Proving Correctness of a Garbage Collector via Local Reasoning

Lars Birkedal [birkedal@itu.dk], Noah Torp-Smith [noah@itu.dk]

The IT University of Copenhagen

Joint work with John C. Reynolds, Carnegie Mellon University
Motivation
Motivation

- Copying garbage collectors widely used, for example in implementations of functional languages.
Motivation

- Copying garbage collectors widely used, for example in implementations of functional languages.
- A “non-toy” example. Yang’s proof of the Schoor-Waite algorithm is another such.
Motivation

- Copying garbage collectors widely used, for example in implementations of functional languages.
- A “non-toy” example. Yang’s proof of the Schoor-Waite algorithm is another such.
- Proof Carrying Code theory assumes an underlying memory allocator, but doesn’t treat it further.
Setup: A user language and an implementation language, both standard while-languages, but with different memory interactions:

\[
\begin{align*}
C_{user} & ::= \cdots \mid x := \text{cons}(e_1, e_2) \mid x.i := e \mid x := y.i \\
C_{impl} & ::= \cdots \mid [e] := e \mid x := [e]
\end{align*}
\]

User language: No pointer arithmetic, “implicit type system”: \(\text{Vals} = \text{Ints} \cup \text{Ptr}\), heaps map pointers to pairs of values.

Implementation language: Pointer arithmetic, Heaps map locations (a subset of integers) to integers.
**Interface** (informal): The command $\text{cons}(e_1, e_2)$ in the user language results in a call to $\text{alloc}$ in implementation language.

```plaintext
alloc(l, n_1, n_2) {
    if (any_space_left) {
        allocate 2 heap cells;
        store(n_1, n_2);
        return address;
    }
    else {
        Garbage collect;
        alloc(l, n_1, n_2);
    }
}
```
• All values allocated by the user language at runtime (through cons-operations) are pairs of values, so the garbage collector only needs to deal with pairs of locations. A pointer is a first component of such a pair.
Preliminaries (3)

- All values allocated by the user language at runtime (through cons-operations) are pairs of values, so the garbage collector only needs to deal with pairs of locations. A pointer is a first component of such a pair.
- Pointers are divisible by 8.
Preliminaries (3)

- All values allocated by the user language at runtime (through \texttt{cons}-operations) are pairs of values, so the garbage collector only needs to deal with pairs of locations. A \textit{pointer} is a first component of such a pair.
- Pointers are divisible by 8.
- A simplifying assumption: Only one pointer in user language, call it \texttt{root} (more “root pointers” do not add anything interesting to the proof).
Preliminaries (4)

So a picture might look like this:

```
root
   / \   /
  /   \ /   /
/     / 0   /
1     3     4     5
```

Note that some cells cannot be “reached” from the root cell. These are ignored by copying collectors.
Preliminaries (4)

So a picture might look like this:

Note that some cells cannot be “reached” from the root cell. These are ignored by copying collectors.
A **weak heap isomorphism** \( \varphi : (s', h') \cong (s, h) \) is a bijection \( \varphi : \text{dom}(h') \cong \text{dom}(h) \) such that for all \( p \in \text{dom}(h') \),

\[
h(\varphi(p)) = \varphi^*(h'(p)),
\]

where \( \varphi^* \) is the extension of \( \varphi \) to all integers (pointers and nonpointers) with the identity on nonpointers. If also \( \varphi(s'(\text{root})) = s(\text{root}) \), we call \( \varphi \) a **heap isomorphism**.
A weak heap isomorphism $\varphi : (s', h') \cong (s, h)$ is a bijection $\varphi : \text{dom}(h') \cong \text{dom}(h)$ such that for all $p \in \text{dom}(h')$,

$$h(\varphi(p)) = \varphi^*(h'(p)),$$

where $\varphi^*$ is the extension of $\varphi$ to all integers (pointers and nonpointers) with the identity on nonpointers. If also $\varphi(s'(\text{root})) = s(\text{root})$, we call $\varphi$ a heap isomorphism.

$(s, h)$ is a garbage collected version of $(s', h')$, if there is a heap isomorphism $\varphi : \text{prune}(s, h) \cong \text{prune}(s', h')$. We do not have to remove anything.
Preliminaries (5)

- A **weak heap isomorphism** \( \varphi : (s', h') \cong (s, h) \) is a bijection \( \varphi : \text{dom}(h') \cong \text{dom}(h) \) such that for all \( p \in \text{dom}(h') \),
  \[
  h(\varphi(p)) = \varphi^*(h'(p)),
  \]
  where \( \varphi^* \) is the extension of \( \varphi \) to all integers (pointers and nonpointers) with the identity on nonpointers. If also \( \varphi(s'(\text{root})) = s(\text{root}) \), we call \( \varphi \) a **heap isomorphism**.

- \((s, h)\) is a **garbage collected version** of \((s', h')\), if there is a heap isomorphism \( \varphi : \text{prune}(s, h) \cong \text{prune}(s', h') \). We do not have to remove anything.

- So if \( GC \) is our garbage collector, and if \( GC, s, h \leadsto s', h' \) the requirement is that \((s', h')\) is a garbage collected version of \((s, h)\).
Cheney’s Algorithm (1970)

Assumes 2 contiguous “semi-spaces” of equal size, \( OLD = [\text{startOld}; \text{endOld}] \) and \( NEW = [\text{startNew}; \text{endNew}] \)

\[ s(\text{root})^2 \text{OLD} \]

\( \text{ALIVE} = f \quad \text{if} \quad \text{p is reachable} \quad g \).

Copies \( \text{ALIVE} \) to \( \text{NEW} \) in a “structure preserving way” and resumes allocation there.

The example from before:

\[
\begin{align*}
1 & \quad 2 \\
3 & \quad 4 & \quad 5 \\
\text{root} & \quad \text{OLD} & \quad \text{NEW}
\end{align*}
\]

\[
1 & \quad 2 & \quad 3 & \quad 4 & \quad 5 \\
\text{free} & \quad \text{scan} & \quad \text{root}
\]
Cheney’s Algorithm (1970)

Assumes 2 contiguous “semi-spaces” of equal size,

\[
\text{OLD} = [\text{startOld}, \text{endOld}] \quad \text{and} \quad \text{NEW} = [\text{startNew}, \text{endNew}],
\]

\(s(\text{root}) \in \text{OLD}. \quad \text{ALIVE} = \{p \mid p \text{ is reachable}\}. \quad \text{Copies ALIVE to NEW in a “structure preserving way” and resumes allocation there.}\]
Cheney’s Algorithm (1970)

Assumes 2 contiguous “semi-spaces” of equal size,

\[
\text{OLD} = [\text{startOld}, \text{endOld}] \quad \text{and} \quad \text{NEW} = [\text{startNew}, \text{endNew}],
\]

\(s(\text{root}) \in \text{OLD}. \ \text{ALIVE} = \{p \mid p \text{ is reachable}\}. \ \text{Copies ALIVE to NEW in a “structure preserving way” and resumes allocation there.}

The example from before:
An Example

*Initializing code:* Copy the root cell and update the first component to point to the copy:
**An Example (2)**

*Scanning a pointer-component (1):* If the first component of the cell it points to is not a pointer into NEW, we just copy the cell and update its first component. Then we update the component we are scanning:

![Diagram showing the process of scanning a pointer-component](image-url)
An Example (3)

... and again:
An Example (4)

*Scanning a non-pointer component*: Nothing happens.
An Example (5)

Scanning two pointer components as before:
An Example (6)

**Scanning a pointer-component (2):** If the first component of the cell it points to *is* a pointer into NEW, we do *not* make another copy; we just update the component, we are scanning appropriately:
An Example (7)

After this, nothing more interesting happens, and we update root:

```
2 3 5
1 2 3 4 5
```

...and we're done!
An Example (7)

After this, nothing more interesting happens, and we update root:

...and we’re done!
Extension of Separation Logic

To formalize our partition, we extend the term language with *finite sets of pointers*:
Extension of Separation Logic

To formalize our partition, we extend the term language with *finite sets of pointers*:

\[ m ::= \cdots \mid m^f s \oplus e \mid m^f s \ominus e \mid Itv(e, e) \mid \cdots \]
Extension of Separation Logic

To formalize our partition, we extend the term language with _finite sets of pointers:_

\[ m ::= \cdots \mid m^{fs} \oplus e \mid m^{fs} \ominus e \mid Itv(e, e) \mid \cdots \]

We will also need _finite relations:_

\[ f ::= \cdots \mid f \circ g \mid f \odot g \]
Extension of Separation Logic

To formalize our partition, we extend the term language with *finite sets of pointers*:

\[ m ::= \cdots \mid m_{fs} \oplus e \mid m_{fs} \ominus e \mid Itv(e, e) \mid \cdots \]

We will also need *finite relations*:

\[ f ::= \cdots \mid f \circ g \mid f \odot g \]

Semantics for \( \odot \): extension with identity on non-pointers:

\[
[f \odot h] = \{(p, n) \mid ((p, n) \in [h]s \wedge n \notin Ptr) \lor \\
(\exists p' \in Ptr. (p, p') \in [h]s \wedge (p', n) \in [f]s)\}
\]
Extension of Separation Logic (2)

We will also have new assertion forms. We mention some:
Extension of Separation Logic (2)

We will also have new assertion forms. We mention some:

- $p \in m$, $m_1 = m_2$, $(x_1, x_2) \in f$, $\text{iso}(f, m_1, m_2)$, $\text{Tfun}(f, m)$. Semantics is straightforward.
We will also have new assertion forms. We mention some:

- \( p \in m, m_1 = m_2, (x_1, x_2) \in f, \text{iso}(f, m_1, m_2), \text{Tfun}(f, m) \). Semantics is straightforward.

- *Iterated Separating Conjunction* over a finite set:

\[
\forall *p \in m. \ A(p)
\]
Extension of Separation Logic (2)

We will also have new assertion forms. We mention some:

- $p \in m, m_1 = m_2, (x_1, x_2) \in f, \text{iso}(f, m_1, m_2), \text{Tfun}(f, m)$. Semantics is straightforward.

- *Iterated Separating Conjunction* over a finite set:

\[
\forall_* p \in m. \ A(p)
\]

Semantics:

\[
s, h \models \forall_* p \in m. \ A(p) \iff \\
\llbracket m \rrbracket s = \emptyset \text{ implies } s, h \models \text{emp}, \text{ and} \\
\llbracket m \rrbracket s = \{p_1, \ldots, p_k\} \text{ implies} \\
s, h \models A(p_1) * \cdots * A(p_k)
\]
Interlude

Recall from Owicki, Gries:

Let $C$ be a command, and let $AV$ be a set of variables that appear in $C$ only in assignments $x := E$, where $x \in AV$. Then $AV$ is an auxiliary variable set for $C$. 
Recall from Owicki, Gries:

Let $C$ be a command, and let $AV$ be a set of variables that appear in $C$ only in assignments $x := E$, where $x \in AV$. Then $AV$ is an auxiliary variable set for $C$.

Let $AV$ be an auxiliary variable set for $C'$, and $P$ and $Q$ assertions not containing free variables from $AV$. Let $C$ be the command obtained from $C'$ by deleting all assignments to the variables in $AV$. Then

$$\frac{\{P\} C' \{Q\}}{\{P\} C \{Q\}}$$
Local Reasoning

The most interesting part of the proof is when we copy a cell. We prove a local specification and use the Frame Rule. The local specification only mentions the “footprint” of the program fragment ($x$ is cell pointed to by $\text{scan}$):
Local Reasoning

The most interesting part of the proof is when we copy a cell. We prove a local specification and use the Frame Rule. The local specification only mentions the “footprint” of the program fragment ($x$ is cell pointed to by scan):

\[
\begin{align*}
\{(\exists q. (x, q) \in \text{head} \land x \leftrightarrow q) \}^* \\
(\exists q'. (x, q') \in \text{tail} \land x + 4 \leftrightarrow q')^* \\
(\text{scan} \leftrightarrow -)^* (\text{free} \leftrightarrow -, -)
\end{align*}
\]

CopyCell

\[
\begin{align*}
\{((x \leftrightarrow \text{free}, -) \}^* (\text{scan} \leftrightarrow \text{free})^* \\
(\text{free} \leftrightarrow t_1, t_2)) \land \\
(x, t_1) \in \text{head} \land (x, t_2) \in \text{tail}
\end{align*}
\]
At some stage of our example, we had the following situation:
Informal Analysis

At some stage of our example, we had the following situation:

We will partition the cells into different “kinds”.
Sets of Pointers: Some cells in OLD have not yet been copied yet.
Sets of Pointers: Some cells in OLD have not yet been copied yet.

We call the set of these pointers UNFORW.
Sets of Pointers: Some cells in OLD have been copied, they’re marked with a “forward pointer” in the first component.
Sets of Pointers: Some cells in OLD have been copied, they’re marked with a “forward pointer” in the first component.

We call the set of these pointers FORW.
Sets of Pointers: Some cells in NEW have been copied and scanned, they will not be modified (or read) further.
Sets of Pointers: Some cells in NEW have been copied and scanned, they will not be modified (or read) further.

We call the set of these pointers FIN. FIN = [startNew, scan].
Sets of Pointers: Some cells in NEW have been copied but not scanned.

\[ \text{UNFIN} = \text{scan}; \text{free} \]
Sets of Pointers: Some cells in NEW have been copied but not scanned.

We call the set of these pointers UNFIN. UNFIN = [scan, free].
Sets of Pointers: Some cells in NEW have been copied but not scanned.

We call the set of these pointers UNFIN. \( \text{UNFIN} = [\text{scan}, \text{free}] \).

Finally, \( \text{FREE} = [\text{free}, \text{endNew}] \) is “free for allocation”.
The Proof

We will have
The Proof

We will have

- The sets mentioned before
The Proof

We will have

- The sets mentioned before
- Relations head and tail that record the initial heap
The Proof

We will have

- The sets mentioned before
- Relations $\text{head}$ and $\text{tail}$ that record the initial heap
- $\varphi$, a bijection,

\[ \varphi : \text{FORW} \rightarrow \text{BUSY} = \text{FIN} \cup \text{UNFIN} = \text{[startNew, free]} \]
The Proof

We will have

- The sets mentioned before
- Relations $\text{head}$ and $\text{tail}$ that record the initial heap
- $\varphi$, a bijection,

$$\varphi : \text{FORW} \rightarrow \text{BUSY} = \text{FIN} \cup \text{UNFIN} = [\text{startNew}, \text{free}]$$

These are all added to the program as auxiliary variables, and will be part of the proof.
The Proof (2)

Analysis of each set:
The Proof (2)

Analysis of each set:

- UNFORW is not yet modified, so we can use head, tail.

\[
A_{Uf} \equiv \forall_{p} p \in \text{UNFORW}. ((\exists q. (p, q) \in \text{head} \land p \leftrightarrow q)^* \\
(\exists q'. (p, q') \in \text{tail} \land p + 4 \leftrightarrow q'))
\]
The Proof (2)

Analysis of each set:

- UNFORW is not yet modified, so we can use head, tail.

\[ A_{Uf} \equiv \forall _*p \in \text{UNFORW. } (((\exists q. (p, q) \in \text{head} \land p \leftrightarrow q)^* \\
(\exists q'. (p, q') \in \text{tail} \land p + 4 \leftrightarrow q')) \]

- FORW: First component points to cell determined by \( \varphi \):

\[ A_{Fw} \equiv \forall _*p \in \text{FORW. } (\exists q. (p, q) \in \varphi \land p \leftrightarrow q, \neg) \]
The Proof (2)

Analysis of each set:

- UNFORW is not yet modified, so we can use head, tail.

\[ A_{Uf} \equiv \forall \ast p \in \text{UNFORW. } ((\exists q. (p, q) \in \text{head} \land p \leftrightarrow q) \ast \newline\]
\[ (\exists q'. (p, q') \in \text{tail} \land p + 4 \leftrightarrow q')) \]

- FORW: First component points to cell determined by \( \varphi \):

\[ A_{Fw} \equiv \forall \ast p \in \text{FORW. } (\exists q. (p, q) \in \varphi \land p \leftrightarrow q, -) \]

- FREE. Pointers here are in the domain of the heap:

\[ A_{Fr} \equiv \forall \ast p \in \text{FREE. } p \leftrightarrow -, - \]
The Proof (3)

Analysis of each set, ct’d:

- **UNFIN**: Each cell is a copy of the cell in FORW that points to it:

\[
A_{Un} \equiv \forall \exists^* p \in \text{UNFIN. } ((\exists q. \langle p, q \rangle \in \text{head } \circ \varphi^T \land p \rightarrow q)^* \\
(\exists q'. \langle p, q' \rangle \in \text{tail } \circ \varphi^T \land p + 4 \rightarrow q'))
\]
The Proof (4)

- FIN: scanned version of cells in UNFIN. Scanning means updating component to $\varphi$-value (but only if the component is a pointer). This is captured by $\circ$:

$$A_{Fn} \equiv \forall p \in \text{UNFIN. } ((\exists q. (p, q) \in \varphi \circ (\text{head} \circ \varphi^T) \land p \rightarrow q)^* \ (\exists q'. (p, q') \in \varphi \circ (\text{tail} \circ \varphi^T) \land p + 4 \rightarrow q'))$$
The Proof (5)

The Precondition:

\[ \text{InitAss} \equiv \]
\[ \text{Ptr}(\text{startNew}) \land \text{Ptr}(\text{endNew}) \land \text{Ptr}(\text{root}) \land \text{Disjoint}(\text{OLD}, \text{NEW}) \land \]
\[ \text{SbSet}(\text{ALIVE}, \text{OLD}) \land \text{Reachable}(\text{ALIVE}, \text{root}) \land \]
\[ \#\text{NEW} = \#\text{OLD} \land \text{PtrRg}(\text{head}, \text{ALIVE}) \land \text{PtrRg}(\text{tail}, \text{ALIVE}) \land \]
\[ \text{Tfun}(\text{head}, \text{ALIVE}) \land \text{Tfun}(\text{tail}, \text{ALIVE}) \land \]
\[ ( (\forall_* p \in \text{ALIVE}. ((\exists q. (p, q) \in \text{head} \land p \mapsto q)^*) \]
\[ (\exists q. (p, q') \in \text{tail} \land p + 4 \mapsto q'))^* ) \]
\[ (\forall_* p \in \text{NEW}. p \mapsto -, -, -)^* T ) \]

The \( T \) deals with “unreachable” cells (they are framed out).
The Proof (6)

The Invariant:

\[ I \equiv \]
\[ \text{iso}(\varphi, \text{FORW}, \text{BUSY}) \land \text{isUnion}(	ext{FORW}, \text{UNFORW}, \text{ALIVE}) \land \]
\[ \#\text{ALIVE} \leq \#\text{NEW} \land \text{root} \in \text{FORW} \land \text{scan} \leq \text{free} \land \]
\[ \text{Disjoint} (\text{ALIVE}, \text{NEW}) \land \text{Ptr} (\text{free}) \land \text{Ptr} (\text{scan}) \land \text{Ptr} (\text{offset}) \land \]
\[ \text{Ptr} (\text{maxFree}) \land \text{Reachable} (\text{ALIVE}, \text{root}) \land \]
\[ \text{PtrRg} (\text{head}, \text{ALIVE}) \land \text{PtrRg} (\text{tail}, \text{ALIVE}) \land \]
\[ \text{Tfun} (\text{head}, \text{ALIVE}) \land \text{Tfun} (\text{tail}, \text{ALIVE}) \land \]
\[ (A_{Uf} \ast A_{Fw} \ast A_{Fn} \ast A_{Un} \ast A_{Fn}) \]
The Proof (7)

Remarks about the Proof:

The proof of the specification is entirely formal: uses only proof-rules, not "semantical arguments". The proof that the invariant is strong enough to conclude that there is a heap isomorphism, is almost logical: We prove logically that,

\[ I^{\text{scan}} = \text{free!} (p_{2}\text{ALIVE}^{\text{!}} (q, r) (p_{r})_{2}^{!}) \]  

Recall equation for heap isos:

\[ h_{0}(p) = h(p)_{p} \]
Remarks about the Proof:

- The proof of the specification is entirely formal: uses only proof-rules, not “semantical arguments”.

Recall equation for heap isos:
The Proof (7)

Remarks about the Proof:

- The proof of the specification is entirely formal: uses only proof-rules, not “semantical arguments”.
- The proof that the invariant is strong enough to conclude that there is a heap isomorphism, is almost logical: We prove logically that,

\[ I \land \text{scan} = \text{free} \rightarrow (p \in \text{ALIVE} \land (p, q) \in \varphi \rightarrow (q \leftrightarrow r \leftrightarrow (p, r) \in \varphi \odot \text{head})) \]

Recall equation for heap isos:

\[ h'(\varphi(p)) = \varphi^*(h(p)) \]
Conclusion and Future Work

Formal proof of an algorithm that is used in practice. Method of finite sets and relations is believed to be widely applicable, so further study is needed. A more precise formulation of interface issues is needed. A technical report will be available soon.
Conclusion and Future Work

- Formal proof of an algorithm that is used in practice.
Conclusion and Future Work

- Formal proof of an algorithm that is used in practice.
- Method of finite sets and relations is believed to be widely applicable, so further study is needed.
Conclusion and Future Work

- Formal proof of an algorithm that is used in practice.
- Method of finite sets and relations is believed to be widely applicable, so further study is needed.
- A more precise formulation of *interface issues* is needed.
Conclusion and Future Work

- Formal proof of an algorithm that is used in practice.
- Method of finite sets and relations is believed to be widely applicable, so further study is needed.
- A more precise formulation of interface issues is needed.
- A technical report will be available soon.