Automated Verification of Practical Garbage Collectors

Chris Hawblitzel
Microsoft Research
One Microsoft Way
Redmond, WA 98052
USA
Chris.Hawblitzel@microsoft.com

Erez Petrank *
Dept of Computer Science
Technion
Haifa 32000
Israel
erez@cs.technion.ac.il

Abstract

Garbage collectors are notoriously hard to verify, due to their low-level interaction with the underlying system and the general difficulty in reasoning about reachability in graphs. Several papers have presented verified collectors, but either the proofs were hand-written or the collectors were too simplistic to use on practical applications. In this work, we present two mechanically verified garbage collectors, both practical enough to use for real-world C# benchmarks. The collectors and their associated allocators consist of x86 assembly language instructions and macro instructions, annotated with preconditions, postconditions, invariants, and assertions. We used the Boogie verification generator and the Z3 automated theorem prover to verify this assembly language code mechanically. We provide measurements comparing the performance of the verified collector with that of the standard Bartok collectors on off-the-shelf C# benchmarks, demonstrating their competitiveness.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

General Terms Verification

1. Introduction

Garbage collectors automatically reclaim dynamically allocated objects that will never be accessed again by the program. Garbage collection is widely acknowledged for supporting fast development of reliable and secure software. It has been incorporated into modern languages, such as Java and C#. Many recent projects have attempted to verify the safety or correctness of garbage collectors. The goal of this verification is to reduce the trusted computing base of a system and increase the system's reliability. This is particularly important for secure systems based on proof-carrying code (PCC) [23] or typed assembly language (TAL) [22]; typical large-scale PCC/TAL systems can verify the safety of the mutator (the program), but not of the run-time system that manages memory and other resource on the mutator's behalf. This prevents untrusted programs from customizing the run-time system. Furthermore, bugs in the unverified run-time systems could result in security vulnerabilities that undermine the guarantees promised by PCC and TAL.

Proving that garbage collectors are safe and correct has been a challenge. In this work, we provide the first fully mechanized correctness proofs of garbage collectors and allocators realistic enough to run large, off-the-shelf benchmarks. To make this verification tractable, we exploit recent advances in automated theorem proving technology, using the Boogie [3] and Z3 [8] tools to provide *automated* verification of the correctness properties. Our key contribution is the expression of garbage collector specifications and invariants in a style that allows efficient, automated verification.

We verify two collectors, both practical enough for use with real-world C# benchmarks: a Cheney copying collector [20, 7], with a bump allocator; and a mark-sweep collector [18], with a local-cache allocator that allows fast bump-pointer allocation. Both are simple enough to verify, yet efficient enough to support realistic benchmarks competitively. The collectors and their associated allocators consist of x86 assembly language instructions and macro instructions, annotated with preconditions, postconditions, invariants, and assertions. These annotations require significant human effort to write, but once they are written, the Boogie verification condition generator and the Z3 theorem prover verify the annotated collectors automatically, with no further human intervention. The collectors and allocators are entirely self-contained, relying on no unverified library code, and the verification relies on only a minimal set of trusted axioms and definitions describing 32-bit arithmetic, x86 instructions, memory words, and the interface to the mutator.

We show how to define higher-level abstractions, particularly abstractions drawn from region-based type systems, in terms of these trusted axioms and definitions; these higher-level abstractions provide forms of local reasoning that make automated verification tractable. The verification ensures that if an allocation or garbage collection operation completes, then the physical heap managed by the allocator and collector faithfully represents the abstract graph of objects defined by the mutator. The verification also ensures that the garbage collector deallocates all objects unreached during the collection. The verification does not prove termination; verified collectors or allocators could fail to terminate because of an infinite loop, or fail to terminate properly because of a 32-bit integer overflow exception, or an explicit halt operation. (The allocators and collectors halt if they run out of memory, or if the mutator relies on a feature not supported by our collectors, such as multithreading.)

The collectors and allocators include support for objects, arrays, strings, header words, interior pointers, static data scanning, stack scanning, object descriptors, stack frame descriptors, return-address lookup tables, and bit-level data manipulation, making them realistic enough to support off-the-shelf single-threaded C#

* Part of this work was done while the author was on a sabbatical leave at Microsoft Research. Supported by THE ISRAEL SCIENCE FOUNDATION (grant No. 845/06).

POPL'09, January 18–24, 2009, Savannah, Georgia, USA.

benchmarks compiled with the Bartok compiler, using the native Bartok memory layouts and descriptor formats. To assess the efficacy of the proposed collectors, we ran the verified collectors with the Bartok runtime and compared their performance with the standard Bartok mark-sweep and generational copying collectors. The verified collectors demonstrated competitive performance.

The contributions in this paper include:

1. We provide the first mechanically verified garbage collectors that support a real-world object model, including vtables, arrays, object descriptors, stacks, etc.
2. We provide the first mechanically verified garbage collectors that can link to code generated by a real-world, optimizing compiler (Bartok).
3. We demonstrate how to apply *automated* verification to garbage collectors, including both copying and mark-sweep garbage collectors. This automation allows scaling the verification to realistic collectors without employing a huge human effort.
4. We propose a simple, efficient, easy-to-verify mark-sweep collector and allocator based on local caches.
5. We provide the first performance measurements of off-the-shelf C# benchmarks running on top of verified garbage collectors.

Outline. Section 2 discusses previous work on garbage collector verification. Section 3 describes Boogie and Z3. Section 4 presents a complete example mark-sweep collector and allocator in the BoogiePL programming language [3], describing the specification and invariants in detail. Section 5 generalizes Section 4's ideas to cover copying collectors, borrowing ideas from region-based type systems. Section 6 presents two simple, yet practical, collectors (and their allocators): a Cheney-queue copying collector and an iterative mark-sweep collector. Section 7 shows that the practical collectors perform reasonably well compared to Bartok's native collectors on a range of off-the-shelf C# benchmarks. Section 8 concludes.

Code availability. The garbage collectors were coded in an x86-like subset of the BoogiePL language; a small tool automatically extracted the x86 instructions, which were assembled and linked with the benchmarks (see Section 6.3). The complete BoogiePL code for the two practical collectors is available as part of the public Microsoft Research Singularity RDK2 source (in "Source Code", in the base/Imported/Bartok/runtime/verified/GCs directory, which can be browsed without downloading all of Singularity) at:

`http://www.codeplex.com/singularity`

The Boogie and Z3 tools (April 2008 release), used to verify the two collectors, are available from:

`http://research.microsoft.com/specsharp/`

2. Background and related work

Hand-written proofs of garbage collector correctness, at least for abstract models of garbage collectors, go back decades (e.g., [9, 10, 4, 17]). The work of Birkedal *et al*[4] is noteworthy for formally proving a Cheney copying collector correct, rather than a mark-sweep collector, and emphasizing *local reasoning* based on separation logic. Nevertheless, the local reasoning is used mainly to separate pieces of the invariant at a coarse granularity (e.g. separating invariants about forwarded objects from unforwarded objects); we offer a different perspective on local reasoning in section 5.

Other work [25, 13, 14, 15, 12] has mechanically proven garbage collector correctness, but only for mark-sweep collectors, and only using abstract models of memory (for instance, representing the heap as just a mathematical graph and the root set as just a mathematical set), and only using abstract models of programs rather than programs executable on real hardware. These papers used interactive theorem provers, with the exception of Russinoff[25], and even this paper required over 100 explicitly user-declared lemmas (each of which was automatically proved). More recently, McCreight *et al* [19] used an interactive theorem prover to verify the correctness of both mark-sweep and copying collectors written in a RISC-like assembly language, with a more realistic memory model. This required an enormous effort though, relying on over 10000 lines of Coq scripts per collector, and the treatment of the memory still falls short of what realistic compilers expect: the collectors assume that every object has exactly two fields, and there is no stack, no static data area, no object and stack frame descriptors, and so on. We adopt McCreight *et al*'s definition of correctness as a starting point for our work.

Several papers [27, 21] use typed regions to implement type-safe copying garbage collectors; these garbage collectors copy live data from an old region to a new region, and then (safely) delete the old region. Type safety is a weaker property than correctness, though, and these techniques don't obviously extend to mark-sweep collection. We borrow ideas from typed regions to help us verify our copying collector.

Vechev *et al.* [26] describe how to mechanically fit prefabricated, high-level garbage collection building blocks together in a provably correct way, but they do not mechanically verify the building blocks themselves.

3. Boogie and Z3

BoogiePL is a simple imperative programming language designed to support automated program verification. It includes pure (side-effect free) expressions, written in a standard C/C#/Java syntax, imperative statements (which may update local variables and global variables), pure functions, and imperative procedures. Procedures support preconditions and postconditions, written with the keywords `requires` and `ensures`, that specify what must be true upon entry to the procedure and what the procedure guarantees is true upon exit from the procedure. Within a procedure, loop invariants for `while` loops are written with the `invariant` keyword. The following example shows a pure function `Pos`, which returns true if its argument is positive, and a procedure `IncreaseX` that adds a positive number `y` to a global variable `x`:

```
function{:expand true} Pos(i:int)returns(bool){i>0}
var x:int;
procedure IncreaseX(y:int)
  requires Pos(y);
  modifies x;
  ensures  x > old(x);
{
  x := x + y;
}
```

In this example, the expression `old(x)` refers to the value of `x` at the beginning of the procedure's execution, so that the postcondition "`ensures  x > old(x);`" says that `x` will have a larger value upon exit from the procedure than upon entry to the procedure. A procedure must disclose all the global variables it modifies (just `x` in this example); this allows callers of the procedure to know which variables remain unmodified by the procedure. The `expand true` annotation turns a function definition into a macro that is expanded to its definition whenever it is used, so that "`requires Pos(y);`" is just an abbreviation for "`requires y > 0;`". (Recursive or mutually recursive macro defintions are disallowed.)

Our programs occasionally use the statement "`assert P;`", which asks the verifier to prove `P`, which is then used as a lemma for

subsequent proving. (We do not use the statement "`assume P;`", which introduces a new lemma `P` *without* proof, since this would make our verification unsound.)

The Boogie tool generates verification conditions from the BoogiePL code. These verification conditions are logical formulas that, if valid, guarantee that each procedure call satisfies the procedure's precondition, each procedure guarantees its postcondition, and each loop invariant holds on entry to the loop and is maintained by each loop iteration. Boogie passes these verification conditions to an automated theorem prover, which attempts to prove the validity of the verification conditions. We use the Z3 theorem prover, which is efficient, scales to large formulas, and reasons about many useful first-order logic theories, including integers, bit vectors, arrays, and uninterpreted functions.

BoogiePL's data types are more purely mathematical than the data types in conventional programming languages. The type `int` represents mathematical integers, ranging from negative infinity to positive infinity, while `bv32` represents 32-bit values. The theorem prover support for `int` is more mature and efficient than for `bv32`, so we used `int` wherever possible (section 6 describes how we reconciled this approach with the x86's native 32-bit words).

BoogiePL also supports array types `[int]t` for any element type `t`, defining arrays as simple mappings from mathematical integers to elements. The BoogiePL "select" expression `a[i]` retrieves element `i` from array `a`, where `i` can be any integer. The BoogiePL "update" expression `a[i := v]` generates a new array, equal to `a` except at element `i`, where the new array contains the value `v`, so that `(a[i := v])[i] == v` is true for any `a`, `i`, and `v`. For convenience, the statement "`a[i] := v;`" is an abbreviation for "`a := (a[i := v]);`". Arrays can also be multidimensional: an array `a` of type `[int,int]t` supports a select expression `a[i1,i2]` and an update expression `a[i1,i2 := v]`. Note that BoogiePL arrays lack many properties of say, Java arrays. For example, BoogiePL arrays are not references, so there's no issue of aliasing: the statement "`a := b;`" assigns a copy of array `b` to variable `a`.

Due to formatting constraints, the BoogiePL code shown in this paper omits most type annotations. We abbreviate `a<=b && b<c` as `a<=b<c`, and `function{:expand true}` as `fun`. The notation "$\forall^T$" is an abbreviation for the universal quantifier "$\forall$" with a particular *trigger* "T", used as a hint to Z3, as described further in Section 4.3. For now, the reader may ignore the "T".

4. A miniature collector in BoogiePL

This section presents a miniature allocator and mark-sweep collector written in the BoogiePL programming language, introducing some of the invariants used by the more realistic collectors in subsequent sections. The allocator and collector are implemented as a single BoogiePL file, shown in its entirety in Figures 1-5. When run on this example garbage collector, Boogie verifies all 7 procedures in the collector in less than 2 seconds; since Boogie and Z3 process BoogiePL files entirely automatically, no human assistance or proof scripts are required.

The miniature collector assumes that every object has exactly two fields, numbered `0` and `1`, and each field holds a non-null pointer to some object. The collector manages memory addresses in the range `memLo...memHi - 1`, where `memLo` and `memHi` are constants such that `0 < memLo <= memHi - 1`, but whose values are otherwise unspecified (see Figure 1). Memory is object addressed, rather than byte addressed or word addressed, so that each memory location in the range `memLo...memHi - 1` contains either an entire object, or free space big enough to allocate an object in. The variable `Mem`, of type `[int,int]int`, represents all of memory; for each address `i` in the range `memLo...memHi - 1` and field `field` in the range 0...1, the value `Mem[i,field]` holds the contents of the field `field` in the object at address `i`.

```
function{:expand false} T(i) { true }
const NO_ABS:int, memLo:int, memHi:int;
axiom 0 < memLo <= memHi;
fun memAddr(i) { memLo <= i < memHi }

fun Unalloc(i) { i == 0 }
fun White(i) { i == 1 }
fun Gray(i) { i == 2 }
fun Black(i) { i == 3 }

var Mem:[int,int]int, Color:[int]int;
var $toAbs:[int]int, $AbsMem:[int,int]int;

fun WellFormed($toAbs) {
  (∀T i1 ∀T i2. memAddr(i1) && memAddr(i2)
     && $toAbs[i1] != NO_ABS
     && $toAbs[i2] != NO_ABS
     && i1 != i2
   ==> $toAbs[i1] != $toAbs[i2])
}
fun Pointer($toAbs, ptr, $abs) {
    memAddr(ptr) && $abs != NO_ABS
 && $toAbs[ptr] == $abs
}
fun ObjInv(i, $toAbs, $AbsMem, Mem) {
  $toAbs[i] != NO_ABS ==>
      Pointer($toAbs, Mem[i,0], $AbsMem[$toAbs[i],0])
   && Pointer($toAbs, Mem[i,1], $AbsMem[$toAbs[i],1])
}
fun GcInv(Color, $toAbs, $AbsMem, Mem) {
    WellFormed($toAbs)
 && (∀T i. memAddr(i) ==>
        ObjInv(i, $toAbs, $AbsMem, Mem)
     && 0 <= Color[i] < 4
     && (Black(Color[i]) ==> !White(Color[Mem[i,0]])
                          && !White(Color[Mem[i,1]])))
     && ($toAbs[i] == NO_ABS <==> Unalloc(Color[i])))
}
fun MutatorInv(Color, $toAbs, $AbsMem, Mem) {
    WellFormed($toAbs)
 && (∀T i. memAddr(i) ==>
        ObjInv(i, $toAbs, $AbsMem, Mem)
     && 0 <= Color[i] < 2
     && ($toAbs[i] == NO_ABS <==> Unalloc(Color[i])))
}
```

Figure 1. ***Miniature Collector: Definitions.***

The allocator and collector use a variable `Color` to represent the state of memory at each address. If `Color[i]` is `0`, the memory at address `i` is free. Otherwise, the memory is occupied by an object and is either colored white (`Color[i] == 1`), gray (`Color[i] == 2`), or black (`Color[i] == 3`).

4.1 Concrete and abstract states

To verify a garbage collector, we must specify what it means for a collector to be correct. For the mark-sweep collector, the most obvious criterion is that it frees all objects unreachable from the root and leaves all reachable objects unmodified. However, this definition of correctness is specific to one particular class of collectors; it doesn't account for collectors that move objects, and doesn't account for mutator-collector interaction, such as write barriers and read barriers. We'd like one definition of correctness that encompasses many classes of collectors, so we follow a more general approach advocated by McCreight et al [19]. In this approach, the mutator de-

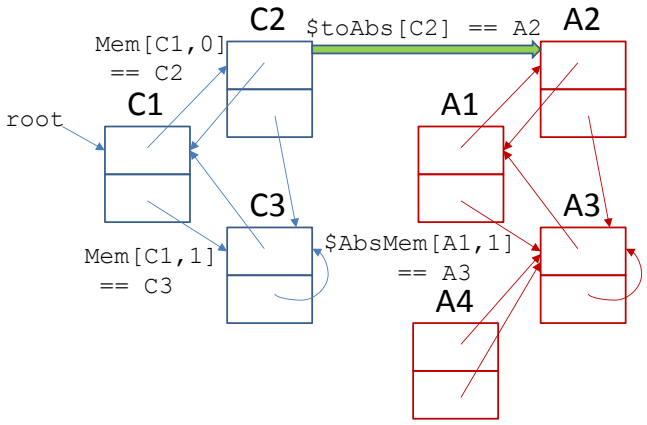

Figure 2. Concrete and abstract graphs

```
procedure Initialize()
  modifies $toAbs, Color;
  ensures  MutatorInv(Color, $toAbs, $AbsMem, Mem);
  ensures  WellFormed($toAbs);
{
  var ptr;
  ptr := memLo;
  while (ptr < memHi)
    invariant T(ptr) && memLo <= ptr <= memHi;
    invariant (∀^T i. memLo <= i <ptr ==>
      $toAbs[i] == NO_ABS && Unalloc(Color[i]));
  {
    Color[ptr] := 0;
    $toAbs[ptr] := NO_ABS;
    ptr := ptr + 1;
  }
}

procedure ReadField(ptr, field) returns (val)
  requires MutatorInv(Color, $toAbs, $AbsMem, Mem);
  requires Pointer($toAbs, ptr, $toAbs[ptr]);
  requires field == 0 || field == 1;
  ensures  Pointer($toAbs, val,
                   $AbsMem[$toAbs[ptr],field]);
{
  assert T(ptr);
  val := Mem[ptr,field];
}

procedure WriteField(ptr, field, val)
  requires MutatorInv(Color, $toAbs, $AbsMem, Mem);
  requires Pointer($toAbs, ptr, $toAbs[ptr]);
  requires Pointer($toAbs, val, $toAbs[val]);
  requires field == 0 || field == 1;
  modifies $AbsMem, Mem;
  ensures  MutatorInv(Color, $toAbs, $AbsMem, Mem);
  ensures  $AbsMem ==
    old($AbsMem)[$toAbs[ptr],field := $toAbs[val]];
{
  assert T(ptr) && T(val);
  Mem[ptr,field] := val;
  $AbsMem[$toAbs[ptr],field] := $toAbs[val];
}
```

Figure 3. *Miniature Collector: Initialize, ReadField, WriteField.*

```
procedure GarbageCollect(root)
  requires MutatorInv(Color, $toAbs, $AbsMem, Mem);
  requires root != 0 ==>
             Pointer($toAbs, root, $toAbs[root]);
  modifies Color, $toAbs;
  ensures  MutatorInv(Color, $toAbs, $AbsMem, Mem);
  ensures  root != 0 ==>
             Pointer($toAbs, root, $toAbs[root]);
  ensures  (∀^T i.memAddr(i) && $toAbs[i] != NO_ABS ==>
             $toAbs[i] == old($toAbs)[i]);
  ensures  root != 0 ==>
             $toAbs[root] == old($toAbs)[root];
{
  assert T(root);
  if (root != 0) {
    call Mark(root);
  }
  call Sweep();
}

procedure Alloc(root, $abs) returns (newRoot,ptr)
  requires MutatorInv(Color, $toAbs, $AbsMem, Mem);
  requires root != 0 ==>
             Pointer($toAbs, root, $toAbs[root]);
  requires $abs != NO_ABS;
  requires (∀^T i. memAddr(i) ==> $toAbs[i] != $abs);
  requires $AbsMem[$abs,0] == $abs;
  requires $AbsMem[$abs,1] == $abs;
  modifies Color, $toAbs, Mem;
  ensures  MutatorInv(Color, $toAbs, $AbsMem, Mem);
  ensures  root != 0 ==>
             Pointer($toAbs,newRoot,old($toAbs)[root]);
  ensures  Pointer($toAbs, ptr, $abs);
  ensures  WellFormed($toAbs);
{
  while (true)
    invariant MutatorInv(Color, $toAbs, $AbsMem, Mem);
    invariant root != 0 ==>
      Pointer($toAbs, root, $toAbs[root]);
    invariant (∀^T i. memAddr(i) ==> $toAbs[i] != $abs);
    invariant root != 0 ==>
      $toAbs[root] == old($toAbs)[root];
  {
    ptr := memLo;
    while (ptr < memHi)
      invariant T(ptr) && memLo <= ptr <= memHi;
    {
      if (Unalloc(Color[ptr])) {
        Color[ptr] := 1; // make white
        $toAbs[ptr] := $abs;
        Mem[ptr,0] := ptr;
        Mem[ptr,1] := ptr;
        newRoot := root;
        return;
      }
      ptr := ptr + 1;
    }
    call GarbageCollect(root);
  }
}
```

Figure 4. *Miniature Collector: Garbage Collection and Allocation.*

fines an abstract state, consisting of an abstract graph of abstract nodes. A memory manager is responsible for representing the abstract state in memory. The memory manager exposes procedures to initialize memory, allocate memory, read memory, and write memory (see `Initialize`, `Alloc`, `ReadField`, and `WriteField` in Figures 3, 4). Correctness means that each of these procedures faithfully represent the abstract state.

To make this notion of correctness precise, the variable `$AbsMem` of type `[int,int]int` defines the abstract state as a mapping from abstract nodes and fields to abstract values. In the miniature memory model presented so far, each field contains a pointer to a node, so the abstract values stored in the abstract graph are always abstract nodes. (Section 6 extends the set of abstract values with other values, such as primitive integers and null.) For example, Figure 2 shows an abstract graph consisting of 4 nodes, `A1`, `A2`, `A3`, and `A4`, each having two fields numbered `0` (on top) and `1` (on the bottom). In this example, `A1`'s bottom field points to `A3`, so `$AbsMem[A1,1] == A3`. Integers represent abstract nodes, but these integers can be any mathematical integers, and need not be related to the addresses used by the computer's actual memory. In fact, the variable `$AbsMem` is not represented at run-time at all; it is used solely for verification. We call such variables "ghost variables" (also known as "auxillary variables"), and we use a naming convention that prefixes each ghost variable with a dollar sign.

The function `MutatorInv(...)` defines the invariant that holds on the memory manager's data while the mutator is running. Initialize establishes `MutatorInv`, while `Alloc`, `ReadField`, and `WriteField` require `MutatorInv` as a precondition and guarantee `MutatorInv` as a postcondition. Each collector defines `MutatorInv(var1...varn)` as it wishes. The mutator is not allowed to modify any of the variables `var1...varn` directly, but instead must use `ReadField`, `WriteField`, and `Alloc` to affect these variables. Since `MutatorInv` varies across collectors, a mutator that wants to work with all collectors should treat `MutatorInv` as abstract. In this framework, the specifications for `Initialize`, `Alloc`, `ReadField`, and `WriteField` are exactly the same across all collectors, except for the differing definitions of `MutatorInv`.

The function `$toAbs:[int]int` maps each concrete memory address in the range `memLo...memHi - 1` to an abstract node, or to `NO_ABS`. The memory management procedures ensure that `$toAbs` is well formed (`WellFormed($toAbs)`), which says that any two distinct concrete addresses `i1` and `i2` map to distinct abstract nodes, unless they map to `NO_ABS`. In Figure 2, `$toAbs` maps addresses `C1`, `C2`, and `C3` to abstract nodes `A1`, `A2`, and `A3`, respectively, while all other concrete addresses map to `NO_ABS`. The function `Pointer($toAbs,ptr,$abs)` says that `$toAbs` maps the concrete address `ptr` to the abstract node `$abs`.

Suppose the mutator calls `ReadField(C1,0)`, which will return the contents of field `0` of the object at address `C1`. The precondition `Pointer($toAbs,ptr,$toAbs[ptr])` requires `C1` to be a valid pointer, mapped to some abstract node (`A1` in this example). In the miniature memory model presented so far, all fields hold pointers, so the return value will also be a pointer; the postcondition for `ReadField` ensures that the returned value is the pointer corresponding to the abstract node `$AbsMem[$toAbs[ptr],field] = $AbsMem[A1,0] = A2`. Since only one pointer, `C2`, maps to `A2`, the postcondition forces `ReadField(C1,0)` to return exactly the address `C2`. (The well-formedness condition, `WellFormed($toAbs)` ensures that no node other than `C2` maps to `A2`.) Once the mutator obtains the pointer `C2` from `ReadField(C1,0)`, it may call, say, `ReadField(C2,1)` to obtain the pointer `C3`. In this way, the specification of `ReadField` allows the mutator to traverse the reachable portion of memory, even though the specification never mentions reachability directly. The specification does not obligate the memory manager to retain unreachable objects. Since `A1`, `A2`, and `A3` do not point to `A4`, the memory manager need not devote any physical memory for representing `A4`. In Figure 2, there is no concrete address that maps to `A4`. (Note: in our practical verified collectors, the mutator does not make actual run-time procedure calls to `ReadField` and `WriteField`; instead, the postconditions of `ReadField` and `WriteField` prove the properties that the mutator needs to read or write memory, without actually reading or writing the memory. For example, `ReadField` ensures that `Pointer($toAbs,Mem[ptr,field],...)`.)

The mutator allocates new abstract nodes by calling `Alloc`, passing in a fresh abstract node `$abs` whose fields initially point to itself. Unlike `ReadField` and `WriteField`, `Alloc` modifies `$toAbs`, which potentially invalidates any pointers that the mutator possesses. (The mutator can't use an invalid pointer that refers to an old version of `$toAbs`, because `Pointer($toAbs,...)` for an old `$toAbs` won't satisfy the preconditions for `ReadField` and `WriteField`, which are in terms of the current `$toAbs`.) Therefore, the mutator may pass in a root pointer, and the `Alloc` procedure returns a new root pointer that points to the same abstract node as the old pointer. We could also allow `ReadField` and `WriteField` to modify `$toAbs`, in which case these procedures would also require a root to be passed in. In practice, though, this would be an onerous burden on the mutator.

4.1.1 Verifying collection effectiveness

The specification described so far hides the garbage collection process behind the `Initialize`, `ReadField`, `WriteField`, and `Alloc` interfaces. We also verify one internal property of the garbage collector, invisible to the mutator: after a collection, only abstract nodes that the collector reached have physical memory dedicated to them; unreached abstract nodes are not represented in memory. It's easy to define an axiom for reachability for any particular abstract graph: for any node `A`, if `A` is reachable, then `A`'s children are also reachable. It's difficult, though, to track reachability as the edges in a graph evolve. For the two collectors presented here, the `$AbsMem` graph remains unmodified throughout collection, but in general, this is not true: incremental collectors interleave short spans of garbage collection with short spans of mutator activity, and the mutator activity modifies `$AbsMem`. Therefore, we adopt a looser criterion: rather than checking that all remaining allocated nodes at the end of a collection are *reachable* from the root, we merely check that all remaining allocated nodes were *reached* from the root at some time since the start of the collection. Verifying this property was only a small extension to the rest of the verification.

4.2 Allocation, marking, and sweeping

Figure 4's `Alloc` procedure performs an (inefficient) linear search for a free memory address; if no free space remains, `Alloc` calls the garbage collector. The collector recursively marks all nodes reachable from some root pointer (the "mark phase"), and then deallocates all unmarked objects (the "sweep phase"). Figure 5 shows the code for both the `Mark` and `Sweep` procedures. The next few paragraphs trace the preconditions and postconditions for `Mark` and `Sweep` backwards, starting with `Sweep`'s postconditions.

A key property of `Sweep` is that it leaves no dangling pointers (pointers from allocated objects to free space). This property is part of `MutatorInv`: each memory address `i` satisfies `ObjInv(i, ...)`, which ensures that if some object lives at `i` (if `$toAbs[i] != NO_ABS`), then the object's fields contain valid pointers to allocated objects (see Figure 1). Specifically, the fields `Mem[i,0]` and `Mem[i,1]` are, like `i`, mapped to some abstract nodes, so that `$toAbs[Mem[i,0]] != NO_ABS` and `$toAbs[Mem[i,1]] != NO_ABS`. To maintain this property, `Sweep` must ensure that any object it deallocates had no pointers from objects that remain allocated. Since `Sweep` deallocates white objects

```
procedure Mark(ptr)
  requires GcInv(Color, $toAbs, $AbsMem, Mem);
  requires memAddr(ptr) && T(ptr);
  requires $toAbs[ptr] != NO_ABS;
  modifies Color;
  ensures  GcInv(Color, $toAbs, $AbsMem, Mem);
  ensures  (∀T i. !Black(Color[i]) ==>
                Color[i] == old(Color)[i]);
  ensures  !White(Color[ptr]);
{
  if (White(Color[ptr])) {
    Color[ptr] := 2; // make gray
    call Mark(Mem[ptr,0]);
    call Mark(Mem[ptr,1]);
    Color[ptr] := 3; // make black
  }
}

procedure Sweep()
  requires GcInv(Color, $toAbs, $AbsMem, Mem);
  requires (∀T i. memAddr(i) ==> !Gray(Color[i]));
  modifies Color, $toAbs;
  ensures  MutatorInv(Color, $toAbs, $AbsMem, Mem);
  ensures  (∀T i. memAddr(i) ==>
      (Black(old(Color)[i]) ==> $toAbs[i] != NO_ABS)
   && ($toAbs[i] != NO_ABS ==>
        $toAbs[i] == old($toAbs)[i]));
{
  var ptr;
  ptr := memLo;
  while (ptr < memHi)
    invariant T(ptr) && memLo <= ptr <= memHi;
    invariant WellFormed($toAbs);
    invariant (∀T i. memAddr(i) ==>
        0 <= Color[i] < 4
     && !Gray(Color[i])
     && (Black(old(Color)[i]) ==>
           $toAbs[i] != NO_ABS
         && ObjInv(i, $toAbs, $AbsMem, Mem)
         && (Mem[i,0] >= ptr ==>
               !White(Color[Mem[i,0]]))
         && (Mem[i,1] >= ptr ==>
               !White(Color[Mem[i,1]])))
     && ($toAbs[i] == NO_ABS <==> Unalloc(Color[i]))
     && ($toAbs[i] != NO_ABS ==>
          $toAbs[i] == old($toAbs)[i])
     && (ptr <= i ==> Color[i] == old(Color)[i])
     && (i < ptr ==> 0 <= Color[i] < 2)
     && (i < ptr && White(Color[i]) ==>
          Black(old(Color)[i])));
  {
    if (White(Color[ptr])) {
      Color[ptr] := 0; // deallocate
      $toAbs[ptr] := NO_ABS;
    }
    else if (Black(Color[ptr])) {
      Color[ptr] := 1; // make white
    }
    ptr := ptr + 1;
  }
}
```

Figure 5. *Miniature Collector: Mark and Sweep.*

and leaves gray and black objects allocated, `Sweep`'s preconditions requires that no gray-to-white or black-to-white pointers exist.

To rule out gray-to-white pointers, `Sweep`'s second precondition requires that no gray objects exist at all:

```
requires (∀T i. memAddr(i) ==> !Gray(Color[i]));
```

The `GcInv` function (see Figure 1) prohibits black-to-white pointers: every black object has fields pointing to non-white objects. (This is known as the tri-color or three color invariant [9].)

The `Mark` procedure's postconditions must satisfy `Sweep`'s preconditions. To ensure that no gray objects exist at the end of the mark phase, `Mark`'s second postcondition says that any non-black object at the end of the mark phase retained its original color from the beginning of the mark phase. For example, any leftover gray objects must have been gray at the beginning of the mark phase. Since no gray objects existed at the beginning, no gray objects exist at the end. `Mark` obeys the ban on black-to-white pointers by coloring an object black *after* its children are black. (Before coloring a node's children, `Mark` temporarily colors the node gray to indicate the node is "in progress"; without this intermediate step, a cycle in the graph would send `Mark` into an infinite loop.)

4.3 Quantifiers and triggers

In the absence of universal and existential quantifiers, many theories are decidable and have practical decision procedures. These include the theory of arrays, the theory of linear arithmetic, the theory of uninterpreted functions, and the combination of these theories. Unfortunately, adding quantifiers makes the theories either undecidable or very slow to decide: the combination of linear arithmetic and arrays, for example, is undecidable in the presence of quantifiers. This forces verification to rely on heuristics for instantiating quantifiers. The choice of heuristics determines the success of the verification.

Many automated theorem provers, including Z3, use programmer-supplied *triggers* to guide quantifier instantiation. Consider again `Sweep`'s precondition prohibiting gray objects. Here are two ways to write this in BoogiePL syntax, each with a different trigger:

```
forall i::{memAddr(i)}memAddr(i)==>!Gray(Color[i]))
forall i::{Color[i]}  memAddr(i)==>!Gray(Color[i]))
```

Both have the same logical meaning, but use different instantiation strategies. The first asks `i` to be instantiated with expression `e` whenever an expression `memAddr(e)` appears during an attempt to prove a theorem. The second asks `i` to be instantiated with `e` whenever `Color[e]` appears. Selecting appropriate triggers is challenging in general. With an overly selective trigger, a quantified formula may never get instantiated, leaving a theorem unproved. With an overly liberal trigger, a quantified formula may be instantiated too often (even infinitely often), drowning the theorem prover in unwanted information.

Shaz Qadeer suggested that we look at formulas of form `forall i::{f(i)}f(i) ==> P`, using `f(i)` as a trigger. For example, we could use `memAddr(i)` as a trigger, although this appears in so many places that it would be easy to accidentally introduce an infinite instantiation loop. (The appearance of `memAddr(ptr)` inside the `Pointer` function, which in turn appears in the `ObjInv` function, which in turn appears in the `GcInv` function, is one example of such a loop.) To avoid accidental loops, we introduce a function `T(i:int)`, defined to be true for all `i`, solely for use as a trigger, writing the invariants above as:

```
forall i::{T(i)}T(i)==>memAddr(i)==>!Gray(Color[i]))
```

(Note that the `==>` operator is right associative.) For conciseness, we abbreviate `forall i::{T(i)}T(i)==>` as $\forall^T$`i`. To avoid instantiation loops, we never write a formula of the form

$\forall^T$i.(...T(e)...), where e is some expression other than a simple quantified variable.

Based on the trigger T(i), we use two strategies to ensure sufficient instantiation of quantified formulas. First, we write explicit assertions of T(e) for various expressions e that appear in the program. This helps Z3 prove formulas ($\forall^T$i.P(i))==>P(e). For example, the ReadField procedure explicitly asserts T(ptr) to instantiate the quantifiers in MutatorInv at the value ptr.

Second, we use the trigger T(i) to prove formulas of the form ($\forall^T$i.P(i))==>($\forall^T$j.Q(j)). In this case, since T appears in both quantifiers, Z3 automatically instantiates P at i=j to prove Q(j). This second strategy isn't sufficient for all P and Q; for example, knowing $\forall^T$i.a[i + 5] == 0 does not prove $\forall^T$j.a[j + 6] == 0, even though mathematically, both these formulas are equivalent. Nevertheless, this strategy works well for purely local reasoning. For example, Sweep's loop invariant maintains the property $\forall^T$i.memAddr(i)==>!Gray(Color[i]). If the loop updates Color by changing Color[ptr] to 1 (white), then the theorem prover attempts to prove:

```
    (memAddr(i)==>!Gray(Color[i]))
==> (memAddr(i)==>!Gray(Color'[i]))
```

where Color' == Color[ptr := 1]. In the case where i != ptr, Color[i] == Color'[i] and the proof is trivial. In the case where i == ptr, !Gray(Color'[i]) == !Gray(1) == true. The proof is easy because the formula memAddr(i) ==> !Gray(Color[i]) is entirely local; it depends only on array elements at index i.

Many formulas depend on non-local array elements, though. Consider how Mark maintains this piece of the tri-color invariant (no black-to-white pointers) from GcInv in Figure 1:

```
Black(Color[i]) ==> !White(Color[Mem[i,0]])
```

This depends not only on i's color, but on the color of some other node Mem[i,0]. For non-local formulas, the local instantiation strategy suffices for some programs but not for others. For example, it suffices for the collector in Figures 1-5 (we invite the reader to write out the verification conditions by hand to see), but did not suffice for an analogous copying collector that we wrote (it did not sufficiently instantiate information about objects pointed to by forwarding pointers).

5. Regions

A mark-sweep collector appears easier to verify than a copying collector, because the mark-sweep collector doesn't modify pointers inside objects. As the previous section mentioned, the mark-sweep collector in Figures 1-5 passed verification even with a very simple triggering strategy, while the analogous copying collector did not. Therefore, this section augments the two strategies described in the previous section with a third instantiation strategy, based on *regions*. Together, these three strategies were sufficient for both mark-sweep and copying collectors.

Regions have proven useful for verifying the type safety of copying collectors [27, 21], which suggests that they might also help verify the *correctness* of copying collectors. Type systems for regions are similar to the verification presented in section 4: section 4's verification mapped concrete addresses to abstract nodes, while type systems type-check a region by mapping concrete addresses in the region to types (e.g., a type system with types Parent and Child might map Figure 2's C1 to Parent and C2 and C3 to Child). This suggests a strategy for importing regions (and the ease of verifying copying collectors via regions) from type systems: rather than defining just one concrete-to-abstract mapping $toAbs, allow multiple regions, where each region is an independent concrete-to-abstract mapping.

For example, consider how Figure 1's object invariant uses $toAbs:

```
ObjInv(i,$toAbs,$AbsMem,Mem) =
  $toAbs[i] != NO_ABS ==>
     Pointer($toAbs, Mem[i,0], $AbsMem[$toAbs[i],0])
     ...
```

Expanding the Pointer function exposes a non-local invariant:

```
  $toAbs[i] != NO_ABS ==>
     ... $toAbs[Mem[i,0]] != NO_ABS ...
```

This invariant is crucial; as discussed in section 4, it ensures that no dangling pointers exist. However, it's not obvious how to prove that this invariant is maintained when $toAbs[Mem[i,0]] changes. Therefore, the remainder of this paper adopts a region-based object invariant:

```
ObjInv(i,$rs,$rt,$toAbs,$AbsMem,Mem) =
  $rs[i] != NO_ABS ==>
     Pointer($rt, Mem[i,0], $AbsMem[$toAbs[i],0])
 ...
```

This object invariant describes an object living in a source region $rs, whose fields point to some target region $rt. Expanding the Pointer function yields:

```
  $rs[i] != NO_ABS ==>
     ... $rt[Mem[i,0]] != NO_ABS ...
```

Now we adopt another idea from region-based type systems: regions only grow over time, and are then deallocated all at once; deallocating a single object from a region is not allowed. In our setting, this means that for any address j and region $r, $r[j] may change monotonically from NO_ABS to some particular abstract node, but thereafter $r[j] is fixed at that abstract node. The function RExtend expresses this restriction; the memory manager only changes $r to some new $r' if RExtend($r,$r') holds:

```
fun RExtend($r:[int]int,$r':[int]int) {
  (forall i::{$r[i]}{$r'[i]}
    $r[i] != NO_ABS ==> $r[i] == $r'[i])
}
```

RExtend's quantifier is *not* based on T; instead, it can trigger on either $r[i] or $r'[i]. (Note that RExtend introduces no instantiation loops, because it only mentions r and r' at index i, and does not mention T at all.) In combination with the second strategy from section 4, this triggering allows Z3 to prove formulas of the form ($\forall^T$i.P(r[e]))==>($\forall^T$i.P(r'[e])), where e depends on i. For example, given the guarantee that RExtend($rt,$rt'), the object invariant ensures that if $rt[Mem[i,0]] != NO_ABS, then $rt'[Mem[i,0]] != NO_ABS.

Given this region-based object invariant, a memory manager can express all other invariants about node i as purely local invariants. For example, our region-based mark-sweep collector relates i's color to i's region state using purely local reasoning:

```
   (White(Color[i]) ==>
        $r1[i] != NO_ABS && $r2[i] == NO_ABS
     && ObjInv(i,$r1,$r1,$toAbs,$AbsMem,Mem))
&& (Gray(Color[i]) ==>
        $r1[i] != NO_ABS && $r2[i] != NO_ABS
     && $r1[i] == $r2[i]
     && ObjInv(i,$r1,$r1,$toAbs,$AbsMem,Mem))
&& (Black(Color[i]) ==>
        $r1[i] != NO_ABS && $r2[i] != NO_ABS
     && $r1[i] == $r2[i]
     && ObjInv(i,$r2,$r2,$toAbs,$AbsMem,Mem))
```

If i is black, then ObjInv(i,$r2,$r2,...) ensures that i's fields point to members of region $r2. Members of $r2 cannot be white, since the invariant above forces white nodes to *not* be members of $r2. Thus, the invariant indirectly expresses the standard tri-color invariant (no black-to-white pointers), and the collector need not state the tri-color invariant directly.

We briefly sketch the region lifetimes during a mark-sweep garbage collection. The collector's mark phase begins with $r1 equal to $toAbs and $r2 empty (i.e. $r2 maps all nodes to NO_ABS). At the beginning of the mark phase, all allocated objects are white, so the invariant above needs ObjInv(i,$r1,$r1,...), and requires that no objects be members of $r2. As the mark phase marks each reached node i gray, it adds i to $r2, so that $r2[i] != NO_ABS. At the end of the mark phase, $r2 contains exactly the reached objects, while $r1 and $toAbs are the same as at the beginning of the mark phase. The sweep phase then removes unreached objects from $toAbs until $toAbs == $r2; Sweep leaves $r1 and $r2 unmodified. After sweeping, the mutator takes an action analogous to "deallocating" region $r1: it simply forgets about $r1, throwing out all invariants relating to $r1 and keeping only the invariants for $r2. In the next collection cycle, $r2 becomes the new $r1, and the process repeats.

6. Practical verified collectors

This section applies the previous section's region-based verification to realistic copying and mark-sweep collectors, replacing the naive recursive algorithm of Figures 1-5 with more efficient iterative algorithms in subsections 6.1 and 6.2, then replacing high-level language constructs with assembly language in subsection 6.3, and then replacing the miniature 2-field, 1-root memory model with a Bartok-compatible memory model in subsection 6.4. If sections 1-5 were the inspiration, this section is the perspiration; the code for the realistic collectors is far longer than Figures 1-5, but not fundamentally much more interesting. We present only short description and selected highlights of invariants; the reader can find the full code and complete invariants in the public release.

6.1 A copying collector

The copying collector is a standard two-space Cheney-queue collector [7]. The heap consists of two equally sized spaces. At any given time, one of the spaces is called *from-space* and the other is called *to-space*. The mutator allocates objects in from-space until from-space fills up. Then the collector traverses all from-space objects reachable from the root pointer, and copies these objects into to-space. (All objects left in from-space are garbage, and are simply ignored by the mutator and collector.) From-space becomes to-space, to-space becomes from-space, and control returns to the mutator.

For each object copied to to-space, the collector sets a *forwarding pointer* that points from the old from-space object to the new to-space copy. This provides a means to find the copied version of each object in to-space and ensures that the collector doesn't copy the same object twice.

When the collector copies an object to to-space, the fields of the copied object initially point back to from-space. The collector later fixes up the pointers to point to to-space, by either copying the referent into to-space, or using the forwarding pointer of an already-copied object. The set of objects not yet fixed form a contiguous work area in to-space. The collection algorithm (shown in Figure 6) treats this work area as a queue, adding newly copied objects to the back of the queue, and fixing objects from the front of the queue. When the queue is empty, all objects are fixed, and the collection is done.

The real collector stores the forwarding pointer in the header field of a from-space object after the from-space object is copied, overwriting the vtable (virtual method table) pointer in the header. (The collector can distinguish a vtable pointer from a forwarding pointer, because vtables do not live in to-space.)

```
while (QFront < QBack)
  for each pointer field f of object QFront
    target := Mem[QFront,f];
    if(FwdPtr[target] != null)
      // target object already copied
      Mem[QFront,f] := FwdPtr[target];
    else
      // copy target object to QBack
      Mem[QBack,0] := Mem[target,0];
      Mem[QBack,1] := Mem[target,1];
      $toAbs[QBack] := $r1[target];
      $r2[QBack] := $r1[target];
      $toAbs[target] := NO_ABS;
      FwdPtr[target] := QBack;
      Mem[QFront,f] := QBack;
      QBack++;
  QFront++;
```

Figure 6. *Copying collector pseudo-code (including ghost variable updates)*

The copying collector shares the same region-based ObjInv from section 5. Other invariants differ from the mark-sweep collector, though. For example, the copying collector has no colors, so there is no invariant to relate colors to regions. There are invariants that relate the forwarding pointer to regions, though. For example, each object i in from-space satisfies this invariant, which ensures that only unforwarded objects are present in $toAbs, and that any forwarding pointer points to a resident of $r2:

```
   (FwdPtr[i]!=null <==> $toAbs[i]==NO_ABS)
&& (FwdPtr[i]!=null ==> Pointer($r2,FwdPtr[i],$r1[i]))
```

The region $r2 is empty at the start of the collection. The collector adds each object that it creates in to-space to $r2, while leaving $r1 unchanged. The collector also updates $toAbs to point to to-space objects rather than from-space objects; at the end, the collector assigns $r2 to $toAbs, and throws out all invariants related to $r1.

During the collection, each fixed object in to-space points from region $r2 to region $r2:

```
ObjInv(i,$r2,$r2,$toAbs,$AbsMem,Mem)
```

Each object still in the to-space queue points from region $r2 back to region $r1:

```
ObjInv(i,$r2,$r1,$toAbs,$AbsMem,Mem)
```

6.2 A mark-sweep collector

Our verified mark-sweep collector uses the standard 3-color invariant. In the beginning of the collection all objects are white and the goal is to mark black all objects reachable from the roots. After this marking process, the sweep process can go over the objects to reclaim all white objects and to mark all black objects white in preparation for the next collection. In the beginning of the collection all objects directly reachable from the roots are put into a list denoted the *mark-stack*. All objects in this list are colored gray, meaning that they have been reached, but their descendants have not yet been traversed. After the roots have been scanned, the collector proceeds by iteratively choosing a gray object O from the mark-stack, inserting O's direct descendants into the mark-stack and marking O black. The black color signifies that the object is reachable and all its direct descendants have been noticed (i.e., put in the mark-stack). The *unallocated* color labels free space.

Keeping the object color requires two bits per object. The colors can be kept in the object header or in a separate table. Following previous work (e.g., [11, 1, 16]) we have chosen the latter. Bartok assumes that objects are 4-byte aligned. Therefore, it is enough to keep two color bits per 4 bytes (creating a space overhead of 6%).

The algorithm follows a very simple collection scheme. One could choose a simpler scheme for verification, for example, by giving up the mark-stack and searching the heap for gray objects, or employing recursion. One could also complicate the scheme and make it more efficient, for example, by using bit-wise sweep. However, we attempted to find the middle way between simplicity and efficiency, in order to enable verification while maintaining the practicability of the collector.

6.2.1 The allocator

A major performance consideration is the allocator. Therefore, we paid special attention to making the allocator efficient, cache-friendly, scalable, and simple. We chose the local allocation cache method that was first invented and used with the IBM JVM allocator [5] and later employed and explained in [2, 16]. This method provides efficiency by allowing bump-pointer allocation with a mark-sweep collection. The mutator holds a local cache in which it allocates small objects by simply bumping a pointer. When the space in the cache is exhausted, the mutator acquires a new local cache from the first chunk in the free list. If that chunk is too large (larger than some threshold maxCacheSize), then only maxCacheSize bytes of the first chunk are taken for the local cache, and the rest is left for future cache allocations. Allocation of large objects use the free list directly; however, since most allocated objects in typical programs are small, most allocation work is efficient. Furthermore, these allocations are cache-friendly since the spatial order of allocated objects in the memory matches the temporal order in which the program allocates them.

Since the mutator only acquires objects or spaces of substantial size from the free list, there is no need to keep small chunks in it. Thus, sweep only fills the free-list with large enough spaces; in our implementation the minimum cache size was set to 256 bytes and objects of size 192 or up are considered large (and are thus directly allocated from the free list).

The mark-sweep collector invariants follow the region-based approach of section 5, sharing the definition of `Pointer` and `ObjInv` with the copying collector. Unlike earlier sections, though, this mark-sweep collector has a free list with non-trivial structure. We use two ghost variables, `$fs` and `$fn` to represent the size of each free list entry and the next-list-entry pointer in each free list entry. Any address i where `$fs[i] != 0` holds a free list entry. Each free list entry must be at least 8 bytes: 4 bytes to store the next pointer, and 4 bytes to store the size. The central invariant ensures, among other things, that the space occupied by each free list entry does not overlap with any object or any other free list entry:

```
$fs[i] != 0 ==>
    $toAbs[i] == NO_ABS
 && i + 8 <= i + $fs[i] && i + $fs[i] <= memHi
 && (∀^T j. i < j && j < i + $fs[i] ==>
       $toAbs[j] == NO_ABS && $fs[j] == 0)
 && ...
```

6.3 From BoogiePL to x86

So far, this paper has expressed all memory management code in BoogiePL or in pseudocode, neither of which were designed to execute on real computers. We decided to write our real copying and mark-sweep collectors (and allocators) in x86 assembly language, for two reasons. First, we didn't want a high-level language compiler in our trusted computing base. Second, the mutator-to-allocator interface requires some assembly language to read the stack pointer, so that the collectors can scan the roots on the stack. We still wanted to use Boogie to verify our code, so this left us with a choice: translate annotated x86 into BoogiePL, or translate BoogiePL into x86. The former approach is the most common way to use BoogiePL, but we chose the latter approach, for the following reason. Since the garbage collectors are written in BoogiePL, the Boogie and Z3 tools guarantee that we really have verified the collectors, at least in BoogiePL form, even if our BoogiePL-to-x86 translation subsequently turns the verified BoogiePL into erroneous x86 code. (If we had translated x86 to BoogiePL, we would have had to ask the reader to trust that our translator didn't just produce a trivially verifiable BoogiePL program.)

We wrote a small tool to automatically translate an x86-like subset of BoogiePL into MASM-compatible x86 code, which we then assemble and link with Bartok-compiled benchmarks. The x86-like subset of BoogiePL consists of top-level variable declarations, non-recursive pure function declarations, and non-recursive procedure declarations. Each procedure is either a macro that gets inline-expanded, or a run-time procedure called with the x86 CALL instruction. The tool enforces matching CALL and RETURN instructions; the BoogiePL code may read the stack pointer at any time, but may not write it. Each procedure consists of local variable declarations followed by a sequence of statements. Since there's no recursion, local variables are statically allocated, as in early FORTRAN. Global and local variables may be ghost variables, of any type, or physical variables, of type `int`. The predefined global variables `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp`, and `esp`, all of type `int`, represent the x86 registers. We maintain the invariant that all registers, physical variables, and words in memory hold an integer in the range $0..2^{32}-1$ at all time.

Each statement in a procedure is a label (used as a jump or branch target), an assignment to a ghost variable (ignored by the translation), an assignment to a register or physical variable, a procedure call, or a control statement. Control statements are either unconditional jumps ("`goto label;`") or conditional branches:

```
if(operand1 cmp operand2) { goto label; }
```

where `operand1` and `operand2` are registers, physical variables, or integer constants, and `cmp` is a comparison operator. Most statements are translated into single x86 instructions, but conditional branches translate into 2 x86 instructions (a compare and a branch). A procedure call either translates into an inline expansion of the called procedure, or a single x86 CALL instruction.

Each assignment statement is either a simple move operation "`operand1 := operand2;`", an arithmetic operation, or a memory operation. Arithmetic operations can either statically check for 32-bit integer overflow, or check at run-time. For example, the statement "`call eax := Sub(eax, 5);`" statically verifies that `eax - 5` does not overflow, because of the (tool-supplied) specification of `Sub` (where `word(e)` means that $0 \leq e < 2^{32}$):

```
procedure Sub(x:int, y:int) returns(ret:int);
  requires word(x - y);
  ensures  ret == x - y;
```

The program is not allowed to modify predefined global variables, like `Mem`, directly. To read or write memory, the program must call tool-supplied `Load` and `Store` procedures, which the tool translates into x86 MOV instructions. The preconditions for `Load` and `Store` guarantee that the verified code does not read or write outside its allowed memory area, and that all reads and writes are to 4-byte aligned addresses. In contrast to the two-dimensional memory `Mem[objAddress,field]` presented earlier, `Load` and `Store` work with a one-dimensional memory `Mem[byteAddress]`.

6.4 The Bartok memory model

Our verified garbage collectors form a critical piece of our long-term goal: an entire verified run-time system for Bartok-compiled code. Because the existing Bartok run-time system contains over 70,000 lines of code, we decided to take an incremental approach towards creating a verified run-time system, starting with as small a run-time system as possible, so as to make the verification as easy as possible. We still wanted to be able to run real Bartok-compiled benchmarks, though, and these benchmarks rely on many non-trivial run-time system features. So before attempting to verify any run-time system code, we examined the 12 large benchmarks used in previous papers [6, 24] to see which features could be evicted from the run-time system. We found that we could remove two major features, while still supporting 10 of the 12 benchmarks:

- Only one benchmark (SpecJBB) was multithreaded, so we omitted support for multithreading from our run-time system.
- Only one of the remaining benchmarks (mandelform) relied on GC support for unsafe code, such as pinning objects (to cast GC-managed pointers to unmanaged pointers for unsafe code) and handling callbacks from unsafe code to safe code. Our verified GC simply halts any program that tries to use these features.

This still left a moderately large set of features to support:

- Objects have a header word, pointing to a virtual method table (vtable). Before the header word, there is a "pre-header" that holds a hash code or other primitive value.
- Non-indexed object types can have any number of fields. Indexed object types can be strings, single-dimensional arrays, or multi-dimensional arrays, each having a different memory layout. Array element types can be pointers, primitive values, structs without pointers, or structs with pointers. We implemented only partial support for arrays of structs with pointers, since the 10 benchmarks did not rely on full support.
- Pointers point to an object's header word, with one exception: root pointers may be *interior pointers* that point to data inside an object, ranging from the header word up to, and including, the address of the end of the object.
- An object's virtual method table has fields that the collector can read to compute the length of an object and to determine which fields of an object are pointers. Bartok's pointer-tracking representation consists of 2 compact bit-level formats for non-indexed objects, 1 non-compact format for non-indexed objects, 1 format for strings, 2 formats for single-dimensional arrays, and 2 formats for multi-dimensional arrays. Our collectors support all of these (except for some arrays of structs with pointers).
- Roots may live on the stack or in static data segments. Each static data segment has a bitmap, with one bit per static data word, indicating pointers and non-pointers in the segment. Finding pointers on the stack is more complicated; the collector has to traverse frame pointers to find the stack frames, and it has to look up return addresses in a sorted table of return addresses to find a descriptor for each frame. To simplify finding pointers, we set a compiler flag telling Bartok to treat all registers as caller-save registers, with no callee-save registers.

Space constraints preclude a complete, detailed description of the specification and collector implementation of the features above. We provide just one example. One of the compact pointer-tracking formats is a dense format, using one bit per field. The specification for this says that if the tag of an object for abstract node `$abs`, with vtable `vt`, is `DENSE_TAG`, then each field is a pointer if and only if the corresponding bit in the vtable's mask field is `1`:

```
tag(vt)==DENSE_TAG ==> (∀^T j.2<=j<numFields($abs)==>
  VFieldPtr($abs,j)==(j<30 && getBit(mask(vt),2+j)))
```

where `mask` looks up a 32-bit value from the vtable (in readonly memory), and `tag` and `getBit` extract bits from words:

```
fun mask(vt:int) { ro32(vt + ?VT_MASK) }
fun tag(vt:int) { and(mask(vt), 15) }
fun getBit(x:int,i:int) { 1 == and(shr(x, i), 1) }
```

The mutator-allocator interface specification uses the uninterpreted function `VFieldPtr` to state which physical values are primitive values, and which are pointer values. The `Value` function states the meaning of values in each of these two cases:

```
fun Value(isPtr,$r,v,$abs) {
   (isPtr && word(v) && gcAddrEx(v) && !word($abs)
                     && Pointer($r, v - 4, $abs))
|| (isPtr && word(v) && !gcAddrEx(v) && $abs == v)
|| (!isPtr && word(v) && $abs == v)
}
```

For primitive data, the data's abstract value equals its concrete value. Pointer data may point to GC memory, under the collector's control, or they may point outside GC memory, in which case the collector treats them the same as primitive values. The "`- 4`" in the Pointer specification converts a pointer to a header word into a pointer to the beginning of the object (the pre-header).

Interior pointers are defined like the ordinary pointers shown above, but may have offsets larger than 4, which forces the collector to search for the beginning of the object. The mark-sweep collector already has a table of colors, so it simply searches backwards from the interior pointer to find the first word whose color isn't `unallocated`. We also had to add an analogous bit map to the copying collector, with one bit per heap word, solely for the purpose of handling interior pointers. (On the bright side, these bit maps did give us a chance to exercise Z3's bit vector support.)

Before we added support for Bartok's memory model, the trusted mutator-allocator specification was fairly short and readable. After adding Bartok's memory model, the specification ballooned to hundreds of lines of bit-level details. At this point, we started to wonder if the specification itself had bugs. We used two techniques to test the specification. First, we used Boogie's "smoke" feature, which attempts to prove false at various points in the program. This did not turn up any bugs. Second, we hand-translated the specification into C# code, and then added run-time assertions to the original Bartok garbage collectors based on this C# code. We saw many assertion violations, which led us to 5 specification bugs, ranging from mundane (forgetting to multiply by 4 to convert a word address to a byte address) to subtle (forgetting that Bartok compresses the sorted return address tables by omitting any entry whose descriptor is identical to the previous entry).

7. Performance

This section presents performance results, measured on a single core of a 4-core, 2.4GHz Intel Core2 with 4GB of RAM, 4MB of L2 cache, and a 64KB L1 cache.

Verifying the copying collector, mark-sweep collector, and the code shared between the collectors took 115 seconds, 70 seconds, and 12 seconds, respectively (see Table 1). This fast verification reflects our choice of a simple trigger `T(i)`. The copying collector and mark-sweep collector contained 802 x86 instructions (before inlining) and 865 x86 instructions (before inlining), plus 177 x86 instructions (before inlining) shared between the collec-

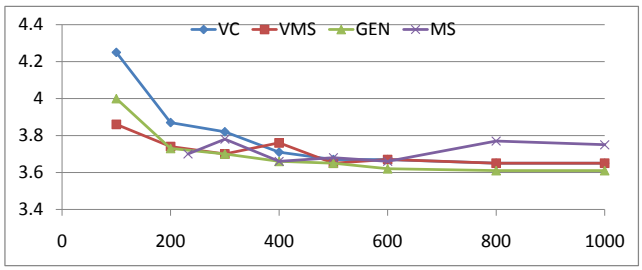

Figure 7. *Othello Performance Comparison: overall running time (in seconds) versus heap size(in KB).*

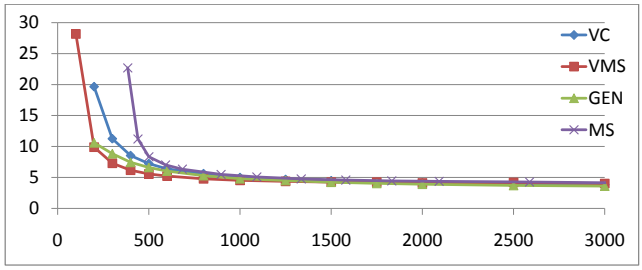

Figure 8. *Ahc Performance Comparison: overall running time (in seconds) versus heap size(in KB).*

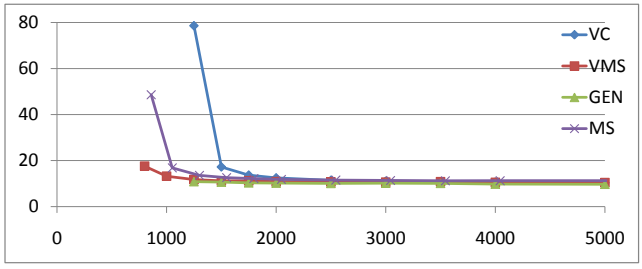

Figure 9. *Go Performance Comparison: overall running time (in seconds) versus heap size(in KB).*

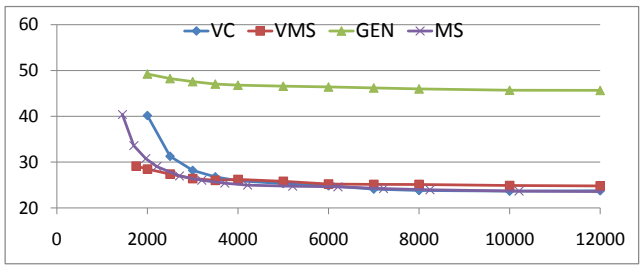

Figure 10. *Xlisp Performance Comparison: overall running time (in seconds) versus heap size(in KB).*

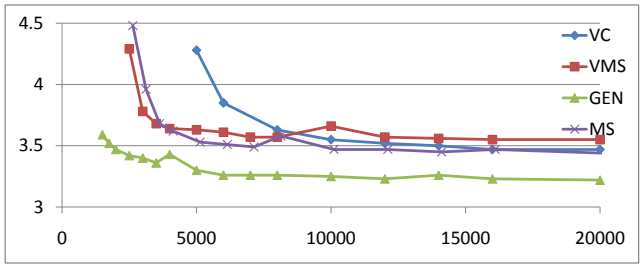

Figure 11. *Crafty Performance Comparison: overall running time (in seconds) versus heap size(in KB).*

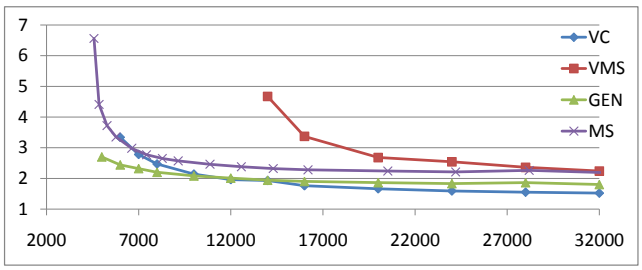

Figure 12. *Zinger Performance Comparison: overall running time (in seconds) versus heap size(in KB).*

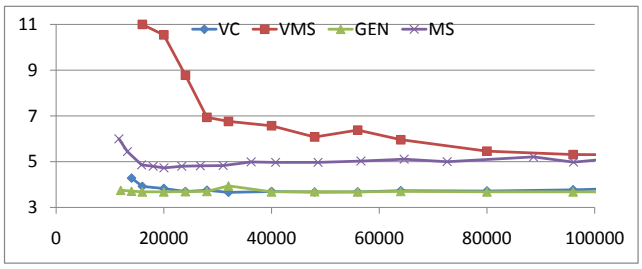

Figure 13. *Sat Performance Comparison: overall running time (in seconds) versus heap size(in KB).*

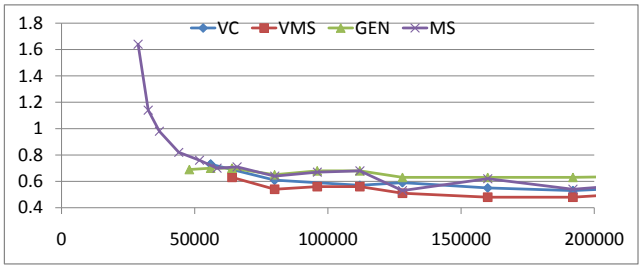

Figure 14. *Asmlc Performance Comparison: overall running time (in seconds) versus heap size(in KB).*

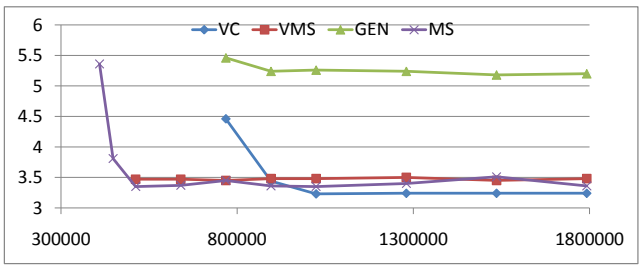

Figure 15. *Lcsc Performance Comparison: overall running time (in seconds) versus heap size(in KB).*

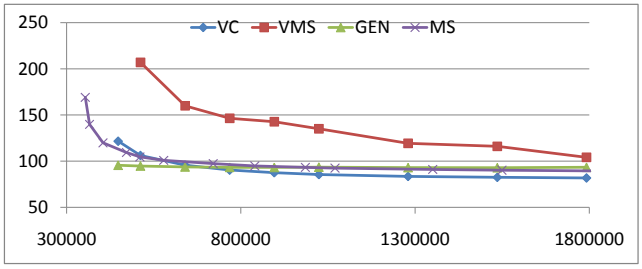

Figure 16. *Bartok Performance Comparison: overall running time (in seconds) versus heap size(in KB).*

	BoogiePL code (non-comment, non-blank lines)	x86 instructions (before inlining)	Time to verify (seconds)
Trusted defs	546		
Shared code	779	177	12
Copying	2398	802	115
Mark-sweep	3038	865	70

Table 1. Verification times for practical collectors

tors. The BoogiePL files for the copying and mark-sweep collectors contained 2398 non-comment, non-blank lines and 3038 non-comment, non-blank lines, plus 779 non-comment, non-blank lines of BoogiePL code shared between the collectors. The trusted definitions, including x86 instruction specifications and the Bartok interface specification, occupied 546 non-blank, non-comment lines.

Figures 7-16 show the performance of the 10 benchmarks described in section 6 as a function of heap size, both for our verified memory managers and for Bartok's native run-time system. We denote the verified copying collector by VC, the verified mark-sweep collector by VMS, the generational copying Bartok collector by GEN, and the Bartok standard mark-sweep collector by MS. These results demonstrate that (a) our collectors work on real benchmarks, and (b) the space and time consumption is in the same ballpark as Bartok's native run-time system. We emphasize the "ballpark" nature of the comparison between the verified collectors and the native Bartok collectors, because this comparison is highly unfair to the native collectors, which support more features than the verified collectors. In particular, Bartok's native run-time system supports multithreading, which adds synchronization overhead to the mutator and memory manager.

Bartok's native collectors were not designed to be used with a fixed heap size; they expect to grow the heap as needed. To get a time vs. space plot for the Bartok collectors, we varied the triggering mechanism used for heap growth, and then measured the actual heap space used. For the generational collector, we set the nursery size to 4MB or 1/4 of the maximum heap size, whichever was smaller.

Several benchmarks created fragmentation that made it difficult for the verified mark-sweep collector to find space for very large objects. The standard Bartok mark-sweep collector simply grows the heap when the current heap lacks space for a very large object; we configured the verified mark-sweep collector to set aside part of the heap as a wilderness area, used as a last resort for very large object allocation. While this wilderness area enabled these benchmarks to keep running under heavy fragmentation, the performance still suffered. For other benchmarks, though, the verified mark-sweep collector performed well across a large spectrum of heap sizes. The verified copying collector, as expected, required a larger minimum heap size, but performed asymptotically well as the heap size increased.

8. Conclusion

We have presented two simple collectors with the minimal set of properties required to make them reasonably efficient in a practical setting. We have mechanically verified that these collectors maintain a heap representation that is faithful to a mutator-defined abstract heap, and have run the collector on large, off-the-shelf C# benchmarks.

Given the large size of the mutator-allocator specification, we were very curious to see whether our collectors would run correctly the first time. Alas, running the verified copying collector revealed two specification bugs that we hadn't caught before: `Initialize`'s postcondition forgot to ensure that the `ebp` register was saved, and the allocation postcondition specified a return value that was off by 4 bytes (a header/pre-header confusion). Thus, the copying collector ran correctly the *third* time we tried it, which is still no small achievement for a garbage collector hand-coded in assembly language. Furthermore, we were then able to verify the mark-sweep collector against the debugged specification, so that the mark-sweep collector ran correctly the first time we tried it. In addition, having a clear and well-tested specification is useful for TAL/PCC verifiers: based on the specification, we found a bug in our TAL verifier [6], which didn't check that the sparse pointer tracking formats mention no field more than once; this bug can allow TAL code to crash when linked to Bartok's native sliding/compacting collector.

The fast verification times give us hope that there is still room to grow to support more features and better GC algorithms. In particular, multithreading and pinning are essential for many applications and libraries. Pinning should be easy for the mark-sweep collector, but would complicate the copying collector: pinned objects fragment the heap, forcing the allocator to allocate from a non-contiguous free space. Multithreading would require reasoning about mutual exclusion (e.g. to keep allocators in different threads from allocating the same memory simultaneously), reasoning about suspending mutator threads during collection, and support for a more elaborate object pre-header word (for monitor operations on objects). As the collectors grow, modularity becomes more important, so we're interested to see if the Boogie/Z3 approach can be combined with modular verification approaches based on separation logic and/or higher-order logic; hopefully, the automation provided by Boogie/Z3 will allow verification at a scale where modularity becomes essential.

Acknowledgments

The authors would like to thank Nikolaj Bjørner, Shaz Qadeer, Shuvendu Lahiri, Bjarne Steensgaard, Jeremy Condit, Juan Chen, Zhaozhong Ni, and the anonymous reviewers for many helpful discussions, suggestions, and comments.

References

[1] Hezi Azatchi, Yossi Levanoni, Harel Paz, and Erez Petrank. An on-the-fly mark and sweep garbage collector based on sliding view. In *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Anaheim, CA, November 2003. ACM Press.

[2] Katherine Barabash, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, Yoav Ossia, Avi Owshanko, and Erez Petrank. A parallel, incremental, mostly concurrent garbage collector for servers. *ACM Transactions on Programming Languages and Systems*, 27(6):1097–1146, November 2005.

[3] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005.

[4] Lars Birkedal, Noah Torpe-Smith, and John C. Reynolds. Local reasoning about a copying garbage collector. In POPL, pages 220–231, Venice, January 2004. ACM Press.

[5] Sam Borman. Sensible sanitation — understanding the IBM Java garbage collector, part 1: Object allocation. *IBM developerWorks*, August 2002.

[6] Juan Chen, Chris Hawblitzel, Frances Perry, Mike Emmi, Jeremy Condit, Derrick Coetzee, and Polyvios Pratikakis. Type-preserving compilation for large-scale optimizing object-oriented compilers. In PLDI, pages 183–192, Tucson, AZ, June 2008.

[7] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, November 1970.

[8] Leonardo de Moura and Nikolaj Bjorner. Z3: An Efficient SMT Solver. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2008.

[9] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. In *Lecture Notes in Computer Science, No. 46*. Springer-Verlag, New York, 1976.

[10] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In POPL, Portland, OR, January 1994. ACM Press.

[11] Tamar Domani, Elliot Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for Java. In PLDI, Vancouver, June 2000. ACM Press.

[12] Healfdene Goguen, Richard Brooksby, and Rod Burstall. An abstract formulation of memory management, December 1998. draft.

[13] Georges 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.

[14] Klaus Havelund and N. Shankar. A mechanized refinement proof for a garbage collector. Technical report, Aalborg University, 1997. Submitted to Formal Aspects of Computing.

[15] Paul B. Jackson. Verifying a garbage collection algorithm. In *Proceedings of 11th International Conference on Theorem Proving in Higher Order Logics TPHOLs'98*, volume 1479 of *Lecture Notes in Computer Science*, pages 225–244, Canberra, September 1998. Springer-Verlag.

[16] Haim Kermany and Erez Petrank. The Compressor: Concurrent, incremental and parallel compaction. In PLDI, pages 354–363, Ottawa, June 2006. ACM Press.

[17] Yossi Levanoni and Erez Petrank. An on-the-fly reference counting garbage collector for Java. In PLDI, 28(1), January 2006.

[18] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

[19] Andrew McCreight, Zhong Shao, Chunxiao Lin, and Long Li. A general framework for certifying garbage collectors and their mutators. In PLDI, San Diego, CA, June 2007.

[20] Marvin L. Minsky. A Lisp garbage collector algorithm using serial secondary storage. Technical Report Memo 58 (rev.), Project MAC, MIT, Cambridge, MA, December 1963.

[21] Stefan Monnier, Bratin Saha, and Zhong Shao. Principled scavenging. In PLDI, Snowbird, Utah, June 2001.

[22] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In POPL, pages 85–97, January 1998.

[23] George Necula. Proof-Carrying Code. In POPL, pages 106–119. ACM Press, 1997.

[24] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A study of concurrent real-time garbage collectors. In PLDI, pages 33–44, Tucson, AZ, June 2008.

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

[26] Martin Vechev, Eran Yahav, David Bacon, and Noam Rinetzky. CGCExplorer: A semi-automated search procedure for provably correct concurrent collectors. In PLDI, San Diego, CA, June 2007.

[27] Daniel C. Wang and Andrew W. Appel. Type-preserving garbage collectors. In POPL, January 2001.