *et al.* [32] discuss a high level semantics of garbage collection, which is similar to our work in that it involves reasoning about interference between the mutator and the garbage collector.

Our work builds on work by Birkedal *et al.* on verifying the implementation of a Cheney collector [4]. They use separation logic combined with Hoare logic to reason about GC implementations, and the same kind of heap isomorphism predicate to describe the behavior of a copying collector. However, they only consider the verification of the collector and do not consider hiding the representation of the garbage collected heap. Our main contribution over [4] is the general framework for verifying mutator and collector together while using different levels of abstraction, even in the presence of read/write barriers needed for incremental collection. Additionally, we have verified more algorithms, and our proofs are machine checked.

Other work focuses on the mutator side of mutator-GC interaction. Calcagno *et al.* [8] give a means of reasoning about the behavior of a copying collector from the perspective of the mutator, and develop a program logic that is sound in the presence of garbage collection being performed at any step. Hunter and Krishnamurthi [23] show that adding garbage collection (as an atomic step) to a formal model of Java is sound. Vanderwaart and Crary [42] present a type system able to describe the interface of the mutator with a modern, realistic garbage collector. While this work all provides a variety of useful models for GC interfaces, it does not deal with the verification of the collector itself, and thus is unable to verify an entire program within a single system.

There is also prior work focusing on type safe garbage collection [44, 30, 31, 20]. This work allows verifying the safety of both mutator and collector in a single framework, while giving a well-defined interface between the two that allows them to be separately checked. However, these seem fairly linked to particular algorithms (aside from [20]) and notions of safety. If you wanted to prove a stronger property than memory safety, you would need to construct a more expressive type system and reprove soundness.

Hawblitzel *et al.* [20] mechanically verify the safety of an implementation of the Cheney collector. Their implementation uses a more complex object model, but for safety has an unnecessary bounds check that we are able to avoid. While it may be possible for them to avoid this, their system still cannot prove partial correctness, in contrast to our system. Additionally, the overall soundness of their system is dependent on a paper proof of the safety of their type system, and on their own implementation of a type checker for their system, while our system is entirely mechanically verified using Coq, an existing general purpose proof system (except of course for the definitions of safety and the machine semantics).

O'Hearn *et al.* [37] also combine separation logic with a form of abstraction. Their approach uses a "hypothetical frame rule" that allows one part of a program (like a garbage collector) to hide some of the heap from another part of a program (like the mutator), in contrast to our approach of using an abstract heap representation predicate *repr* to hide information. While the hypothetical frame rule can completely hide parts of the heap, it is very coarse-grained: to the client each slice of the heap is either completely hidden or completely exposed. This seems insufficient to, for instance, give the Cheney and Baker collectors the same interface, as our approach is able to do. Furthermore, in contrast to our approach, the soundness of the hypothetical frame rule depends on the heap predicates being restricted in some way.

As previously mentioned, our work also draws heavily on prior work on formal reasoning about programs, such as Hoare logic, rely guarantee reasoning about concurrent programs [25], and separation logic [40].

We have demonstrated a framework for the formal verification of garbage collectors, including the specification of a general interface that is general enough to handle a variety of collectors. By combining this work with existing work on typed compilation and machine level Hoare logics for concurrency [45], it should be possible to produce FPCC [1] using a modern concurrent garbage collector.

Acknowledgment

We thank anonymous referees for their suggestions and comments on an earlier version of this paper. This research is based on work supported in part by National Science Foundation (of USA) under Grant CCR-0524545, gifts from Intel (USA), Microsoft, and Intel China Research Center, Innovation Funds from Chinese Academy of Sciences, and the National Natural Science Foundation of China

under Grant No. 60673126. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

[1] A. W. Appel. Foundational proof-carrying code. In *Symp. on Logic in Comp. Sci. (LICS'01)*, pages 247–258. IEEE Comp. Soc., June 2001.

[2] H. G. Baker, Jr. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–294, 1978.

[3] M. Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Trans. Program. Lang. Syst.*, 6(3):333–344, 1984.

[4] L. Birkedal, N. Torp-Smith, and J. C. Reynolds. Local reasoning about a copying garbage collector. In *POPL '04: Proc. of the 31st ACM SIGPLAN-SIGACT symp. on Principles of prog. lang.*, pages 220–231, New York, NY, USA, 2004. ACM Press.

[5] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Exp.*, 18(9):807–820, 1988.

[6] R. A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *LFP '84: Proc. of the 1984 ACM Symp. on LISP and functional prog.*, pages 256–262, New York, NY, USA, 1984. ACM Press.

[7] L. Burdy. B vs. Coq to prove a garbage collector. In R. J. Boulton and P. B. Jackson, editors, *14th Int'l Conference on Theorem Proving in Higher Order Logics: Supplemental Proc.*, pages 85–97, Sept. 2001. Report EDI–INF–RR–0046, Division of Informatics, University of Edinburgh.

[8] C. Calcagno, P. O'Hearn, and R. Bornat. Program logic and equivalence in the presence of garbage collection. *Theoretical Comp. Sci.*, 298(3):557–581, 2003.

[9] J. Chen and D. Tarditi. A simple typed intermediate language for object-oriented languages. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 38–49, New York, NY, USA, 2005. ACM Press.

[10] Coq Development Team. The Coq proof assistant reference manual. Coq release v8.0, Oct. 2005.

[11] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM*, 21(11):966–975, 1978.

[12] X. Feng, Z. Ni, Z. Shao, and Y. Guo. An open framework for foundational proof-carrying code. In *The third ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 67–78, Nice, France, Jan. 2007. ACM Press.

[13] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni. Modular verif. of assembly code with stack-based control abstractions. In *Proc. 2006 ACM Conf. on Prog. Lang. Design and Impl.*, June 2006.

[14] G. Gonthier. Verifying the safety of a practical concurrent garbage collector. In R. Alur and T. Henzinger, editors, *Computer Aided Verification CAV'96*, Lecture Notes in Computer Science, New Brunswick, NJ, 1996. Springer-Verlag.

[15] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.

[16] D. Gries. An exercise in proving parallel programs correct. *Commun. ACM*, 20(12):921–930, 1977.

[17] D. Gries. Corrigendum. *Commun. ACM*, 21(12):1048, 1978.

[18] J. Guy L. Steele. Multiprocessing compactifying garbage collection. *Commun. ACM*, 18(9):495–508, 1975.

[19] K. Havelund. Mechanical verification of a garbage collector. In *FMPPTA'99*, 1999.

[20] C. Hawblitzel, H. Huang, L. Wittie, and J. Chen. A garbage-collecting typed assembly language. In *The Third ACM SIGPLAN Workshop on Types in Lang. Design and Impl.* ACM Press, Jan. 2007.

[21] A. Hejlsberg, S. Wiltamuth, and P. Golde. *The C# Programming Language*. Addison-Wesley, Boston, Mass., 2004.

[22] G. Hunt, J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, Redmond, WA, 2005.

[23] R. Hunter and S. Krishnamurthi. A model of garbage collection for oo languages. In *Tenth Int'l Workshop on Foundations of Object-Oriented Lang. (FOOL10)*, 2003.

[24] P. Jackson. Verifying a garbage collection algorithm. In *Proc. of 11th Int'l Conference on Theorem Proving in Higher Order Logics TPHOLs'98*, volume 1479 of *Lecture Notes in Computer Science*, pages 225–244, Canberra, Sept. 1998. Springer-Verlag.

[25] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

[26] R. E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[27] C. Lin, A. McCreight, Z. Shao, Y. Chen, and Y. Guo. Foundational typed assembly language with certified garbage collection. In *1st IEEE & IFIP International Symposium on Theoretical Aspects of Software Engineering (TASE 2007)*, June 2007.

[28] A. McCreight, Z. Shao, C. Lin, and L. Li. A general framework for certifying garbage collectors and their mutators (extended version). Technical Report YALEU/DCS/TR-1378, Yale University, New Haven, CT, Mar. 2007.

[29] A. McCreight, Z. Shao, C. Lin, and L. Li. A general framework for certifying garbage collectors and their mutators (implementation). `http://flint.cs.yale.edu/flint/publications/hgc.html`, Jan. 2007.

[30] S. Monnier, B. Saha, and Z. Shao. Principled scavenging. In *Proc. 2001 ACM Conf. on Prog. Lang. Design and Impl.*, pages 81–91, New York, 2001. ACM Press.

[31] S. Monnier and Z. Shao. Typed regions. Technical Report YALEU/DCS/TR-1242, Dept. of Comp. Sci., Yale University, New Haven, CT, Oct. 2002.

[32] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *FPCA '95: Proc. of the 7th Int'l conference on Functional prog. lang. and comp. architecture*, pages 66–77, New York, NY, USA, 1995. ACM Press.

[33] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

[34] Mozilla. Mozilla foundation security advisory 2006-01. `http://www.mozilla.org/security/announce/2006/mfsa2006-01.html`.

[35] L. P. Nieto and J. Esparza. Verifying single and multi-mutator garbage collectors with owicki-gries in isabelle/hol. In *MFCS '00: Proc. of the 25th Int'l Symp. on Mathematical Foundations of Comp. Sci.*, pages 619–628, London, UK, 2000. Springer-Verlag.

[36] NIST. Vulnerability summary cve-2006-3451. `http://nvd.nist.gov/nvd.cfm?cvename=CVE-2006-3451`.

[37] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 268–280, New York, NY, USA, 2004. ACM Press.

[38] W. Paul, M. Broy, and T. In der Rieden. The verisoft project. URL: `http://www.verisoft.de` , 2007.

[39] C. Paulin-Mohring. Inductive definitions in the system Coq—rules and properties. In *Proc. TLCA*, volume 664 of *Lecture Notes in Computer Science*, 1993.

[40] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02: Proc. of the 17th Annual IEEE Symp. on Logic in Comp. Sci.*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[41] D. M. Russinoff. A mechanically verified incremental garbage collector. *Formal Aspects of Computing*, 6:359–390, 1994.

[42] J. C. Vanderwaart and K. Crary. A typed interface for garbage collection. In *TLDI '03: Proc. of the 2003 ACM SIGPLAN Int'l Workshop on Types in Lang. Design and Impl.*, pages 109–122, New York, NY, USA, 2003. ACM Press.

[43] M. T. Vechev, E. Yahav, and D. F. Bacon. Correctness-preserving derivation of concurrent garbage collection algorithms. In *PLDI '06: Proc. of the 2006 ACM SIGPLAN conference on Prog. Lang. Design and Impl.*, pages 341–353, New York, NY, USA, 2006. ACM Press.

[44] D. C. Wang and A. W. Appel. Type-preserving garbage collectors. In *Proc. of the 28th ACM Symp. on Principles of prog. lang.*, pages 166–178, New York, NY, USA, 2001. ACM Press.

[45] D. Yu and Z. Shao. Verification of safety properties for concurrent assembly code. In *Proc. 9th ACM SIGPLAN International Conference on Functional Programming*, September 2004.

[46] T. Yuasa. Real-time garbage collection on general-purpose machines. *J. Syst. Softw.*, 11(3):181–198, 1990.