Polishing a Rough Diamond
An Enhanced Separation Logic for Heap Space under Garbage Collection

Alexandre Moine    Arthur Charguéraud    François Pottier
ASL’22
A program logic to verify heap space bounds...
A program logic to verify **heap space bounds**...

...for an **imperative λ-calculus**...
A program logic to verify heap space bounds...

...for an imperative $\lambda$-calculus...

...equipped with a Garbage Collector.
let rec revapp l1 l2 =
  match l1 with
  | [] -> l2
  | x::l1' -> revapp l1' (x::l2)

With a GC, what is the heap usage of `revapp`?
A Motivating Example

let rec revapp l1 l2 =
    match l1 with
    | [] -> l2
    | x::l1' -> revapp l1' (x::l2)

With a GC, what is the heap usage of \texttt{revapp}?

It depends on the call site!

- If \( l_1 \) is used “elsewhere” \( O(\text{length } l_1) \)
A Motivating Example

```
let rec revapp l1 l2 =
  match l1 with
  | [] -> l2
  | x::l1' -> revapp l1' (x::l2)
```

With a GC, what is the heap usage of `revapp`?

It depends on the call site!

- If `l1` is used “elsewhere” $O(\text{length } l_1)$
- If `l1` is not used elsewhere $O(1)$:
  The GC can claim the front cell at each step.
Prior Work

SpaceLang by Madiot and Pottier (2022)

- **Space as a resource**, Space Credits
  \[ \ell \leftrightarrow_1 L \]
- Pointed-by assertions to track predecessors
  \[ \ell \leftrightarrow_1 b \ast \ell \leftrightarrow_1 \emptyset \Rightarrow \Diamond \text{size}(b) \ast \uparrow \ell \]
- **Free as a Ghost Update**

But...

- Target a low level language
- Bookkeeping of roots with stack cells
- Heavy reasoning rules
Prior Work

SpaceLang by Madiot and Pottier (2022)

- **Space as a resource**, Space Credits ◊₁
- Pointed-by assertions to track predecessors \( \ell \leftrightarrow₁ L \)
- Free as a Ghost Update \( \ell \mapsto₁ b \ast \ell \leftrightarrow₁ \emptyset \Rightarrow \diamond \text{size}(b) \ast \dagger \ell \)

But...

- Target a low level language
- Bookkeeping of roots with stack cells
- Heavy reasoning rules
Contributions

- A Separation Logic with Space Credits for an imperative $\lambda$-calculus
- New Stackable assertion to track roots
- Enhancement of pointed-by assertions

- Possibly-null fractions
- Signed multisets

- Examples: Lists & Stacks
- Mechanized in Coq with Iris
An imperative $\lambda$-calculus - Syntax

Values

- Unit & numbers
- Memory locations of blocks
- Closed functions (code pointers). See next paper for closures 😊
**An imperative \( \lambda \)-calculus - Syntax**

**Values**

- Unit & numbers
- Memory locations of blocks
- Closed functions (code pointers). See next paper for closures 😊

**Terms**

- Arithmetic, conditional, code pointer call, let definition
- Heap allocation, load and store
- No explicit deallocation instruction!
An imperative $\lambda$-calculus - Semantics

- Standard small-step call-by-value semantics, with a maximal live heap size
  → allocation fails if there is not enough space
- Substitution-based
- Interleave GC steps with reduction steps
The GC can deallocate unreachable locations.
The location $\ell$ is unreachable $\iff$ there is no path from a root to $\ell$. 
The GC can deallocate unreachable locations. The location $\ell$ is unreachable $\iff$ there is no path from a root to $\ell$.

Nontrivial to reason about paths. Madiot & Pottier’s solution: the location $\ell$ is unreachable when

- $\ell$ is not a root; and,
- $\ell$ is not pointed by any heap block.
What is a root?

let rec revapp l1 l2
  (* l1 is a root according to the FVR *)
  match l1 with
  | [] -> l2
  | x :: l1' -> (* l1 is not a root anymore according to the FVR *)
    revapp l1' (x :: l2)
What is a root?

The Free Variable Rule (Morrisett et al., 1995). The roots are:

- Syntactically, live bound variables
- Operationally, live locations in the stackframe
About Unreachability: the Free Variable Rule

What is a root?

The Free Variable Rule (Morrisett et al., 1995). The roots are:

- Syntactically, live bound variables
- Operationally, live locations in the stackframe

```
let rec revapp l1 l2 =
  (* l1 is a root according to the FVR *)
match l1 with
  | []  -> l2
  | x::l1' ->
    (* l1 is not a root anymore according to the FVR *)
    revapp l1' (x::l2)
```
Visible Roots vs Invisible Roots

- Roots may appear in the evaluation context
- We want to reason locally, on subterms

\[ \{ \Phi \} \ t \ \{ \Psi \} \] does not involve any evaluation context
Visible Roots vs Invisible Roots

- Roots may appear in the evaluation context
- We want to reason **locally**, on subterms
  \{\Phi\} t \{\Psi\} does not involve any evaluation context

With a subterm \( t \) of a program \( K[t] \), the location \( \ell \) is not a root:

- If \( \ell \) is not a **visible** root \( \ell \notin \text{locs}(t) \); and,
- If \( \ell \) is not an **invisible** root \( \ell \notin \text{locs}(K) \)
Visible Roots vs Invisible Roots

- Roots may appear in the evaluation context
- We want to reason locally, on subterms
  \(\{\Phi\} \ t \ \{\Psi\}\) does not involve any evaluation context

With a subterm \(t\) of a program \(K[t]\), the location \(\ell\) is not a root:

- If \(\ell\) is not a visible root \(\ell \not\in \text{locs}(t)\); and, inspect the term
- If \(\ell\) is not an invisible root \(\ell \not\in \text{locs}(K)\) \(\text{Stackable assertion}\)
Free as a Ghost Update

New ghost update parameterized by the visible roots.

\[ \Phi \Rightarrow_{\text{locs}(t)} \Phi' \quad \{\Phi'\} \quad t \quad \{\Psi\} \]

\[ \{\Phi\} \quad t \quad \{\Psi\} \]
Free as a Ghost Update

New ghost update parameterized by the visible roots.

\[ \Phi \Rightarrow_{\text{locs}(t)} \Phi' \quad \{\Phi'\} \quad t \quad \{\Psi\} \]

\[ \{\Phi\} \quad t \quad \{\Psi\} \]

Our logical FREE rule.

\[ \ell \mapsto_1 b \quad * \quad \ell \leftrightarrow_1 \emptyset \quad * \quad \neg \ell \notin V^\updownarrow \quad * \quad \text{Stackable } \ell \quad 1 \quad \Rightarrow_V \quad \Diamond \text{size}(b) \quad * \quad \dagger \ell \]

We provide a more general rule to deallocate cycles.
Pointed-by and *Stackable* assertions are created upon allocation.

$$\{\Diamond n\} \text{ alloc } n \quad \left\{ \begin{array}{l} \lambda \ell. \quad \ell \mapsto_1 ()^n \\ \text{Stackable } \ell \; 1 \end{array} \right\}$$
The Stackable Assertion

Our extended let rule for a simple context.

\[
\{ \Phi \} \ t_1 \ \{ \Psi' \} \quad \forall v. \ \{ \Psi' \ v \} \ [v/x] t_2 \ \{ \Psi \} \\
\{ \} \ \Phi \} \ \text{let} \ x = t_1 \ \text{in} \ t_2 \ \{ \Psi \}
\]
The *Stackable Assertion*

Our extended let rule for a simple context.

\[
\text{locs}(t_2) = \{\ell\}
\]

\[
\begin{array}{c}
\{\Phi\} \ t_1 \ \{\Psi'\} \quad \forall v. \ \{ \Psi' \ v \} \ [v/x]t_2 \ \{\Psi\} \\
\{ \Phi \} \ \text{let} \ x = t_1 \ \text{in} \ t_2 \ \{\Psi\} \\
\end{array}
\]
The *Stackable Assertion*

Our extended let rule for a simple context.

\[
\text{locs}(t_2) = \{\ell\} \\
\{\Phi\} t_1 \{\Psi'\} \quad \forall v. \{\text{Stackable } \ell \ p \ast \Psi' \ v\} [v/x]t_2 \ \{\Psi\} \\
\{\text{Stackable } \ell \ p \ast \Phi\} \ \text{let } x = t_1 \text{ in } t_2 \ \{\Psi\}
\]
The **Stackable Assertion**

Our extended let rule for a simple context.

\[
\text{locs}(t_2) = \{\ell\}
\]

\[
\frac{\{\Phi\} t_1 \{\Psi'\} \quad \forall v. \{\text{Stackable } \ell \ p \ast \Psi' \ v\} [v/x] t_2 \{\Psi\}}{\{\text{Stackable } \ell \ p \ast \Phi\} \text{ let } x = t_1 \text{ in } t_2 \{\Psi\}}
\]

- **Stackable */p cannot appear in */\Phi*
- Hence, **Stackable */l 1 cannot appear in */\Phi*
- Hence, *//l cannot be logically deallocated in */\{\Phi\} t_1 \{\Psi'\}*

We provide a more general rule for arbitrary contexts.
Triples with Souvenir

*Stackable* assertions seems difficult to manage in practice.

Introducing **triples with souvenir** \( \langle R \rangle \{ \Phi \} t \{ \Psi \} \)

“Give a Stackable assertion once and thats it”
Triples with Souvenir

*Stackable* assertions seems difficult to manage in practice.

Introducing *triples with souvenir* $\langle R \rangle \{ \Phi \} t \{ \Psi \}$

“*Give a Stackable assertion once and thats it*”

\[
\begin{align*}
\text{locs}(t_2) &= \{ \ell \} \\
\langle R \cup \{ \ell \} \rangle \{ \Phi \} t_1 \{ \Psi' \} &\quad \forall v. \langle R \rangle \{ \text{Stackable } \ell p * \Psi' v \} [v/x]t_2 \{ \Psi \} \\
\langle R \rangle \{ \text{Stackable } \ell p * \Phi \} \text{ let } x = t_1 \text{ in } t_2 \{ \Psi \}
\end{align*}
\]
Triples with Souvenir

*Stackable* assertions seems difficult to manage in practice.

Introducing **triples with souvenir** \( \langle R \rangle \{ \Phi \} t \{ \Psi \} \)

“Give a *Stackable* assertion once and thats it”

\[
\begin{align*}
\text{locs}(t_2) &= \{ \ell \} \\
\langle R \cup \{ \ell \} \rangle \{ \Phi \} t_1 \{ \Psi' \} &\quad \forall v. \langle R \rangle \{ \text{Stackable } \ell \ p \ast \ \Psi' \ v \} [v/x]t_2 \{ \Psi \} \\
\langle R \rangle \{ \text{Stackable } \ell \ p \ast \ \Phi \} &\quad \text{let } x = t_1 \text{ in } t_2 \{ \Psi \}
\end{align*}
\]

\[
\begin{align*}
\text{locs}(t_2) &= \{ \ell \} \quad \ell \in R \\
\langle R \rangle \{ \Phi \} t_1 \{ \Psi' \} &\quad \forall v. \langle R \rangle \{ \Psi' \ v \} [v/x]t_2 \{ \Psi \} \\
\langle R \rangle \{ \Phi \} &\quad \text{let } x = t_1 \text{ in } t_2 \{ \Psi \}
\end{align*}
\]
Proving that a Location is not a Visible Root

The goal \( \ell \not\in V \) is not trivial: one must take aliasing into account.

\[
\ell \mapsto_1 b \quad \& \quad \ell \leftarrow_1 \emptyset \quad \& \quad \lnot \ell \not\in V \quad \& \quad \text{Stackable } \ell 1 \quad \Rightarrow_V \quad \diamond \text{size}(b) \quad \& \quad \lceil \ell \rceil
\]
The goal $\ell \not\in V$ is not trivial: one must take aliasing into account.

\[ \ell \rightarrow_1 b \ast \ell \leftarrow_1 \emptyset \ast \neg \ell \not\in V \ast \text{Stackable } \ell \ 1 \Rightarrow_V \Diamond \text{size}(b) \ast \uparrow \ell \]

Thankfully

- We are developing a Separation Logic
- We can use the separating conjunction
- With $\ell \rightarrow_1 b$ and $\ell \leftarrow_1 L$ and Stackable $\ell \ 1$
- Simple cases can be automated!
Possibly Null Fractions & Signed Multisets

\[ l_3 \leftarrow q \{ +l_1; +l_1; +l_2 \} \]
Possibly Null Fractions & Signed Multisets

\[ l_3 \leftarrow_q \{ +l_1; +l_1; +l_2 \} \ast l_3 \leftarrow_0 \{-l_1\} \]
Possibly Null Fractions & Signed Multisets

\[ \ell_3 \leftarrow_q \{ +\ell_1; +\ell_1; +\ell_2 \} \ast \ell_3 \leftarrow_0 \{ -\ell_1 \} \]
\[ \equiv \ell_3 \leftarrow_{(q+0)} (\{ +\ell_1; +\ell_1; +\ell_2 \} \uplus \{ -\ell_1 \}) \]
\[ \equiv \ell_3 \leftarrow_q \{ +\ell_1; +\ell_2 \} \]
The soundness theorem is about safety.

**Theorem**

If $\langle \emptyset \rangle \{\Diamond S\} t \{\Psi\}$ holds, then, with $S$ initial memory words, $t$ is safe.
What are we Proving?

The soundness theorem is about safety.

**Theorem**

If \( \langle \emptyset \rangle \{ \Diamond S \} t \{ \Psi \} \) holds, then, with \( S \) initial memory words, \( t \) is safe.

Safety means that if \( t \) reduces to \( t' \), then either,

- \( t' \) is a value; or,
- after a full garbage collection, \( t' \) can reduce.

In other words: the maximal live heap size never exceeds \( S \).
The List Predicate

Pointed-by and *Stackable* assertions often go together.

\[ v \leftarrow_p L \triangleq v \leftarrow_p L \ast Stackable v p \]
The List Predicate

Pointed-by and *Stackable* assertions often go together.

\[ v \leftrightarrow_p L \triangleq v \leftrightarrow_p L \ast \text{Stackable} v p \]

The predicate *List*, for lists *without sharing*

\[
\text{List } L l \triangleq \text{match } L \text{ with } \\
| [] \Rightarrow l \mapsto [0] \\
| (v, p) :: L' \Rightarrow \exists l'. \\
\quad l \mapsto [1; v; l'] \ast v \leftrightarrow_p \{l\} \ast l' \leftarrow_1 \{l\} \ast \text{List } L' l'
\]
Back to the Example: Destructive Specification

\[
\langle \emptyset \rangle \left\{ \begin{array}{l}
\text{List } L_1 \ l_1 \ * \ l_1 \leftarrow_1 \emptyset \\
\text{List } L_2 \ l_2 \ * \ l_2 \leftarrow_1 \emptyset
\end{array} \right\} \text{ revapp } (l_1, l_2) \left\{ \begin{array}{l}
\lambda l. \ \text{List } (\text{rev } L_1 \ ++ \ L_2) \ l
\end{array} \right\}
\]

- Consumes its two arguments
- Generates one space credit
Back to the Example: Non-Destructive Specification

\[ \langle \{l_1\} \rangle \left\{ \begin{array}{l}
List L_1 \; l_1 \ast \diamond (3 \times |L_1|) \\
List L_2 \; l_2 \ast l_2 \leftrightarrow_1 \emptyset 
\end{array} \right\} \text{revapp} (l_1, l_2) \left\{ \begin{array}{l}
\lambda l. \; \text{List} \left( \text{rev} \left( \frac{1}{2} L_1 \right) \leftrightarrow L_2 \right) \; l \\
l \leftrightarrow_1 \emptyset 
\end{array} \right\} \]

- A souvenir of \( l_1 \): requires the framing of \textit{Stackable} \( l_1 \) \( p \) assertion
- Requires space credits
- Split fractions
A Separation Logic with Space Credits for a $\lambda$-calculus with a GC.
Conclusion & Future Work

- A Separation Logic with Space Credits for a $\lambda$-calculus with a GC.

Future Work

- See next paper for closures 😊
- Weak Pointers & Ephemerons
- Concurrency
- Link with the cost semantics of CakeML (Gómez-Londoño et al., 2020)
Thank you for your attention!
The Bind Rule

\[
\text{Stackables } M \triangleq \star \text{ Stackable } \ell \ p \\
(\ell, p) \in M
\]

**BIND**

\[
\text{dom}(M) = \text{locs}(K) \\
\{\Phi\} \ t \ \{\Psi'\} \quad \forall v. \ \{\Psi' \ v \ * \ Stackables \ M\} \ K[v] \ \{\Psi\} \\
\{\Phi \ * \ Stackables \ M\} \ K[t] \ \{\Psi\}
\]