Int J Software Informatics, Vol.3, No.1, March 2009, pp. 67–88 E-mail: ijsi@iscas.ac.cn
International Journal of Software and Informatics, ISSN 1673-7288 http://www.ijsi.org
 Tel: +86-10-62661040

Verification of an Incremental Garbage Collector in Hoare-Style Logic*

Chunxiao Lin[1,2], Yiyun Chen[1,2], and Bei Hua[1,2]

[1](Department of Computer Science and Technology, University of Science and Technology of China, Anhui 230026, China)

[2](Software Security Laboratory, Suzhou Institute for Advanced Study, University of Science and Technology of China, Jiangsu 215123, China)

Abstract Many of the current software systems rely on garbage collectors for automatic memory management. This is also the case for various software systems in real-time applications. However, a real-time application often requires an incremental working style of the underlying garbage collection, which renders the garbage collector more complex and less trustworthy. We present a formal verification of the Yuasa incremental garbage collector in Hoare-style logic. The specification and proof of the collector are built on a concrete machine model and cover detailed behaviors of the collector which may lead to safety problems but are often ignored in high-level verifications. The work is fully implemented with the Coq proof assistant and can be packed as foundational proof-carrying-code packages. Our work makes an important step toward providing high-assurance garbage collection for mission-critical real-time systems.

Key words: program verification; incremental garbage collector; separation logic; proof-carrying code

Lin CX, Chen YY, Hua B. Verification of an incremental garbage collector in Hoare-style logic. *Int J Software Informatics*, **2009, 3(1): 67–88. http://www.ijsi.org/1673-7288/3/67.htm**

1 Introduction

Many of the current software systems rely on garbage collectors for automatic memory management. This is also the case for various software systems in real-time applications. By using garbage collectors, there is no more need for explicit memory de-allocation in programs, which greatly improves the safety of these programs by reducing the possibility of memory leak and other memory-related bugs.

However, for a garbage-collected software system, its safety depends heavily on the correct implementation of the underlying garbage collector. Bugs in the collector may lead to unexpected memory leak, or even corruption of important user data in

* This work is sponsored by the National Natural Science Foundation of China under Grant Nos.90718026 and 60673126 and Intel China Research Center
Corresponding author: Chunxiao Lin, Email: cxlin3@mail.ustc.edu.cn
Manuscript received 2007-08-06; revised 2008-07-23; accepted 2009-04-07.cc

the system. Unfortunately, garbage collectors often employ very complex algorithms and are hard to be implemented correctly. The situation is even worse for real-time systems, in which the process of garbage collection is interleaved with the execution of the user program to guarantee real-time properties. This kind of incremental garbage collectors must maintain intricate invariants and impose subtle restrictions on the behavior of the user program, in order to prevent garbage collection from being disrupted. Many incremental algorithms[1−7] have been proposed and some were reported to contain bugs[2]. In fact, as we will show in a later section, a large class of incremental algorithms are incorrect to be used directly in real-world settings.

For mission-critical real-time systems, safety and reliability are commonly considered as the most important issues. Thus it is very important to provide high-assurance incremental garbage collection for these systems.

Proof-carrying code (PCC)[8] is a promising technique for building verified software for mission-critical applications. In the PCC style, programs are verified as machine-level implementations. And a verified software system contains not only the executable code, but also a machine-checkable proof saying that the code will run safely. Safety is thus ensured by checking the proof before running the corresponding program. Besides the original proposal for safe distribution of application-level mobile code, PCC-style verification has also been applied to system-level software[9, 10], so that the whole software stack can be linked as verified machine code.

The aim of our work is to verify incremental garbage collectors in the PCC-style. And with the proof-carrying collectors, garbage-collected software systems can then be built as fully verified PCC packages to improve the safety and reliability of mission-critical real-time applications.

In this paper, we present the verification of the Yuasa incremental garbage collector in a Hoare-style PCC framework, the *Stack-based Certified Assembly Programming* (SCAP) system[11] with embedded separation-logic[12] primitives. The verification ensures that the collector always preserves the heap objects reachable by the mutator. Some of the specification constructs follow our previous work on verifying a stop-the-world mark-sweep collector[13]. And our model of mutator-collector interaction, which is the basis of the collector-side verification, is based on the work by McCreight *et al.*[14]. Building on the existing work, this paper makes the following new contributions:

- As far as we know, our work is the first to successfully verify an incremental mark-sweep garbage collector in Hoare-style PCC framework. Though there is extensive work on high-level verification (mostly model checking) of incremental garbage collection algorithms[15−20], none of them is able to explore the low-level behaviors of the collector on the concrete machine model. On the other hand, we verify the collector as directly runnable assembly code, rather than some abstract algorithm. Thus our specification and proof of the collector are built on a concrete machine model and cover many detailed behaviors of the collector which would otherwise be ignored in high-level verifications. This makes our verification more trustworthy and useful for building proof-carrying software system with real-time garbage collection.

- We formalize the Yuasa collector's heap invariant (based on the weak tricolor

invariant[21]) using separation logic, and prove the important properties about this invariant. The heap invariant describes the detailed restrictions on the object heap and structures that are crucial for the correct executions of collector routines. It is well known that an incremental collector on a sequential model corresponds to a concurrent collector on the shared-memory multiprocessor model[7]. So, the formalization of the heap invariant may also be used in the verification of concurrent collectors [1] based on the weak tricolor invariant.

- Our work is fully implemented in the Coq proof assistant[22], and the verified collector can be shipped immediately as foundational-PCC (FPCC)[23] packages. Following the ideas in[24], the verification of the collector can easily be ported into an open-FPCC framework[25] and link with other verifications. Besides, building proofs mechanically makes our verification more rigorous and trustworthy than the paper-and-pencil proofs.

The rest of the paper is organized as follows: in Section 2, we discuss the the basic assumptions of our verification and the general issues of real-time garbage collection. Then, we present the collector we verified in Section 3. In Section 4, we present the formalization of the collector's heap invariant built with the weak tricolor invariant. The specification and proof of the collector are discussed in Section 5, and our Coq implementation is briefly evaluated in Section 6. In Section 7, we present a discussion of the related work as well as a comparison with our past research. Finally, we draw a conclusion in Section 8.

Note that since all the lemmas mentioned in this paper are mechanically proved in Coq, their detailed proofs are skipped here. Readers who are interested in may find them in our Coq implementation[26].

2 Real-Time Garbage Collection

In real-time applications, whenever control is transferred to a garbage collector routine, it must return to the mutator within a bounded time span. That is, each delay imposed by the collector must be shorter than a small constant value. In the traditional stop-the-world collectors, a collector routine has to traverse the whole heap to collect unused objects[7], which may cause an intolerable interruption to the mutator. Similarly, although generational collectors can effectively shorten a garbage collection interrupt in average cases, they cannot guarantee their performance in the worst cases and thus are also not suitable for real-time applications[7]. The only solution is to make garbage collection incrementally. That is, a garbage collection cycle is broken into pieces and every time the collector gets control, it does only a small amount of collection and returns control to the mutator.

During an incremental garbage collection cycle, the mutator operations may break the invariant maintained by the collector and cause the collector to falsely collect the accessible data. Thus, it is important to make clear the model of mutator-collector interaction and the possible set of mutator operations before we dive into the verification of the collector.

[1] However, to verify a real-world concurrent collector, one may have to deal with fine-grained concurrency, weak memory mode and other intricate problems, which makes the task much harder than the verification described here.

2.1 Mutator-Collector interaction model

Our model of mutator-collector interaction follows the one in Ref.[14]. We define the root set of the collection as the pointers stored in a set of registers (root registers). We also adopt a simple accurate setting, where all non-pointer values are separated from object pointers by a test bit. And an object is *accessible*, or *reachable*, in the object heap, only if it can be visited with a chain of heap load operations starting from a pointer in the root set.

The interaction between the mutator and collector is modelled as function call/return. As shown in Fig.1, the mutator calls the corresponding barriers for heap accesses and object allocation, and the barriers make sure that these operations preserve the invariant of the collector. It is also obvious that if the mutator is able to manipulate the root registers arbitrarily (e.g. doing pointer arithmetics), there will be no chance for an incremental collector to work correctly. Thus, we follow Ref.[14] to restrict the mutator operations on root registers exactly as the following:

- root registers can be copied to other root registers;
- non-pointer values can be moved to root registers;
- field values of the heap object pointed to by a root register can be loaded to root registers;
- return value of an allocation is stored to a root register.

Following these restrictions, it is trivially provable that the mutator operations preserve the collector's invariant between each call to the barriers based on the ideas in Ref.[14].

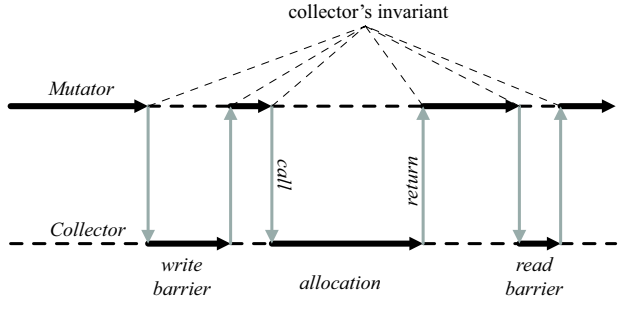

Figure 1. Mutator-Collection interaction

A general framework is proposed in Ref.[14] to verify the mutator in a restricted abstract view as well as the collector in a concrete view of state, and links them together to form a fully verified system. In the rest of this paper, we will mainly focus on the verification of the collector itself with consideration of the mutator-collector model described here. Putting the verified collector into the general framework will not be a hard task based on our previous experience on a stop-the-world mark-sweep collector[14], and we leave this to our future work.

2.2 Incremental collectors

We now discuss the various incremental collectors and explain our choice of verifying the Yuasa collector based on our mutator-collector interaction model.

For an incremental copying collector[5, 6], every heap load instruction in the mutator must be substituted by a read barrier, which imposes great overhead. On the other hand, non-copying mark-sweep collectors[1−4] only require write barriers, and are thus much lightweight compared with the copying collectors[7].

An incremental mark-sweep collector is commonly based on the following tricolor abstraction[2]:

- the allocated objects are in three colors;
- *black* objects are reachable objects with all fields already examined by the collector, and they will not be visited again;
- *gray* objects are reachable objects which are scheduled to be examined;
- the rest objects are in *white* color;
- if there are no more *gray* objects, the *white* objects are unreachable and can be collected.

We show a simple example of the incremental marking process in Fig.2. There are four consecutive states from 1 to 4, each of which contains a set of roots along with heap objects A, B and C. The color of the objects changes during the marking process following the tricolor abstraction.

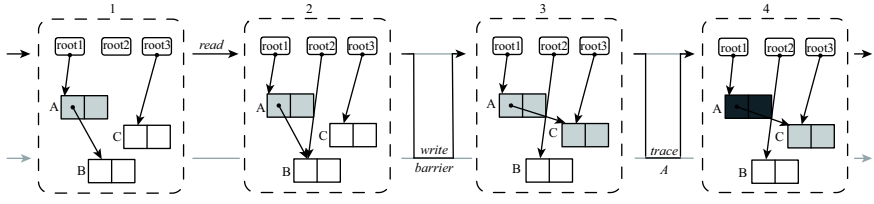

Figure 2. Incremental marking

Typically, a write barrier is called when the mutator intends to update an object field with a value in the register. Besides updating the actual field, the barrier also changes the color of some related objects to ensure that none of the reachable objects would be collected mistakenly[21]. For example, in Fig.2, the write barrier does not only update the first field of object A with the pointer in root3, but also shades the color of C into *gray* in state 3.

Based on the behaviors of their write barriers, incremental mark-sweep collectors are commonly divided into two major classes, namely the *incremental-updating* collectors and the *snapshot-at-beginning* collectors[7]. Take Fig.2 as an example again, the first class of collectors will shade either A or C in state 3, which enables the collectors to log the heap mutation incrementally. On the other hand, a collector in the second class will shade the object B in state 3, which ensures that all the objects reachable at the beginning of a collection cycle will be preserved by the collector.

However, there is a subtle bug of the incremental-updating collectors when they are used in our mutator-collector interaction model. This is also noticed by Ref.[27]. As illustrated in Fig.2, if the link from A is the only link that keeps B reachable in state 1, these series of moves by the mutator and collector will eventually make the collector falsely collect the root reachable object B (a *white* object), no matter whether the barrier shades A or C, since it will not be examined by the collector again and will remain white at the end of the current marking process. In high-level verifications, this problem is ignored with the assumption that the root set is unchanged throughout the collection[2]. But this assumption makes the collector useless in a realistic setting. The implementations of this class of algorithms, on the other hand, often employ a final phase for a stop-the-world tracing from the modified root registers[28], or iterate the marking phase many times[29] to mark the root reachable objects that would otherwise be collected mistakenly.

This defect of the incremental-updating collectors illustrates again the necessity of doing verification on a concrete implementation, instead of just the high-level algorithm. It is also the reason why we choose to verify the snapshot-at-beginning-based Yuasa collector[4]. And as we will show later, the execution of each Yuasa collector interface routine correctly preserves the objects reachable on its entry state.

3 The Collector Verified

We begin with the heap layout used by the collector we verified in Fig.3. We assume for simplicity that the size of each heap object is two words. This implies that our allocator only works with mutators that always require two-word objects. However, it does not affect the computational power of the LISP-style mutator. We also assume that all heap objects (including the free objects) reside in a continuous subheap from ST to ED. And the collector also keeps the mark bits to tell white objects from colored objects, a mark stack for keeping gray objects and a free list of unallocated objects.

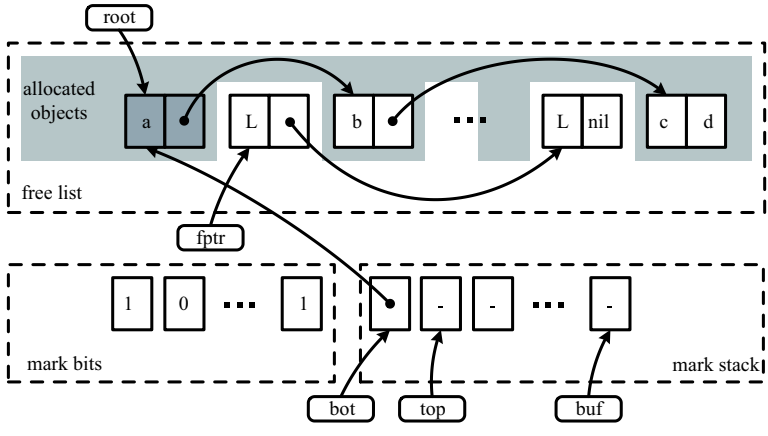

Figure 3. Collector's heap layout

In the original algorithm proposed by Yuasa[4], the user stack is also taken as part of the root set. However, to simplify the problem, our root set contains only

```
/* interface procedures */
alloc() {
  count = 0;
  if (phase == MARK) mark();
  else if (phase == SWEEP) sweep();
  else if (fcount < MIN) {
    mark_field(root1);
    mark_field(root2);
    sptr = ST;
    phase = MARK;
  }
  if (fptr == NULL) inf_loop();
  ret = fptr;
  fptr = fptr->next;
  if (ret >= sptr) markbit(ret) = BLACK;
  return ret;
}

write(des, fld, newval) {
  if (phase == MARK) mark_field(des->fld);
  des->fld = newval;
}
\vspace{5pt} \footnotesize \hspace{1em}
  \begin{minipage}[l]{0.51\linewidth}
\begin{verbatim}
/* internal procedures */
mark_field(val) {
  if (atom(val) || markbit(val) == BLACK)
    return;
  markbit(val) = BLACK;
  stack_push(val);
}
```

```
/* internal procedures */
mark() {
  while (!stack_empty() &&
         count++ < MAX) {
    ptr=stack_pop();
    mark_field(ptr->first);
    mark_field(ptr->second);
  }
  phase = stack_empty() ? SWEEP : MARK;
}

sweep(){
  while (sptr < ED && count++ < MAX)
    if (sptr->first == LEAVEME) return;
    else if (markbit(sptr) == WHITE) {
      sptr->first = fptr;
      fptr=sptr;
      fcount++;
    } else markbit(addr) = WHITE;
  phase = (sptr == ED) ? IDLE : SWEEP;
}

inf_loop() { while (1); }
markbit(x) { return (ED+(x-ST)/2); }
stack_push(ptr) {
  if (top >= buf) inf_loop();
  *(top++)=ptr;
}
stack_pop() { return *(--top); }
stack_empty() { return (bot == top); }
```

Figure 4. Pseudo code of the Yuasa collector

two registers, but it is not hard to extend the system with more root registers. We also follow Ref.[14] to have all the atomic (non-pointer) values 31bit encoded. Thus, `atom()` in Fig.4 is simply implemented as a parity test.

With the assumptions above, we show our version of the pseudo code of the Yuasa collector in Fig.4. The interface of the collector contains the two procedures:

- `alloc()`, performs garbage collection and object allocation.
- `write()`, the write barrier

The collector also keeps global variables to record the collection state and internal data structures, which are:

- `phase`, indicates the status of the collection cycle
- `sptr`, indicates the object that `sweep()` is currently working on
- `fptr` and `fcount`, the head and length of the free list, respectively
- `bot`, `top` and `buf`, the mark stack pointers
- `ST` and `ED`, the heap boundaries

$$
\begin{array}{llll}
(State) & \mathbb{S} & ::= & (\mathbb{H}, \mathbb{R}) \\
(Heap) & \mathbb{H} & ::= & \{\mathtt{l} \rightsquigarrow \mathtt{w}\}^* \\
(RFile) & \mathbb{R} & ::= & \{\mathtt{r} \rightsquigarrow \mathtt{w}\}^* \\
(Reg) & \mathtt{r} & ::= & \{\mathtt{r}k\}^{k \in \{0 \ldots 31\}} \\
(Wd) & \mathtt{w} & ::= & 0 \mid 1 \mid 2 \mid 3 \mid \ldots \\
(Address) & \mathtt{l} & ::= & 0 \mid 4 \mid 8 \mid 12 \mid \ldots \\
(SPred) & \mathrm{p}, \mathrm{q} & \in & State \rightarrow Prop \\
(HPred) & A, B & \in & Heap \rightarrow Prop
\end{array}
$$

Figure 5. Machine state and state assertions

According to the code in Fig.4, `alloc()` does a small piece of collection before each allocation. The collection work differs in different phases of collection. This may be tracing a few reachable objects in `mark()`, when `phase` equals to `MARK`; or sweeping and collecting some white objects into the free list in `sweep()`, when `phase` equals to `SWEEP`; or marking the root set, when `phase` equals to `IDLE` and the free-object count is low. The number of objects being processed at each time is bounded by the constant `MAX` and the size of the root set. Both `mark()` and `write()` calls `mark_field()` to update the color of an object, that is, change it from white to gray. The first field of the objects in the free list contains a special constant `LEAVEME` to keep them from being swept again in `sweep()`.

4 Invariant Formalization

We formalize the invariant of the collector in Section 3 by using Hoare-style state assertions with separation-logic primitives. Before presenting the invariant, we firstly go over some necessary backgrounds for understanding the rest of the paper, which include the meta-logic framework we use, our model of the machine state, as well as separation-logic primitives defined on the state model.

4.1 Backgrounds

The work in the rest of this paper is formalized within a mechanized meta-logic, the *Calculus of inductive Constructions* (CiC)[30]. CiC is a higher-order predicate logic extended with inductive definitions. The CiC terms in this paper are written with standard logic notations. We let *Prop* be the universe of all logical propositions, and let *Set* be the universe of all computational terms.

As shown in Fig.5, we model a machine state $\mathbb{S}$ as a pair of heap $\mathbb{H}$ and register file $\mathbb{R}$. A heap $\mathbb{H}$ is a partial map from address `l` (aligns to 4) to word value `w`. While a register file $\mathbb{R}$ is a map from register `r` to word value. We write $X(z)$ for the value bound to z in the map X, and $X\{z \rightsquigarrow v\}$ for the map obtained by updating the binding of z to v in X. We also write $\mathbb{S}.\mathbb{R}$ for the register file in state $\mathbb{S}$.

We use CiC directly as our assertion language. Following the Hoare-logic style, state predicates (p, q) with the type $State \rightarrow Prop$ are used to assert the program behavior. And separation-logic primitives (A, B) are embedded into the assertion

$$
\begin{array}{rcl}
T & \in & \mathit{Set} \\
\mathtt{l} \mapsto \mathtt{w} & \stackrel{\text{def}}{=} & \lambda\mathbb{H}.\ \mathbb{H} = \{\mathtt{l} \rightsquigarrow \mathtt{w}\} \\
\mathsf{emp} & \stackrel{\text{def}}{=} & \lambda\mathbb{H}.\ \mathsf{dom}(\mathbb{H}) = \emptyset \\
\mathsf{true} & \stackrel{\text{def}}{=} & \lambda\mathbb{H}.\ \mathsf{True} \\
A\ *\ B & \stackrel{\text{def}}{=} & \lambda\mathbb{H}.\ \exists\mathbb{H}_1, \mathbb{H}_2.\mathbb{H}_1 \uplus \mathbb{H}_2 = \mathbb{H}\ \wedge\ (A\ \mathbb{H}_1) \wedge (B\ \mathbb{H}_2) \\
\exists x : T.\ A & \stackrel{\text{def}}{=} & \lambda\mathbb{H}.\ \exists x : T.\ (A\ \mathbb{H}) \\
!(P) & \stackrel{\text{def}}{=} & \lambda\mathbb{H}.\ P \wedge (\mathsf{emp}\ \mathbb{H}) \\
\forall_* x \in \emptyset.\ A & \stackrel{\text{def}}{=} & \mathsf{emp} \\
\forall_* x \in \{n\} \cup S.\ A & \stackrel{\text{def}}{=} & A[n/x]\ *\ \forall_* x \in S - n.\ A
\end{array}
$$

Figure 6. Separation logic primitives

$$
\begin{array}{rcl}
\mathsf{null} & ::= & 0 \\
\mathsf{st}, \mathsf{ed} & ::= & 8 \mid 16 \mid 24 \mid \ldots \\
\mathsf{ptrs} & \stackrel{\text{def}}{=} & \{\mathtt{l} \mid (\mathtt{l} \bmod 8 = 0) \wedge (\mathsf{st} \leqslant \mathtt{l} < \mathsf{ed})\} \\
\mathsf{vptr}(\mathtt{l}) & \stackrel{\text{def}}{=} & \mathtt{l} \in \mathsf{ptrs} \\
\mathsf{rchrt}((\mathbb{H}, \mathbb{R}), \mathtt{l}) & \stackrel{\text{def}}{=} & \exists \mathtt{r} \in \{\mathtt{r1}, \mathtt{r2}\}.\ \mathsf{reach}(\mathbb{H}, \mathbb{R}(\mathtt{r}), \mathtt{l})
\end{array}
$$

$$\frac{\mathsf{vptr}(\mathtt{l})}{\mathsf{reach}(\mathbb{H}, \mathtt{l}, \mathtt{l})} \quad (\textsc{refl})$$

$$\frac{\mathsf{vptr}(\mathtt{l}) \quad \mathsf{vptr}(\mathtt{l}') \quad \mathsf{reach}(\mathbb{H}, \mathtt{l}'', \mathtt{l}') \quad \mathbb{H}(\mathtt{l}) = \mathtt{l}'' \vee \mathbb{H}(\mathtt{l}+4) = \mathtt{l}''}{\mathsf{reach}(\mathbb{H}, \mathtt{l}, \mathtt{l}')} \quad (\textsc{next})$$

Figure 7. Reachability

language as heap predicates with the type $\mathit{Heap} \to \mathit{Prop}$ to specify the collector's heap manipulation. We shallowly embed these primitives in CiC, as listed in Figure. 6. The definitions are consistent with the semantics described in Ref.[12]. We write $\mathbb{H} \Vdash A$ if $(A\ \mathbb{H})$ is a valid proposition in CiC. We also follow the standard separation-logic abbreviations: $\mathtt{l} \mapsto \mathtt{w}_1, \mathtt{w}_2$ for $\mathtt{l} \mapsto \mathtt{w}_1\ *\ \mathtt{l}+4 \mapsto \mathtt{w}_2$, $\mathtt{l} \mapsto -$ for $\exists x : \mathit{Nat}.\ \mathtt{l} \mapsto x$, etc.

With such knowledge, we move on to build the invariant of the collector.

4.2 Reachability

Following the informal definition of reachability in Section 2 and the heap layout in Fig.3, we give the formal definition of reachability in Fig.7. The lower and upper boundaries of the collector's allocatable heap, st and ed, are both aligned to 8, which is the size of a heap object. A value l is a valid pointer ($\mathsf{vptr}(\mathtt{l})$) only if it is the address of an allocatable heap object.

The reachability predicate $\mathsf{reach}(\mathbb{H}, \mathtt{l}, \mathtt{l}')$ is inductively defined. In the base case, a valid pointer is self-reachable. And in the inductive case, $\mathtt{l}'$ is reachable from l if

$$(HeapPath) \quad \chi \quad ::= \quad [] \mid \mathtt{l} :: \chi$$

$\mathsf{head}([]) \stackrel{\text{def}}{=} -$
$\mathsf{head}(\mathtt{l} :: \chi) \stackrel{\text{def}}{=} \mathtt{l}$

$\mathsf{last}([]) \stackrel{\text{def}}{=} -$
$\mathsf{last}(\mathtt{l} :: []) \stackrel{\text{def}}{=} \mathtt{l}$
$\mathsf{last}(\mathtt{l} :: \chi) \stackrel{\text{def}}{=} \mathsf{head}(\chi)$

$\mathsf{heap\_next}(\mathbb{H}, \mathtt{l}, \mathtt{l}') \stackrel{\text{def}}{=} \mathbb{H}(\mathtt{l}) = \mathtt{l}' \vee \mathbb{H}(\mathtt{l}+4) = \mathtt{l}'$

$\mathsf{wfpth}([], \mathbb{H}, W) \stackrel{\text{def}}{=} \mathsf{true}$
$\mathsf{wfpth}(\mathtt{l} :: [], \mathbb{H}, W) \stackrel{\text{def}}{=} \mathtt{l} \in W$
$\mathsf{wfpth}(\mathtt{l} :: \mathtt{l}' :: \chi', \mathbb{H}, W) \stackrel{\text{def}}{=} \mathtt{l} \in W \wedge \mathsf{heap\_next}(\mathbb{H}, \mathtt{l}, \mathtt{l}') \wedge \mathsf{wfpth}(\mathtt{l}' :: \chi', \mathbb{H}, W)$

$\mathsf{white\_path}(\mathbb{H}, G, W, \mathtt{l}) \stackrel{\text{def}}{=} \exists \chi, \mathtt{l}_{\mathrm{G}} \in G, \mathtt{l}_{\mathrm{W}} \in W.$
$\mathsf{heap\_next}(\mathbb{H}, \mathtt{l}_{\mathrm{G}}, \mathtt{l}_{\mathrm{W}}) \wedge \mathsf{wfpth}(\chi, \mathbb{H}, W) \wedge \mathsf{head}(\chi) = \mathtt{l}_{\mathrm{W}} \wedge \mathsf{last}(\chi) = \mathtt{l}$

$\mathsf{weak\_tricolor}((\mathbb{H}, \mathbb{R}), B, G, W) \stackrel{\text{def}}{=} (\forall \mathtt{r} \in \{\mathtt{r1}, \mathtt{r2}\}.\ \mathbb{R}(\mathtt{r}) \in W \rightarrow \mathsf{white\_path}(\mathbb{H}, G, W, \mathbb{R}(\mathtt{r}))) \wedge$
$(\forall \mathtt{l}_{\mathrm{B}} \in B, \mathtt{l}_{\mathrm{W}} \in W.\ \mathsf{heap\_next}(\mathbb{H}, \mathtt{l}_{\mathrm{B}}, \mathtt{l}_{\mathrm{W}}) \rightarrow \mathsf{white\_path}(\mathbb{H}, G, W, \mathtt{l}_{\mathrm{W}}))$

Figure 8. Weak tricolor invariant

it is reachable from the pointers in the heap object at $\mathtt{l}$. The predicate $\mathsf{rchrt}(\mathbb{S}, \mathtt{l})$ asserts that $\mathtt{l}$ points to a root reachable heap object in the state $\mathbb{S}$.

4.3 The weak tricolor invariant

In both the stop-the-world and incremental-updating mark-sweep collectors, the following invariant, known as the *strong tricolor invariant*[21], holds during the mark phase of the collection:

There are no pointers from black objects to white objects.

This invariant guarantees that when the mark phase finishes at a state with no *gray* objects, the set of *black* objects will form a closed subheap with reachable objects, and the *white* objects are unreachable and thus can be collected safely.

However, since the Yuasa write barrier never shades the newly written value, we can easily write a *white* pointer into a *black* object and break the strong tricolor invariant. But the collect still guarantees the property that the *black* objects form a closed heap when the mark phase is over. And this is achieved by maintaining a more intricate invariant, the *weak tricolor invariant*[21]:

All white objects pointed to by a black object, or stored in a root register, are reachable from some gray object through a chain of white objects.

Note that in our mutator-collector interaction model, we take mutable registers as roots, and the invariant is thus slightly different from the one in Ref.[21].

As the most important part of the collector's invariant, the weak tricolor invariant is formalized in Fig.8. A heap path χ is inductively defined as a list of addresses.

$$\mathsf{ok\_val}(S, \mathtt{w}) \stackrel{\text{def}}{=} \mathsf{atom}(\mathtt{w}) \vee \mathtt{w} \in S$$
$$\mathsf{ok\_fld}(S, \mathtt{l}) \stackrel{\text{def}}{=} \exists \mathtt{w}.\ !(\mathsf{ok\_val}(S, \mathtt{w})) \ * \ \mathtt{l} \mapsto \mathtt{w}$$
$$\mathsf{ok\_obj}(S, \mathtt{l}) \stackrel{\text{def}}{=} \mathsf{ok\_fld}(S, \mathtt{l}) \ * \ \mathsf{ok\_fld}(S, \mathtt{l} + 4)$$
$$\mathsf{obj\_hp}(S', S) \stackrel{\text{def}}{=} \forall_* x \in S.\ \mathsf{ok\_obj}(S', x)$$

$$\mathsf{flst}(\mathtt{l}, \emptyset) \stackrel{\text{def}}{=} !(\mathtt{l} = \mathsf{null})$$
$$\mathsf{flst}(\mathtt{l}, \{\mathtt{l}\} \cup S) \stackrel{\text{def}}{=} !(\mathtt{l} \neq \mathsf{null}) * \exists \mathtt{l}'.\ \mathtt{l} \mapsto \mathsf{leaveme}, \mathtt{l}' \ * \ \mathsf{flst}(\mathtt{l}', S - \mathtt{l})$$
$$\mathsf{flist}(\mathbb{R}, S) \stackrel{\text{def}}{=} \mathsf{flst}(\mathbb{R}(\mathtt{rfptr}), S)$$

$$\mathsf{array\_set}(\mathtt{l}, \emptyset) \stackrel{\text{def}}{=} \mathsf{emp}$$
$$\mathsf{array\_set}(\mathtt{l}, \{\mathtt{w}\} \cup S) \stackrel{\text{def}}{=} \mathtt{l} \mapsto \mathtt{w} \ * \ \mathsf{array\_set}(\mathtt{l} + 4, S - \mathtt{w})$$
$$\mathsf{buffer}(\mathtt{l}, \mathtt{l}') \stackrel{\text{def}}{=} \forall_* x \in \{x \mid x \bmod 4 = 0 \wedge \mathtt{l} \leqslant x < \mathtt{l}'\}.\ x \mapsto -$$
$$\mathsf{mstk}(S, x, y, z) \stackrel{\text{def}}{=} !(y - x = \mathsf{size}(S)) * \mathsf{array\_set}(x, S) * \mathsf{buffer}(y, z)$$
$$\mathsf{mstack}(\mathbb{R}, S) \stackrel{\text{def}}{=} \mathsf{mstk}(S, \mathbb{R}(\mathtt{rbot}), \mathbb{R}(\mathtt{rtop}), \mathbb{R}(\mathtt{rbuf}))$$

$$\mathsf{hdr}(\mathtt{l}) \stackrel{\text{def}}{=} (\mathsf{ed} + (\mathtt{l} - \mathsf{st})/2)$$
$$\mathsf{mbits}(S, n) \stackrel{\text{def}}{=} \forall_* x \in S.\ \mathsf{hdr}(x) \mapsto n$$
$$\mathsf{bmbits}(S, \mathtt{l}, m, n) \stackrel{\text{def}}{=} \forall_* x \in S.\ \mathsf{hdr}(x) \mapsto (\mathsf{if}\ x < \mathtt{l}\ \mathsf{then}\ m\ \mathsf{else}\ n)$$

Figure 9. Auxiliary heap definitions

The functions $\mathsf{head}(\chi)$ and $\mathsf{last}(\chi)$ return the first and last address on the path χ, respectively. Note that both $\mathsf{head}(\chi)$ and $\mathsf{last}(\chi)$ are undefined when χ is empty. The predicate $\mathsf{heap\_next}(\mathbb{H}, \mathtt{l}, \mathtt{l}')$ asserts that there is a pointer from object $\mathtt{l}$ to object $\mathtt{l}'$ in $\mathbb{H}$. A heap path χ is well-formed with some pointer set W in heap $\mathbb{H}$ (asserted by $\mathsf{wfpth}(\chi, \mathbb{H}, W)$) if the objects in χ form a linked list and are all in the set W. Finally, the definition of the weak tricolor invariant predicate $\mathsf{weak\_tricolor}(\mathbb{S}, B, G, W)$ and the white path predicate $\mathsf{white\_path}(\mathbb{H}, G, W, \mathtt{l})$ follow exactly with their informal descriptions mentioned earlier.

4.4 Heap components

Following the collector's heap layout in Fig.3, we formalize the various heap components of the collector in Fig.9. This part of formalization resembles the work in Ref.[13], and is only briefly discussed here. Readers who are interested in may find the detailed explanation in Ref.[13].

The heap predicate $\mathsf{obj\_hp}(S', S)$ asserts an object heap containing exactly the objects in set S, and all the pointers stored in it belong to the set S'. So, once $\mathbb{H} \Vdash \mathsf{obj\_hp}(S, S)$ holds, $\mathbb{H}$ will be a *closed* heap with no outgoing pointers. The relation between obj_hp and reach is stated in Lemma 1.

Lemma 1 (Object Heap Reachability).
If $\mathbb{H} \Vdash \mathsf{obj\_hp}(S, S)$, $\mathtt{l} \in S$, *and* $\mathsf{reach}(\mathbb{H}, \mathtt{l}, \mathtt{l}')$, *then* $\mathtt{l}' \in S$.

The heap predicate $\mathsf{flist}(\mathbb{R}, S)$ asserts a list of free objects in S with the list header in register `rfptr`. Similarly, $\mathsf{mstack}(\mathbb{R}, S)$ asserts a mark stack with objects in S, and the stack pointers (`top`, `bot` and `buf`) are stored in the corresponding registers. There

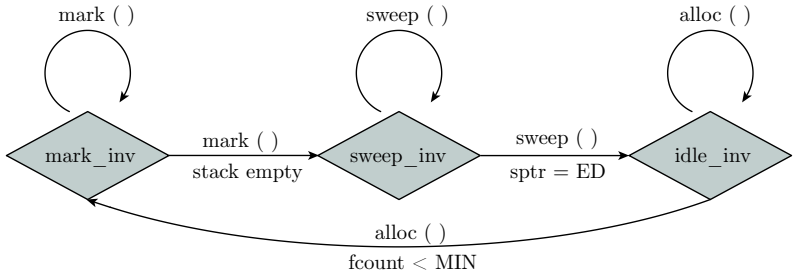

Figure 10. Invariant transition

are two predicates for mark bits, where $\mathsf{mbits}(S, n)$ asserts that there is a mark bit n for each member x of S, while $\mathsf{bmbits}(S, \mathtt{l}, m, n)$ asserts that there are different headers for objects in S regarding their relative position to the address `l`.

4.5 Collector's invariant

With the building blocks defined in Subsections 4.3 and 4.4, we form the global invariant of the collector in Hoare-style state predicate.

A garbage collection cycle is divided into three phases, namely the mark, sweep and idle phase. We firstly define for each phase a state invariant that must be satisfied when entering or exiting a collector routine in that phase. The transition between these invariants is done by executing the corresponding collector routines, as shown in Fig.10.

We define the mark phase invariant mark_inv in Fig.11, which asserts that in the mark phase:

1. global object set ptrs is divided into four subsets, in which the allocated objects are grouped into the sets B, G, and W according to their mark bits and the content of the mark stack, while the set F contains the objects on the free list;

2. flag register `rph` and the sweep pointer `rsptr` are properly set, and the borders of the object heap are loaded to the corresponding registers ($\mathsf{sted\_ok}(\mathbb{R})$);

3. values in the root registers are either atomic or pointers to the allocated objects (in the set $B \cup G \cup W$);

4. weak tricolor invariant holds on the state with the sets B, G and W;

5. global heap contains all the necessary parts: the object heap, the free list, the mark bits and the mark stack.

The closed subheap $\mathbb{H}_{\mathrm{T}}$ represents the mutator's view of the heap, which corresponds to the heap of the abstract state discussed in Ref.[14]. It is also used to show that each collector routine preserves the reachable objects, as we will discuss in the next section.

The mark phase finishes with no *gray* object. And the weak tricolor invariant guarantees that the *black* objects form a closed heap, which is the only object heap

$$\mathsf{eq}(\mathbb{H}) \stackrel{\text{def}}{=} \lambda\mathbb{H}'.\ \mathbb{H}' = \mathbb{H}$$
$$\mathsf{sted\_ok}(\mathbb{R}) \stackrel{\text{def}}{=} \mathbb{R}(\mathtt{rst}) = \mathsf{st} \ \wedge\ \mathbb{R}(\mathtt{red}) = \mathsf{ed}$$
$$\mathsf{roots\_ok}(S, \mathbb{R}) \stackrel{\text{def}}{=} \forall \mathtt{r} \in \{\mathtt{r1}, \mathtt{r2}\}.\ \mathsf{ok\_val}(S, \mathbb{R}(\mathtt{r}))$$

$$\begin{aligned}
&\mathsf{mark_inv}((\mathbb{H}, \mathbb{R}), \mathbb{H}_{\text{T}}) \stackrel{\text{def}}{=} \exists B, G, W, F.\ B \cup G \cup W \cup F = \mathsf{ptrs} \wedge \\
&\quad \mathbb{R}(\mathtt{rph}) = \mathsf{mark} \ \wedge\ \mathbb{R}(\mathtt{rsptr}) = \mathsf{st} \ \wedge \mathsf{sted_ok}(\mathbb{R}) \ \wedge\ \mathsf{roots_ok}(B \cup G \cup W, \mathbb{R}) \ \wedge \\
&\quad \mathsf{weak_tricolor}((\mathbb{H}, \mathbb{R}), B, G, W) \ \wedge \\
&\quad \mathbb{H} \Vdash \mathsf{eq}(\mathbb{H}_{\text{T}}) * \mathsf{flist}(\mathbb{R}, F) * \mathsf{mstack}(\mathbb{R}, G) * \mathsf{mbits}(B, 1) * \mathsf{mbits}(G, 1) * \mathsf{mbits}(W, 0) * \mathsf{mbits}(F, 0) \ \wedge \\
&\quad \mathbb{H}_{\text{T}} \Vdash \mathsf{obj_hp}(B \cup G \cup W, B \cup G \cup W)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{sweep_inv}((\mathbb{H}, \mathbb{R}), \mathbb{H}_{\text{T}}) \stackrel{\text{def}}{=} \exists B, W, F.\ B \cup W \cup F = \mathsf{ptrs} \wedge \\
&\quad \mathbb{R}(\mathtt{rph}) = \mathsf{sweep} \ \wedge\ \mathbb{R}(\mathtt{rsptr}) \in \mathsf{ptrs} \ \wedge \mathsf{sted_ok}(\mathbb{R}) \ \wedge\ \mathsf{roots_ok}(B, \mathbb{R}) \ \wedge \\
&\quad (\forall x \in W.\ x \geq \mathbb{R}(\mathtt{rsptr})) \ \wedge \\
&\quad \mathbb{H} \Vdash \mathsf{eq}(\mathbb{H}_{\text{T}}) * \mathsf{obj_hp}(\mathsf{ptrs}, W) * \mathsf{flist}(\mathbb{R}, F) * \mathsf{mstack}(\mathbb{R}, \emptyset) * \\
&\qquad \mathsf{bmbits}(B, \mathbb{R}(\mathtt{rsptr}), 0, 1) * \mathsf{mbits}(W, 0) * \mathsf{mbits}(F, 0) \ \wedge \\
&\quad \mathbb{H}_{\text{T}} \Vdash \mathsf{obj_hp}(B, B)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{idle_inv}((\mathbb{H}, \mathbb{R}), \mathbb{H}_{\text{T}}) \stackrel{\text{def}}{=} \exists W, F.\ W \cup F = \mathsf{ptrs} \wedge \\
&\quad \mathbb{R}(\mathtt{rph}) = \mathsf{idle} \ \wedge\ \mathbb{R}(\mathtt{rsptr}) = \mathsf{ed} \ \wedge \mathsf{sted_ok}(\mathbb{R}) \ \wedge\ \mathsf{roots_ok}(W, \mathbb{R}) \ \wedge \\
&\quad \mathbb{H} \Vdash \mathsf{eq}(\mathbb{H}_{\text{T}}) * \mathsf{flist}(\mathbb{R}, F) * \mathsf{mstack}(\mathbb{R}, \emptyset) * \mathsf{mbits}(W, 0) * \mathsf{mbits}(F, 0) \ \wedge \\
&\quad \mathbb{H}_{\text{T}} \Vdash \mathsf{obj_hp}(W, W)
\end{aligned}$$

$$\mathsf{gc\_inv}(\mathbb{S}, \mathbb{H}_{\text{T}}) \stackrel{\text{def}}{=} \mathsf{mark\_inv}(\mathbb{S}, \mathbb{H}_{\text{T}}) \vee \mathsf{sweep\_inv}(\mathbb{S}, \mathbb{H}_{\text{T}}) \vee \mathsf{idle\_inv}(\mathbb{S}, \mathbb{H}_{\text{T}})$$

Figure 11. Collector's invariant

preserved during the sweep phase. We thus move the *white* objects out of $\mathbb{H}_{\text{T}}$ in the sweep phase invariant sweep_inv, which asserts that:

1. global object set ptrs is divided into the subsets B, W and F, which have the same meaning as in mark_inv;
2. flag register rph is properly set, and the sweep pointer $\mathbb{R}(\mathtt{rsptr})$ is a valid object pointer in ptrs. The heap borders are loaded to the corresponding registers;
3. values in the root registers are either atomic or pointers in the allocated subheap (in the set B);
4. all objects in W are behind the sweep pointer;
5. global heap contains all the necessary parts: the object heap, the free list, the mark bits and the mark stack.

The heap predicate $\mathsf{bmbits}(B, \mathbb{R}(\mathtt{rsptr}), 0, 1)$ asserts that the mark bits of the *black* objects before the sweep pointer $\mathbb{R}(\mathtt{rsptr})$ has already been reset by the collector.

The sweep phase finishes when the sweep pointer reaches the end of the object heap ($\mathbb{R}(\mathtt{rsptr}) = \mathsf{ed}$), which ensures that all the *white* objects are collected ($W = \emptyset$). With this, the system finishes garbage collection and enters the idle phase. The invariant idle_inv of the idle phase reassembles sweep_inv, except that the set W is empty and all the mark bits of objects in B are reset.

Finally, the global invariant gc_inv, which holds at every control transfer between the mutator and the collector (as shown in Fig.1), is simply the disjunction of the three cases based on the flag register rph.

5 Specification and Proof of the Collector

Our specification and proof of the collector is constructed within the SCAP framework[11], which is a Hoare-style logic for modular verification of assembly-level program with stack-based control abstraction.

We begin this section with a brief introduction to SCAP. Then, we present the SCAP specifications of the collector's major procedures and discuss the proof construction issues.

5.1 SCAP

SCAP is an FPCC system where programs are verified as assembly implementation. The basic units of specification and verification are *instruction sequences* (aka. *code blocks*) ending with jumps. And an SCAP specification of a code block is a pair of state predicates (p, g) with the types:

$$\begin{array}{lcl} (Pre) & \mathrm{p} & \in \quad State \to Prop \\ (Guar) & \mathrm{g} & \in \quad State \to State \to Prop \end{array}$$

The precondition p resembles the precondition in Hoare logic, while the guarantee g relates to the entry state of the code block with the return state of the corresponding procedure. Thus the behavior of a procedure, in terms of state transition, is asserted by the specification g of its entry block. Note that in the rest of this section, when we talk about a specification of a particular procedure, we actually mean the SCAP specification of its entry block.

SCAP employs a set of inference rules for building well-formedness proofs for code blocks against their specifications. And there are also rules for grouping the well-formed code blocks into a well-formed program.

The rules are built according to an operational-semantics-based abstract machine model. And the soundness of SCAP ensures that well-formed program will run without sticking to the machine model, and each code block, or procedure, functions according to its specification.

The uncovered details of the machine model and SCAP are not necessarily needed for understanding the rest of this paper. However, interested readers may refer to Refs.[11, 13, 14]. Besides, for clarity of presentation in this section and the rest, we are not going to unfold the collector procedures into their assembly implementations. Still, readers who have interest should be aware that our proofs[26] are actually built for the assembly implementation of the collector.

5.2 Collector specifications

We discuss here the SCAP specifications of the collector's major procedures, as listed in Fig.12. For the basic procedures like stack operations and `mark_field()`, their specifications are trivially identical to those used in a stop-the-world collector[13],

$\mathsf{hp\_sub}(\mathbb{H}, \mathbb{H}', \mathbb{R}) \stackrel{\text{def}}{=} \mathbb{H} \Vdash \mathsf{eq}(\mathbb{H}') * \mathsf{true} \ \wedge \exists S.\ \mathbb{H}' \Vdash \mathsf{obj\_hp}(S, S) \ \wedge \ \mathsf{roots\_ok}(S, \mathbb{R})$
$\mathsf{hp\_ext}(\mathbb{H}, \mathbb{H}', \mathbb{R}) \stackrel{\text{def}}{=} \mathbb{H}' \Vdash \mathsf{eq}(\mathbb{H}) * \mathbb{R}(\mathtt{r2}) \mapsto -, -$
$\mathsf{reg\_ok}((\mathbb{H}, \mathbb{R}), (\mathbb{H}', \mathbb{R}')) \stackrel{\text{def}}{=} \forall \mathtt{r} \in \{\mathtt{r1}, \mathtt{r2}\}.\ \mathbb{R}(\mathtt{r}) = \mathbb{R}'(\mathtt{r})$

$\mathrm{p}_{\mathtt{mark}} \stackrel{\text{def}}{=} \lambda\mathbb{S}.\ \exists\mathbb{H}_{\mathrm{T}}.\ \mathsf{mark\_inv}(\mathbb{S}, \mathbb{H}_{\mathrm{T}})$
$\mathrm{g}_{\mathtt{mark}} \stackrel{\text{def}}{=} \lambda\mathbb{S}, \mathbb{S}'.\ \mathsf{reg\_ok}(\mathbb{S}, \mathbb{S}') \wedge$
$\quad \forall\mathbb{H}_{\mathrm{T}}.\ \mathsf{mark\_inv}(\mathbb{S}, \mathbb{H}_{\mathrm{T}}) \rightarrow \mathsf{mark\_inv}(\mathbb{S}', \mathbb{H}_{\mathrm{T}}) \ \vee \exists\mathbb{H}'_{\mathrm{T}}.\ \mathsf{hp\_sub}(\mathbb{H}_{\mathrm{T}}, \mathbb{H}'_{\mathrm{T}}, \mathbb{S}.\mathbb{R}) \ \wedge \ \mathsf{sweep\_inv}(\mathbb{S}', \mathbb{H}'_{\mathrm{T}})$

$\mathrm{p}_{\mathtt{sweep}} \stackrel{\text{def}}{=} \lambda\mathbb{S}.\ \exists\mathbb{H}_{\mathrm{T}}.\ \mathsf{sweep\_inv}(\mathbb{S}, \mathbb{H}_{\mathrm{T}})$
$\mathrm{g}_{\mathtt{sweep}} \stackrel{\text{def}}{=} \lambda\mathbb{S}, \mathbb{S}'.\ \mathsf{reg\_ok}(\mathbb{S}, \mathbb{S}') \wedge$
$\quad \forall\mathbb{H}_{\mathrm{T}}.\ \mathsf{sweep\_inv}(\mathbb{S}, \mathbb{H}_{\mathrm{T}}) \rightarrow \mathsf{sweep\_inv}(\mathbb{S}', \mathbb{H}_{\mathrm{T}}) \vee \mathsf{idle\_inv}(\mathbb{S}', \mathbb{H}_{\mathrm{T}})$

$\mathrm{p}_{\mathtt{alloc}} \stackrel{\text{def}}{=} \lambda\mathbb{S}.\ \exists\mathbb{H}_{\mathrm{T}}.\ \mathsf{gc\_inv}(\mathbb{S}, \mathbb{H}_{\mathrm{T}})$
$\mathrm{g}_{\mathtt{alloc}} \stackrel{\text{def}}{=} \lambda\mathbb{S}, \mathbb{S}'.\ \mathbb{S}.\mathbb{R}(\mathtt{r1}) = \mathbb{S}'.\mathbb{R}(\mathtt{r1}) \wedge$
$\quad \forall\mathbb{H}_{\mathrm{T}}.\ \mathsf{gc\_inv}(\mathbb{S}, \mathbb{H}_{\mathrm{T}}) \rightarrow \exists\mathbb{H}'_{\mathrm{T}}, \mathbb{H}''_{\mathrm{T}}.\ \mathsf{gc\_inv}(\mathbb{S}', \mathbb{H}'_{\mathrm{T}}) \wedge \mathsf{hp\_sub}(\mathbb{H}_{\mathrm{T}}, \mathbb{H}''_{\mathrm{T}}, \mathbb{S}.\mathbb{R}) \wedge \mathsf{hp\_ext}(\mathbb{H}''_{\mathrm{T}}, \mathbb{H}'_{\mathrm{T}}, \mathbb{S}'.\mathbb{R})$

$\mathrm{p}_{\mathtt{write}}(i) \stackrel{\text{def}}{=} \lambda\mathbb{S}.\ \exists\mathbb{H}_{\mathrm{T}}.\ \mathsf{gc\_inv}(\mathbb{S}, \mathbb{H}_{\mathrm{T}})$
$\mathrm{g}_{\mathtt{write}}(i) \stackrel{\text{def}}{=} \lambda\mathbb{S}, \mathbb{S}'.\ \mathsf{reg\_ok}(\mathbb{S}, \mathbb{S}') \wedge$
$\quad \forall\mathbb{H}_{\mathrm{T}}.\ \mathsf{gc\_inv}(\mathbb{S}, \mathbb{H}_{\mathrm{T}}) \rightarrow \mathsf{gc\_inv}(\mathbb{S}', \mathbb{H}_{\mathrm{T}}\{\mathbb{S}.\mathbb{R}(\mathtt{r1}) + i \rightsquigarrow \mathbb{S}.\mathbb{R}(\mathtt{r2})\})$
where $i \in \{0, 4\}$

Figure 12. Collector specifications

and are thus skipped here. Readers may refer to Ref.[26] for the detailed specifications of all the code blocks of the collector.

We begin with the auxiliary definitions, and go on to explain the specifications in Fig.12 one by one.

The predicate $\mathsf{hp\_sub}(\mathbb{H}, \mathbb{H}', \mathbb{R})$ asserts that $\mathbb{H}'$ is a closed subheap of $\mathbb{H}$ with root reachable objects; $\mathsf{hp\_ext}(\mathbb{H}, \mathbb{H}', \mathbb{R})$ asserts that $\mathbb{H}'$ extents $\mathbb{H}$ with exactly one more object pointed to by $\mathbb{R}(\mathtt{r2})$; and $\mathsf{reg\_ok}(\mathbb{S}, \mathbb{S}')$ asserts that the root registers are preserved from $\mathbb{S}$ to $\mathbb{S}'$.

5.2.1 Mark

We know from Figs.4 and 10 that the `mark()` procedure is always invoked in the mark phase. Thus its precondition $\mathrm{p}_{\mathtt{mark}}$ only requires that the entry state satisfies the mark phase invariant $\mathsf{mark\_inv}$ in Fig.11. The guarantee $\mathrm{g}_{\mathtt{mark}}$ is divided into two parts. Firstly, the root registers are preserved. Secondly, if the entry state satisfies $\mathsf{mark\_inv}$, either $\mathsf{mark\_inv}$ or $\mathsf{sweep\_inv}$ will hold on the return state. Following the definition of $\mathsf{mark\_inv}$, the universal quantification of the subheap $\mathbb{H}_{\mathrm{T}}$ ensures that all reachable objects in state $\mathbb{S}$ are preserved by the execution of `mark()`.

Lemma 2 (Mark safety).
If $\mathrm{g}_{\mathtt{mark}}\ (\mathbb{H}, \mathbb{R})\ (\mathbb{H}', \mathbb{R}')$ *and* $\mathsf{reach}(\mathbb{H}, \mathbb{R}(\mathtt{r}), \mathtt{l})$ *for some root register* `r`*, then* $\mathbb{H}(\mathtt{l}) = \mathbb{H}'(\mathtt{l})$ *and* $\mathbb{H}(\mathtt{l}+4) = \mathbb{H}'(\mathtt{l}+4)$.

Proof sketch:
For each of the two cases in $\mathrm{g}_{\mathtt{mark}}$, we are able to find an untouched subheap $\mathbb{H}_{\mathrm{s}}$ satisfying both $\mathbb{H}_{\mathrm{s}} \Vdash \mathsf{obj\_hp}(S, S)$ and $\mathsf{roots\_ok}(S, \mathbb{R})$ for some set S. By Lemma 1, we know that all the reachable objects are untouched.

We have $\mathbb{H}_{\mathrm{T}}$ for the first case, and $\mathbb{H}'_{\mathrm{T}}$ for the second, following the definition of mark_inv and hp_sub. □

5.2.2 Sweep

The specification of `sweep()` follows the same idea for `mark()`. The precondition $\mathrm{p}_{\texttt{sweep}}$ only asserts that the sweep phase invariant sweep_inv holds. On the other hand, the guarantee $\mathrm{g}_{\texttt{sweep}}$ ensures that the procedure makes correct transition from sweep_inv to either sweep_inv or idle_inv, while the mutator's view of the heap ($\mathbb{H}_{\mathrm{T}}$) is identical in the two states. Like Lemma 2, we also have the safety lemma for `sweep()`.

Lemma 3 (Sweep safety).
If $\mathrm{g}_{\texttt{sweep}}\ (\mathbb{H}, \mathbb{R})\ (\mathbb{H}', \mathbb{R}')$ *and* $\mathsf{reach}(\mathbb{H}, \mathbb{R}(\texttt{r}), \texttt{l})$ *for some root register* `r`*, then* $\mathbb{H}(\texttt{l}) = \mathbb{H}'(\texttt{l})$ *and* $\mathbb{H}(\texttt{l}+4) = \mathbb{H}'(\texttt{l}+4)$.

5.2.3 Alloc and write

The two interface procedures `alloc()` and `write()` can be invoked in any phases and thus require the global invariant gc_inv holds on their entry states, as specified by $\mathrm{p}_{\texttt{alloc}}$ and $\mathrm{p}_{\texttt{write}}(i)$. Here i separates the write operations on the first and second field of an object. On the other hand, both the guarantees $\mathrm{g}_{\texttt{alloc}}$ and $\mathrm{g}_{\texttt{write}}(i)$ require that the corresponding procedures preserve gc_inv between their entry and return states.

Besides, the guarantee $\mathrm{g}_{\texttt{alloc}}$ of `alloc()` asserts that the mutator's subheap $\mathbb{H}'_{\mathrm{T}}$ in the return state is an extension of a possibly garbage-collected version ($\mathbb{H}''_{\mathrm{T}}$) of the subheap $\mathbb{H}_{\mathrm{T}}$ in the entry state.

Lemma 4 (Alloc safety).
If $\mathrm{g}_{\texttt{alloc}}\ (\mathbb{H}, \mathbb{R})\ (\mathbb{H}', \mathbb{R}')$ *and* $\mathsf{reach}(\mathbb{H}, \mathbb{R}(\texttt{r}), \texttt{l})$ *for some root register* `r`*, then* $\mathbb{H}(\texttt{l}) = \mathbb{H}'(\texttt{l})$ *and* $\mathbb{H}(\texttt{l}+4) = \mathbb{H}'(\texttt{l}+4)$.

Proof sketch:
Following the definition of hp_sub and hp_ext, we apply Lemma 1 with $\mathbb{H}''_{\mathrm{T}}$, and get the conclusion with the same idea used in the proof of Lemma 2. □

The guarantee $\mathrm{g}_{\texttt{write}}(i)$ of the write barrier ensures that the collector's invariant still holds even after the mutator changes the content of the object heap $\mathbb{H}_{\mathrm{T}}$, and the reachable objects, except for the one been written on, are preserved.

Lemma 5 (Write safety).
If $\mathrm{g}_{\texttt{write}}(i)\ (\mathbb{H}, \mathbb{R})\ (\mathbb{H}', \mathbb{R}')$, $\mathsf{reach}(\mathbb{H}, \mathbb{R}(\texttt{r}), \texttt{l})$ *for some root register* `r`*, and* $\texttt{l} \neq \mathbb{R}(\texttt{r1})$*, then* $\mathbb{H}(\texttt{l}) = \mathbb{H}'(\texttt{l})$ *and* $\mathbb{H}(\texttt{l}+4) = \mathbb{H}'(\texttt{l}+4)$.

5.2.4 Discussion

The readers may notice that specifications in Fig.12 and Lemmas 2, 3, 4 and 5 mention only the safety of the collector. That is, all reachable objects are preserved.

Incremental collectors are all working in a somehow conservative style[7]. Thus, it is not possible to prove the correctness property that a successful garbage collection collects all the unreachable objects, which is enjoyed by stop-the-world collectors[13].

We also skip the proof of the real-time property of the collector, due to the limitation of the SCAP system. However, a time-based SCAP can be developed for this purpose, and we leave this to the future research.

5.3 Proof construction

With the improvements we made to SCAP, and our various tactics and lemma libraries for reasoning on finite sets and separation logic[13], the effort of proof construction is focused mainly on the domain-specific problems of the collector.

The main difficulty left in building proofs for the Yuasa collector is to show that the weak tricolor invariant $\mathsf{weak\_tricolor}(\mathbb{S}, B, G, W)$ is preserved by various operations on the state $\mathbb{S}$ as well as the object sets B, G, and W. To tackle this problem, we analyze the behaviors of the collector routines and form the following important lemmas on the properties of the white path predicate $\mathsf{white\_path}(\mathbb{H}, G, W, \mathtt{1})$, which is the key component in the definition of $\mathsf{weak\_tricolor}(\mathbb{S}, B, G, W)$.

In the loop body of the `mark()` procedure, as shown in Figure. 4, the *white* objects pointed to by object `ptr` is moved to the *gray* set before `ptr` joins the *black* set. Meanwhile, the mark stack and mark bits are also changed to keep consistent with the colors of the objects. We have Lemmas 6 and 7 for these operations.

Lemma 6 states that, first, moving objects from *white* to *gray* preserves the white path, and second, it is correct to move an object out of *gray* if both its fields contain atomic value or pointers to non-*white* objects.

Lemma 6 (White path move).
If $\mathtt{1} \in W/\{x\}$ *and* $\mathsf{white\_path}(\mathbb{H}, G, W, \mathtt{1})$*, then:*
1. $\mathsf{white\_path}(\mathbb{H}, G \cup \{x\}, W/\{x\}, \mathtt{1})$*;*
2. $y \notin W$*,* $m \in G$ *and* $\mathbb{H} \Vdash m \mapsto y, x * \mathsf{true}$ *implies*
$\mathsf{white\_path}(\mathbb{H}, (G/\{m\}) \cup \{x\}, W/\{x\}, \mathtt{1})$*;*

Proof sketch:
By unfolding the definition of white_path we get the existentially quantified path χ and the *gray* head of the path h.

We consider two cases for the first conclusion:
Case 1: If x is on the path χ, we break χ at x and form the new white path with x being its *gray* head.
Case 2: The conclusion is trivially proved by induction on χ, if x is not on χ.

For the second conclusion, we have to consider the additional case when m equals h, and the rest proof follows the one for the first conclusion. □

Lemma 7 states the fact that modifications on other heap components do not disrupt the properties of the white path.

Lemma 7 (White path subheap).
If $\mathbb{H} \Vdash \mathsf{eq}(\mathbb{H}_{\mathrm{T}}) * A$*,* $\mathbb{H}' \Vdash \mathsf{eq}(\mathbb{H}_{\mathrm{T}}) * B$*,* $\mathbb{H}_{\mathrm{T}} \Vdash \mathsf{obj\_hp}(S, S)$*, and* $\mathsf{white\_path}(\mathbb{H}, G, W, \mathtt{1})$*, where* G *and* W *are disjoint subsets of* S*, then* $\mathsf{white\_path}(\mathbb{H}', G, W, \mathtt{1})$*.*

Proof sketch:
By unfolding the definition of white_path and induction on the existentially quantified path χ. □

Note that Lemma 7 is also used to show that the allocation operation in `alloc()` preserves the white path invariant.

The write barrier `write()` stores a new value into an object field and shades the old field value *gray*. And Lemma 8 ensures that no matter the old value is

Lines	Component
962	Basic properties and tactics
1387	Ordered heap library
451	Abstract machine encoding and lemmas
1209	Ordered finite set library
910	Separation logic library
547	SCAP, VCGen and related tactics
471	Collector's heap definitions and lemmas
932	Weak tricolor invariant
2554	Code, specification and proof of the collector
9423	Total

Figure 13. Proof script breakdown

originally *white* or not, the white path property is preserved with some corresponding modifications to the *white* and *gray* sets.

Lemma 8 (White path update).
If $\mathbb{H}(m+i) = x$ *and* $\mathsf{white\_path}(\mathbb{H}, G, W, \mathtt{1})$*, where* G *and* W *are disjoint object pointer sets and* m *is an object pointer, then:*
1. $x \notin W$ *implies* $\mathsf{white\_path}(\mathbb{H}\{m+i \rightsquigarrow y\}, G, W, \mathtt{1})$*;*
2. $x \in W$ *and* $\mathtt{1} \in W/\{x\}$ *implies*
$\mathsf{white\_path}(\mathbb{H}\{m+i \rightsquigarrow y\}, G \cup \{x\}, W/\{x\}, \mathtt{1})$.
where $i \in \{0, 4\}$.

Proof sketch:
The first conclusion is proved by induction on the χ packed in the premise $\mathsf{white\_path}(\mathbb{H}, G, W, \mathtt{1})$.

For the second conclusion, we perform case analysis on whether x is on χ or not, which follows the idea used in the proof of Lemma 6. □

With the help of these lemmas, the main SCAP well-formedness theorems of the collector routines can be constructed with much less efforts.

6 Coq Implementation

Our work is fully implemented within the Coq proof assistant. In our Coq proofs, we do not assume any axioms except the *Law of Excluded Middle*, which is also removable by following the ordered set model in Ref.[14]. All the proofs are machine-checkable and the implementation can be packed immediately as FPCC packages.

Coq[22] is an interactive proof assistant based on a sound meta-logic, the CiC system. Following the *Curry-Howard isomorphism*[31], theorem proving in Coq is actually a constructive process for building terms with certain types. On the other hand, proof checking in Coq, which relies only on the simple type-checker of CiC, is trustworthy compared with other theorem provers like PVS, and is thus suitable for

code development in the FPCC style. Besides, Coq also provides strong facilities for building inductive definitions and induction-based proofs, which are very important for our work.

Our implementation includes both the construction of the verification framework and the verification of the collector. For the first part, we build the machine model, SCAP and its soundness proof, the VCGen and various lemma libraries. And the second part includes the basic constructs of the mark-sweep GC specifications, the Yuasa collector invariant and the codes and proofs of the collector's assembly-level implementation. In Fig.13, we show the script size of our implementation in terms of non-empty non-commented lines of Coq script. The script includes the definitions, lemmas, theorems and their proofs as lists of tactics. The implementation takes several man-months for programmers familiar with the Coq system.

Compared with the implementation in Ref.[13], it has cost us notable effort to build the weak tricolor invariant and prove its various properties, which are the most intricate parts of this work. The proofs of these lemmas, especially the ones in Subsection 5.3, often require detailed analysis for various subtle cases, and proving by induction is also frequently encountered. The code complexity of the Yuasa collector also imposes non-trivial increment of difficulty on the proof construction process compared with[13]. For example, the proof of the `alloc()` procedure must reflect the fact that the code is correct to run in any of the mark, sweep and idle phase of the collection, with different phase invariants.

The Coq code of our implementation is freely available through the Internet[26]. Interested readers may also refer to Ref.[13] for some details of the implementation of the verification framework, which is skipped here.

7 Related Work

The existing work on mechanized verification of garbage collectors (such as Refs. [15-20]) mainly focuses on abstract algorithms. Our certified collector, on the other hand, is a real machine-level implementation with concrete specifications and it can run directly on real machine. Verifying the collector against concrete specifications makes us focus on those subtle details of the collector which may lead to safety problems but are often ignored in a high-level verification. However, our verification only addresses the safety of the collector, not any liveness properties, such as the real-time property of the collector. Besides, we also benefit a lot from the ideas of these high-level verifications.

The work of Birkedal *et al.*[32] uses separation logic with a variety of new constructors to prove the correctness of a copying collector. Many of our ideas on formalizing the concrete heap model come from their work. Hawblitzel *et al.*[33] proposed a linear type system for type-checking the assembly implementation of a copying collector. However, the type system is very complex and has no mechanized soundness proof. Our FPCC-style verification is thus more trustworthy than theirs.

McCreight *et al.*[14] proposed a general framework for certifying collectors and their mutators. Within this framework, the verification of the mutators and collectors can be done separately according to a common abstraction, and then linked together with a set of cast lemmas. We follow their mutator-collector interaction model and the style for composing collector specifications. Our verification of the incremental

mark-sweep collector is also a good supplement to their work.

The work on CAP systems[34, 35, 11, 25] provides a good way to build FPCC packages. Our work builds on the SCAP system in Ref.[11] and can be ported to the OCAP system[25] to interact with mutators verified in other verification systems.

We proved the correctness of a conservative variant of the stop-the-word mark-sweep collector[13] and linked it with a foundational typed assembly language system[24] before. These are very useful experiences for the work described in this paper. On the other hand, the work in this paper also contains non-trivial increment and is different from the work in Ref.[13]. Our main focus is on formalizing and reasoning the weak tricolor invariant, which is much more complex than the invariant used in Ref.[13]. The comparison of the implementation efforts is discussed in Section 6.

8 Conclusion

We present in this paper the verification of the Yuasa incremental garbage collector in a Hoare-style PCC framework. We take a realistic mutator-collector interaction model as the basis of our verification. And the collector specifications are given on the machine-level state model using separation logic. Our verification is fully implemented in the Coq proof assistant, and can be packed immediately as FPCC packages. This work can be used as a model for building garbage-collected real-time PCC-style software systems. Besides, our formalization of the collector's invariant can also be used in the Hoare-style verification of concurrent mark-sweep collectors.

Acknowledgments

We thank Professor Zhong Shao and Andrew McCreight for their kindly help and suggestions. The valuable comments by Yan Guo and Cheng Liu are also greatly appreciated. This research is based on work partly supported by the National Natural Science Foundation of China under Grant Nos.90718026 and 60673126, and Intel China Research Center. Any opinions, findings, and conclusions presented in this document are those of the authors and do not reflect the views of these agencies.

References

[1] Steele GL Jr. Multiprocessing compactifying garbage collection. Commun. ACM, 1975, 18(9): 495–508.

[2] Dijkstra EW, Lamport L, Martin AJ, Scholten CS, Steffens EFM. On-the-fly garbage collection: an exercise in cooperation. Commun. ACM, 1978, 21(11): 966–975.

[3] Kung HT, Song SW. An efficient parallel garbage collection system and its correctness Proof. IEEE Symp. on Foundations of Comp. Sci. IEEE Computer Society, 1977: 120–131.

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

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

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

[7] Jones RE. Garbage Collection: Algorithms for Automatic Dynamic Memory Management. Wiley, 1996.

[8] Necula G. Proof-carrying code. Proc. 24th ACM Symp. on Principles of Prog. Lang. ACM Press, New York, January, 1997: 106–119.

[9] Hunt GC, Larus JR. Singularity: rethinking the software stack. SIGOPS Operating System Review, 2007, 41(2): 37–49.

[10] Feng XY, Shao Z, Dong Y, Guo Y. Certifying Low-Level programs with hardware interrupts and preemptive threads. PLDI '08: Proc. of the 2008 ACM SIGPLAN Conf. on Prog. Lang. Design and Impl., ACM Press, Tucson, Arizona, USA, Jun 2008: 170-182.

[11] Feng XY, Shao Z, Vaynberg A, Xiang S, Ni ZZ. Modular verification of assembly code with stack-based control abstractions. PLDI '06: Proc. of the 2006 ACM SIGPLAN Conf. on Prog. Lang. Design and Impl., ACM Press, New York, NY, USA, 2006: 401–414.

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

[13] Lin CX, Chen YY, Li L, Hua B. Garbage collector berification for proof-carrying code. J. Comp. Sci. and Tech., May 2007, 22(3): 426-437.

[14] McCreight A, Shao Z, Lin CX, Li L. A general framework for certifying garbage collectors and their mutators. PLDI '07: Proc. of the 2007 ACM SIGPLAN Conf. on Prog. Lang. Design and Impl., June 2007.

[15] Russinoff DM. A mechanically verified incremental garbage collector. Formal Aspects of Computing, 1994, 6: 359–390.

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

[17] Havelund K. Mechanical verification of a garbage collector. Proc. of the 11th IPPS/SPDP'99 Workshops, Springer-Verlag, London, UK, 1999: 1258–1283.

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

[19] Vechev MT, Yahav E, Bacon DF. Correctness-preserving derivation of concurrent garbage collection algorithms. LDI '06: Proc. of the 2006 ACM SIGPLAN Conf. on Prog. Lang. Design and Impl., ACM Press, 2006: 341–353.

[20] Vechev MT, Yahav E, Bacon DF, Rinetzky N. CGCExplorer: a semi-automated search procedure for provably correct concurrent collectors. PLDI '07: Proc. of the 2007 ACM SIGPLAN Conf. on Prog. Lang. Design and Impl., ACM Press, 2007: 456–467.

[21] Pekka P. Pirinen. Barrier techniques for incremental tracing. *ISMM '98: Proc. of the 1st Int'l Symp. on Memory Management*, New York, NY, USA, ACM Press, 1998: 20–25.

[22] Coq Development Team. The Coq proof assistant reference manual. Coq release v8.1, 2006.

[23] Appel AW. Foundational proof-carrying code. Symp. on Logic in Comp. Sci.(*LICS'01*), IEEE Comp. Soc., June 2001: 247–258.

[24] Lin CX, McCreight A, Shao Z, Chen YY, Guo Y. Foundational typed assembly language with certified garbage collection. Proc. of 1st IEEE IFIP Int'l Symp. on Theoretical Aspects of Soft. Eng. (TASE 2007), IEEE Computer Society, Shanghai, China, Jun 2007: 326–335.

[25] Feng XY, Ni ZZ, Shao Z, Guo Y. An open framework for foundational proof-carrying code. Proc. 3rd ACM Workshop on Types in Lang. Design and Impl., ACM Press, Nice, France, January 2007: 67–78.

[26] Lin CX, Chen YY, Hua B. Verification of a real-time garbage collector in Hoare-style logic (Coq implementation). http://ssg.ustcsz.edu.cn/ cxlin/yuasa, 2007.

[27] Doligez D, Leroy X. A concurrent, generational garbage collector for a multithreaded implementation of ml. POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, New York, NY, USA, 1993: 113–123.

[28] Boehm H-J, Demers AJ, Shenker S. Mostly parallel garbage collection. PLDI '91: Proc. of the ACM SIGPLAN 1991 Conf. on Prog. Lang. Design and Impl., ACM Press, New York, NY, USA, 1991: 157–164.

[29] Hudson RL, Moss JEB. Sapphire: Copying gc without stopping the world. Joint ACM Java Grande – ISCOPE 2001 Conference, 2001.

[30] Paulin-Mohring C. Inductive definitions in the system Coq—rules and properties. Proc. TLCA,

volume 664 of Lecture Notes in Computer Science, 1993.

[31] Howard WA. The formulas-as-types notion of construction. In: Curry THB ed. Essays on Combinatory Logic, Lambda Calculus, and Formalism. Academic Press, 1980: 479–490.

[32] Birkedal L, Torp-Smith N, Reynolds JC. Local reasoning about a copying garbage collector. Proc. of the 31st ACM Symp. on Principles of Prog. Lang., New York, NY, USA, NY, USA, ACM Press, 2004: 220–231.

[33] Hawblitzel C, Huang H, Wittie L, Chen J. A garbage-collecting typed assembly language. Proc. 3rd ACM SIGPLAN Int'l Workshop on Types in Lang. Design and Impl. ACM Press, January 2007.

[34] Yu D, Hamid NA, Shao Z. Building certified lbiraries for PCC: Dynamic storage allocation. Science of Comp. Prog., March 2004, 50(1-3): 101–127.

[35] Ni ZZ, Shao Z. Certified assembly programming with embedded code pointers. Proc. 33rd ACM SIGPLAN-SIGACT Symp. on Principles of Prog. Lang., January 2006.