Polishing a Rough Diamond

An Enhanced Separation Logic for Heap Space under Garbage Collection

Alexandre Moine   Arthur Charguéraud   François Pottier

ASL'22

Goal

A program logic to verify heap space bounds...

Goal

A program logic to verify heap space bounds...

...for an imperative λ-calculus...

Goal

A program logic to verify heap space bounds...

...for an imperative λ-calculus...

...equipped with a Garbage Collector.

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`?

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” $O(\text{length } l_1)$

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 $\diamond 1$
- Pointed-by assertions to track predecessors $\ell \leftarrow\!\!\dashv_1 L$
- Free as a Ghost Update $\ell \mapsto_1 b \ast \ell \leftarrow\!\!\dashv_1 \emptyset \Rrightarrow \diamond size(b) \ast \dagger \ell$

Prior Work

SpaceLang by Madiot and Pottier (2022)

- Space as a resource, Space Credits $\diamond 1$
- Pointed-by assertions to track predecessors $\ell \leftarrow\!\!\dashv_1 L$
- Free as a Ghost Update $\ell \mapsto_1 b \ast \ell \leftarrow\!\!\dashv_1 \emptyset \Rrightarrow \diamond size(b) \ast \dagger \ell$

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 λ-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 λ-calculus - Syntax

Values

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

An imperative λ-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 λ-calculus - Semantics

- Standard small-step call-by-value semantics,
  with a maximal live heap size
  $\rightarrow$ allocation fails if there is not enough space
- Substitution-based
- Interleave GC steps with reduction steps

About Unreachability: Heap Paths

The GC can deallocate unreachable locations.

The location ℓ is unreachable $\iff$ there is no path from a root to ℓ.

About Unreachability: Heap Paths

The GC can deallocate unreachable locations.
The location ℓ is unreachable $\iff$ there is no path from a root to ℓ.

Nontrivial to reason about paths.
Madiot & Pottier's solution: the location ℓ is unreachable when

- ℓ is not a root; and,
- ℓ is not pointed by any heap block

About Unreachability: the Free Variable Rule

What is a root?

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

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 ℓ is not a root:

- If ℓ is not a visible root $\ell \notin locs(t)$; and,
- If ℓ is not an invisible root $\ell \notin locs(K)$

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 ℓ is not a root:

- If ℓ is not a visible root $\ell \notin locs(t)$; and, inspect the term
- If ℓ is not an invisible root $\ell \notin locs(K)$ *Stackable* assertion

Free as a Ghost Update

New ghost update parameterized by the visible roots.

$$\frac{\Phi \Rrightarrow_{locs(t)} \Phi' \qquad \{\Phi'\}\ t\ \{\Psi\}}{\{\Phi\}\ t\ \{\Psi\}}$$

Free as a Ghost Update

New ghost update parameterized by the visible roots.

$$\frac{\Phi \Rrightarrow_{locs(t)} \Phi' \qquad \{\Phi'\}\ t\ \{\Psi\}}{\{\Phi\}\ t\ \{\Psi\}}$$

Our logical FREE rule.

$$\ell \mapsto_1 b \ * \ \ell \hookleftarrow_1 \emptyset \ * \ \ulcorner \ell \notin V \urcorner \ * \ \mathit{Stackable}\ \ell\ 1 \quad \Rrightarrow_V \quad \diamond size(b) \ * \ \dagger \ell$$

We provide a more general rule to deallocate cycles.

Allocation

Pointed-by and *Stackable* assertions are created upon allocation.

$$\left\{\diamond n\right\}\ \text{alloc}\ n\ \left\{\lambda \ell.\ \begin{array}{c} \ell \mapsto_1 ()^n \\ \ell \hookleftarrow_1 \emptyset \\ \mathit{Stackable}\ \ell\ 1 \end{array}\right\}$$

The *Stackable* Assertion

Our extended let rule for a simple context.

$$\frac{\{\Phi\}\ t_1\ \{\Psi'\} \qquad \forall v.\,\{\qquad\qquad\qquad\qquad \Psi'\ v\}\ [v/x]t_2\ \{\Psi\}}{\{\qquad\qquad\qquad\qquad \Phi\}\ \mathrm{let}\, x = t_1\ \mathrm{in}\ t_2\ \{\Psi\}}$$

The *Stackable* Assertion

Our extended let rule for a simple context.

$$\frac{locs(t_2) = \{\ell\} \qquad \{\Phi\}\ t_1\ \{\Psi'\} \qquad \forall v.\{\qquad \Psi'\ v\}\ [v/x]t_2\ \{\Psi\}}{\{\qquad \Phi\}\ \mathsf{let}\, x = t_1\, \mathsf{in}\, t_2\ \{\Psi\}}$$

The *Stackable* Assertion

Our extended let rule for a simple context.

$$\frac{locs(t_2) = \{\ell\} \qquad \{\Phi\}\ t_1\ \{\Psi'\} \qquad \forall v.\, \{\mathit{Stackable}\ \ell\ p * \Psi'\ v\}\ [v/x]t_2\ \{\Psi\}}{\{\mathit{Stackable}\ \ell\ p * \Phi\}\ \mathsf{let}\, x = t_1\, \mathsf{in}\, t_2\ \{\Psi\}}$$

The *Stackable* Assertion

Our extended let rule for a simple context.

$$\frac{\{\Phi\}\ t_1\ \{\Psi'\} \qquad \forall v.\ \{\mathit{Stackable}\ \ell\ p * \Psi'\ v\}\ [v/x]t_2\ \{\Psi\} \qquad \mathit{locs}(t_2) = \{\ell\}}{\{\mathit{Stackable}\ \ell\ p * \Phi\}\ \mathrm{let}\, x = t_1\, \mathrm{in}\, t_2\ \{\Psi\}}$$

- $\mathit{Stackable}\ \ell\ p$ cannot appear in Φ
- Hence, $\mathit{Stackable}\ \ell\ 1$ cannot appear in Φ
- 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"*

$$\frac{locs(t_2) = \{\ell\} \qquad \langle R \cup \{\ell\}\rangle\,\{\Phi\}\,t_1\,\{\Psi'\} \qquad \forall v.\,\langle R\rangle\,\{\mathit{Stackable}\;\ell\;p * \Psi'\;v\}\,[v/x]t_2\,\{\Psi\}}{\langle R\rangle\,\{\mathit{Stackable}\;\ell\;p * \Phi\}\,\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2\,\{\Psi\}}$$

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"*

$$\frac{locs(t_2) = \{\ell\} \qquad \langle R \cup \{\ell\} \rangle \; \{\Phi\} \; t_1 \; \{\Psi'\} \qquad \forall v. \; \langle R \rangle \; \{Stackable \; \ell \; p * \Psi' \; v\} \; [v/x]t_2 \; \{\Psi\}}{\langle R \rangle \; \{Stackable \; \ell \; p * \Phi\} \; \mathrm{let} \, x = t_1 \; \mathrm{in} \; t_2 \; \{\Psi\}}$$

$$\frac{locs(t_2) = \{\ell\} \qquad \ell \in R \qquad \langle R \rangle \; \{\Phi\} \; t_1 \; \{\Psi'\} \qquad \forall v. \; \langle R \rangle \; \{\Psi' \; v\} \; [v/x]t_2 \; \{\Psi\}}{\langle R \rangle \; \{\Phi\} \; \mathrm{let} \, x = t_1 \; \mathrm{in} \; t_2 \; \{\Psi\}}$$

Proving that a Location is not a Visible Root

The goal $\ell \notin V$ is not trivial: one must take aliasing into account.

$$\ell \mapsto_1 b \ * \ \ell \leftarrow\!\!\dashv_1 \emptyset \ * \ \ulcorner \ell \notin V \urcorner \ * \ \mathit{Stackable}\ \ell\ 1 \quad \Rrightarrow_V \quad \diamond \mathit{size}(b) \ * \ \dagger \ell$$

Proving that a Location is not a Visible Root

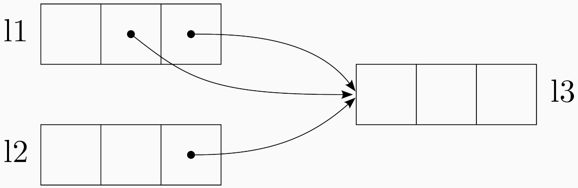
The goal $\ell \notin V$ is not trivial: one must take aliasing into account.

$$\ell \mapsto_1 b \ * \ \ell \hookleftarrow_1 \emptyset \ * \ \ulcorner \ell \notin V \urcorner \ * \ \mathit{Stackable}\ \ell\ 1 \quad \Rrightarrow_V \quad \diamond \mathit{size}(b) \ * \ \dagger \ell$$

Thankfully

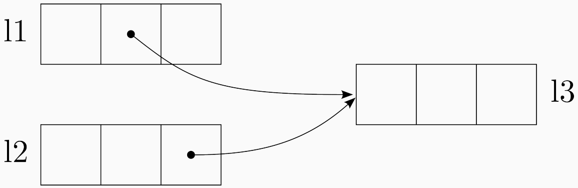
- We are developing a Separation Logic
- We can use the separating conjunction
- With $\ell \mapsto_1 b$ and $\ell \hookleftarrow_1 L$ and $\mathit{Stackable}\ \ell\ 1$
- Simple cases can be automated!

Possibly Null Fractions & Signed Multisets



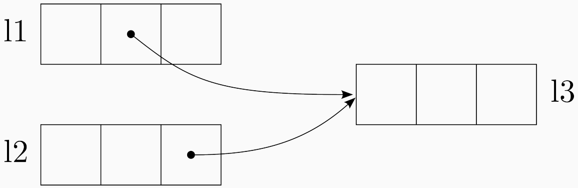
$$\ell_3 \hookleftarrow_q \{ +\ell_1; +\ell_1; +\ell_2 \}$$

Possibly Null Fractions & Signed Multisets



$$\ell_3 \hookleftarrow_q \{\, +\ell_1;\ +\ell_1;\ +\ell_2 \,\} \ * \ \ell_3 \hookleftarrow_0 \{-\ell_1\}$$

Possibly Null Fractions & Signed Multisets



$$\ell_3 \hookleftarrow_q \{\, +\ell_1;\ +\ell_1;\ +\ell_2 \,\} \ * \ \ell_3 \hookleftarrow_0 \{-\ell_1\}$$
$$\equiv \ell_3 \hookleftarrow_{(q+0)} \big(\{\, +\ell_1;\ +\ell_1;\ +\ell_2 \,\} \uplus \{-\ell_1\}\big)$$
$$\equiv \ell_3 \hookleftarrow_q \{\, +\ell_1;\ +\ell_2 \,\}$$

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.*

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.

$$v \hookleftarrow_p L \quad \triangleq \quad v \leftarrow\!\dashv_p L \; * \; \mathit{Stackable}\; v\; p$$

The List Predicate

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

$$v \hookleftarrow_p L \quad \triangleq \quad v \leftarrow\!\!\dashv_p L \ * \ \mathit{Stackable}\ v\ p$$

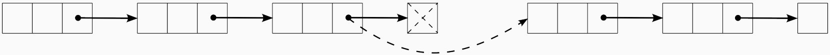
The predicate *List*, for lists without sharing

$$\begin{aligned}
&\mathit{List}\ L\ l \ \triangleq\ \text{match } L \text{ with} \\
&\quad \mid [\,] \ \Rightarrow\ l \mapsto [0] \\
&\quad \mid (v, p) :: L' \ \Rightarrow\ \exists l'. \\
&\qquad l \mapsto [1; v; l'] \ * \ v \hookleftarrow_p \{l\} \ * \ l' \hookleftarrow_1 \{l\} \ * \ \mathit{List}\ L'\ l'
\end{aligned}$$

Back to the Example: Destructive Specification

$$\langle \emptyset \rangle \left\{ \begin{array}{l} \mathit{List}\ L_1\ l_1 \ * \ l_1 \hookleftarrow_1 \emptyset \\ \mathit{List}\ L_2\ l_2 \ * \ l_2 \hookleftarrow_1 \emptyset \end{array} \right\} \text{revapp}\,(l_1, l_2)\ \left\{ \lambda l. \begin{array}{c} \mathit{List}\ (\mathit{rev}\ L_1 +\!\!+\ L_2)\ l \\ l \hookleftarrow_1 \emptyset \ * \ \diamond 1 \end{array} \right\}$$

- Consumes its two arguments
- Generates one space credit



Back to the Example: Non-Destructive Specification

$$\langle\,\{l_1\}\,\rangle \left\{\begin{matrix} \mathit{List}\ L_1\ l_1 * \diamond(3 \times |L_1|) \\ \mathit{List}\ L_2\ l_2 * l_2 \hookleftarrow_1 \emptyset \end{matrix}\right\} \ \mathsf{revapp}\,(l_1, l_2) \ \left\{ \begin{matrix} \mathit{List}\ (\frac{1}{2}L_1)\ l_1 \\ \lambda l.\ \mathit{List}\ (\mathit{rev}\,(\frac{1}{2}L_1) +\!\!+ L_2)\ l \\ l \hookleftarrow_1 \emptyset \end{matrix} \right\}$$

- A souvenir of l_1: requires the framing of *Stackable* l_1 p assertion
- Requires space credits
- Split fractions

Conclusion & Future Work

- A Separation Logic with Space Credits for a λ-calculus with a GC.

Conclusion & Future Work

- A Separation Logic with Space Credits for a λ-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

$$\mathit{Stackables}\ M \triangleq \mathop{\ast}_{(\ell, p) \in M} \mathit{Stackable}\ \ell\ p$$

Bind

$$\frac{dom(M) = locs(K) \qquad \{\Phi\}\ t\ \{\Psi'\} \qquad \forall v.\, \{\Psi'\, v \ast \mathit{Stackables}\ M\}\ K[v]\ \{\Psi\}}{\{\Phi \ast \mathit{Stackables}\ M\}\ K[t]\ \{\Psi\}}$$

References i

Alejandro Gómez-Londoño, Johannes Åman Pohjola, Hira Taqdees Syeda, Magnus O. Myreen, and Yong Kiam Tan. Do you have space for dessert? A verified space cost semantics for CakeML programs. Proceedings of the ACM on Programming Languages, 4(OOPSLA): 204:1–204:29, 2020. URL https://doi.org/10.1145/3428272.

Jean-Marie Madiot and François Pottier. A separation logic for heap space under garbage collection. Proceedings of the ACM on Programming Languages, (POPL), January 2022. URL http://cambium.inria.fr/~fpottier/publis/madiot-pottier-diamonds-2022.pdf.

Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In Functional Programming Languages and Computer Architecture, June 1995. URL https://www.cs.cmu.edu/~rwh/papers/gc/fpca95.pdf.