Foundational Typed Assembly Language with Certified Garbage Collection

Chunxiao Lin[†] Andrew McCreight[‡] Zhong Shao[‡] Yiyun Chen[†] Yu Guo[†]
[†]*Department of Computer Science and Technology*
University of Science and Technology of China
Hefei, Anhui 230026, China
{cxlin3,guoyu}@mail.ustc.edu.cn *yiyun@ustc.edu.cn*
[‡]*Department of Computer Science*
Yale University
New Haven, CT 06520-8285, U.S.A
{aem,shao}@cs.yale.edu

Abstract

Type-directed certifying compilation and typed assembly language (TAL) aim to minimize the trusted computing base of safe languages by directly type-checking low-level machine code. However, the safety of TAL still heavily relies on its safe interaction with the underlying garbage collector. Based on a recent variant of foundational proof-carrying code (FPCC), we introduce a general methodology for combining foundational TAL with a certified garbage collector. We demonstrate the practicality of this approach by linking a typical TAL with a conservative garbage collector. This includes proving the safety of the collector, the soundness of TAL, and the safe interaction between TAL programs and the garbage collector. Our work is fully mechanized in the Coq proof assistant and the certified programs can be shipped immediately as FPCC packages.

1. Introduction

Type-safe languages such as Java and C# provide several protection mechanisms for the safe execution of programs. The implementation of a safe language, on the other hand, is a complex system with many components which must be trusted. These unverified components form the *trusted computing base* (TCB). Techniques such as type-directed certifying compilation and *typed assembly language* (TAL) [21] reduce the size of the TCB of these type safe languages. By preserving type information during compilation and directly type-checking the assembly code, these techniques remove the compiler from the TCB of a type-safe language.

However, the safety of a TAL system still relies on the soundness of its type system, the correctness of the type-checker, and the correct implementation of macro instructions such as `malloc`. Because it is difficult to certify the implementation of `free` in the presence of memory aliasing, TAL often requires a trusted garbage collector be part of the memory management run-time.

Some recent research focuses on building type-safe garbage collectors to remove the collector from the TCB of a TAL system. Wang and Appel [26] and Monnier *et al.* [19] propose to use languages with region-based type systems and intensional type analysis for type-checking a standard copying garbage collector [14]. But their approaches work only with specific GC algorithms and not, for example, with mark-sweep collectors. The complexity of the type system may also increase the TCB of their systems.

GTAL [11] uses a linear type system to encode new types from individual memory words. By building up appropriate abstractions, GTAL is capable of type-checking a variety of garbage collection mechanisms. Still, the new features in the type system result in a large TCB. Also, the metatheory of GTAL is not mechanized.

Foundational proof-carrying code (FPCC) [1, 9] can eliminate a large portion of TAL's TCB by mechanically proving the soundness of its type system, the correctness of the type checker, and the safety of the associated garbage collector in a foundational logic. The minimized TCB contains only the soundness of the foundational logic, the correctness of its proof checker, and the machine model. The recently proposed separation logic [24] also provides a powerful approach to reasoning about the safety of garbage collectors, as demonstrated by the work of Birkedal *et al.* [2].

In this paper we present a new methodology for building foundational TAL with a certified garbage collector. We combine the general framework for mutator-collector verification by McCreight *et al.* [17] with the open FPCC system by Feng *et al.* [8, 7]. We demonstrate the practicality of our approach by linking a TAL with a simple conservative garbage collector [3] in the FPCC setting. This includes proving the safety of the collector, the soundness of TAL, and the safe interaction between TAL programs and the collector. Our paper makes the following new contributions:

- As far as we know, the work presented in this paper is the first to successfully link a TAL program with a garbage collector to generate complete FPCC

packages. The type system of our TAL contains nontrivial features such as mutable references, recursive types, and union types, and is capable of typing mutable recursive data structures. The collector we verified is a conservative variant of a standard stop-the-world mark-sweep collector [14].

- Although our current paper only shows how to safely link TAL with a certified conservative collector, our methodology is general, capable of verifying more complex situations, such as the TAL type system keeping complex information for an accurate collector or even an incremental collector with read/write barriers.
- We specify the collector using the *stack-based certified assembly programming* (SCAP) framework [8] extended with separation-logic primitives [24]. Our specification asserts the machine-level behavior of the collector and is general enough for various mutator safety requirements besides type safety. The safety proof of our collector, which is done using SCAP, is a nontrivial work by itself.
- The work presented in this paper is fully mechanized within the Coq proof assistant [5]. Thus, the linked code of the mutator and collector can immediately be shipped as an FPCC package with a minimal TCB. We have also developed various mechanisms in Coq to simplify proof construction.

In Section 2, we introduce the preliminary knowledge needed to understand the rest of the paper. In Section 3, we present our general methodology for building TAL with certified garbage collection. In Sections 4–6 we apply this methodology to verify the safe interaction of a TAL with a conservative collector. We discuss the implementation in Section 7. Finally, we discuss related work and conclude.

Note that all lemmas and theorems in this paper are mechanically proved in Coq. Their detailed proofs are available on our project web site [16].

2. Basic setting

Before presenting our general methodology we give a general introduction to the FPCC system that our work uses. This includes a MIPS32-style [18] abstract machine model and the *lightweight open framework for certified assembly programming* [7] (LOCAP), a program logic for reasoning about the interaction of two different verification systems. We also present the embedding of SCAP [8] in LOCAP. As demonstrated by Feng *et al.* [8, 7], SCAP is suitable for certifying system level libraries, and we use it to construct the safety proof for our garbage collector.

Both the machine model and the program logic are formalized within a mechanized meta-logic, the *Calculus of Inductive Construction* (CiC) [23], as implemented in the Coq proof assistant [5]. CiC is a higher-order predicate logic extended with inductive definitions. The CiC terms in this paper are written using standard logical notation. We let $Prop$ be the universe of all logical propositions and Set be the universe of all computational terms.

$$
\begin{array}{llll}
(Prog) & \mathbb{P} & ::= & (\mathbb{C}, \mathbb{S}, \mathbb{I}) \\
(CdHeap) & \mathbb{C} & ::= & \{\mathtt{f} \rightsquigarrow \mathbb{I}\}^* \\
(State) & \mathbb{S} & ::= & (\mathbb{H}, \mathbb{R}) \\
(Heap) & \mathbb{H} & ::= & \{\mathtt{l} \rightsquigarrow \mathtt{w}\}^* \\
(RFile) & \mathbb{R} & ::= & \{\mathtt{r} \rightsquigarrow \mathtt{w}\}^* \\
(Reg) & \mathtt{r} & ::= & \{\mathtt{r}k\}^{k \in \{0 \ldots 31\}} \\
(Wd, Lab) & \mathtt{w}, \mathtt{f} & ::= & 0 \mid 1 \mid 2 \mid \ldots \\
(Address) & \mathtt{l} & ::= & 0 \mid 4 \mid 8 \mid \ldots \\
(ISeq) & \mathbb{I} & ::= & c; \mathbb{I} \mid \mathtt{beq}\ \mathtt{r_s},\ \mathtt{r_t},\ \mathtt{f}; \mathbb{I} \\
 & & \mid & \mathtt{bne}\ \mathtt{r_s},\ \mathtt{r_t},\ \mathtt{f}; \mathbb{I} \\
 & & \mid & \mathtt{j}\ \mathtt{f} \mid \mathtt{jal}\ \mathtt{f},\ \mathtt{f_{ret}} \mid \mathtt{jr}\ \mathtt{r_s} \\
(Comm) & c & ::= & \mathtt{addu}\ \mathtt{r_d},\ \mathtt{r_s},\ \mathtt{r_t} \mid \mathtt{addiu}\ \mathtt{r_d},\ \mathtt{r_s},\ \mathtt{w} \\
 & & \mid & \mathtt{subu}\ \mathtt{r_d},\ \mathtt{r_s},\ \mathtt{r_t} \mid \mathtt{srl}\ \mathtt{r_d},\ \mathtt{r_s},\ \mathtt{1} \\
 & & \mid & \mathtt{sltu}\ \mathtt{r_d},\ \mathtt{r_s},\ \mathtt{r_t} \mid \mathtt{andi}\ \mathtt{r_d},\ \mathtt{r_s},\ \mathtt{7} \\
 & & \mid & \mathtt{lw}\ \mathtt{r_d},\ \mathtt{w}(\mathtt{r_s}) \mid \mathtt{sw}\ \mathtt{r_s},\ \mathtt{w}(\mathtt{r_d})
\end{array}
$$

Figure 1. Abstract machine syntax

2.1. Abstract machine

Figure 1 gives the syntax of our abstract machine. 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 partial map from code labels $\mathtt{f}$ to instruction sequences $\mathbb{I}$. A machine state $\mathbb{S}$ contains a data heap $\mathbb{H}$ and a register file $\mathbb{R}$. A data heap $\mathbb{H}$ is a partial map from 4-byte aligned addresses $\mathtt{l}$ to word values $\mathtt{w}$, while a register file $\mathbb{R}$ is a map from registers $\mathtt{r}$ to word values, with $\mathtt{r0}$ always mapped to 0. A command c is a non-control-flow instruction such as a register add or a heap load. An instruction sequence $\mathbb{I}$, or code block, is a series of commands and branches 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 an instruction sequence instead of the standard `pc` register. This results in the additional return address $\mathtt{f_{ret}}$ in the jump and link instruction $\mathtt{jal}\ \mathtt{f},\ \mathtt{f_{ret}}$. By expanding this instruction to the MIPS instruction pair $\mathtt{jal}\ \mathtt{f}$ and $\mathtt{j}\ \mathtt{f_{ret}}$, all our certified code [16] can directly run on the SPIM simulator [15].

Following [18], we give the small step operational semantics of the abstract machine in Figure 2. 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. Note that for a `lw`/`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 an invalid label or the next state of its first command is undefined.

if $\mathbb{I}$ =	then $(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{I}) \longmapsto$
$\mathtt{j}\ \mathtt{f}$	if $\mathtt{f} \in \mathsf{dom}(\mathbb{C})$, $(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{C}(\mathtt{f}))$
$\mathtt{jal}\ \mathtt{f},\ \mathtt{f}_{\mathtt{ret}}$	if $\mathtt{f} \in \mathsf{dom}(\mathbb{C})$, $(\mathbb{C}, (\mathbb{H}, \mathbb{R}\{\mathtt{r31} \rightsquigarrow \mathtt{f}_{ret}\}), \mathbb{C}(\mathtt{f}))$
$\mathtt{jr}\ \mathtt{r_s}$	if $\mathbb{R}(\mathtt{r_s}) \in \mathsf{dom}(\mathbb{C})$, $(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{C}(\mathbb{R}(\mathtt{r_s})))$
$\mathtt{beq}\ \mathtt{r_s},\ \mathtt{r_t},\ \mathtt{f}; \mathbb{I}'$	if $\mathbb{R}(\mathtt{r_s}) \neq \mathbb{R}(\mathtt{r_t})$, $(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{I}')$, else if $\mathtt{f} \in \mathsf{dom}(\mathbb{C})$, $(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{C}(\mathtt{f}))$
$\mathtt{bne}\ \mathtt{r_s},\ \mathtt{r_t},\ \mathtt{f}; \mathbb{I}'$	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 $\mathsf{Next}_c(\mathbb{H}, \mathbb{R})$=
$\mathtt{addu}\ \mathtt{r_d},\ \mathtt{r_s},\ \mathtt{r_t}$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r_d} \rightsquigarrow \mathbb{R}(\mathtt{r_s}) + \mathbb{R}(\mathtt{r_t})\})$
$\mathtt{addiu}\ \mathtt{r_d},\ \mathtt{r_s},\ \mathtt{w}$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r_d} \rightsquigarrow \mathbb{R}(\mathtt{r_s}) + \mathtt{w}\})$
$\mathtt{subu}\ \mathtt{r_d},\ \mathtt{r_s},\ \mathtt{r_t}$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r_d} \rightsquigarrow \mathbb{R}(\mathtt{r_s}) - \mathbb{R}(\mathtt{r_t})\})$
$\mathtt{srl}\ \mathtt{r_d},\ \mathtt{r_s},\ \mathtt{1}$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r_d} \rightsquigarrow \mathbb{R}(\mathtt{r_s})/2\})$
$\mathtt{sltu}\ \mathtt{r_d},\ \mathtt{r_s},\ \mathtt{r_t}$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r_d} \rightsquigarrow k\})$ if $\mathbb{R}(\mathtt{r_s}) < \mathbb{R}(\mathtt{r_t})$, $k = 1$, else $k = 0$
$\mathtt{andi}\ \mathtt{r_d},\ \mathtt{r_s},\ \mathtt{7}$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r_d} \rightsquigarrow \mathbb{R}(\mathtt{r_s}) \bmod 8\})$
$\mathtt{lw}\ \mathtt{r_d},\ \mathtt{w}(\mathtt{r_s})$	if $(\mathbb{R}(\mathtt{r_s}) + \mathtt{w}) \in \mathsf{dom}(\mathbb{H})$, $(\mathbb{H}, \mathbb{R}\{\mathtt{r_d} \rightsquigarrow \mathbb{H}(\mathbb{R}(\mathtt{r_s}) + \mathtt{w})\})$
$\mathtt{sw}\ \mathtt{r_s},\ \mathtt{w}(\mathtt{r_d})$	if $(\mathbb{R}(\mathtt{r_d}) + \mathtt{w}) \in \mathsf{dom}(\mathbb{H})$, $(\mathbb{H}\{\mathbb{R}(\mathtt{r_d}) + \mathtt{w} \rightsquigarrow \mathbb{R}(\mathtt{r_s})\}, \mathbb{R})$

Figure 2. Abstract machine semantics

2.2. Program logic

The readers may view LOCAP as a simplified OCAP [7], or an extended CAP0 [8]. We use it to embed two verification systems, namely TAL and SCAP. As listed in Figure 3, the specification of a code block is given by θ. This may be a state predicate in Hoare logic [12], a register file type in TAL, or anything else. LOCAP is a simplification of OCAP because there are only two kinds of code block specifications, so the language dictionary of OCAP is not needed. A code heap specification Ψ is a set of $(\mathtt{f}, \theta)$ pairs. Therefore, a code block may have more than one kind of specification in Ψ. We utilize this property to specify the GC interface for TAL. The interpretation function $[\![\ ]\!]$ translates θ into a predicate $\mathtt{a}$ over the environment Ψ and the machine state, to allow $\mathtt{a}$ to specify the code pointers (labels of code blocks) in Ψ. Both θ and $[\![\ ]\!]$ will be instantiated for TAL and SCAP in our following discussion. Finally, $\Rightarrow$ is the implication relation on assertions and a lifted assertion $\langle \mathtt{a} \rangle_\Psi$ combines $\mathtt{a}$ with all the information in Ψ.

We show the LOCAP inference rules in Figure 4. A well-formed program is a well-formed code heap with an appropriate initial state. A code heap $\mathbb{C}$ is well-formed with respect to Ψ if each pair in Ψ corresponds to a well-formed code block in $\mathbb{C}$. Interested readers may find the detailed explanation of the rules in [7], but this is not required for understanding the rest of the paper.

$$
\begin{array}{llll}
(CdSpec) & \theta & ::= & \cdots \mid \cdots \\
(ChSpec) & \Psi & ::= & \{(\mathtt{l}, \theta)\}^* \\
(Assert) & \mathtt{a} & \in & ChSpec \rightarrow State \rightarrow Prop \\
(Interp) & [\![\ ]\!] & \in & CdSpec \rightarrow Assert
\end{array}
$$

$$
\begin{array}{rcl}
\mathtt{a} \Rightarrow \mathtt{a}' & \stackrel{\text{def}}{=} & \forall \Psi, \mathbb{S}.\ \mathtt{a}\ \Psi\ \mathbb{S} \ \rightarrow\ \mathtt{a}'\ \Psi\ \mathbb{S} \\
\Psi \subseteq \Psi' & \stackrel{\text{def}}{=} & \forall (\mathtt{f}, \theta).\ (\mathtt{f}, \theta) \in \Psi \rightarrow (\mathtt{f}, \theta) \in \Psi' \\
\langle \mathtt{a} \rangle_{\Psi'} & \stackrel{\text{def}}{=} & \lambda \Psi, \mathbb{S}.\ \Psi' \subseteq \Psi \ \wedge\ \mathtt{a}\ \Psi\ \mathbb{S}
\end{array}
$$

Figure 3. LOCAP specification constructs

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

$$\frac{\Psi \vdash \mathbb{C} : \Psi \qquad (\mathtt{a}\ \Psi\ \mathbb{S}) \qquad \vdash \{\mathtt{a}\}\ \mathbb{I}}{\Psi \vdash (\mathbb{C}, \mathbb{S}, \mathbb{I})} \quad \text{(PROG)}$$

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

$$\frac{\vdash \{\langle [\![\theta]\!] \rangle_\Psi\}\ \mathbb{C}(\mathtt{f}) \qquad \forall (\mathtt{f}, \theta) \in \Psi'}{\Psi \vdash \mathbb{C} : \Psi'} \quad \text{(CDHP)}$$

$\boxed{\vdash \{\mathtt{a}\}\ \mathbb{I}}$ (*Well-formed Instruction Sequence*)

$$\frac{\mathtt{a} \Rightarrow \lambda \Psi, \mathbb{S}.\ \exists \theta.\ (\mathtt{f}, \theta) \in \Psi \wedge [\![\theta]\!]\ \Psi\ \mathbb{S}}{\vdash \{\mathtt{a}\}\ \mathtt{j}\ \mathtt{f}} \quad \text{(J)}$$

$$\frac{\begin{array}{l}\mathtt{a} \Rightarrow \lambda \Psi, (\mathbb{H}, \mathbb{R}).\ \exists \theta. \\ \qquad (\mathbb{R}(\mathtt{r_s}), \theta) \in \Psi \wedge [\![\theta]\!]\ \Psi\ (\mathbb{H}, \mathbb{R})\end{array}}{\vdash \{\mathtt{a}\}\ \mathtt{jr}\ \mathtt{r_s}} \quad \text{(JR)}$$

$$\frac{\begin{array}{l}\mathtt{a} \Rightarrow \lambda \Psi, (\mathbb{H}, \mathbb{R}).\ \exists \theta. \\ \qquad (\mathtt{f}, \theta) \in \Psi \wedge [\![\theta]\!]\ \Psi\ (\mathbb{H}, \mathbb{R}\{\mathtt{r31} \rightsquigarrow \mathtt{f_{ret}}\})\end{array}}{\vdash \{\mathtt{a}\}\ \mathtt{jal}\ \mathtt{f},\ \mathtt{f_{ret}}} \quad \text{(JAL)}$$

$$\frac{\vdash \{\mathtt{a}'\}\ \mathbb{I} \qquad \mathtt{a} \Rightarrow \lambda \Psi, \mathbb{S}.\ \mathtt{a}'\ \Psi\ \mathsf{Next}_c(\mathbb{S})}{\vdash \{\mathtt{a}\}\ c; \mathbb{I}} \quad \text{(SEQ)}$$

Figure 4. LOCAP inference rules (excerpt)

The weakening lemma states that if a code block is well-formed with some $\mathtt{a}'$, it is also well-formed with a stronger assertion $\mathtt{a}$, 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 $\mathtt{a} \Rightarrow \mathtt{a}'$ and $\vdash \{\mathtt{a}'\}\ \mathbb{I}$, then: $\vdash \{\mathtt{a}\}\ \mathbb{I}$;
2. If $\Psi \subseteq \Psi'$ and $\vdash \{\langle \mathtt{a} \rangle_\Psi\}\ \mathbb{I}$, then: $\vdash \{\langle \mathtt{a} \rangle_{\Psi'}\}\ \mathbb{I}$.

The soundness-correctness theorem of the CAP system ensures that a well-formed program will run forever without reaching any undefined steps in Figure 2, and the partial correctness of the program against its specification holds.

Theorem 1 (Soundness-Correctness).
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 $(\mathbb{C}, \mathbb{S}, \mathbb{I}) \longmapsto_{\mathrm{n}} (\mathbb{C}, \mathbb{S}', \mathbb{I}')$; and if $(\mathbb{C}, \mathbb{S}', \mathbb{I}') \longmapsto (\mathbb{C}, \mathbb{S}'', \mathbb{C}(\mathtt{f}))$, then there exists a θ, such that $(\mathtt{f}, \theta) \in \Psi$ and $[\![\theta]\!]\ \Psi\ \mathbb{S}''$.

$$
\begin{array}{llll}
(SPred) & \mathrm{p}, \mathrm{q} & \in & State \rightarrow Prop \\
(Guar) & \mathrm{g} & \in & State \rightarrow State \rightarrow Prop \\
(CdSpec) & \theta & ::= & (\mathrm{p}, \mathrm{g})
\end{array}
$$

$$
\begin{array}{l}
\mathsf{wfst}(0, \mathrm{q}, \Psi) \stackrel{\text{def}}{=} \forall(\mathbb{H}, \mathbb{R}).\ \mathrm{q}\ (\mathbb{H}, \mathbb{R}) \rightarrow \\
\qquad \exists\Gamma.\ (\mathbb{R}(\mathtt{r31}), \Gamma) \in \Psi \wedge [\![\Gamma]\!]_{\text{TAL}}\ \Psi\ (\mathbb{H}, \mathbb{R}) \\
\mathsf{wfst}(n+1, \mathrm{q}, \Psi) \stackrel{\text{def}}{=} \forall(\mathbb{H}, \mathbb{R}).\ \mathrm{q}\ (\mathbb{H}, \mathbb{R}) \rightarrow \\
\qquad \exists \mathrm{p}, \mathrm{g}.\ (\mathbb{R}(\mathtt{r31}), (\mathrm{p}, \mathrm{g})) \in \Psi \wedge \mathrm{p}\ (\mathbb{H}, \mathbb{R}) \wedge \\
\qquad\qquad \mathsf{wfst}(n, \mathrm{g}\ (\mathbb{H}, \mathbb{R}), \Psi) \\
[\![(\mathrm{p}, \mathrm{g})]\!]_{\text{SCAP}} \stackrel{\text{def}}{=} \lambda\Psi, \mathbb{S}.\ \mathrm{p}\ \mathbb{S} \wedge \exists n.\ \mathsf{wfst}(n, \mathrm{g}\ \mathbb{S}, \Psi)
\end{array}
$$

$$
\Psi \vdash_{\text{SCAP}} \{(\mathrm{p}, \mathrm{g})\}\ \mathbb{I} \stackrel{\text{def}}{=} \vdash \{\langle [\![(\mathrm{p}, \mathrm{g})]\!]_{\text{SCAP}} \rangle_\Psi\}\ \mathbb{I}
$$

$\boxed{\Psi \vdash_{\text{SCAP}} \{(\mathrm{p}, \mathrm{g})\}\ \mathbb{I}}$ *(Well-formed Instruction Sequence Lemmas)*

$$
\frac{\begin{array}{l}
(\mathtt{f}, (\mathrm{p}', \mathrm{g}')) \in \Psi \qquad (\mathtt{f}_{\mathtt{ret}}, (\mathrm{p}'', \mathrm{g}'')) \in \Psi \\
\forall(\mathbb{H}, \mathbb{R}).\ \mathrm{p}\ (\mathbb{H}, \mathbb{R}) \rightarrow \mathrm{p}'\ (\mathbb{H}, \mathbb{R}\{\mathtt{r31} \rightsquigarrow \mathtt{f}_{\mathtt{ret}}\}) \wedge \\
\quad \forall\mathbb{S}'.\ \mathrm{g}'\ (\mathbb{H}, \mathbb{R}\{\mathtt{r31} \rightsquigarrow \mathtt{f}_{\mathtt{ret}}\})\ \mathbb{S}' \rightarrow \\
\qquad \mathrm{p}''\ \mathbb{S}' \wedge \forall\mathbb{S}''.\ \mathrm{g}''\ \mathbb{S}'\ \mathbb{S}'' \rightarrow \mathrm{g}\ (\mathbb{H}, \mathbb{R})\ \mathbb{S}'' \\
\forall(\mathbb{H}, \mathbb{R}), (\mathbb{H}', \mathbb{R}'). \\
\quad \mathrm{g}'\ (\mathbb{H}, \mathbb{R})\ (\mathbb{H}', \mathbb{R}') \rightarrow \mathbb{R}(\mathtt{r31}) = \mathbb{R}'(\mathtt{r31})
\end{array}}{\Psi \vdash_{\text{SCAP}} \{(\mathrm{p}, \mathrm{g})\}\ \mathtt{jal\ f,\ f_{ret}}} \ (\text{CALL})
$$

$$
\frac{\forall\mathbb{S}.\ \mathrm{p}\ \mathbb{S} \rightarrow \mathrm{g}\ \mathbb{S}\ \mathbb{S}}{\Psi \vdash_{\text{SCAP}} \{(\mathrm{p}, \mathrm{g})\}\ \mathtt{jr\ r31}} \ (\text{RETURN})
$$

Figure 5. SCAP in LOCAP

Embedding of SCAP. Following [7], we show the embedding of SCAP in LOCAP in Figure 5. An SCAP code specification is a pair consisting of a precondition p and a guarantee g. Here p resembles a precondition in Hoare logic while g relates the current state to the return state (of the current procedure). A guarantee g at the entry of a procedure can be used to assert its safety guarantee, as we will see later.

The SCAP interpretation $[\![(\mathrm{p}, \mathrm{g})]\!]_{\text{SCAP}}$ asserts that the whole machine state satisfies p, and there is a well-formed control stack somewhere in the state. The abstract stack predicate $\mathsf{wfst}(n, \mathrm{g}\ \mathbb{S}, \Psi)$ generally asserts that the current function can return to the label stored in `r31` in the return state. n is the number of stack frames. When n is 0, the caller of the SCAP function must be a TAL program. A set of lemmas is also proved for building well-formed code blocks with SCAP code specifications. A detailed knowledge of wfst and the lemmas is not needed for understanding the rest of the paper; interested readers may refer to [8, 7] for their explanations.

3. The general methodology

Our basic idea comes from the analysis in Section 2.2: if we are able to prove that the client program is well-formed using a TAL-style type system, and that the collector is well-formed using SCAP, then we can link the client with the collector to form a well-formed complete code heap in LOCAP. Following Theorem 1, the code in a well-formed code heap will run safely forever from a correct initial state. This is exactly what we want, as it implies the safe interaction of the TAL program and the garbage collector. This leads to the following steps to combine foundational TAL with certified garbage collection:

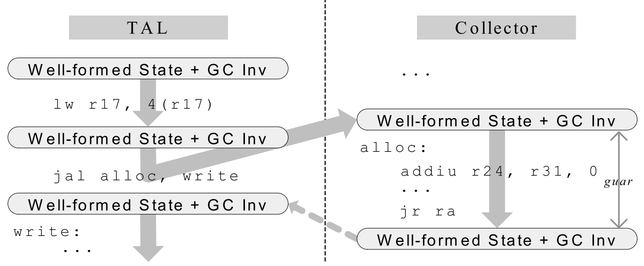

Figure 6. TAL and GC steps

Certifying the collector. We prove the well-formedness of the collector with SCAP specifications. For each collector routine with the specification (p, g), assertion p should include all of the information required by the collector routine, while g should capture its basic safety guarantee.

Embedding of TAL. We get a foundational TAL by embedding its type system in LOCAP (much like how we embed SCAP in LOCAP in Section 2.2). The soundness of TAL follows directly from the soundness of LOCAP. The type system of TAL must also reflect our choice of collector in that it must contain enough information to meet the requirements of the SCAP specifications of the collector routines.

Collector interface compatibility. We must also provide the TAL specifications for the collector routines to type-check the TAL client codes. Therefore in the code heap specification of the global code heap (which contains both the client and the collector), we have both the SCAP and TAL specifications for the collector interface. For each collector interface, we supply the missing proof required by the CDHP rule using Lemma 1, if the interpretation of its TAL specification implies the interpretation of its SCAP one.

In the rest of this section, we discuss several general aspects of embedding TAL into LOCAP with respect to various garbage collectors.

3.1. Typed assembly language in LOCAP

The register file type Γ of the original TAL [21] is a natural candidate for the TAL instantiation of the LOCAP code

```
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);
}
gc() {
  mark_field(root1);
  ...
  mark_field(rootn);
  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;
}
alloc() {
  if (freeptr == NULL) gc();
  if (freeptr == NULL) loop();
  l = freeptr; freeptr = freeptr->first;
  return l;
}
```

Figure 7. A conservative collector

specification θ, and its interpretation $[\![\ ]\!]_{\text{TAL}}$, as shown bellow, is a variant of the one used in [7].

$$[\![\Gamma]\!]_{\text{TAL}} \stackrel{\text{def}}{=} \lambda\Psi, (\mathbb{H}, \mathbb{R}).\ \exists \mathbb{H}_{\text{T}}.\ \Psi \vdash_{\text{TAL}} (\mathbb{H}_{\text{T}}, \mathbb{R}) : \Gamma \wedge \mathsf{gc_inv}((\mathbb{H}, \mathbb{R}), \mathbb{H}_{\text{T}})$$

The interpretation allows us to partition the state, reasoning about TAL code as though it were running on a virtual heap $\mathbb{H}_{\text{T}}$ provided by the collector. Both the TAL state typing rules and the collector invariant depend on the collector used. With an precise collector, $\Psi \vdash_{\text{TAL}} (\mathbb{H}_{\text{T}}, \mathbb{R}) : \Gamma$ must contain pointer information for each heap object, while with a conservative collector this is not necessary. The $\mathsf{gc_inv}$ invariants of various precise collectors are described in [17].

The invariant $\Psi \vdash_{\text{TAL}} (\mathbb{H}_{\text{T}}, \mathbb{R}) : \Gamma$ corresponds to the well-formed state relation of the original TAL, but with additional information required by the collector routines to correctly trace the live objects in $\mathbb{H}_{\text{T}}$. The garbage collector representation invariant $\mathsf{gc_inv}((\mathbb{H}, \mathbb{R}), \mathbb{H}_{\text{T}})$ specifies the collector's data structures in $(\mathbb{H}, \mathbb{R})$ and their relationship with the virtual heap $\mathbb{H}_{\text{T}}$ accessed by TAL clients.

In addition, the TAL instruction sequence lemmas, which correspond to the instruction typing rules of the original TAL, must ensure that the invariants of $[\![\Gamma]\!]_{\text{TAL}}$ hold at any step in the execution of a well-formed instruction sequence proved with these lemmas, as shown in Figure 6. That is, the execution of TAL code preserves state well-formedness, and never breaks the collector's invariant.

$$\begin{array}{rcl}
\mathsf{null} & ::= & 0 \\
\mathsf{st}, \mathsf{ed} & ::= & 8 \mid 16 \mid 24 \mid \ldots \\
\mathsf{ptrs} & \stackrel{\text{def}}{=} & \{\mathtt{l} \mid (\mathtt{l} \bmod 8 = 0) \wedge (\mathsf{st} \le \mathtt{l} < \mathsf{ed})\} \\
\mathsf{vptr}(\mathtt{l}) & \stackrel{\text{def}}{=} & \mathtt{l} \in \mathsf{ptrs} \\
\mathsf{roots} & \stackrel{\text{def}}{=} & \{\mathtt{r17}, \mathtt{r18}, \mathtt{r31}, \mathtt{r0}\}
\end{array}$$

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

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

$$\mathsf{rchrts}((\mathbb{H}, \mathbb{R}), \mathtt{l}) \stackrel{\text{def}}{=} \exists \mathtt{r} \in \mathsf{roots}.\ \mathsf{reach}(\mathbb{H}, \mathbb{R}(\mathtt{r}), \mathtt{l})$$

Figure 8. Pointer validity and reachability

On the other hand, to prove collector interface compatibility, we must show that the successful execution of each collector routine also preserves these invariants, as shown in Figure 6. That is, for each collector routine, its guarantee g satisfies the following relation, where Γ and Γ' are defined by the behavior of this routine.

$$\forall \Psi, \mathbb{S}, \mathbb{S}'.\ \mathrm{g}\ \mathbb{S}\ \mathbb{S}' \rightarrow [\![\Gamma]\!]_{\text{TAL}}\ \Psi\ \mathbb{S} \rightarrow [\![\Gamma']\!]_{\text{TAL}}\ \Psi\ \mathbb{S}'$$

Next we will present a case study that demonstrates the practicality and effectiveness of our methodology.

4. A certified conservative collector

Like TALx86 [20] and TALT [6], we choose a conservative garbage collector [3]. This kind of collector treats all values as potential pointers, eliminating the need to keep complex pointer location information in the TAL type system and simplifying the collector interface.

Our collector is a standard stop-the-world mark-sweep collector [14] and uses the valid pointer check procedure of the Boehm-Demers-Weiser collector [3]. To simplify the problem, our collector only allocates heap chunks with a fixed size of two words. The pseudo code of our collector is presented in Figure 7.

4.1. The specification interface

We define in Figure 8 the view of the heap that the collector and TAL must agree on. The constant address null is 0. The variables st and ed are the lower and upper bounds of the collector's allocatable heap, and are aligned at 8, the size of a heap chunk. Thus, a value $\mathtt{l}$ is a valid pointer ($\mathsf{vptr}(\mathtt{l})$) only if it falls in the range of the allocatable heap and points to the start of a heap chunk.

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

$$\mathsf{eq}(\mathbb{H}) \stackrel{\text{def}}{=} \lambda\mathbb{H}'.\ \mathbb{H}' = \mathbb{H}$$
$$\ldots$$
$$\begin{array}{l}\mathsf{gc_inv}((\mathbb{H},\mathbb{R}),\mathbb{H}_{\mathrm{T}}) \stackrel{\text{def}}{=} \exists B, F.\\ \quad \mathsf{sted_ok}(\mathbb{R}) \wedge B \cup F = \mathsf{ptrs} \wedge\\ \quad \mathsf{dom}(\mathbb{H}_{\mathrm{T}}) = \{\mathtt{l}, \mathtt{l}+4 \mid \mathtt{l} \in B\} \wedge\\ \quad \mathbb{H} \Vdash \mathsf{eq}(\mathbb{H}_{\mathrm{T}}) * \mathsf{flst}(F,\mathbb{R}) * \mathsf{mbits}(\mathsf{ptrs},0) * \mathsf{mstack}(\emptyset,\mathbb{R})\end{array}$$
$$\begin{array}{l}\mathsf{chkeq}(\mathbb{H},\mathbb{H}',\mathtt{l}) \stackrel{\text{def}}{=} \mathbb{H}(\mathtt{l}) = \mathbb{H}'(\mathtt{l}) \wedge \mathbb{H}(\mathtt{l}+4) = \mathbb{H}'(\mathtt{l}+4)\\ \mathsf{gc_step}((\mathbb{H},\mathbb{R}),(\mathbb{H}',\mathbb{R}')) \stackrel{\text{def}}{=}\\ \quad (\forall \mathtt{l}.\ \mathsf{rchrts}((\mathbb{H},\mathbb{R}),\mathtt{l}) \rightarrow \mathsf{chkeq}(\mathbb{H},\mathbb{H}',\mathtt{l})) \wedge\\ \quad (\forall \mathtt{r} \in \mathsf{roots}.\ \mathbb{R}(\mathtt{r}) = \mathbb{R}'(\mathtt{r}))\end{array}$$
$$\begin{array}{l}\mathsf{alloc_step}((\mathbb{H},\mathbb{R}),(\mathbb{H}',\mathbb{R}')) \stackrel{\text{def}}{=} \exists \mathtt{l}.\\ \quad \mathtt{l} \notin \mathsf{dom}(\mathbb{H}) \wedge (\mathtt{l}+4) \notin \mathsf{dom}(\mathbb{H}) \wedge\\ \quad \mathbb{H}' = \mathbb{H}\{\mathtt{l} \rightsquigarrow -\}\{\mathtt{l}+4 \rightsquigarrow -\} \wedge \mathbb{R}' = \mathbb{R}\{\mathtt{r18} \rightsquigarrow \mathtt{l}\}\end{array}$$
$$\begin{array}{l}\mathrm{p}_{\mathtt{a}} \stackrel{\text{def}}{=} \lambda\mathbb{S}.\ \exists\mathbb{H}_{\mathrm{T}}.\ \mathsf{gc_inv}(\mathbb{S},\mathbb{H}_{\mathrm{T}})\\ \mathrm{g}_{\mathtt{a}} \stackrel{\text{def}}{=} \lambda(\mathbb{H},\mathbb{R}),(\mathbb{H}',\mathbb{R}').\ \forall\mathbb{H}_{\mathrm{T}}.\ \mathsf{gc_inv}((\mathbb{H},\mathbb{R}),\mathbb{H}_{\mathrm{T}}) \rightarrow\\ \quad \exists\mathbb{H}'_{\mathrm{T}},\mathbb{H}^{\dagger}_{\mathrm{T}},\mathbb{R}^{\dagger}.\ \mathsf{gc_inv}((\mathbb{H}',\mathbb{R}'),\mathbb{H}'_{\mathrm{T}}) \wedge\\ \qquad \mathsf{gc_step}((\mathbb{H}_{\mathrm{T}},\mathbb{R}),(\mathbb{H}^{\dagger}_{\mathrm{T}},\mathbb{R}^{\dagger})) \wedge\\ \qquad \mathsf{alloc_step}((\mathbb{H}^{\dagger}_{\mathrm{T}},\mathbb{R}^{\dagger}),(\mathbb{H}'_{\mathrm{T}},\mathbb{R}'))\end{array}$$

Figure 9. Collector interface specification

itself. In the inductive case, $\mathtt{l}'$ is reachable from $\mathtt{l}$ if it is reachable from the pointers in the heap chunk at $\mathtt{l}$.

We define the collector's root set roots as the set of registers used by TAL. For simplicity, we have four registers in this set, but it would not be difficult make more registers usable in TAL. The predicate $\mathsf{rchrts}(\mathbb{S},\mathtt{l})$ asserts that $\mathtt{l}$ points to a live heap chunk in state $\mathbb{S}$.

4.2. Specification and proof construction

We now present the collector's safety specification and a discussion of the construction of the safety proof.

SCAP Specification. Our specification of the collector interface (`alloc`) includes the precondition $\mathrm{p}_{\mathtt{a}}$ and the guarantee $\mathrm{g}_{\mathtt{a}}$, as defined in Figure 9.

The collector's representation invariant gc_inv is defined using separation logic [24]. We write $\mathbb{H} \Vdash P$ if the heap predicate P, which has the type $Heap \rightarrow Prop$, is valid with $\mathbb{H}$. $\mathbb{H} \Vdash P * Q$ is valid if $\mathbb{H}$ can be split into two disjoint subheaps $\mathbb{H}_1$ and $\mathbb{H}_2$, such that both $\mathbb{H}_1 \Vdash P$ and $\mathbb{H}_2 \Vdash Q$ are valid propositions. The precondition of `alloc`, as defined with gc_inv, asserts that:

- The heap boundaries st and ed are stored in $\mathbb{R}$ ($\mathsf{sted_ok}(\mathbb{R})$). The set of allocatable pointers (ptrs) is split into the allocated subset B and the free subset F, while the allocated subheap $\mathbb{H}_{\mathrm{T}}$ contains exactly the heap chunks in B.
- The global heap $\mathbb{H}$ contains the allocated subheap $\mathbb{H}_{\mathrm{T}}$, the free list with the head pointer in $\mathbb{R}$ ($\mathsf{flst}(F,\mathbb{R})$), the mark bits for all the pointers in ptrs ($\mathsf{mbits}(\mathsf{ptrs},0)$), and the mark stack with the stack pointers stored in $\mathbb{R}$ ($\mathsf{mstack}(\emptyset,\mathbb{R})$).

$$\begin{array}{llll}(IFlag) & \varphi & ::= & 1 \mid 0\\ (WTy) & \tau & ::= & \alpha \mid \mathsf{nul} \mid \mathsf{int}\\ & & \mid & \Gamma \mid \langle\tau^{\varphi},\tau^{\varphi}\rangle \mid \mu\alpha.\tau \mid \tau\vee\tau\\ (RfTy) & \Gamma & ::= & \{\mathtt{r} \rightsquigarrow \tau\}^*\\ (CdSpec) & \theta & ::= & \Gamma\\ (DhSpec) & \Phi & ::= & \{\mathtt{l} \rightsquigarrow (\tau^{\varphi},\tau^{\varphi})\}^*\end{array}$$

$$\Psi \vdash_{\mathrm{TAL}} \Gamma : \mathbb{I} \stackrel{\text{def}}{=} \vdash \{\langle[\![\Gamma]\!]_{\mathrm{TAL}}\rangle_{\Psi}\}\ \mathbb{I}$$

$\boxed{\Psi \vdash_{\mathrm{TAL}} \Gamma : \mathbb{I}}$ *(Well-formed Instruction Sequence Lemmas)*

$$\frac{(\mathtt{f},\Gamma') \in \Psi \quad \vdash_{\mathrm{TAL}} \Gamma \leq \Gamma'}{\Psi \vdash_{\mathrm{TAL}} \Gamma : \mathtt{j}\ \mathtt{f}} \quad (\mathrm{J})$$

$$\frac{\begin{array}{c}(\mathtt{f},\Gamma') \in \Psi \quad (\mathtt{f}_{\mathtt{ret}},\Gamma'') \in \Psi\\ \vdash_{\mathrm{TAL}} \Gamma\{\mathtt{r31} \rightsquigarrow \Gamma''\} \leq \Gamma'\end{array}}{\Psi \vdash_{\mathrm{TAL}} \Gamma : \mathtt{jal}\ \mathtt{f},\ \mathtt{f}_{\mathtt{ret}}} \quad (\mathrm{JAL})$$

$$\frac{\Gamma(\mathtt{r_s}) = \tau \quad \vdash_{\mathrm{TAL}} \tau \leq \tau' \quad \Psi \vdash_{\mathrm{TAL}} \Gamma\{\mathtt{r_d} \rightsquigarrow \tau'\} : \mathbb{I}'}{\Psi \vdash_{\mathrm{TAL}} \Gamma : \mathtt{addiu}\ \mathtt{r_d},\ \mathtt{r_s},\ 0; \mathbb{I}'} \quad (\mathrm{MOV})$$

$$\frac{\begin{array}{c}\Psi(\mathtt{f}) = \Gamma' \quad \Gamma(\mathtt{r_s}) = \mathsf{nul} \vee \tau\\ \vdash_{\mathrm{TAL}} \Gamma\{\mathtt{r_s} \rightsquigarrow \mathsf{nul}\} \leq \Gamma' \quad \Psi \vdash_{\mathrm{TAL}} \Gamma\{\mathtt{r_s} \rightsquigarrow \tau\} : \mathbb{I}'\end{array}}{\Psi \vdash_{\mathrm{TAL}} \Gamma : \mathtt{beq}\ \mathtt{r_s},\ \mathtt{r0},\ \mathtt{f}; \mathbb{I}'} \quad (\mathrm{NULL})$$

Figure 10. TAL in LOCAP

The guarantee $\mathrm{g}_{\mathtt{a}}$ specifies the situation where a free chunk is successfully allocated. It simply divides the state transition of `alloc` into a collection phase and an allocation phase with an auxiliary state $(\mathbb{H}^{\dagger}_{\mathrm{T}},\mathbb{R}^{\dagger})$, and asserts that:

- The representation invariant gc_inv is preserved between the entry state $(\mathbb{H},\mathbb{R})$ and the return state $(\mathbb{H}',\mathbb{R}')$, with allocated subheaps $\mathbb{H}_{\mathrm{T}}$ and $\mathbb{H}'_{\mathrm{T}}$, respectively.
- The collection phase turns $(\mathbb{H}_{\mathrm{T}},\mathbb{R})$ into $(\mathbb{H}^{\dagger}_{\mathrm{T}},\mathbb{R}^{\dagger})$ and the gc_step relation asserts that the live chunks are equal in the two heaps, while the values of the root registers are equal in the two register files. The allocation phase turns $(\mathbb{H}^{\dagger}_{\mathrm{T}},\mathbb{R}^{\dagger})$ into $(\mathbb{H}'_{\mathrm{T}},\mathbb{R}')$ and the alloc_step relation asserts that $\mathbb{H}'_{\mathrm{T}}$ has exactly one more heap chunk than $\mathbb{H}^{\dagger}_{\mathrm{T}}$, with its pointer stored in $\mathbb{R}'(\mathtt{r18})$.

Proof Construction. The verification of the collector involves two main steps. We first form the verification environment Ψ_{GC} with the SCAP specifications for each label $\mathtt{l}$ in the collector's code heap $\mathbb{C}_{\mathrm{GC}}$. Then for each $\mathtt{l}$ we prove the CAP well-formedness of the corresponding code block $\mathbb{C}_{\mathrm{GC}}(\mathtt{l})$ with the SCAP lemmas in Figure 5. Due to space

$\boxed{\vdash_{\text{TAL}} * \quad \vdash_{\text{TAL}} * \leq *}$ (*Well-formed Type, Subtyping*)

$$\frac{\mathsf{ftv}(\tau) = \emptyset}{\vdash_{\text{TAL}} \tau} \text{ (WORD)} \qquad \frac{\Gamma(\mathtt{r}) = \tau \quad \vdash_{\text{TAL}} \tau \quad \forall\, \mathtt{r} \in \mathsf{dom}(\Gamma) \subseteq \mathsf{roots}}{\vdash_{\text{TAL}} \Gamma} \text{ (RFILE)}$$

$$\frac{\Phi(\mathtt{l}) = (\tau_0^{\varphi_0}, \tau_4^{\varphi_4}) \quad \vdash_{\text{TAL}} \tau_i \quad \mathsf{vptr}(\mathtt{l}) \quad \forall\, \mathtt{l} \in \mathsf{dom}(\Phi)}{\vdash_{\text{TAL}} \Phi} \text{ (HEAP)}$$

$$\frac{\Gamma(\mathtt{r}) = \Gamma'(\mathtt{r}) \quad \forall\, \mathtt{r} \in \mathsf{dom}(\Gamma')}{\vdash_{\text{TAL}} \Gamma \leq \Gamma'} \text{ (SUB)} \qquad \frac{\Gamma(\mathtt{r}) = \mu\alpha.\tau}{\vdash_{\text{TAL}} \Gamma \leq \Gamma\{\mathtt{r} \rightsquigarrow \tau[\mu\alpha.\tau/\alpha]\}} \text{ (UNFOLD)}$$

$$\frac{\Gamma(\mathtt{r}) = \tau[\mu\alpha.\tau/\alpha]}{\vdash_{\text{TAL}} \Gamma \leq \Gamma\{\mathtt{r} \rightsquigarrow \mu\alpha.\tau\}} \text{ (FOLD)} \qquad \frac{}{\vdash_{\text{TAL}} \tau^{\varphi} \leq \tau^{\varphi}} \text{ (REFL)} \qquad \frac{}{\vdash_{\text{TAL}} \tau^{1} \leq \tau^{0}} \text{ (0-1)}$$

$$\frac{}{\vdash_{\text{TAL}} \tau \leq \tau \vee \tau'} \text{ (UNIONL)} \qquad \frac{}{\vdash_{\text{TAL}} \tau' \leq \tau \vee \tau'} \text{ (UNIONR)} \qquad \frac{}{\vdash_{\text{TAL}} \mathsf{nul} \leq \mathsf{int}} \text{ (NULL-INT)}$$

$\boxed{\Psi; \Phi \vdash_{\text{TAL}} * : * \quad \Psi \vdash_{\text{TAL}} \mathbb{S} : \Gamma}$ (*Value, Heap, Rfile, State Typing*)

$$\frac{}{\Psi; \Phi \vdash_{\text{TAL}} 0 : \mathsf{nul}} \text{ (NULL)} \qquad \frac{}{\Psi; \Phi \vdash_{\text{TAL}} \mathtt{w} : \mathsf{int}} \text{ (INT)} \qquad \frac{(\mathtt{f}, \Gamma) \in \Psi}{\Psi; \Phi \vdash_{\text{TAL}} \mathtt{f} : \Gamma} \text{ (CODE)}$$

$$\frac{\vdash_{\text{TAL}} \mathsf{fst}(\Phi(\mathtt{l})) \leq \tau_0^{\varphi_0} \quad \vdash_{\text{TAL}} \mathsf{snd}(\Phi(\mathtt{l})) \leq \tau_4^{\varphi_4}}{\Psi; \Phi \vdash_{\text{TAL}} \mathtt{l} : \langle \tau_0^{\varphi_0}, \tau_4^{\varphi_4} \rangle} \text{ (TUPLE)} \qquad \frac{\Psi; \Phi \vdash_{\text{TAL}} \mathtt{w} : \tau \quad \vdash_{\text{TAL}} \tau \leq \tau'}{\Psi; \Phi \vdash_{\text{TAL}} \mathtt{w} : \tau'} \text{ (SUBTY)}$$

$$\frac{\Psi; \Phi \vdash_{\text{TAL}} \mathtt{w} : \tau[\mu\alpha.\tau/\alpha]}{\Psi; \Phi \vdash_{\text{TAL}} \mathtt{w} : \mu\alpha.\tau} \text{ (REC)} \qquad \frac{\Psi; \Phi \vdash_{\text{TAL}} \mathtt{w} : \tau}{\Psi; \Phi \vdash_{\text{TAL}} \mathtt{w} : \tau^{\varphi}} \text{ (INIT)} \qquad \frac{}{\Psi; \Phi \vdash_{\text{TAL}} \mathtt{w} : \tau^{0}} \text{ (JUNK)}$$

$$\frac{\vdash_{\text{TAL}} \Phi \quad \Phi(\mathtt{l}) = (\tau_0^{\varphi_0}, \tau_4^{\varphi_4}) \quad \Psi; \Phi \vdash_{\text{TAL}} \mathbb{H}(\mathtt{l} + i) : \tau_i^{\varphi_i} \quad \forall\, \mathtt{l} \in \mathsf{dom}(\Phi')}{\Psi; \Phi \vdash_{\text{TAL}} \mathbb{H} : \Phi'} \text{ (HEAP)}$$

$$\frac{\vdash_{\text{TAL}} \Gamma \quad \Psi; \Phi \vdash_{\text{TAL}} \mathbb{R}(\mathtt{r}) : \Gamma(\mathtt{r}) \quad \forall\, \mathtt{r} \in \mathsf{dom}(\Gamma)}{\Psi; \Phi \vdash_{\text{TAL}} \mathbb{R} : \Gamma} \text{ (RFILE)} \qquad \frac{\Psi; \Phi \vdash_{\text{TAL}} \mathbb{H} : \Phi \quad \Psi; \Phi \vdash_{\text{TAL}} \mathbb{R} : \Gamma}{\Psi \vdash_{\text{TAL}} \mathbb{S} : \Gamma} \text{ (STATE)}$$

Figure 11. TAL state typing rules

limitations, we omit detailed discussion of the collector's proof construction. Interested readers will find the assembly code implementation, SCAP specification and proof of the collector in [16].

5. A typed assembly language with GC

We show in Figure 10 our definition of TAL types, which includes code types, mutable reference types, recursive types and union types. We do not include a polymorphic code type, as it is orthogonal to our primary concern, memory management, and this extension should not be hard for our system.

Our TAL type system is built over the abstract machine in Section 2.1 and based on the definitions in Section 4.1, and thus is different from the original TAL [21] in several ways. Since both the registers and heap cells contain only word-size values, we use one value type τ for all values, instead of having *small value* and *heap value* types as in the original TAL. We also use fixed-sized tuple types to make it consistent with our collector, which allocates heap chunks with a size of two words. This does not reduce the expressiveness of our type system, since a tuple with arbitrary size can be encoded into a list of our fixed-sized tuples.

The TAL typing rules listed in Figure 11 are similar to those of the original TAL. However, we require that the domain of a well-formed code heap specification contains only valid heap pointers, and a well-formed register file type asserts only the root register set roots defined in Figure 8.

As partly listed in Figure 10, our TAL lemmas for CAP resemble the instruction typing rules of the original TAL. The definition of $\Psi \vdash_{\text{TAL}} \Gamma : \mathbb{I}$ is based on the TAL interpretation $[\![\ ]\!]_{\text{TAL}}$ in Section 3.1, and the representation invariant gc_inv in Figure 9. Instead of using the `malloc` macro of the original TAL, our TAL supports heap allocation by making a function call to the garbage collector (`jal alloc`, $\mathtt{f}_{\mathtt{ret}}$).

The readers should note that there are other possible sets of TAL lemmas for our type system besides the ones we used. The choice of these lemmas may also depend on the actual type-checking algorithm.

A well-formed TAL instruction sequence proved with the TAL lemmas keeps the invariant that at any step of its

execution the machine state of TAL is well-formed and the collector's invariant holds. We follow the soundness proof of the original TAL to prove that the execution preserves the TAL state typing relation. The preservation of the collector's invariant is proved by observing the fact that well-formed TAL instructions never change the heap's domain.

5.1. Collector interface compatibility

As the final step, we prove that the SCAP specification of the collector interface in Figure 9 is compatible with its TAL specification $\Gamma_{\mathtt{a}}$, which asserts that the function returns a pointer to a new heap chunk in register `r18` and that the types of the other TAL registers are preserved.

Theorem 2 (Collector interface compatibility).
For any code heap specification Ψ and any instantiation of word value types τ_a, τ_b, τ_0 and τ_4, we have:

$$\langle[\![\Gamma_{\mathtt{a}}]\!]\rangle_\Psi \Rightarrow \langle[\![(\mathrm{p}_{\mathtt{a}}, \mathrm{g}_{\mathtt{a}})]\!]\rangle_\Psi$$

where:

$$\Gamma_{\mathtt{a}} \stackrel{\text{def}}{=} \{\mathtt{r17} \rightsquigarrow \tau_a, \mathtt{r0} \rightsquigarrow \tau_b, \mathtt{r31} \rightsquigarrow \{\mathtt{r17} \rightsquigarrow \tau_a, \mathtt{r0} \rightsquigarrow \tau_b, \mathtt{r18} \rightsquigarrow \langle \tau_0^0, \tau_4^0 \rangle\}\}.$$

After unfolding the two interpretations, we get $\mathrm{p}_{\mathtt{a}}$ directly from $\langle[\![\Gamma_{\mathtt{a}}]\!]\rangle_\Psi$. Then, we instantiate the first parameter of wfst to 0. From Lemmas 3 and 5, we know from $\mathrm{g}_{\mathtt{a}}$ that the return state of `alloc` satisfies $\langle[\![\{\mathtt{r17} : \tau_a, \mathtt{r0} : \tau_b, \mathtt{r18} : \langle \tau_0^0, \tau_4^0 \rangle\}]\!]\rangle_\Psi$, as required by the unfolded wfst predicate. We list here the most important lemmas for proving Theorem 2.

Lemma 2 (Heap pruning).
If $\Psi; \Phi \vdash_{\text{TAL}} \mathbb{H} : \Phi$ and $\Psi; \Phi \vdash_{\text{TAL}} \mathbb{R} : \Gamma$, then:
1. $\Psi; \Phi_{/(\mathbb{H},\mathbb{R})} \vdash_{\text{TAL}} \mathbb{H} : \Phi_{/(\mathbb{H},\mathbb{R})}$;
2. $\Psi; \Phi_{/(\mathbb{H},\mathbb{R})} \vdash_{\text{TAL}} \mathbb{R} : \Gamma$.

where $\Phi_{/\mathbb{S}}$ is the data heap specification formed with exactly the live labels in the state $\mathbb{S}$ from Φ.

The proof of Lemma 2 follows the proof of the *heap update* lemma of the original TAL, but with additional case analysis to separate root-reachable pointers from the rest of the word values.

Lemma 3 (GC step).
If $\mathsf{gc_step}(\mathbb{S}, \mathbb{S}')$ and $\Psi \vdash_{\text{TAL}} \mathbb{S} : \Gamma$, then: $\Psi \vdash_{\text{TAL}} \mathbb{S}' : \Gamma$.

Lemma 3 is proved using Lemma 2 by observing that both $\Psi \vdash_{\text{TAL}} \mathbb{S} : \Gamma$ and $\Psi \vdash_{\text{TAL}} \mathbb{S}' : \Gamma$ can be proved using the same data heap specification $\Phi_{/\mathbb{S}}$.

Lemma 4 (Heap extension).
If $\Psi; \Phi \vdash_{\text{TAL}} \mathbb{H} : \Phi$, $\mathsf{vptr}(\mathtt{l})$, $\mathtt{l} \notin \mathsf{dom}(\mathbb{H})$, $\vdash_{\text{TAL}} \tau_0^{\varphi_0}$, and $\vdash_{\text{TAL}} \tau_4^{\varphi_4}$, then:
1. $\Psi; \Phi' \vdash_{\text{TAL}} \mathbb{R} : \Gamma$.
2. If $\Psi; \Phi \vdash_{\text{TAL}} \mathtt{w}_0 : \tau_0^{\varphi_0}$, and $\Psi; \Phi \vdash_{\text{TAL}} \mathtt{w}_4 : \tau_4^{\varphi_4}$, then $\Psi; \Phi' \vdash_{\text{TAL}} \mathbb{H}\{\mathtt{l} \rightsquigarrow \mathtt{w}_0\}\{\mathtt{l} + 4 \rightsquigarrow \mathtt{w}_4\} : \Phi'$.

where Φ' stands for $\Phi\{\mathtt{l} \rightsquigarrow (\tau_0^{\varphi_0}, \tau_4^{\varphi_4})\}$.

```
chase(list * i) {
  while(i <> NULL){
    i = i->next;
    i = alloc(0, i);
  }
}
```

Figure 12. An example

Lemma 5 (Allocation step).
If $\mathsf{alloc_step}(\mathbb{S}, (\mathbb{H}', \mathbb{R}'))$, $\Psi \vdash_{\text{TAL}} \mathbb{S} : \Gamma$, $\vdash_{\text{TAL}} \tau_0$, and $\vdash_{\text{TAL}} \tau_4$, then: $\Psi \vdash_{\text{TAL}} (\mathbb{H}', \mathbb{R}') : \Gamma\{\mathtt{r} : \langle \tau_0^0, \tau_4^0 \rangle\}$.

The proof of Lemma 4 resembles the proof of the *heap extension* lemma of the original TAL. Lemma 5 is trivially derivable from Lemma 4.

6. An example of linked code

We now give an example to show the safe linking of code verified in TAL with our collector. The pseudo code of `chase` is given in Figure 12, which repeatedly removes a node from a list and appends a new one. If `i` is not null initially, the program will surely run out of memory without a collector. We type check the assembly implementation $\mathbb{C}_{\text{C}}$ in Figure 13 with the following code heap specification. The skipped Γs are listed at the corresponding labels in Figure 13.

$$\Psi_{\text{C}} \stackrel{\text{def}}{=} \{(\mathtt{alloc}, \{\mathtt{r17} \rightsquigarrow \mathsf{list}, \mathtt{r0} \rightsquigarrow \mathsf{int}, \mathtt{r31} \rightsquigarrow \{\mathtt{r17} \rightsquigarrow \mathsf{list}, \mathtt{r0} \rightsquigarrow \mathsf{int}, \mathtt{r18} \rightsquigarrow \langle \mathsf{int}^0, \mathsf{list}^0 \rangle\}\}), (\mathtt{init}, \cdots), (\mathtt{chase}, \cdots), (\mathtt{write}, \cdots), (\mathtt{ret}, \cdots)\}.$$

When the instructions pass type checking, for each $(\mathtt{l}, \Gamma)$ pair in Ψ_{C} we get the proof that:

$$\vdash \{\langle[\![\Gamma]\!]\rangle_{\Psi_{\text{C}}}\}\ \mathbb{C}_{\text{C}}(\mathtt{l})$$

From Section 4, we have for each $(\mathtt{l}, (\mathrm{p}, \mathrm{g}))$ pair in the collector's code heap specification Ψ_{GC} that:

$$\vdash \{\langle[\![(\mathrm{p}, \mathrm{g})]\!]\rangle_{\Psi_{\text{GC}}}\}\ \mathbb{C}_{\text{GC}}(\mathtt{l})$$

We also obtain from Theorem 2 and Lemma 1 that:

$$\vdash \{\langle[\![(\Gamma_{\mathtt{a}})]\!]\rangle_{\Psi_{\text{GC}}}\}\ \mathbb{C}_{\text{GC}}(\mathtt{alloc})$$

We form the global code heap $\mathbb{C}$ and its specification Ψ:

$$\mathbb{C} \stackrel{\text{def}}{=} \mathbb{C}_{\text{C}} \cup \mathbb{C}_{\text{GC}} \qquad \Psi \stackrel{\text{def}}{=} \Psi_{\text{C}} \cup \Psi_{\text{GC}}$$

With Lemma 1, we have for each $(\mathtt{l}, \theta)$ pair in Ψ that:

$$\vdash \{\langle[\![\Psi(\mathtt{l})]\!]\rangle_\Psi\}\ \mathbb{C}(\mathtt{l})$$

Finally, we obtain the well-formedness of the linked code $\mathbb{C}$ with the CDHP rule in Figure 4.

```
list ≝ μα. nul ∨ ⟨int¹, α¹⟩

init:                               {r17 ⇝ list, r0 ⇝ int}
  j chase                                         # unfold

chase:           {r17 ⇝ nul ∨ ⟨int¹, list¹⟩, r0 ⇝ int}
  beq r17, r0, ret                             # null elim
                         {r17 ⇝ ⟨int¹, list¹⟩, r0 ⇝ int}
  lw r17, 4(r17)                               # load next
                                  {r17 ⇝ list, r0 ⇝ int}
  jal alloc, write

write:     {r17 ⇝ list, r18 ⇝ ⟨int⁰, list⁰⟩, r0 ⇝ int}
  sw r0, 0(r18)                                # write val
           {r17 ⇝ list, r18 ⇝ ⟨int¹, list⁰⟩, r0 ⇝ int}
  sw r17, 4(r18)                              # write next
           {r17 ⇝ list, r18 ⇝ ⟨int¹, list¹⟩, r0 ⇝ int}
  addiu r17, r18, 0                                 # move
{r17 ⇝ nul ∨ ⟨int¹, list¹⟩, r18 ⇝ ⟨int¹, list¹⟩, r0 ⇝ int}
  j chase                                     # sub domain

ret:                                                  { }
  j ret
```

Figure 13. An example (assembly)

7. Implementation

Our verification is fully mechanized within Coq [5], an interactive theorem prover that uses CiC as its underlying logic, where specifications and proofs are constructed as types and terms in CiC, respectively. Proof checking in Coq is thus type checking of terms in CiC, which is easier to implement and more trustworthy. Coq also provides a rich language for defining both logical and computational constructors, with the ability to construct inductive predicates and well-formed recursive functions. Using this, we build the abstract machine model and the sound program logics.

The tricky part of the implementation is to obtain the pruned data heap specification $\Phi_{/\mathbb{S}}$ mentioned in Section 5.1, which implies that every label `l` in its domain satisfies $\mathsf{rchrts}(\mathbb{S}, \mathtt{l})$. As Φ is a mapping function in the Set universe, we cannot get $\Phi_{/\mathbb{S}}$ by a case analysis on the proof of the decidability of $\mathsf{rchrts}(\mathbb{S}, \mathtt{l})$ (if it can be constructed directly), as this will break the proof-irrelevance axiom that is commonly accepted. To solve this problem, we define in the Set universe a well-formed recursive Boolean function which is equivalent to the predicate rchrts, and obtain $\Phi_{/\mathbb{S}}$ by case analysis on the return value of this function.

To simplify the proof construction, we have implemented (in Coq) a verification condition generator (VCGen) for SCAP and proved its correctness. We have also built various automated proof tactics such as those involving separation logic. This results in a proof which is about $1/4$ the size of our first proof and is much easier to follow.

We omit the implementation of a type-checker for TAL, since it is orthogonal to the main goal of this work. Building a certified TAL type checker is not hard since it has a very straight-forward (and decidable) type-checking algorithm.

Lines	Component
833	Basic properties and tactics
1941	Abstract machine encoding and lemmas
1263	Finite set library
884	Separation logic library
398	LOCAP
1188	TAL in LOCAP
874	Reachability properties
237	GC Safety for TAL
360	SCAP in LOCAP, VCGen and related tactics
154	Code, specification and proof of `chase`
2618	Code, specification and proof of the collector
276	Link up `chase` and the collector in LOCAP

Figure 14. Proof script size

In Figure 14 we give a breakdown of the size of our proofs for our foundational TAL with certified GC. For each component we give the number of non-empty lines of Coq proof scripts. The work took several man-months (by programmers who are familiar with the Coq system) to complete. Interested readers can obtain the Coq implementation from our project web site [16].

8. More related work and conclusion

Much work has been done concerning TAL and garbage collector safety in addition to those mentioned in Section 1. TALT [6] considered the impact of garbage collection on the soundness of its type system and mechanically proved GC safety assuming a conservative collector, but the type system interface for the collector is not clearly defined and it is unclear how their definition of GC safety can be used to link with a real collector. Vanderwaart and Crary [25] proposed a type system with an interface for an accurate garbage collector. But again, this work addresses only the mutator (TAL) side of GC safety, while our work complements their work by mechanically proving the safety of both TAL and the collector, including their interaction.

Earlier work on mechanized verification of garbage collectors (such as [13, 10, 4]) focused mostly 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. However, our verification only addresses the safety of the collector, not any liveness properties.

The work on CAP systems [27, 22, 8, 7] provides a nice way to build FPCC packages. Our work builds on the CAP0 system in [8] and the OCAP system in [7]. Feng *et al.* [7] described linking TAL with a certified `malloc` function.

Our idea of using interpretation to specify the TAL/collector interface is borrowed from this work.

We introduce in this paper a general methodology based on a new variant of FPCC for combining foundational TAL with a certified garbage collector. We demonstrate the practicality of this approach by linking a typical TAL with a conservative garbage collector. Our work is fully mechanized in the Coq proof assistant and the certified programs can be shipped immediately as FPCC packages. In the future we plan to extend this work by applying our methodology to link TAL with more complex accurate collectors.

Acknowledgment

This research is based on work supported in part by National Science Foundation (of USA) under Grant CCR-0524545, gifts from Intel (USA), Microsoft, and Intel China Research Center, Innovation Funds from Chinese Academy of Sciences, and the National Natural Science Foundation of China under Grant No. 60673126. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

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

[2] L. Birkedal, N. Torp-Smith, and J. C. Reynolds. Local reasoning about a copying garbage collector. In *Proc. of the 31st ACM symp. on Principles of Prog. Lang.*, pages 220–231, New York, NY, USA, 2004. ACM Press.

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

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

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

[6] K. Crary. Toward a foundational typed assembly language. In *Proc. of the 30th ACM Symp. on Principles of Prog. Lang.*, pages 198–212, Jan. 2003.

[7] X. Feng, Z. Ni, Z. Shao, and Y. Guo. An open framework for foundational proof-carrying code. In *Proc. 3rd ACM Workshop on Types in Language Design and Implementation*, pages 67–78, Nice, France, Jan. 2007. ACM Press.

[8] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni. Modular verification of assembly code with stack-based control abstractions. In *PLDI '06: Proc. of the 2006 ACM SIGPLAN conference on Prog. Lang. Design and Impl.*, June 2006.

[9] N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. In *Proc. 17th Annual IEEE Symp. on Logic in Computer Science*, pages 89–100, July 2002.

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

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

[12] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, Oct. 1969.

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

[14] R. E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[15] J. Larus. SPIM: a MIPS32 simulator. v7.3, 2006.

[16] C. Lin, A. McCreight, Z. Shao, Y. Chen, and Y. Guo. Foundational typed assembly language with certified garbage collection (documents and Coq implementation). `http://flint.cs.yale.edu/publications/talgc.html`, 2007.

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

[18] MIPS Technologies, Inc. MIPS32$^{\mathrm{TM}}$ Architecture For Programmers Volume II: Instruction Set, v2.50.

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

[20] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, USA, May 1999.

[21] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *TOPLAS*, 21(3):527–568, 1999.

[22] Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *Proc. 33rd ACM SIGPLAN-SIGACT symp. on Principles of prog. lang.*, Jan. 2006.

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

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

[25] J. C. Vanderwaart and K. Crary. A typed interface for garbage collection. In *Proc. 1st ACM SIGPLAN Int'l Workshop on Types in Lang. Design and Impl.*, pages 109–122, New York, NY, USA, 2003. ACM Press.

[26] D. C. Wang and A. W. Appel. Type-preserving garbage collectors. In *Proc. 28th ACM Symp. on Principles of Prog. Lang.*, pages 166–178. ACM Press, 2001.

[27] D. Yu, N. A. Hamid, and Z. Shao. Building certified libraries for PCC: Dynamic storage allocation. *Science of Computer Programming*, 50(1-3):101–127, 2004.