Lin CX, Chen YY, Li L *et al.* Garbage collector verification for proof-carrying code. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 22(3): 426~437 May 2007

Garbage Collector Verification for Proof-Carrying Code

Chun-Xiao Lin (林春晓), Yi-Yun Chen (陈意云), Long Li (李 隆), and Bei Hua (华 蓓)

Department of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China

E-mail: {cxlin3, liwis}@mail.ustc.edu.cn; {yiyun, bhua}@ustc.edu.cn

Received September 19, 2006; revised March 20, 2007.

Abstract We present the verification of the machine-level implementation of a conservative variant of the standard mark-sweep garbage collector in a Hoare-style program logic. The specification of the collector is given on a machine-level memory model using separation logic, and is strong enough to preserve the safety property of any common mutator program. Our verification is fully implemented in the Coq proof assistant and can be packed immediately as foundational proof-carrying code package. Our work makes important attempt toward building fully certified production-quality garbage collectors.

Keywords program verification, garbage collector, proof-carrying code, program safety

1 Introduction

The latest developments in theorem provers, certifying compilers and other program verification tools have enabled the building of more trustworthy software systems using the Proof-Carrying Code (PCC) paradigm[1]. A PCC package contains the native code of the software, its safety specification, and a machine checkable proof saying that the code behaves as specified. The client can then use a tiny checker to check the safety proof before running the software.

Software verified in a high-level language is hardly trustworthy if the compiler fails to correctly translate the verified program into native code, which is often the case due to the various bugs in the compiler. However, in the PCC style of verification, the specification and proof are directly given on, or translated to, the machine level, and the compiler is thus removed from the Trusted Computing Base (TCB). Therefore, the PCC packages are endowed with a higher level of safety.

However, in existing PCC frameworks like the Touchstone-PCC[1] and *Typed Assembly Language* (TAL)[2], memory management is either regarded as trusted primitive instructions, or included in the trusted library. On the other hand, automatic memory management, especially Garbage Collection (GC)[3], is acknowledged as complex and error-prone. Thus, the overall safety level of the PCC system will be undermined if an unverified garbage collector is included in TCB.

In this paper, we present the formal verification of a conservative variant[4] of the standard mark-sweep garbage collector[3] in the PCC style. The technical highlights and contributions of this paper include:

• Our verification is performed directly on the machine level with the Hoare-style program logic *Stack-based Certified Assembly Programming* (SCAP)[5]. Thus, we are forced to specify and prove the concrete machine-level behavior of the collector, which though often abstracted out in a high-level language implementation, is indispensable by the PCC-style mutators for the safe usage of the collector. On the other hand, as we will show later, the major effort we spend in proving the collector is on the manipulation of language independent properties like heap predicates and finite sets. Thus, verifying a low-level language, especially C, implementation of the collector would require almost the same effort. And a certifying compiler, if one could be constructed, is still needed for making the C-level verification useful to other PCC packages.

• We use *separation logic*[6] to define the machine-level heap predicates for building the specification of the collector. With separation-logic operators, we define new heap predicates to assert reachability on a concrete memory model, which could evade troubles caused by cyclic links in heap objects. Our specification of the collector states that all the live objects are preserved, and all the unreachable objects (in a conservative definition) are collected, hence is strong enough to preserve the safety of common mutator programs.

• Our verification is fully mechanized within the Coq proof assistant[7]. We follow the Foundational-PCC (FPCC)[8] style to give the proof of the collector along side the soundness proof of the whole verification framework. The collector's proof in SCAP can be ported to an open FPCC framework[9] without much difficulty as shown in [9], which enables our collector's possible interoperability with other mutator systems. Our verification can also be used as a model for other PCC systems where the verification of a garbage collector is necessary.

• During our verification of the collector, we make important improvements to the verification framework. We build a sound verification condition generator (VC-Gen) for the SCAP system, which alleviates the user from understanding the details of the SCAP rules, and introduces the potential for automatic verification of basic safety properties. We also build automatic tactics in Coq to deal with proof goals involving separation logic

Regular Paper

Supported by the National Natural Science Foundation of China under Grant No. 60673126 (Verification and Compilation of Software Safety); Intel China Research Center.

```
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);
}
```

```
gc() {
 mark_field(root);
 while(!stack_empty()) {
  ptr=stack_pop();
  mark_field(ptr->first);
  mark_field(ptr->second);
 }
for(addr=ST; addr<ED; addr++)
 if (markbit(addr)==WHITE) {
  addr->first=freeptr;
  freeptr=addr;
 }
 else markbit(addr)=WHITE;
}
```

```
mark_field(val) {
 if (val<ST || val>=ED) return;
 if (val mod 8 != 0) return;
 if (markbit(val)==BLACK) return;
 markbit(val)=BLACK;
 stack_push(val);
}
alloc() {
 if (freeptr==NULL) gc();
 if (freeptr==NULL) loop();
 l=freeptr;
 freeptr=freeptr->first;
 return l;
}
```

Fig.1. Pseudo code of our collector.

and finite set operations. These improvements greatly simplify the proof construction.

The work presented in this paper is a part of our ongoing project[10] to build an unified framework for certifying the mutator-collector interaction based on modern garbage collectors like the incremental ones and the generational ones. As an extension to the work in this paper, we have successfully linked a TAL with the verified collector[11] following the ideas in [9], however, this part is beyond the scope of this paper. We also believe that our improvements to the verification framework will benefit future researches in this field.

The rest of this section is a brief introduction to the collector we verified. Then, we introduce in Section 2 the verification framework and our extension. In Section 3, we discuss the heap predicates that formalize the heap during a collection, and give the safety specification of the collector using these predicates. Section 4 outlines the methodology we used to certify the collector in the SCAP framework. In Section 5, we discuss and evaluate our Coq implementation. Finally, we talk about the related work in Section 6 and give the conclusion in Section 7.

Note that since all the lemmas and theorems in this paper are mechanically proved in Coq, we skip their detailed proofs, interested readers may refer to [12].

1.1 Collector

Fig.1 shows the pseudo code of the garbage collector we verified, a conservative variant of the standard stop-the-world mark-sweep collector. To simplify the problem, we adopt a heap layout shown in Fig.2: the size of each heap object is two words; all heap objects reside in a continuous subheap; the collector also keeps the mark bits, a mark stack and a record of global variables. This simplification implies that our allocator only works with mutators that always require two-word objects.

The conservative valid-pointer check in the `mark_field` procedure follows [4]. A value is considered as a valid object pointer only if it is inside the boundaries of the allocatable heap, and aligns to the size of two words.

A GC cycle begins with tracing and marking the objects reachable from root. A mark stack is used to temporarily store the marked objects whose fields are to be examined. Then, the collector examines the mark bits of all the objects, reclaims those unreachable objects to form a free list, and resets all the mark bits. The free list can then be used by the allocator when the GC cycle is finished.

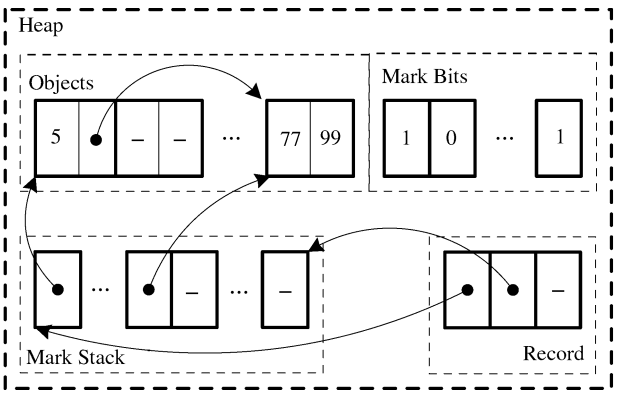

Fig.2. Collector's heap layout.

2 Verification Framework

We discuss in this section the three components of our verification framework: a MIPS-like abstract machine model; a Hoare-style program logic, the SCAP system; and a heap specification language with separation-logic operators.

The whole framework is formalized within a mechanized meta logic, the Calculus of inductive Construction (CiC)[13]. CiC is a higher-order predicate logic extended with inductive definitions. The CiC terms in this paper are written with the standard logic notations. We let *Prop* be the universe of all logical propositions, and let *Set* be the universe of all computational terms.

2.1 Abstract Machine

We show the syntax of the abstract machine in Fig.3. A program $\mathbb{P}$ is a triple of a code heap $\mathbb{C}$, a machine

state $\mathbb{S}$ and an instruction sequence $\mathbb{I}$. A code heap $\mathbb{C}$ is a map from code label `f` to instruction sequence $\mathbb{I}$. A machine state $\mathbb{S}$ contains a data heap $\mathbb{H}$ and a register file $\mathbb{R}$. A data heap $\mathbb{H}$ is a 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, with `r0` always maps to 0. We use the standard MIPS register aliases[14] in the rest of the paper. A command c is a non-controlflow instruction such as a register add or a heap load. An instruction sequence $\mathbb{I}$, or code block, is a series of commands followed by an unconditional jump instruction. For simplicity, we separate the code heap $\mathbb{C}$ from the mutable data heap $\mathbb{H}$. Also, we use instruction sequence instead of the standard `pc` register, and this results in the additional return address $\mathtt{f}_{\mathtt{ret}}$ in the jump and link instruction `jal f`, $\mathtt{f}_{\mathtt{ret}}$, which can be viewed as a macro for the MIPS instruction pair `jal f` and `j` $\mathtt{f}_{\mathtt{ret}}$.

(*Prog*)	$\mathbb{P}$	::=	$(\mathbb{C}, \mathbb{S}, \mathbb{I})$
(*CdHeap*)	$\mathbb{C}$	::=	$\{\mathtt{f} \rightsquigarrow \mathbb{I}\}^*$
(*State*)	$\mathbb{S}$	::=	$(\mathbb{H}, \mathbb{R})$
(*Heap*)	$\mathbb{H}$	::=	$\{\mathtt{l} \rightsquigarrow \mathtt{w}\}^*$
(*RFile*)	$\mathbb{R}$	::=	$\{\mathtt{r} \rightsquigarrow \mathtt{w}\}^*$
(*Reg*)	`r`	::=	$\{\mathtt{r}k\}^{k \in \{0 \ldots 31\}}$
(*Wd, Lab*)	`w`,`f`	::=	$0 \mid 1 \mid 2 \mid 3 \mid \ldots$
(*Address*)	`l`	::=	$0 \mid 4 \mid 8 \mid 12 \mid \ldots$
(*ISeq*)	$\mathbb{I}$	::=	$c; \mathbb{I} \mid$ `beq` $\mathtt{r}_s, \mathtt{r}_t$, `f`; $\mathbb{I}$
		$\mid$	`bne` $\mathtt{r}_s, \mathtt{r}_t$, `f`; $\mathbb{I}$
		$\mid$	`j f` $\mid$ `jal f`, $\mathtt{f}_{\mathtt{ret}} \mid$ `jr` $\mathtt{r}_s$
(*Comm*)	c	::=	`addu` $\mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t \mid$ `addiu` $\mathtt{r}_d, \mathtt{r}_s$, `w`
		$\mid$	`subu` $\mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t \mid$ `srl` $\mathtt{r}_d, \mathtt{r}_s$, `1`
		$\mid$	`sltu` $\mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t \mid$ `andi` $\mathtt{r}_d, \mathtt{r}_s$, `7`
		$\mid$	`lw` $\mathtt{r}_d$, `w`$(\mathtt{r}_s) \mid$ `sw` $\mathtt{r}_s$, `w`$(\mathtt{r}_d)$

Fig.3. Abstract machine syntax.

Following [14], we give the small step operational semantics of the abstract machine in Fig.4. 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 $\mathbb{S}$. Note that for an `lw` or `sw` command, if the source address is not in the domain of the heap, the next state is undefined. The next step of a program is undefined if it jumps to a wrong label, or the next state of its initial command is undefined.

2.2 Program Logic

The SCAP system[5] is a Hoare-style program logic for modular verification of assembly code with procedure call/return. Procedures can be verified separately and then linked together to form a verified program. As shown in Fig.5, an SCAP code specification is a partial correctness specification containing a pair of precondition p and guarantee g. p reassembles the precondition in Hoare logic, while g relates the entry state of the code block to the return state of the corresponding procedure. Thus the g at the entry label of a procedure asserts the guarantee of the procedure, as we will see in the later sections. A code heap specification Ψ maps the labels of the code blocks to their specifications. We also define the implication relation $\Rightarrow$ on SCAP specifications.

if $\mathbb{I}$ =	then $(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{I}) \longmapsto$
`j f`	if $\mathtt{f} \in \mathsf{dom}(\mathbb{C})$, $(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{C}(\mathtt{f}))$
`jal f`, $\mathtt{f}_{\mathtt{ret}}$	if $\mathtt{f} \in \mathsf{dom}(\mathbb{C})$, $(\mathbb{C}, (\mathbb{H}, \mathbb{R}\{\mathtt{r}_{31} \rightsquigarrow \mathtt{f}_{\mathtt{ret}}\}), \mathbb{C}(\mathtt{f}))$
`jr` $\mathtt{r}_s$	if $\mathbb{R}(\mathtt{r}_s) \in \mathsf{dom}(\mathbb{C})$, $(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{C}(\mathbb{R}(\mathtt{r}_s)))$
`beq` $\mathtt{r}_s, \mathtt{r}_t$, `f`; $\mathbb{I}'$	if $\mathbb{R}(\mathtt{r}_s) \neq \mathbb{R}(\mathtt{r}_t)$, $(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{I}')$, else if $\mathtt{f} \in \mathsf{dom}(\mathbb{C})$, $(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{C}(\mathtt{f}))$
`bne` $\mathtt{r}_s, \mathtt{r}_t$, `f`; $\mathbb{I}'$	if $\mathbb{R}(\mathtt{r}_s) = \mathbb{R}(\mathtt{r}_t)$, $(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{I}')$, else if $\mathtt{f} \in \mathsf{dom}(\mathbb{C})$, $(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{C}(\mathtt{f}))$
$c; \mathbb{I}'$	if $\mathsf{Next}_c((\mathbb{H}, \mathbb{R})) = \mathbb{S}'$, $(\mathbb{C}, \mathbb{S}', \mathbb{I}')$

if c =	then $\mathsf{Next}_c((\mathbb{H}, \mathbb{R}))$ =
`addu` $\mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s) + \mathbb{R}(\mathtt{r}_t)\})$
`addiu` $\mathtt{r}_d, \mathtt{r}_s$, `w`	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s) + \mathtt{w}\})$
`subu` $\mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s) - \mathbb{R}(\mathtt{r}_t)\})$
`srl` $\mathtt{r}_d, \mathtt{r}_s$, `1`	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s)/2\})$
`sltu` $\mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow k\})$ if $\mathbb{R}(\mathtt{r}_s) < \mathbb{R}(\mathtt{r}_t)$, $k = 1$, else $k = 0$
`andi` $\mathtt{r}_d, \mathtt{r}_s$, `7`	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s) \bmod 8\})$
`lw` $\mathtt{r}_d$, `w`$(\mathtt{r}_s)$	if $(\mathbb{R}(\mathtt{r}_s) + \mathtt{w}) \in \mathsf{dom}(\mathbb{H})$, $(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{H}(\mathbb{R}(\mathtt{r}_s) + \mathtt{w})\})$
`sw` $\mathtt{r}_s$, `w`$(\mathtt{r}_d)$	if $(\mathbb{R}(\mathtt{r}_d) + \mathtt{w}) \in \mathsf{dom}(\mathbb{H})$, $(\mathbb{H}\{\mathbb{R}(\mathtt{r}_d) + \mathtt{w} \rightsquigarrow \mathbb{R}(\mathtt{r}_s)\}, \mathbb{R})$

Fig.4. Abstract machine semantics.

(*SPred*)	p, q	$\in$	$State \to Prop$
(*Guar*)	g	$\in$	$State \to State \to Prop$
(*BSpec*)	σ	::=	(p, g)
(*CHSpec*)	Ψ	::=	$\{\mathtt{l} \rightsquigarrow \sigma\}^*$
$\mathsf{WFST}(0, g, \mathbb{S}, \Psi)$		$\stackrel{\text{def}}{=}$	$\neg\exists \mathbb{S}'.g\ \mathbb{S}\ \mathbb{S}'$
$\mathsf{WFST}(n, g, \mathbb{S}, \Psi)$		$\stackrel{\text{def}}{=}$	$\forall \mathbb{S}'.g\ \mathbb{S}\ \mathbb{S}' \to \mathbb{S}'.\mathbb{R}(\mathtt{ra}) \in \mathsf{dom}(\Psi) \wedge p'\ \mathbb{S}' \wedge \mathsf{WFST}(n-1, g', \mathbb{S}', \Psi)$ where $(p', g') = \Psi(\mathbb{S}'.\mathbb{R}(\mathtt{ra}))$
$(p, g) \Rightarrow (p', g')$		$\stackrel{\text{def}}{=}$	$\forall \mathbb{S}, p\ \mathbb{S} \to (p'\ \mathbb{S} \wedge (\forall \mathbb{S}'.g'\ \mathbb{S}\ \mathbb{S}' \to g\ \mathbb{S}\ \mathbb{S}'))$

Fig.5. SCAP specification constructs.

A set of operational-semantics-based inference rules, as partly shown in Fig.6, is provided to build a well-formed program from bottom-up. The abstract stack predicate $\mathsf{WFST}(n, g, \mathbb{S}, \Psi)$ in Fig.5 generally asserts that the current procedure can return to the label in `ra` of its return state. n indicates the number of stack frames, once it becomes 0, the current code will never return. A detailed knowledge of these rules and WFST is not required for understanding the rest of the paper, interested readers may refer to [5].

Lemma 1 states that if a code block is well-formed with some σ', it is also well-formed with a stronger assert σ, and the proof of a well-formed code block can be lifted from a local Ψ to a global Ψ'.

Lemma 1 (Weakening).

1. *If* $\sigma \Rightarrow \sigma'$ *and* $\Psi; \sigma' \vdash \mathbb{I}$, *then*: $\Psi; \sigma \vdash \mathbb{I}$.
2. *If* $\Psi \subseteq \Psi'$ *and* $\Psi; \sigma \vdash \mathbb{I}$, *then*: $\Psi'; \sigma \vdash \mathbb{I}$.

Theorem 1 ensures that a well-formed program will run safely without stopping at any program step undefined in Fig.4.

$\boxed{\Psi \vdash \mathbb{P}}$ (*Well-Formed Program*)

$$\frac{\Psi \vdash \mathbb{C} : \Psi \; p\mathbb{S} \;\; \Psi; (p,g) \vdash \mathbb{I} \quad \exists n.\mathsf{WFST}(n, g, \mathbb{S}, \Psi)}{\Psi \vdash (\mathbb{C}, \mathbb{S}, \mathbb{I})} \quad (\textsc{prog})$$

$\boxed{\Psi \vdash \mathbb{C} : \Psi'}$ (*Well-Formed Code Heap*)

$$\frac{\Psi' \, \Psi(\mathtt{f}) \vdash \mathbb{C}(\mathtt{f}) \qquad \forall \mathtt{f} \in \mathsf{dom}(\Psi)}{\Psi' \vdash \mathbb{C} : \Psi} \quad (\textsc{cdhp})$$

$\boxed{\Psi; (p,g) \vdash \mathbb{I}}$ (*Well-Formed Instruction Sequence*)

$$\frac{\begin{array}{l}\Psi; (p,g) \vdash \mathbb{I} \\ \forall \mathbb{S}.\; p\;\mathbb{S} \to \exists \mathbb{S}'.\mathsf{Next}_c(\mathbb{S}) = \mathbb{S}' \wedge p'\;\mathbb{S}' \wedge \\ \quad \dfrac{\forall \mathbb{S}''.\; g'\;\mathbb{S}'\;\mathbb{S}'' \to g\;\mathbb{S}\;\mathbb{S}''}{\Psi; (p,g) \vdash c; \mathbb{I}}\end{array}}{} \quad (\textsc{seq})$$

$$\frac{\begin{array}{l}\Psi(\mathtt{f}) = (p', g') \\ \forall \mathbb{S}.\; p\;\mathbb{S} \to p'\;\mathbb{S} \wedge \forall \mathbb{S}'.\; g'\;\mathbb{S}\;\mathbb{S}' \to g\;\mathbb{S}\;\mathbb{S}'\end{array}}{\Psi; (p,g) \vdash \mathtt{j\ f}} \quad (\textsc{j})$$

$$\frac{\begin{array}{l}\Psi(\mathtt{f}) = (p', g') \quad \Psi(\mathtt{f}_{\mathtt{ret}}) = (p'', g'') \\ \forall (\mathbb{H}, \mathbb{R}).\; p\;(\mathbb{H}, \mathbb{R}) \to\; p'\;(\mathbb{H}, \mathbb{R}\{\mathtt{ra} \rightsquigarrow \mathtt{f}_{\mathtt{ret}}\}) \wedge \\ \quad \forall \mathbb{S}'.\; g'\;(\mathbb{H}, \mathbb{R}\{\mathtt{ra} \rightsquigarrow \mathtt{f}_{\mathtt{ret}}\})\;\mathbb{S}' \to \\ \quad\; p''\;\mathbb{S}' \wedge \forall \mathbb{S}''.\; g''\;\mathbb{S}'\;\mathbb{S}'' \to g\;(\mathbb{H}, \mathbb{R})\;\mathbb{S}''\;\forall (\mathbb{H}, \mathbb{R}), (\mathbb{H}', \mathbb{R}').\end{array}}{\begin{array}{l} g'\;(\mathbb{H}, \mathbb{R})\;(\mathbb{H}', \mathbb{R}') \to \mathbb{R}(\mathtt{ra}) = \mathbb{R}'(\mathtt{ra}) \\ \hline \Psi; (p,g) \vdash \mathtt{jal\ f}, \mathtt{f}_{\mathtt{ret}}\end{array}} \quad (\textsc{call})$$

$$\frac{\forall \mathbb{S}.\; p\;\mathbb{S} \to g\;\mathbb{S}\;\mathbb{S}}{\Psi; (p,g) \vdash \mathtt{jr\ ra}} \quad (\textsc{return})$$

Fig.6. SCAP inference rules (excerpt).

Theorem 1 (Soundness). *If* $\Psi \vdash (\mathbb{C}, \mathbb{S}, \mathbb{I})$, *for all natural number* n *there exists a* $(\mathbb{C}, \mathbb{S}', \mathbb{I}')$, *such that* $(\mathbb{C}, \mathbb{S}, \mathbb{I}) \mapsto_n (\mathbb{C}, \mathbb{S}', \mathbb{I}')$.

We extend SCAP by building for it a VCGen, which is partly shown in Fig.7. $\mathsf{wp}(\Psi, \mathbb{I})$ is a code block specification formed from the code heap specification Ψ and the code block $\mathbb{I}$. The predicate $\mathsf{rapred}(\Psi, \mathbb{I})$ makes sure that if $\mathbb{I}$ calls procedure `f`, `f` will keep `ra` unchanged, as required by the CALL rule in Fig.6. Since it is hard to directly embed rapred into wp, we just make it a standalone predicate. With these definitions, we obtain Theorem 2 from Lemma 1 and the CDHP rule in Fig.6.

Theorem 2 (VCGen Correctness).

1. *If* $\sigma \Rightarrow \mathsf{wp}(\Psi, \mathbb{I}) \wedge \mathsf{rapred}(\Psi, \mathbb{I})$, *then* $\Psi; \sigma \vdash \mathbb{I}$.
2. *If* $\mathsf{vc}(\Psi, \mathbb{C})$, *then* $\Psi \vdash \mathbb{C}$: Ψ.

With the help of VCGen, the SCAP proof construction is lifted onto the code block level. For each block, only the proof of a σ implication is required, instead of the proofs of σ implications for each instructions following the rules in Fig.6. This alleviates the user from understanding the complex details of the SCAP rules, simplifies the proof construction, and makes the reasoning more natural. Besides, by eliminating the need for instantiating the specifications for each instruction inside a code block, the VCGen introduces the potential of automatic proof construction for programs verified against simple properties.

$$\mathsf{vc}(\Psi, \mathbb{C}) \stackrel{\text{def}}{=} \forall \mathtt{f} \in \mathsf{dom}(\Psi).(\Psi(\mathtt{f}) \Rightarrow \mathsf{wp}(\Psi, \mathbb{C}(\mathtt{f}))) \wedge \mathsf{rapred}(\Psi, \mathbb{C}(\mathtt{f}))$$

if $\mathbb{I} =$	then $\mathsf{wp}(\Psi, \mathbb{I}) =$	in case that
$c; \mathbb{I}'$	$(\lambda \mathbb{S}.p\;\mathsf{Next}_c(\mathbb{S}), \lambda \mathbb{S}, \mathbb{S}'.g\;\mathsf{Next}_c(\mathbb{S})\mathbb{S}')$	$\mathsf{wp}(\Psi, \mathbb{I}') = (p, g)$
`j f`	(p, g)	$\Psi(\mathtt{f}) = (p, g)$
`jal f`, $\mathtt{f}_{\mathtt{ret}}$	$(\lambda(\mathbb{H}, \mathbb{R}).p'\;(\mathbb{H}, \mathbb{R}\{\mathtt{ra} \rightsquigarrow \mathtt{f}_{\mathtt{ret}}\}) \wedge (\forall \mathbb{S}'.g'(\mathbb{H}, \mathbb{R}\{\mathtt{ra} \rightsquigarrow \mathtt{f}_{\mathtt{ret}}\})\;\mathbb{S}' \to p''\;\mathbb{S}',$ $\lambda(\mathbb{H}, \mathbb{R}), \mathbb{S}''.\exists \mathbb{S}'.g'\;(\mathbb{H}, \mathbb{R}\{\mathtt{ra} \rightsquigarrow \mathtt{f}_{\mathtt{ret}}\})\;\mathbb{S}' \wedge g''\;\mathbb{S}'\;\mathbb{S}'')$	$\Psi(\mathtt{f}) = (p', g')$ $\Psi(\mathtt{f}_{\mathtt{ret}}) = (p'', g'')$
`jr ra`	$(\lambda \mathbb{S}.\mathsf{True}, \lambda \mathbb{S}, \mathbb{S}'.\mathbb{S} = \mathbb{S}')$	

if $\mathbb{I} =$	then $\mathsf{rapred}(\Psi, \mathbb{I}) =$	in case that
$c; \mathbb{I}'$	$\mathsf{rapred}(\Psi, \mathbb{I}')$	
`j f`	True	
`jal f`, $\mathtt{f}_{\mathtt{ret}}$	$\forall (\mathbb{H}, \mathbb{R}), (\mathbb{H}', \mathbb{R}').\; g'\;(\mathbb{H}, \mathbb{R})\;(\mathbb{H}', \mathbb{R}') \to \mathbb{R}(\mathtt{ra}) = \mathbb{R}'(\mathtt{ra})$	$\Psi(\mathtt{f}) = (p', g')$
`jr ra`	True	

Fig.7. VCGen (excerpt).

2.3 Heap Predicate

To specify the behavior of a collector, we introduce the separation-logic operators. Separation logic was originally proposed as an extension to Hoare logic for reasoning mutable data structures[6]. In order to employ it in the SCAP system, we directly embed a set of separation-logic operators as heap predicate constructors using CiC, as shown in Fig.8. Thus, the SCAP specifications p and g, which are state predicates, may contain heap specifications build with these predicates.

We write $\mathbb{H} \Vdash A$ if $(A\;\mathbb{H})$ is a valid proposition in CiC. The definitions are consistent with the semantics

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

Fig.8. Separation logic operators.

described in [6]. For simplicity of presentation, we use the same notation to denote the separation-logic connectives and the logic connectives in CiC. We also use 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 $\exists x.\ Nat.\mathtt{l} \mapsto x$ and so on.

The inductively defined iterated separating conjunction $\forall_* x \in S.\ A$, taken from [15], asserts that the heap can be split into a finite set of subheaps according to the finite set S of natural numbers, while for each member n of S, there is a unique subheap on which $A[n/x]$ holds.

We prove as lemmas the properties of these separation-logic operators (axioms in [6]), and provide Lemma 2 for linking the heap predicates with the operational semantics of the `lw` and `sw` instructions.

Lemma 2 (Heap Operations).

1. *If* $\mathbb{H} \Vdash \mathtt{l} \mapsto \mathtt{w} * \mathsf{true}$, *then*: $\mathbb{H}(\mathtt{l}) = \mathtt{w}$.
2. *If* $\mathbb{H} \Vdash \mathtt{l} \mapsto - * A$, *then*: $\mathbb{H}\{\mathtt{l} \rightsquigarrow \mathtt{w}\} \Vdash \mathtt{l} \mapsto \mathtt{w} * A$.

Most of our separation logic lemmas follow the frame-rule[6] style to support local reasoning. For example, the universal quantification of the heap predicate A in Lemma 2 ensures that a local heap update never affects the rest of the heap. We also use this kind of heap predicate quantification in our specifications of the collector to achieve local reasoning in sub-procedures.

3 Specification

We present in this section the SCAP specification of the collector we proved. First, we give the reachability predicate on which our specification is built. Then, we model the collector's heap with separation-logic-based heap predicates. Finally, we give the specification of the collector's interface, the `alloc` procedure.

3.1 Specification Interface

We define in Fig.9 the heap conceptions that should be consistent between the collector and the mutator. The constant address null is set to be 0, while the lower and upper boundaries of the collector's allocatable heap, st and ed, should both align 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 it is reachable from the pointers in the heap object at `l`. The predicate $\mathsf{rchrt}(\mathbb{S}, \mathtt{l})$ asserts that `l` points to a live heap object (reachable from the root) in the state $\mathbb{S}$. For the sake of simplicity, we consider the case of a single root register `t0`, which can be extended to support more root registers without much difficulty.

The heap predicate $\mathsf{obj\_hp}(S', S)$ in Fig.9 asserts an object heap with reachability information. A value `w` is valid with a pointer set S ($\mathsf{ok\_val}(S, \mathtt{w})$) if it is either a member of S or a non-pointer value. $\mathsf{ok\_fld}(S, \mathtt{l})$ asserts that the field value at `l` is valid with S. A valid object ($\mathsf{ok\_obj}(S, \mathtt{l})$) is composed of two valid fields. Thus, the proposition $\mathbb{H} \Vdash \mathsf{obj\_hp}(S', S)$ is true only if $\mathbb{H}$ is formed exactly with the heap objects in S, and all its field values are valid with S'. It is noteworthy that once $\mathbb{H} \Vdash \mathsf{obj\_hp}(S, S)$ holds, $\mathbb{H}$ will be a *closed* heap with no outgoing pointers. The relationship between $\mathsf{obj\_hp}$ and reach is stated in Lemma 3.

$$
\begin{array}{rcl}
\mathsf{nul} & ::= & 0 \\
\mathsf{st},\ \mathsf{ed} & ::= & 8 \mid 16 \mid 24 \mid \ldots \\
\mathsf{ptrs} & \stackrel{\mathrm{def}}{=} & \{\mathtt{l} \mid (\mathtt{l} \bmod 8 = 0) \wedge (\mathsf{st} \leqslant \mathtt{l} < \mathsf{ed})\} \\
\mathsf{vptr}(\mathtt{l}) & \stackrel{\mathrm{def}}{=} & \mathtt{l} \in \mathsf{ptrs} \\
\mathsf{rchrt}((\mathbb{H}, \mathbb{R}), \mathtt{l}) & \stackrel{\mathrm{def}}{=} & \mathsf{reach}(\mathbb{H}, \mathbb{R}(\mathtt{t0}), \mathtt{l})
\end{array}
$$

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

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

$$
\begin{array}{l}
\mathsf{ok\_val}(S, \mathtt{w}) \stackrel{\mathrm{def}}{=} \mathtt{w} \in \mathsf{ptrs} \ \rightarrow\ \mathtt{w} \in S \\
\mathsf{ok\_fld}(S, \mathtt{l}) \stackrel{\mathrm{def}}{=} \exists\, \mathtt{w}.!(\mathsf{ok\_val}(S, \mathtt{w})) * \mathtt{l} \mapsto \mathtt{w} \\
\mathsf{ok\_obj}(S, \mathtt{l}) \stackrel{\mathrm{def}}{=} \mathsf{ok\_fld}(S, \mathtt{l}) * \ \mathsf{ok\_fld}(S, \mathtt{l}+4) \\
\mathsf{obj\_hp}(S', S) \stackrel{\mathrm{def}}{=} \forall_* x \in S.\ \mathsf{ok\_obj}(S', x)
\end{array}
$$

Fig.9. Specification interface.

Lemma 3 (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* $\mathtt{l}' \in S$.

By virtue of the $\mathsf{obj\_hp}$ predicate, we can avoid the problem caused by the possible cyclic links in the object heap when the reach predicate is directly used. This benefit greatly simplifies our verification of the mark phase of the garbage collection.

3.2 Collector's Heap

The object heap in the collector's heap layout (described in Subsection 1.1) is formalized with the $\mathsf{obj\_hp}$ predicate introduced in the previous subsection.

The rest of the formalization of the collector's heap is listed in Fig.10. The free list is inductively defined by the set of free objects. The heap that storing the mark bits n for an objects set S is specified with $\mathsf{mbits}(S, n)$, which asserts that there is a mark bit n for each member x of S at the address $(\mathsf{ed} + (x - \mathsf{st})/2)$. The mark stack $\mathsf{mstk}(S, x, y, z)$ is represented by a continuous memory area, with the lower part containing pointers to a set of objects. The collector's record gcinfo stores three pointers to the collector's internal data structures. $\mathsf{flist}(\mathbb{R}, S)$ and $\mathsf{mstack}(\mathbb{R}, S)$ are used when the pointers are loaded to the corresponding registers. The free list header is stored in `t5`, and the stack pointers (`top`, `bot` and `buf`) are stored in `t2`, `t3` and `t4`.

3.3 Specification of `alloc`

We formalize the SCAP specification of the collector's main procedure `alloc` in Fig.11. The precondition p_{ALLOC} is defined in terms of the predicate $\mathsf{gc\_inv}$. The

first conjunct of $\mathsf{gc\_inv}(\mathbb{S}, \mathbb{H}, \mathtt{l_f})$ asserts that the allocatable object set ptrs is composed of the allocated object set B and the free object set F. The following two heap propositions ensure that the heap contains an allocated subheap $\mathbb{H}$, a free list with head at $\mathtt{l_f}$, an empty mark stack and the record of pointers. This faithfully follows the heap layout in Fig.2.

$$\mathsf{eq}(\mathbb{H}) \stackrel{\text{def}}{=} \lambda\mathbb{H}'.\mathbb{H}' = \mathbb{H}$$
$$\mathsf{flst}(\mathtt{l}, \emptyset) \stackrel{\text{def}}{=} !(\mathtt{l} = \mathsf{nul})$$
$$\mathsf{flst}(\mathtt{l}, \{\mathtt{l}\} \cup S) \stackrel{\text{def}}{=} !(\mathtt{l} \neq \mathsf{nul}) * \exists \mathtt{l}'.\mathtt{l} \mapsto -, \mathtt{l}' * \mathsf{flst}(\mathtt{l}', S - \mathtt{l})$$
$$\mathsf{hdr}(\mathtt{l}) \stackrel{\text{def}}{=} (\mathsf{ed} + (\mathtt{l} - \mathsf{st})/2)$$
$$\mathsf{mbits}(S, n) \stackrel{\text{def}}{=} \forall_* x \in S.\mathtt{hdr}(x) \mapsto n$$
$$\mathsf{array\_set}(\mathtt{l}, \emptyset) \stackrel{\text{def}}{=} \mathsf{emp}$$
$$\mathsf{array\_set}(\mathtt{l}, \{\mathtt{w}\} \cup S) \stackrel{\text{def}}{=} \mathtt{l} \mapsto \mathtt{w} * \mathsf{array\_set}(\mathtt{l} + 4, S - \mathtt{w})$$
$$\mathsf{buffer}(\mathtt{l}, \mathtt{l}') \stackrel{\text{def}}{=} \forall_* x \in \{x \mid x \bmod 4 = 0 \wedge \mathtt{l} \leqslant x < \mathtt{l}'\}.x \mapsto -$$
$$\mathsf{mstk}(S, x, y, z) \stackrel{\text{def}}{=} !(y - x = \mathsf{size}(S)) * \mathsf{array\_set}(x, S) * \mathsf{buffer}(y, z)$$
$$\mathsf{gcinfo}(\mathtt{l}, \mathtt{l}_1, \mathtt{l}_2, \mathtt{l}_3) \stackrel{\text{def}}{=} \mathtt{l} \mapsto \mathtt{l}_1 * \mathtt{l} + 4 \mapsto \mathtt{l}_2 * \mathtt{l} + 8 \mapsto \mathtt{l}_3$$
$$\mathsf{flst}(\mathbb{R}, S) \stackrel{\text{def}}{=} \mathsf{flst}(\mathbb{R}(\mathtt{t5}), S)$$
$$\mathsf{mstack}(\mathbb{R}, S) \stackrel{\text{def}}{=} \mathsf{mstk}(S, \mathbb{R}(\mathtt{t3}), \mathbb{R}(\mathtt{t2}), \mathbb{R}(\mathtt{t4}))$$

Fig.10. Auxiliary heap definitions.

$$\begin{aligned}
&\mathsf{gc_inv}((\mathbb{H}, \mathbb{R}), \mathbb{H}_\mathrm{T}, \mathtt{l_f}) \stackrel{\text{def}}{=} \exists B, F, \mathtt{l_b}, \mathtt{l_t}.\\
&\quad B \cup F = \mathsf{ptrs} \wedge\\
&\quad \mathbb{H} \Vdash \mathsf{ed}(\mathbb{H}_\mathrm{T}) * \mathsf{flst}(\mathtt{l_f}, F) * \mathsf{mbits}(\mathsf{ptrs}, 0) *\\
&\qquad \mathsf{mstk}(\emptyset, \mathtt{l_b}, \mathtt{l_b}, \mathtt{l_t}) * \mathsf{gcinfo}(\mathbb{R}(\mathtt{t1}), \mathtt{l_b}, \mathtt{l_t}, \mathtt{l_f}) \wedge\\
&\quad \mathbb{H}_\mathrm{T} \Vdash \mathsf{obj_hp}(\mathsf{ptrs}, B)\\
&\mathsf{saved_regs} \stackrel{\text{def}}{=} \{\mathtt{s0} \cdots \mathtt{s7}, \mathtt{ra}, \mathtt{t0}, \mathtt{t1}\}\\
&\mathsf{reg_ok}((\mathbb{H}, \mathbb{R}), (\mathbb{H}', \mathbb{R}')) \stackrel{\text{def}}{=} \forall \mathtt{r} \in \mathsf{saved_regs}.\ \mathbb{R}(\mathtt{r}) = \mathbb{R}'(\mathtt{r})\\
&p_\mathrm{ALLOC} \stackrel{\text{def}}{=} \lambda\mathbb{S}.\exists\mathbb{H}, \mathtt{l_f}.\mathsf{gc_inv}(\mathbb{S}, \mathbb{H}, \mathtt{l_f})\\
&g_\mathrm{ALLOC} \stackrel{\text{def}}{=} \lambda\mathbb{S}, \mathbb{S}'.\mathsf{reg_ok}(\mathbb{S}, \mathbb{S}') \wedge\\
&\quad \forall\mathbb{H}, \mathtt{l_f}.\mathsf{gc_inv}(\mathbb{S}, \mathbb{H}, \mathtt{l_f}) \rightarrow \exists\mathbb{H}', \mathtt{l}'_\mathtt{f}.\ \mathsf{gc_inv}(\mathbb{S}', \mathbb{H}', \mathtt{l}'_\mathtt{f}) \wedge\\
&\qquad (\mathtt{l_f} = 0 \rightarrow \exists\mathbb{H}_\mathrm{T}.\ \mathsf{dom}(\mathbb{H}_\mathrm{T}) = \{\mathtt{l}, \mathtt{l} + 4 \mid \mathsf{rchrt}(\mathbb{S}, \mathtt{l})\} \wedge\\
&\qquad\quad \mathbb{H} \Vdash \mathsf{eq}(\mathbb{H}_\mathrm{T}) * \mathsf{true} \wedge\\
&\qquad\quad \mathbb{H}' \Vdash \mathsf{eq}(\mathbb{H}_\mathrm{T}) * \mathbb{S}'.\mathbb{R}(\mathtt{v0}) \mapsto -, -) \wedge\\
&\qquad (\mathtt{l_f} \neq 0 \rightarrow \mathbb{H}' \Vdash \mathsf{eq}(\mathbb{H}) * \mathbb{S}'.\mathbb{R}(\mathtt{v0}) \mapsto -, -)
\end{aligned}$$

Fig.11. Collector specification.

The guarantee g_ALLOC comprises two parts. In the first part, $\mathsf{reg\_ok}(\mathbb{S}, \mathbb{S}')$ asserts that the MIPS callee-saved registers (s0 to s7, and ra), together with the root register t0 and the record pointer t1, are identical in the two states.

The second part asserts that gc_inv is preserved in $\mathbb{S}$ and $\mathbb{S}'$. If the free list is empty ($\mathtt{l_f} = 0$) at the entry state of alloc, a garbage collection will remove all unreachable objects in $\mathbb{H}$. The allocated subheap $\mathbb{H}'$ at the return state thus contains exactly the untouched live object heap $\mathbb{H}_\mathrm{T}$ plus a new object at $\mathbb{S}'.\mathbb{R}(\mathtt{v0})$. Otherwise, $\mathbb{H}'$ is simply formed by extending $\mathbb{H}$ with a new object if the free list is not empty in $\mathbb{S}$.

A mutator program accesses only the live parts of the allocated object heap. A proper specification of a mutator only asserts the values or data structures on this subheap. Since the guarantee g_ALLOC ensures that the live object heap is preserved between the entry/return states of the collector, the validity of any mutator side specification is preserved. In this sense, our GC specification is strong enough to preserve the safety of any common mutator programs. Note that the specification given here is based on the fact that a mark-sweep collector never moves objects, thus it is not suitable for a copying collector[3]. The reader may find a specification for a copying collector in [15].

4 Proof Construction

The SCAP well-formedness proof of the collector is constructed as follows. For each procedure, we firstly form a local code heap specification Ψ_L by composing the specifications for all its code blocks and the procedures it calls. Then, we generate for each code block $\mathbb{I}$ the $\mathsf{wp}(\Psi_\mathrm{L}, \mathbb{I})$, and build their well-formedness proof using Theorem 2.

After verifying each procedure with its local Ψ_L, we sum up the specifications to form a global Ψ_G. With Lemma 1 and the CDHP rule in Fig.6, we prove the well-formedness of the collector's code heap with Ψ_G. A mutator verified in SCAP can then link to this code heap through Lemma 1 and the CDHP rule.

The tricky part of the proof construction is to figure out the correct specification of the collector's internal code blocks. We present these specifications in the following subsection. After that, we briefly discuss the theorem proving issue.

4.1 Code Heap Specification

The SCAP specifications of the collector's code blocks follow the pattern of $(p_\mathrm{ALLOC}, g_\mathrm{ALLOC})$ in Fig.11. Unfamiliar readers may just get the general idea instead of going through the complex specifications in Figs. 12∼15.

We use the following shorthand throughout this section. $\mathsf{nin\_rid}(\mathbb{R}, \mathbb{R}', R)$ asserts that any register which is not a member of the register set R is identical in $\mathbb{R}$ and $\mathbb{R}'$. $\mathsf{sted\_ok}(\mathbb{R})$ asserts that the allocatable heap boundaries are loaded into the corresponding registers in $\mathbb{R}$. We store st in k0 and ed in k1.

$$\mathsf{nin\_rid}(\mathbb{R}, \mathbb{R}', R) \stackrel{\text{def}}{=} \forall \mathtt{r} \notin R.\mathbb{R}(\mathtt{r}) = \mathbb{R}'(\mathtt{r})$$
$$\mathsf{sted\_ok}(\mathbb{R}) \stackrel{\text{def}}{=} \mathbb{R}(\mathtt{k0}) = \mathsf{st} \wedge \mathbb{R}(\mathtt{k1}) = \mathsf{ed}.$$

Mark Stack Operations. We show the assembly code with specifications of the stack-related procedures (is_empty, push and pop) in Fig.12. Each precondition generally asserts that there is a mark stack in the entry

state. The guarantees ensure that if the preconditions are satisfied, the intended operations on the stack are performed. Note that the universally quantified heap predicate A in the guarantees ensures the frame-rule style local reasoning.

Mark an Object. The `mark_field` procedure is also shown in Fig.12. The precondition p_{MFLD} requires that either the argument $\mathbb{R}(\texttt{a0})$ is a non-pointer value, or the heap predicate contains enough information for checking the mark bit of $\mathbb{R}(\texttt{a0})$ and performing stack push. The guarantee g_{MFLD} ensures that the procedure only modifies a few registers if the argument $\mathbb{R}(\texttt{a0})$ is a non-pointer value or a pointer to a marked object. Otherwise, g_{MFLD} guarantees that $\mathbb{R}(\texttt{a0})$ is marked and pushed to the mark stack.

Mark Phase. We show in Fig.13 the code blocks implementing the mark phase of the collection. The specifications of these code blocks follow the same pattern. Each precondition asserts that the corresponding state predicates (mpre, minv, mfst, or msnd) is satisfied. On the other hand, each guarantee ensures that if the corresponding state predicate is satisfied, the `mark` procedure will return on a state satisfying mpost.

On the entry state of mark, as asserted by mpre, all objects are allocated and the heap boundaries are loaded into the corresponding registers. After marking the root register `t0`, we reach the state where the mark loop invariant minv holds. Following the tri-color abstraction[3], minv asserts that the object set ptrs is divided into the sets of *black*, *gray* and *white* objects, which are separated by their mark bits and the mark stack. The *black* and *gray* objects are reachable from root, and the reachability between the three sets are asserted by the obj_hp predicates on $\mathbb{H}_{\text{T}}$.

In the mark loop, a *gray* object is popped into `v0`, and the code blocks `mfirst` and `msecond` examine the two fields of this object. The predicate mfst makes no different than minv except for splitting out the object in `v0` from the *gray* object set G. The predicate msnd goes one step further by asserting that if the first field of the object in `v0` is a valid pointer, it will point to a marked object. The call to `mark_field` in `msecond` returns to `mloop`, where we can safely move the object in `v0` into the *black* object set B and get minv again.

There is no more *gray* object on the return state of `mark`. Thus we get a closed subheap in $\mathbb{H}_{\text{T}}$ formed with the live objects in B.

Sweep Phase. In Fig.14, we specify the first two code blocks of `sweep` in the same way we specify the `mark` code blocks. The predicate spre on the entry state of `sweep` is different from mpost only in that we move the unreachable objects out of the allocated subheap $\mathbb{H}_{\text{T}}$.

In the sweep loop invariant sinv, G stands for the objects before the sweep pointer in `a0`. The unreachable objects in G are collected into the free list with set F, and the mark bits of the reachable objects in G are set to 0. Finally, G becomes a superset of ptrs when the sweep loop finishes. Thus all the unreachable objects will be collected into the free list, as asserted by spost. The other two code block specifications are simply generated by the wp function in Subsection 2.2.

Collector. The rest of the codes, including the `gc` procedure and the `alloc` procedure, are shown in Fig.15. The specification of `gc` is a trivial composition of the `mark` and `sweep` specifications. The guarantee g_{GC} asserts that the reachable subheap $\mathbb{H}'_{\text{T}}$ is preserved during

```
is_empty: (p_EMP, g_EMP)
  beq $t3 $t2 empty2
  addiu $v0 $0 0
  jr $ra
empty2 :
  addiu $v0 $0 1
  jr $ra

push: (p_PUSH, g_PUSH)
  addiu $at $t2 4
  sltu $at $t4 $1
  bne $at $0 stk_loop
  sw $a0 0($t2)
  addiu $t2 $t2 4
  jr $ra

pop: (p_POP, g_POP)
  addiu $v0 $0 4
  subu $t2 $t2 $v0
  lw $v0 0($t2)
  jr $ra
stk_loop:
  j stk_loop

mark_field: (p_MFLD, g_MFLD)
  sltu $t9 $a0 $k0   srl1 $at $at
  bne $t9 $0 return  addu $at $at $k1
  sltu $t9 $a0 $k1   lw $t9 0 $at
  beq $t9 $0 return  addiu $v1 $0 1
  subu $at $a0 $k0   beq $t9 $v1 return
  andi7 $t9 $at      sw $v1 0 $at
  bne $t9 $0 return  j push
```

$$p_{\text{EMP}} \stackrel{\text{def}}{=} \lambda(\mathbb{H},\mathbb{R}).\exists S.\mathbb{H} \Vdash \ \mathsf{mstack}(\mathbb{R},S) * \mathsf{true}$$

$$g_{\text{EMP}} \stackrel{\text{def}}{=} \lambda(\mathbb{H},\mathbb{R}),(\mathbb{H}',\mathbb{R}').\exists n.\ (\mathbb{H}',\mathbb{R}') = (\mathbb{H},\mathbb{R}\{\texttt{v0} \rightsquigarrow n\}) \wedge$$
$$\forall S.\ \mathbb{H} \Vdash \ \mathsf{mstack}(\mathbb{R},S) * \mathsf{true} \rightarrow$$
$$(n = 0 \rightarrow S = \emptyset) \wedge (n \neq 0 \rightarrow S \neq \emptyset)$$

$$p_{\text{PUSH}} \stackrel{\text{def}}{=} \lambda(\mathbb{H},\mathbb{R}).\exists S.\ \mathbb{R}(\texttt{a0}) \notin S \wedge \mathbb{H} \Vdash \mathsf{mstack}(\mathbb{R},S) * \mathsf{true}$$

$$g_{\text{PUSH}} \stackrel{\text{def}}{=} \lambda(\mathbb{H},\mathbb{R}),(\mathbb{H}',\mathbb{R}').\mathsf{nin\_rid}(\mathbb{R},\mathbb{R}',\{\texttt{t2},\texttt{at}\}) \wedge$$
$$\forall S, A.\ \mathbb{H} \Vdash \mathsf{mstack}(\mathbb{R},S) * A \rightarrow$$
$$\mathbb{H}' \Vdash \mathsf{mstack}(\mathbb{R}', S + \mathbb{R}(\texttt{a0})) * A$$

$$p_{\text{POP}} \stackrel{\text{def}}{=} \lambda(\mathbb{H},\mathbb{R}).\exists S \neq \emptyset.\ \mathbb{H} \Vdash \mathsf{mstack}(\mathbb{R},S) * \mathsf{true}$$

$$g_{\text{POP}} \stackrel{\text{def}}{=} \lambda(\mathbb{H},\mathbb{R}),(\mathbb{H}',\mathbb{R}').\mathsf{nin\_rid}(\mathbb{R},\mathbb{R}',\{\texttt{t2},\texttt{v0}\}) \wedge$$
$$\forall S \neq \emptyset, A.\ \mathbb{H} \Vdash \mathsf{mstack}(\mathbb{R},S) * A \rightarrow$$
$$\mathbb{R}'(\texttt{v0}) \in S \wedge \mathbb{H}' \Vdash \mathsf{mstack}(\mathbb{R}', S - \mathbb{R}'(\texttt{v0}) * A$$

$$p_{\text{MFLD}} ::= \lambda(\mathbb{H},\mathbb{R}).\mathsf{sted\_ok}(\mathbb{R}) \wedge$$
$$\neg \mathsf{vptr}(\mathbb{R}(\texttt{a0})) \vee$$
$$\mathsf{vptr}(\mathbb{R}(\texttt{a0})) \wedge \exists G.$$
$$(\mathbb{R}(\texttt{a0}) \in G \wedge \mathbb{H} \Vdash \mathsf{mbits}(G,1) * \mathsf{mstack}(\mathbb{R},G) * \mathsf{true}) \vee$$
$$(\mathbb{R}(\texttt{a0}) \notin G \wedge$$
$$\mathbb{H} \Vdash \mathsf{mbits}(G,1) * \mathsf{mstack}(\mathbb{R},G) * \mathsf{hdr}(\mathbb{R}(\texttt{a0})) \mapsto - * \mathsf{true})$$

$$g_{\text{MFLD}} ::= \lambda(\mathbb{H},\mathbb{R}),(\mathbb{H}',\mathbb{R}').\mathsf{nin\_rid}(\mathbb{R},\mathbb{R}',\{\texttt{t2},\texttt{v1},\texttt{at},\texttt{t9}\}) \wedge$$
$$(\neg \mathsf{vptr}(\mathbb{R}(\texttt{a0})) \vee \mathbb{H}(\mathsf{hdr}(\mathbb{R}(\texttt{a0}))) = 1 \rightarrow$$
$$\mathbb{H} = \mathbb{H}' \wedge \mathbb{R}(\texttt{t2}) = \mathbb{R}'(\texttt{t2})) \wedge$$
$$(\forall G, A.\ \mathsf{vptr}(\mathbb{R}(\texttt{a0})) \rightarrow$$
$$\mathbb{H} \Vdash \mathsf{mbits}(G,1) * \mathsf{mstack}(\mathbb{R},G) * \mathsf{hdr}(\mathbb{R}(\texttt{a0})) \mapsto 0 * A \rightarrow$$
$$\mathbb{H}' \Vdash \mathsf{mbits}(G + \mathbb{R}'(\texttt{a0}),1) * \ \mathsf{mstack}(\mathbb{R}', G + \mathbb{R}'(\texttt{a0}) * A)$$

Fig.12. Assembly code with specifications: stack and `mark_field`.

```
mark: (p_MARK, g_MARK)
  addiu $t6 $ra 0
  addiu $a0 $t0 0
  jal mark_field mloop

mloop: (p_MLOOP, g_MLOOP)
  jal is_empty mloop2
mloop2:
  bne $v0 $0 return1
  jal pop mfirst

mfirst: (p_MFST, g_MFST)
  lw $a0 0($v0)
  jal mark_field msecond

msecond: (p_MSND, g_MSND)
  lw $a0 4($v0)
  jal mark_field mloop
```

$p_{\text{MARK}} \stackrel{\text{def}}{=} \lambda\mathbb{S}.\exists\mathbb{H}, A.\ \mathsf{mpre}(\mathbb{S},\mathbb{H},A)$

$g_{\text{MARK}} \stackrel{\text{def}}{=} \lambda\mathbb{S},\mathbb{S}'.\ (\forall\mathbb{H}, A.\ \mathsf{mpre}(\mathbb{S},\mathbb{H},A) \to \mathsf{mpost}(\mathbb{S}',\mathbb{H},A))\wedge$
$\mathsf{nin\_rid}(\{\mathtt{t6},\ \mathtt{a0},\ \mathtt{at},\ \mathtt{v0},\ \mathtt{t9},\ \mathtt{t2},\ \mathtt{v1}\}, \mathbb{S}.\mathbb{R}, \mathbb{S}'.\mathbb{R})$

$p_{\text{MLOOP}} \stackrel{\text{def}}{=} \lambda\mathbb{S}.\ \exists\mathbb{H}, A.\ \mathsf{minv}(\mathbb{S},\mathbb{H},A)$

$g_{\text{MLOOP}} \stackrel{\text{def}}{=} \lambda\mathbb{S},\mathbb{S}'.\ (\forall\mathbb{H}, A.\ \mathsf{minv}(\mathbb{S},\mathbb{H},A) \to\ \mathsf{mpost}(\mathbb{S}',\mathbb{H},A))\wedge$
$\mathsf{nin\_rid}(\{\mathtt{ra},\ \mathtt{t6},\ \mathtt{a0},\ \mathtt{at},\ \mathtt{v0},\ \mathtt{t9},\ \mathtt{t2},\ \mathtt{v1}\}, \mathbb{S}.\mathbb{R}, \mathbb{S}'.\mathbb{R})\wedge$
$\mathbb{S}'.\mathbb{R}(\mathtt{ra}) = \mathbb{S}.\mathbb{R}(\mathtt{t6})$

$p_{\text{MFST}} \stackrel{\text{def}}{=} \lambda\mathbb{S}.\ \exists\mathbb{H}, A.\ \mathsf{mfst}(\mathbb{S},\mathbb{H},A)$

$g_{\text{MFST}} \stackrel{\text{def}}{=} \lambda\mathbb{S},\mathbb{S}'.\ (\forall\mathbb{H}, A.\ \mathsf{mfst}(\mathbb{S},\mathbb{H},A) \to\ \mathsf{mpost}(\mathbb{S}',\mathbb{H},A))\wedge$
$\mathsf{nin\_rid}(\{\mathtt{ra},\ \mathtt{t6},\ \mathtt{a0},\ \mathtt{at},\ \mathtt{v0},\ \mathtt{t9},\ \mathtt{t2},\ \mathtt{v1}\}, \mathbb{S}.\mathbb{R}, \mathbb{S}'.\mathbb{R})\wedge$
$\mathbb{S}'.\mathbb{R}(\mathtt{ra}) = \mathbb{S}.\mathbb{R}(\mathtt{t6})$

$p_{\text{MSND}} \stackrel{\text{def}}{=} \lambda\mathbb{S}.\ \exists\mathbb{H}, A.\ \mathsf{msnd}(\mathbb{S},\mathbb{H},A)$

$g_{\text{MSND}} \stackrel{\text{def}}{=} \lambda\mathbb{S},\mathbb{S}'.\ (\forall\mathbb{H}, A.\ \mathsf{msnd}(\mathbb{S},\mathbb{H},A)\ \to\ \mathsf{mpost}(\mathbb{S}',\mathbb{H},A))\wedge$
$\mathsf{nin\_rid}(\{\mathtt{ra},\ \mathtt{t6},\ \mathtt{a0},\ \mathtt{at},\ \mathtt{v0},\ \mathtt{t9},\ \mathtt{t2},\ \mathtt{v1}\}), \mathbb{S}.\mathbb{R}, \mathbb{S}'.\mathbb{R})\wedge$
$\mathbb{S}'.\mathbb{R}(\mathtt{ra}) = \mathbb{S}.\mathbb{R}(\mathtt{t6})$

$\mathsf{mpost}\ ((\mathbb{H},\mathbb{R}),\mathbb{H}_{\text{T}},A) \stackrel{\text{def}}{=} \exists B, W.\ \mathsf{sted\_ok}(\mathbb{R}) \wedge B\cup W = \mathsf{ptrs}\wedge$
$\mathsf{ok\_val}(B, \mathbb{R}(\mathtt{t0}))\wedge\ (\forall x\in B.\ \mathsf{reach}(\mathbb{H}_{\text{T}}, \mathbb{R}(\mathtt{t0}), x))\wedge$
$\mathbb{H} \Vdash\ \mathsf{eq}(\mathbb{H}_{\text{T}}) * \mathsf{mbits}(B,1) * \mathsf{mbits}(W,0) * \mathsf{mstack}(\mathbb{R},\emptyset) * A\wedge$
$\mathbb{H}_{\text{T}} \Vdash \mathsf{obj\_hp}(B,B) * \mathsf{obj\_hp}(\mathsf{ptrs}, W)$

$\mathsf{mpre}((\mathbb{H},\mathbb{R}),\mathbb{H}_{\text{T}}),A) \stackrel{\text{def}}{=}\ \mathsf{sted\_ok}(\mathbb{R})\wedge$
$\mathbb{H} \Vdash \mathsf{eq}(\mathbb{H}_{\text{T}}) * \mathsf{mbits}(\mathsf{ptrs},0) *\ \mathsf{mstack}(\mathbb{R},\emptyset) * A\wedge$
$\mathbb{H}_{\text{T}} \Vdash\ \mathsf{obj\_hp}(\mathsf{ptrs},\mathsf{ptrs})$

$\mathsf{minv}((\mathbb{H},\mathbb{R})),\mathbb{H}_{\text{T}},A) \stackrel{\text{def}}{=} \exists B, G, W.$
$\mathsf{sted\_ok}(\mathbb{R}) \wedge B\cup G\cup W =\ \mathsf{ptrs}\wedge$
$\mathsf{ok\_val}(B\cup G, \mathbb{R}(\mathtt{t0}))\wedge\ (\forall x\in (B\cup G).\ \mathsf{reach}(\mathbb{H}_{\text{T}}, \mathbb{R}(\mathtt{t0}), x))\wedge$
$\mathbb{H} \Vdash\ \mathsf{eq}(\mathbb{H}_{\text{T}}) *\ \mathsf{mbits}(B\cup G,1) * \mathsf{mbits}(W,0) *\ \mathsf{mstack}(\mathbb{R},G) * A\wedge$
$\mathbb{H}_{\text{T}} \Vdash\ \mathsf{obj\_hp}(B\cup G, B) *\ \mathsf{obj\_hp}(\mathsf{ptrs}, G\cup W)$

$\mathsf{minv2}((\mathbb{H},\mathbb{R}),\mathbb{H}_{\text{T}},A,S_1,S_2,B,G,W) \stackrel{\text{def}}{=}$
$\mathsf{sted\_ok}(\mathbb{R}) \wedge B\cup G\cup W\cup\{\mathbb{R}(\mathtt{v0})\} = \mathsf{ptrs}\wedge$
$\mathsf{ok\_val}(B\cup G\cup\{\mathbb{R}(\mathtt{v0})\}, \mathbb{R}(\mathtt{t0}))\wedge$
$(\forall x\in B\cup G\cup\{\mathbb{R}(\mathtt{v0})\}.\ \mathsf{reach}(\mathbb{H}_{\text{T}}, \mathbb{R}(\mathtt{t0}), x))\wedge$
$\mathbb{H} \Vdash\ \mathsf{eq}(\mathbb{H}_{\text{T}}) *\ \mathsf{mbits}(B\cup G\cup\{\mathbb{R}(\mathtt{v0})\},1) *\ \mathsf{mbits}(W,0) *$
$\quad \mathsf{mstack}(\mathbb{R},G) * A)\wedge$
$\mathbb{H}_{\text{T}} \Vdash\ \mathsf{obj\_hp}(B\cup G\cup\{\mathbb{R}(\mathtt{v0})\},B) *\ \mathsf{obj\_hp}(\mathsf{ptrs}, G\cup W) *$
$\quad \mathsf{ok\_fld}(S_1,\mathbb{R}(\mathtt{v0})) *\ \mathsf{ok\_fld}(S_2,\mathbb{R}(\mathtt{v0})+4)$

$\mathsf{mfst}(\mathbb{S},\mathbb{H}_{\text{T}},A) \stackrel{\text{def}}{=}\ \exists B, G, W.\ \mathsf{minv2}(\mathbb{S},\mathbb{H}_{\text{T}},A,\mathsf{ptrs},\mathsf{ptrs},B,G,W)$

$\mathsf{msnd}(\mathbb{S},\mathbb{H}_{\text{T}},A) \stackrel{\text{def}}{=}\ \exists B, G, W.$
$\mathsf{minv2}(\mathbb{S},\mathbb{H}_{\text{T}},A,B\cup G\cup\{\mathbb{S}.\mathbb{R}(\mathtt{v0})\},\ \mathsf{ptrs},B,G,W)$

Fig.13. Assembly code with specifications: `mark`.

```
sweep: (p_SWEEP, g_SWEEP)
  addiu $a0 $k0 0
  addiu $t5 $0 0
  j sloop

sloop: (p_SLOOP, g_SLOOP)
  sltu $at $a0 $k1
  beq $at $0 return
  subu $at $a0 $k0
  srl1 $at $at
  addu $at $at $k1
  lw $v0 0($at)
  beq $v0 $0 sadd
  sw $0 0($at)
  j snext

sadd: (p_SADD, g_SADD)
  sw $t5 4($a0)
  addiu $t5 $a0 0
  j snext
return:
  jr $ra

snext: (p_SNXT, g_SNXT)
  addiu $a0 $a0 8
  j sloop
return1:
  addiu $ra $t6 0
  jr $ra
```

$p_{\text{SWEEP}} \stackrel{\text{def}}{=} \lambda\mathbb{S}.\ \exists\mathbb{H}, A.\ \mathsf{spre}(\mathbb{S},\mathbb{H},A)$

$g_{\text{SWEEP}} \stackrel{\text{def}}{=} \lambda\mathbb{S},\mathbb{S}'.\ \mathsf{nin\_rid}(\{(\mathtt{a0},\ \mathtt{t5},\ \mathtt{at},\ \mathtt{v0}\}, \mathbb{S}.\mathbb{R}, \mathbb{S}'.\mathbb{R})\wedge$
$\forall\mathbb{H}, A.\ \mathsf{spre}(\mathbb{S},\mathbb{H},A) \to\ \mathsf{spost}(\mathbb{S}',\mathbb{H},A)$

$p_{\text{SLOOP}} \stackrel{\text{def}}{=}\ \lambda\mathbb{S}.\ \exists\mathbb{H}, A.\ \mathsf{sinv}(\mathbb{S},\mathbb{H},A)$

$g_{\text{SLOOP}} \stackrel{\text{def}}{=}\ \lambda\mathbb{S},\mathbb{S}'.\mathsf{nin\_rid}(\{\mathtt{a0},\ \mathtt{t5},\ \mathtt{at},\ \mathtt{v0}\}, \mathbb{S}.\mathbb{R}, \mathbb{S}'.\mathbb{R})\wedge$
$\forall\mathbb{H}, A.\ \mathsf{sinv}(\mathbb{S},\mathbb{H},A) \to\ \mathsf{spost}(\mathbb{S}',\mathbb{H},A)$

$(p_{\text{SNXT}}, g_{\text{SNXT}}) \stackrel{\text{def}}{=}\ \mathsf{wp}(\{\mathtt{sloop} \rightsquigarrow (p_{\text{SLOOP}}, g_{\text{SLOOP}})\}, \mathbb{I}_{\text{SNXT}})$

$(p_{\text{SADD}}, g_{\text{SADD}}) \stackrel{\text{def}}{=}\ \mathsf{wp}(\{\mathtt{snxt} \rightsquigarrow (p_{\text{SNXT}}, g_{\text{SNXT}})\}, \mathbb{I}_{\text{SADD}})$

$\mathsf{spost}((\mathbb{H},\mathbb{R}),\mathbb{H}_{\text{T}},A) \stackrel{\text{def}}{=}\ \exists B, W.\ \mathsf{sted\_ok}(\mathbb{R}) \wedge B\cup W = \mathsf{ptrs}\wedge$
$\mathsf{ok\_val}(B, \mathbb{R}(\mathtt{t0}))\wedge\ (\forall x\in B.\ \mathsf{reach}(\mathbb{H}_{\text{T}}, \mathbb{R}(\mathtt{t0}), x))\wedge$
$\mathbb{H} \Vdash\ \mathsf{eq}(\mathbb{H}_{\text{T}}) *\ \mathsf{flist}(\mathbb{R},W) *\ \mathsf{mbits}(\mathsf{ptrs},0) *\ \mathsf{mstack}(\mathbb{R},\emptyset) * A\wedge$
$\mathbb{H}_{\text{T}} \Vdash\ \mathsf{obj\_hp}(B,B)$

$\mathsf{spre}((\mathbb{H},\mathbb{R}),\mathbb{H}_{\text{T}},A) \stackrel{\text{def}}{=}\ \exists B, W.$
$\mathsf{sted\_ok}(\mathbb{R}) \wedge B\cup W = \mathsf{ptrs}\wedge$
$\mathsf{ok\_val}(B, \mathbb{R}(\mathtt{t0}))\wedge\ (\forall x\in B.\ \mathsf{reach}(\mathbb{H}_{\text{T}}, \mathbb{R}(\mathtt{t0}), x))\wedge$
$\mathbb{H} \Vdash\ \mathsf{eq}(\mathbb{H}_{\text{T}}) *\ \mathsf{obj\_hp}(\mathsf{ptrs}, W) *\ \mathsf{mstack}(\mathbb{R},\emptyset) *$
$\quad \mathsf{mbits}(B,1) *\ \mathsf{mbits}(W,0) * A\wedge$
$\mathbb{H}_{\text{T}} \Vdash\ \mathsf{obj\_hp}(B,B)$

$\mathsf{sinv}((\mathbb{H},\mathbb{R}),\mathbb{H}_{\text{T}},A) \stackrel{\text{def}}{=}\ \exists B, W, F, G.$
$\mathsf{sted\_ok}(\mathbb{R}) \wedge\ \mathsf{ok\_val}(B, \mathbb{R}(\mathtt{t0}))\wedge$
$G = \{x \mid x\in\ \mathsf{ptrs}\ \wedge x < \mathbb{R}(\mathtt{a0})\}\wedge$
$W\cap G = \emptyset \wedge F\subseteq G \wedge (\mathbb{R}(\mathtt{a0}) - \mathsf{st}) \bmod 8 = 0\wedge$
$B\cup W\cup F = \mathsf{ptrs}\ \wedge\ (\forall x\in B.\ \mathsf{reach}(\mathbb{H}_{\text{T}}, \mathbb{R}(\mathtt{t0}), x))\wedge$
$\mathbb{H} \Vdash\ \mathsf{eq}(\mathbb{H}_{\text{T}}) *\ \mathsf{obj\_hp}(\mathsf{ptrs}, W) *\ \mathsf{flist}(\mathbb{R},F) *\ \mathsf{mstack}(\mathbb{R},\emptyset) *$
$\quad \mathsf{mbits}(W\cup F\cup(B\cap G),0) *\ \mathsf{mbits}(B-G,1) * A\wedge$
$\mathbb{H}_{\text{T}} \Vdash\ \mathsf{obj\_hp}(B,B)$

Fig.14. Assembly code with specifications: `sweep`.

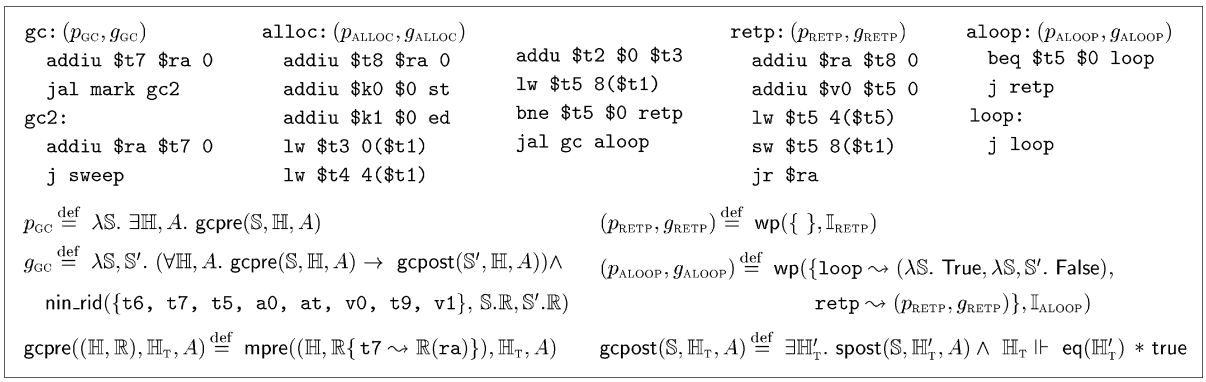

Fig.15. Assembly code with specifications: `gc` and `alloc`.

the collection. The specifications of `alloc` is listed in Fig.11, and the other specifications are generated by the wp function.

4.2 Theorem Proving

With the VCGen in Subsection 2.2, the well-formedness proofs of the code blocks are constructed without the direct use of SCAP rules. Instead, the main difficulty in the proof construction is to solve the domain specific problem. For our collector verification, the problem is to prove those properties about the finite set and the heap predicate, as the specifications are mainly constructed with assertions about them.

We tackle this problem by breaking the proof construction into two steps. First, we prove as lemmas the common properties of the finite set and the heap predicate, such as the introduction/elimination rules of the set operators, and the associative/communicative properties of the separating conjunction $*$. These are collected in two lemma libraries for the finite set and the heap predicate. Both the finite set and the heap predicate contain non-trivial inductive definitions hence it is infeasible to implement fully automatic decision procedures for either of them. However, there are still many patterns in using these lemmas during our proof construction. Following these patterns, we design automatic tactics to simplify the proof construction, an example of such tactics is shown in the next section.

With the well-designed domain specific lemma libraries, the well-formedness proofs of the code blocks are constructed with ease.

5 Coq Implementation

Our verification is fully mechanized within the Coq proof assistant[7]. Coq is an interactive theorem prover based on the formulae-as-type notion[16], where theorems and proofs are constructed in CiC as types and terms, respectively. Proof checking in Coq is thus type checking of terms in CiC, which is simple to implement and more trustworthy than the methodologies used by the other systems like PVS.

Coq provides a rich language which is able to define both logical and computational terms using inductive constructors and pattern matching. With this language, we build the abstract machine, the SCAP rule set, and other components for specifying the collector.

Theorem proving in Coq is goal-directed and tactic-based. A set of predefined tactics is provided to solve the proof goals. This includes the basic tactics like `apply`, `intro` and `elim`, which transform the goal according to the corresponding CiC typing rules; an `auto` tactic, which tries to apply previous lemmas declared as hints; and several predefined tactics representing some well-established decision procedures, such as the `omega` tactic, which solves a goal in Presburger arithmetic. A tactic language is also provided to build user-defined tactics using pattern matching on proof goals, recursion and other programming facilities. Our tactics for the finite set and the heap predicate are defined using this language.

In the following part of this section, we illustrate our Coq implementation methodology with several examples: the state transition function for commands in the abstract machine, the SCAP rule set for well-formed instruction sequence, the wp function in VCGen, and a user-defined tactic for solving goals with heap predicates. Interested readers may get more details of our Coq implementation from [12].

5.1 Verification Framework

The `nextc` function in Fig.16 is defined in Subsection 3.1 as the small step state transition relation for commands. It is implemented as a function with pattern matching. The tags such as `Some` and `None` on the return value of such functions are the constructors of an option type, like `option state` in this function. A value with an option type is either a meaningful value with the tag `Some`, or a bad value represented by `None`. For example, a `None` returned by the heap read function `hget` stands for a bad heap value, while a `None` returned by the program step function stands for a stuck program.

We implement the SCAP well-formed instruction sequence rule set as a recursion function which performs

pattern matching on the instruction sequence and returns different proof obligations based on the rules in Fig.6. The `wp` function in the VCGen follows the same pattern to generate the corresponding specifications.

```
Definition nextc (c:comm) (s:state):  option state :=
 match c with
  | addu rd rs rt =>
   Some (rset s rd (rget s rs + rget s rt))
      ⋮
  | lw rt offset rs =>
   match (hget s (rget s rs + offset)) with
    | Some v => Some (rset s rt v)
    | None => None
   end
 end.
Fixpoint scap_iseq_ok(P:chspec)(p:pre)
 (g:guar)(is:iseq){struct is}:  Prop :=
 match is with
  | jr r31 => forall s, p s -> g s s
      ⋮
  | seq c is' => exists p', exists g',
   (forall s, p s ->
    match (nextc c s) with
     | Some s' => p' s' /\
       forall s'', g' s' s'' -> g s s''
     | None => False
    end) /\ scap_iseq_ok P p' g' is'
 end.
Fixpoint wp(P:chspec)(is:iseq){struct is}
 :  option (pre, guar) :=
 match is with
  | jr r31 => (fun s=>True, fun s s' =>s=s')
      ⋮
  | seq c is' =>
   match (wp P is') with
    | Some (p', g') => Some (fun s =>
     match (nextc c s) with
      | Some s' => p' s' | None => False
     end, fun s s'' =>
     match (nextc c s) with
      | Some s' => g' s' s'' | None => False
     end)
    | None => None
   end
end.
```

Fig.16. Coq implementation.

5.2 Tactics for Automation

The most complicated part of the collector's proof lies in heap manipulation with separation logic, that is to say, to prove that one proposition with heap predicates implies another. For complex heap predicates with many separating conjunctions, like the ones mentioned in Subsection 4.1, this will be enormously difficult using only the communicative, associative and monotonic properties described in [6]. Usually hundreds of lines are needed to prove one such goal, and it is not uncommon to encounter several such goals in proving one code block.

We build a tactic `simplsep` that automatically matches and eliminates the identical or trivial parts of the heap predicates in the hypothesis and proof goal, leaving the different ones for manually proving. For example, the following goal is trivially provable with `simplsep`.

$$\forall \mathbb{H}, A, B, C.\ \mathbb{H} \Vdash A * B * C * \mathsf{emp} \rightarrow \mathbb{H} \Vdash C * \mathsf{true} * B.$$

In most of the cases, a code block changes only a small part of the heap. This tactic alleviates us from complicate reasoning, and greatly reduces the proof script size. The principle of `simplsep` is to regard the heap predicates in the goal and hypotheses as two lists, and prove that one is a permutation of another using the classic algorithm. During this procedure, the identical parts are eliminated using the monotonic lemma of separating conjunction. The running time of the tactic is bounded by the size of the heap predicates.

Lines	Component
816	Basic properties and tactics
1387	Ordered heap library
451	Abstract machine encoding and lemmas
1263	Ordered finite set library
844	Separation logic library
547	SCAP, VCGen and related tactics
529	Collector's heap definitions and lemmas
1930	Code, specification and proof of the collector

Fig.17. Proof script size.

We also implement other automatic tactics for reasoning about heap lookup and update, the manipulation of finite sets, auto rewriting on complicate states values generated by VCGen and so on. The detailed examples of these tactics can be found in [12].

5.3 Evaluation

Fig.17 lists the proof script size of our Coq implementation, in terms of the number of non-empty lines. The work takes several man months for programmers familiar with the Coq system. The intensive use of automatic tactics and the VCGen results in a 3/4 drop of the proof script size and makes the proof much easier to follow. The benefit of these facilities is also demonstrated by our evaluation on the CDSA example in [5]. The `malloc/free` functions are verified within two days by a single person, while each proof script has a length of only 1/6 when comparing to their original ones.

Our implementation relies heavily on the Coq standard library. We also extend these libraries with various lemmas and tactics. The heap model and the finite set is designed with the ideas in [17], where a heap/set is a list of data (as in the Coq `ListSet` library) together with a proof that the list is well-ordered and with no redundancy. Thus we obtain the heap/set extensionally from the Law of Exclude Middle, and the Fixpoint functions on sets, such as the iterated separating conjunction and the mark stack predicate, can be built with ease.

6 Related Work

Intensive efforts have been dedicated to the building of trustworthy PCC systems[1,2,5,8,9]. Our work complements theirs by introducing verified garbage collector into PCC, which will reduces the overall TCB.

The work on the formal correctness proof of a garbage collector dates back to [18, 19], both with informal proofs. Researches on mechanized verification of GC algorithms[20~22] focus mostly on algorithms with abstract memory model, and some are implemented with model checking. Unlike their work, our collector is verified as a machine-level implementation, which forces us to employ a more concrete specification and thus our verified program is more trustworthy to run directly on a real machine.

The recent work by Birkedal *et al.*[15] use the axiom based separation logic to reason a copying collector. They give a formal paper proof of the Cheney collector[14] against a safety specification based on heap isomorphism. The iterated separating conjunction on finite set, which is intensively used in our work, comes from this paper.

Prior work on type-safe garbage collection[23~25] mainly focus on including garbage collection into the mutator's type system. As a result, for reasoning each of the non-trivial safety properties, complex type systems must be constructed and their soundness should be proved, which are unlikely easy tasks. On the other hand, as analyzed in Subsection 3.3, our collector specification is strong enough to preserve any common safety properties of the mutator program.

7 Conclusion

We demonstrate in this paper the mechanical verification of a conservative variant of the standard mark-sweep garbage collector in the PCC style. The specification of our collector is given on a machine-level memory model using separation logic. Our verification is fully implemented in Coq, and can be packed immediately as an FPCC package. The verification can also be used as a model for other PCC systems where the verification of a garbage collector is required. We also make non-trivial improvement to the verification framework, which is helpful for the future researches.

The work presented in this paper is a part of our ongoing project for building an SCAP-based framework to verify the mutator-collector interaction, where we adopt an unified basic safety specification interface for any of the copying, non-copying or incremental collectors. We also utilize the OCAP[9] framework as the platform for verifying the safe interaction between TAL and the collector discussed in this paper. The natural choice of future work includes verifying more efficient collectors, such as incremental and generational ones, and their interaction with various mutator systems.

References

[1] Necula G. Proof-carrying code. In *Proc. 24th ACM Symp. Principles of Prog. Lang.*, New York, ACM Press, January 1997, pp.106~119.

[2] Morrisett G, Walker D, Crary K, Glew N. From system F to typed assembly language. *ACM Trans. Prog. Lang. and Sys.*, 1999, 21(3): 527~568.

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

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

[5] Feng X Y, Shao Z, Vaynberg A *et al.* Modular verification of assembly code with stack-based control abstractions. In *Proc. 2006 ACM Conf. Prog. Lang. Design and Impl.*, Ottawa, Canada, June 2006, ACM Press, pp.401~414.

[6] Reynolds J C. Separation logic: A logic for shared mutable data structures. In *Proc. 17th IEEE Symp. Logic in Comp. Sci.*, Washington DC, USA, IEEE Comp. Soc., 2002, pp.55~74.

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

[8] Appel A W. Foundational proof-carrying code. In *Proc. 16th IEEE Symp. Logic in Comp. Sci.*, IEEE Comp. Soc., Boston, USA, June 2001, pp.247~258.

[9] Feng X, Ni Z, Shao Z, Guo Y. An open framework for foundational proof-carrying code. In *Proc. 3rd ACM SIGPLAN Workshop on Types in Lang. Design and Impl.*, Nice, France, ACM Press, January 2007, pp.67~78.

[10] McCreight A, Shao Z, Lin C, Li L. A General Framework for Certifying Garbage Collectors and Their Mutators. In *Proc. 2007 ACM SIGPLAN Conf. Prog. Lang. Design and Impl.*, San Diego, CA, USA, June 2007, ACM Press. (Paper to appear)

[11] Lin C, McCreight A, Shao Z, Chen Y, Guo Y. Foundational typed assembly language with certified garbage collection. In *Proc. 1st IEEE & IFIP International Symp. Theoretical Aspects of Soft. Eng.*, Shanghai, China, June 2007, IEEE Comp. Soc. (Paper to appear)

[12] Lin C, Chen Y, Li L, Hua B. Garbage collector verification for proof-carrying code (documents and Coq implementation). 2006, http://ssg.ustcsz.edu.cn/~cxlin/gcpaper/.

[13] C Paulin-Mohring. Inductive definitions in the system Coq—Rules and properties. In *Proc. 1st Int. Conf. Typed Lambda Calculi and Applications*, Utrecht, The Netherlands, *LNCS*, Vol.664, Springer-Verlag, 1993, pp.328~345.

[14] MIPS Technologies, Inc. MIPS32^{TM} Architecture for Programmers Volume II: The MIPS32^{TM} Instruction Set. v2.50.

[15] Birkedal L, Torp-Smith N, Reynolds J C. Local reasoning about a copying garbage collector. In *Proc. 31st ACM Symp. Principles of Prog. Lang.*, New York, USA, ACM Press, 2004, pp.220~231.

[16] Howard W A. The formulas-as-types notion of construction. To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism, Academic Press, 1980, pp.479~490.

[17] Marti N, Affeldt R, Yonezawa A. Formal verification of the heap manager of an operating system using separation logic. In *Proc. ICFEM2006, Lecture Notes in Computer Science*, Volume 4260, Canberra, September 1998, Springer-Verlag, pp.225~244.

[18] Dijkstra E W, Lamport L, Martin A J *et al.* On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM*, 1978, 21(11): 966~975.

[19] Ben-Ari M. Algorithms for on-the-fly garbage collection. *ACM Trans. Prog. Lang. and Sys.*, 1984, 6(3): 333~344.

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

[21] Jackson P. Verifying a garbage collection algorithm. In *Proc. 11th Int. Conf. Theorem Proving in Higher Order Logics,*

Lecture Notes in Computer Science, Canberra, Australia, Volume 1479, Springer-Verlag, 2006, pp.225∼244.

[22] L. Burdy. B vs. Coq to prove a garbage collector. In *Proc. 14th Int. Conf. Theorem Proving in Higher Order Logics*, Edinburgh, UK, Boulton R J, Jackson P B (eds.), September 2001, pp.85∼97.

[23] Wang D C, Appel A W. Type-preserving garbage collectors. In *Proc. 28th ACM Symp. Principles of Prog. Lang.*, New York, USA, ACM Press, 2001, pp.166∼178.

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

[25] Hawblitzel C, Huang H, Wittie L, Chen J. A garbage-collecting typed assembly language. In *Proc. The Third ACM SIGPLAN Workshop on Types in Language Design and Implementation*, Nice, France, ACM Press, January 2007, pp.41∼52.

Chun-Xiao Lin is currently a Ph.D. candidate in Dept. Comput. Sci. & Technol. at University of Science & Technology of China (USTC). He received his B.S. degree in computer science from USTC in 2003. His research interests include language based software safety, program verification on assembly code level, and garbage collection verification.

Yi-Yun Chen is a professor in Dept. Comput. Sci. & Technol. at USTC. He received his M.S. degree from East-China Institute of Comput. Technol. in 1982. His research interests include applications of logic (including formal semantics and type theory), techniques for designing and implementing programming languages and software safety and security.

Long Li is currently a Ph.D. candidate in Dept. Comput. Sci. & Technol. at USTC. He received his B.S. degree in computer science from USTC in 2003. His research interests involve language based software safety, program verification on assembly code level, and concurrent program verification.

Bei Hua is an associate professor in Dept. Comput. Sci. & Technol. at USTC. She received the M.S. degree in electronic engineering from Peking Univ. and the Ph.D. degree in computer science from USTC. Her research interests include wireless sensor networks and network processor based algorithms and applications.