Will It Fit?

Alexandre Moine
Arthur Charguéraud and François Pottier

Utrecht, April 25, 2024

Space Credits

Key Idea

Space as a resource.

Following Hofmann [1999], let $\diamondsuit 1$ represent one space credit.

Space Credits

Key Idea

Space as a resource.

Following Hofmann [1999], let $\Diamond 1$ represent one space credit.

- A space credit represents a permission to allocate one memory word.
- Space credits are non-negative numbers.
- Space credits can be split and joined: $\Diamond(n_1 + n_2) \equiv \Diamond n_1 * \Diamond n_2$

Warm Up: No GC

Without garbage collection, one can use the following two reasoning rules:

- "alloc" consumes space:

  $\{\ \Diamond n\ \}\ \text{alloc}\ n\ \{\ \lambda\ell.\ \ell \mapsto [0, \ldots, 0]\ \}$

- "free" produces space:

  $\{\ \ell \mapsto [v_1, \ldots, v_n]\ \}\ \text{free}\,\ell\ \{\ \lambda().\ \Diamond n\ \}$

Warm Up: No GC

Without garbage collection, one can use the following two reasoning rules:

- "alloc" consumes space:

  $\{\ \Diamond n\ \}\ \text{alloc}\ n\ \{\ \lambda\ell.\ \ell \mapsto [0, \ldots, 0]\ \}$

- "free" produces space:

  $\{\ \ell \mapsto [v_1, \ldots, v_n]\ \}\ \text{free}\,\ell\ \{\ \lambda().\ \Diamond n\ \}$

and one can easily prove the following:

Adequacy theorem

If $\{\ \Diamond S\ \}\ t\ \{\ \lambda\_.\ \text{True}\ \}$ holds, then under heap limit S, the program t will not crash.

The GC Enters the Scene

- OCaml, Haskell, Scala, ... have garbage collection.
- There is no "free" instruction.
- At arbitrary times, the garbage collector can reclaim unreachable blocks.

The GC Enters the Scene

- OCaml, Haskell, Scala, ... have garbage collection.
- There is no "free" instruction.
- At arbitrary times, the garbage collector can reclaim unreachable blocks.
- This makes programming easier...

The GC Enters the Scene

- OCaml, Haskell, Scala, ... have garbage collection.
- There is no “free” instruction.
- At arbitrary times, the garbage collector can reclaim unreachable blocks.
- This makes programming easier...
- ...but makes reasoning more difficult. Where can space credits be recovered?

The GC Enters the Scene

- OCaml, Haskell, Scala, ... have garbage collection.
- There is no “free” instruction.
- At arbitrary times, the garbage collector can reclaim unreachable blocks.
- This makes programming easier...
- ...but makes reasoning more difficult. Where can space credits be recovered?

In this work:

- we answer this question,
- while supporting fine-grained shared-memory concurrency.

Free is a Ghost Update

Following Madiot and Pottier [2022], we make deallocation a logical operation:

$$\text{“}\ell\text{ is unreachable”} \;\ast\; \ell \mapsto [v_1, \ldots, v_n] \quad \Rrightarrow \quad \Diamond n$$

If a block is unreachable then by giving up its ownership one obtains space credits.

$\Diamond n$ means that n words are free or can be freed by the GC, once it runs.

Free is a Ghost Update

Following Madiot and Pottier [2022], we make deallocation a logical operation:

$$\text{“}\ell \text{ is unreachable”} \ast \ell \mapsto [v_1, \ldots, v_n] \quad \Rrightarrow \quad \Diamond n$$

If a block is unreachable then by giving up its ownership one obtains space credits.

$\Diamond n$ means that n words are free or can be freed by the GC, once it runs.

How does one prove that a location is unreachable?

Reachability

A location ℓ is reachable if:

1. from some root,
   - a location that is currently stored in a live local variable x of some thread π
2. there is a path, through the heap, to ℓ.

Reachability

A location ℓ is reachable if:

1. from some root,
   - a location that is currently stored in a live local variable x of some thread π
2. there is a path, through the heap, to ℓ.

Therefore, a location ℓ is unreachable if:

1. ℓ is not a root of any thread; and
2. ℓ has no (reachable) predecessor in the heap

Reachability

A location ℓ is reachable if:

1. from some root,
   - a location that is currently stored in a live local variable x of some thread π
2. there is a path, through the heap, to ℓ.

Therefore, a location ℓ is unreachable if:

1. ℓ is not a root of any thread; and $\rightsquigarrow$ the pointed-by-thread assertion
2. ℓ has no (reachable) predecessor in the heap $\rightsquigarrow$ the pointed-by-heap assertion

Pointed-by-Heap Assertions

From Kassios and Kritikos [2013], Madiot and Pottier [2022], Moine et al. [2023].

- $\ell \hookleftarrow_1 A$ asserts that A is an over-approximation of the (reachable) predecessors of ℓ.

Pointed-by-Heap Assertions

From Kassios and Kritikos [2013], Madiot and Pottier [2022], Moine et al. [2023].

- $\ell \leftarrow\!\!\dashv_1 A$ asserts that A is an over-approximation of the (reachable) predecessors of ℓ.
- $\ell \leftarrow\!\!\dashv_1 \emptyset$ means that ℓ has no (reachable) predecessors.
- Pointed-by-heap assertions can be split and joined:

$$\ell \leftarrow\!\!\dashv_1 \{+\ell_1; +\ell_2\} \quad \equiv \quad \ell \leftarrow\!\!\dashv_{\frac{1}{2}} \{+\ell_1\} \ast \ell \leftarrow\!\!\dashv_{\frac{1}{2}} \{+\ell_2\}$$

- For greater comfort, we use signed multisets (not shown here).

What Are the Roots Considered by Real-Life GCs?

What Are the Roots Considered by Real-Life GCs?

The Free Variable Rule (FVR) [Felleisen and Hieb, 1992]

In a substitution-based semantics,
the roots are the locations that occur in the terms that remain to be evaluated.

What Are the Roots Considered by Real-Life GCs?

The Free Variable Rule (FVR) [Felleisen and Hieb, 1992]

In a substitution-based semantics,
the roots are the locations that occur in the terms that remain to be evaluated.

Term	Reachable Heap
let $a = (\mathit{ref}\ 4)$ in let $b = (\mathit{ref}\ 2)$ in $!a + !b$	$\emptyset$

What Are the Roots Considered by Real-Life GCs?

The Free Variable Rule (FVR) [Felleisen and Hieb, 1992]

In a substitution-based semantics,
the roots are the locations that occur in the terms that remain to be evaluated.

	Term	Reachable Heap
	let $a = (\mathit{ref}\ 4)$ in let $b = (\mathit{ref}\ 2)$ in $!a + !b$	$\emptyset$
$\longrightarrow$	let $a = \ell_a$ in let $b = (\mathit{ref}\ 2)$ in $!a + !b$	$\{\ell_a \mapsto 4\}$

What Are the Roots Considered by Real-Life GCs?

The Free Variable Rule (FVR) [Felleisen and Hieb, 1992]

In a substitution-based semantics,
the roots are the locations that occur in the terms that remain to be evaluated.

	Term	Reachable Heap
	let $a = (\mathit{ref}\ 4)$ in let $b = (\mathit{ref}\ 2)$ in $!a + !b$	$\emptyset$
$\longrightarrow$	let $a = \ell_a$ in let $b = (\mathit{ref}\ 2)$ in $!a + !b$	$\{\ell_a \mapsto 4\}$
$\longrightarrow$	let $b = (\mathit{ref}\ 2)$ in $!\ell_a + !b$	$\{\ell_a \mapsto 4\}$

What Are the Roots Considered by Real-Life GCs?

The Free Variable Rule (FVR) [Felleisen and Hieb, 1992]

In a substitution-based semantics,
the roots are the locations that occur in the terms that remain to be evaluated.

	Term	Reachable Heap
	let $a = (\mathit{ref}\ 4)$ in let $b = (\mathit{ref}\ 2)$ in $!a + !b$	$\emptyset$
$\longrightarrow$	let $a = \ell_a$ in let $b = (\mathit{ref}\ 2)$ in $!a + !b$	$\{\ell_a \mapsto 4\}$
$\longrightarrow$	let $b = (\mathit{ref}\ 2)$ in $!\ell_a + !b$	$\{\ell_a \mapsto 4\}$
$\longrightarrow$	let $b = \ell_b$ in $!\ell_a + !b$	$\{\ell_a \mapsto 4, \ell_b \mapsto 2\}$

What Are the Roots Considered by Real-Life GCs?

The Free Variable Rule (FVR) [Felleisen and Hieb, 1992]

In a substitution-based semantics,
the roots are the locations that occur in the terms that remain to be evaluated.

	Term	Reachable Heap
	$\text{let } a = (\mathit{ref}\ 4) \text{ in let } b = (\mathit{ref}\ 2) \text{ in } !a + !b$	$\emptyset$
$\longrightarrow$	$\text{let } a = \ell_a \text{ in let } b = (\mathit{ref}\ 2) \text{ in } !a + !b$	$\{\ell_a \mapsto 4\}$
$\longrightarrow$	$\text{let } b = (\mathit{ref}\ 2) \text{ in } !\ell_a + !b$	$\{\ell_a \mapsto 4\}$
$\longrightarrow$	$\text{let } b = \ell_b \text{ in } !\ell_a + !b$	$\{\ell_a \mapsto 4, \ell_b \mapsto 2\}$
$\longrightarrow$	$!\ell_a + !\ell_b$	$\{\ell_a \mapsto 4, \ell_b \mapsto 2\}$

What Are the Roots Considered by Real-Life GCs?

The Free Variable Rule (FVR) [Felleisen and Hieb, 1992]

In a substitution-based semantics,
the roots are the locations that occur in the terms that remain to be evaluated.

	Term	Reachable Heap
	let $a = (\mathit{ref}\ 4)$ in let $b = (\mathit{ref}\ 2)$ in $!a + !b$	$\emptyset$
$\longrightarrow$	let $a = \ell_a$ in let $b = (\mathit{ref}\ 2)$ in $!a + !b$	$\{\ell_a \mapsto 4\}$
$\longrightarrow$	let $b = (\mathit{ref}\ 2)$ in $!\ell_a + !b$	$\{\ell_a \mapsto 4\}$
$\longrightarrow$	let $b = \ell_b$ in $!\ell_a + !b$	$\{\ell_a \mapsto 4, \ell_b \mapsto 2\}$
$\longrightarrow$	$!\ell_a + !\ell_b$	$\{\ell_a \mapsto 4, \ell_b \mapsto 2\}$
$\longrightarrow$	$4 + !\ell_b$	$\{\cancel{\ell_a \mapsto 4}, \ell_b \mapsto 2\}$

What Are the Roots Considered by Real-Life GCs?

The Free Variable Rule (FVR) [Felleisen and Hieb, 1992]

In a substitution-based semantics,
the roots are the locations that occur in the terms that remain to be evaluated.

	Term	Reachable Heap
	let $a = (\mathit{ref}\ 4)$ in let $b = (\mathit{ref}\ 2)$ in $!a + !b$	$\emptyset$
$\longrightarrow$	let $a = \ell_a$ in let $b = (\mathit{ref}\ 2)$ in $!a + !b$	$\{\ell_a \mapsto 4\}$
$\longrightarrow$	let $b = (\mathit{ref}\ 2)$ in $!\ell_a + !b$	$\{\ell_a \mapsto 4\}$
$\longrightarrow$	let $b = \ell_b$ in $!\ell_a + !b$	$\{\ell_a \mapsto 4, \ell_b \mapsto 2\}$
$\longrightarrow$	$!\ell_a + !\ell_b$	$\{\ell_a \mapsto 4, \ell_b \mapsto 2\}$
$\longrightarrow$	$4 + !\ell_b$	$\{\cancel{\ell_a \mapsto 4}, \ell_b \mapsto 2\}$
$\longrightarrow^*$	6	$\{\cancel{\ell_a \mapsto 4}, \cancel{\ell_b \mapsto 2}\}$

Pointed-By-Thread Assertions

We use a ghost thread identifier π to identify a thread. $\{\,\Phi\,\}\,\pi\colon t\,\{\,\Psi\,\}$

Pointed-By-Thread Assertions

We use a ghost thread identifier π to identify a thread. $\{\, \Phi \,\}\, \pi\colon t \,\{\, \Psi \,\}$

We introduce pointed-by-thread assertions:

- $\ell \Lleftarrow_1 \Pi$ asserts that Π is an over-approximation of the threads in which ℓ is a root.
- $\ell \Lleftarrow_1 \emptyset$ asserts that ℓ is not a root in any thread.
- Pointed-by-thread assertions can be split and joined:

$$\ell \Lleftarrow_{(p_1+p_2)} (\Pi_1 \cup \Pi_2) \quad \equiv \quad \ell \Lleftarrow_{p_1} \Pi_1 \ * \ \ell \Lleftarrow_{p_2} \Pi_2$$

Some Reasoning Rules: Alloc, Load and Fork

Allocation produces points-to, pointed-by-heap, and pointed-by-thread assertions:

$$\{ \Diamond n \}\ \pi\colon \mathsf{alloc}\ n\ \{ \lambda \ell.\ \ell \mapsto [0, \ldots, 0] \ * \ \ell \mathrel{\leftarrow\!\!\dashv}_1 \emptyset \ * \ \ell \mathrel{\Leftarrow\!\!\dashv}_1 \{\pi\} \}$$

Some Reasoning Rules: Alloc, Load and Fork

Allocation produces points-to, pointed-by-heap, and pointed-by-thread assertions:

$$\{\, \Diamond n \,\}\ \pi \colon \mathsf{alloc}\ n\ \{\, \lambda \ell.\ \ell \mapsto [0, \ldots, 0] \ * \ \ell \hookleftarrow_1 \emptyset \ * \ \ell \Lleftarrow_1 \{\pi\} \,\}$$

Reading a location ℓ' from the heap makes it a root in this thread:

$$\frac{0 \leq i < |\vec{v}| \qquad \vec{v}(i) = \ell'}{\{\, \ell \mapsto \vec{v} \ * \ \ell' \Lleftarrow_p \emptyset \,\}\ \pi \colon \ell[i]\ \{\, \lambda w.\ \ulcorner w = \ell' \urcorner \ * \ \ell \mapsto \vec{v} \ * \ \ell' \Lleftarrow_p \{\pi\} \,\}}$$

Some Reasoning Rules: Alloc, Load and Fork

Allocation produces points-to, pointed-by-heap, and pointed-by-thread assertions:

$$\{\, \Diamond n \,\}\; \pi\colon \mathsf{alloc}\; n \;\{\, \lambda\ell.\; \ell \mapsto [0,\ldots,0] \;\ast\; \ell \leftarrow\!\!\dashv_1 \emptyset \;\ast\; \ell \Leftarrow\!\!\dashv_1 \{\pi\} \,\}$$

Reading a location ℓ' from the heap makes it a root in this thread:

$$\frac{0 \le i < |\vec{v}| \qquad \vec{v}(i) = \ell'}{\{\, \ell \mapsto \vec{v} \;\ast\; \ell' \Leftarrow\!\!\dashv_p \emptyset \,\}\; \pi\colon \ell[i] \;\{\, \lambda w.\; \ulcorner w = \ell' \urcorner \;\ast\; \ell \mapsto \vec{v} \;\ast\; \ell' \Leftarrow\!\!\dashv_p \{\pi\} \,\}}$$

Spawning a new thread transfers some roots to it. Here is a (simplified) rule:

$$\frac{\forall \pi'.\; \{\qquad\qquad\qquad \Phi \,\}\; \pi'\colon t \;\{\, \lambda().\; \mathsf{True} \,\}}{\{\qquad\qquad\qquad \Phi \,\}\; \pi\colon \mathsf{fork}\; t \;\{\, \lambda().\; \mathsf{True} \,\}}$$

Some Reasoning Rules: Alloc, Load and Fork

Allocation produces points-to, pointed-by-heap, and pointed-by-thread assertions:

$$\{\, \Diamond n \,\}\ \pi\colon \mathsf{alloc}\ n\ \{\, \lambda \ell.\ \ell \mapsto [0, \ldots, 0] \ \ast\ \ell \mapsfrom_1 \emptyset \ \ast\ \ell \Mapsfrom_1 \{\pi\} \,\}$$

Reading a location ℓ' from the heap makes it a root in this thread:

$$\frac{0 \le i < |\vec{v}| \qquad \vec{v}(i) = \ell'}{\{\, \ell \mapsto \vec{v} \ \ast\ \ell' \Mapsfrom_p \emptyset \,\}\ \pi\colon \ell[i]\ \{\, \lambda w.\ \ulcorner w = \ell' \urcorner \ \ast\ \ell \mapsto \vec{v} \ \ast\ \ell' \Mapsfrom_p \{\pi\} \,\}}$$

Spawning a new thread transfers some roots to it. Here is a (simplified) rule:

$$\frac{\mathit{locs}(t) = \{\ell\} \qquad \forall \pi'.\ \{\, \Phi \,\}\ \pi'\colon t\ \{\, \lambda ().\ \mathsf{True} \,\}}{\{\, \Phi \,\}\ \pi\colon \mathsf{fork}\ t\ \{\, \lambda ().\ \mathsf{True} \,\}}$$

Some Reasoning Rules: Alloc, Load and Fork

Allocation produces points-to, pointed-by-heap, and pointed-by-thread assertions:

$$\{\, \Diamond n \,\}\ \pi \colon \mathsf{alloc}\ n \,\{\, \lambda \ell.\ \ell \mapsto [0, \ldots, 0] \;\ast\; \ell \leftarrow\!\!\dashv_1 \emptyset \;\ast\; \ell \Leftarrow\!\!\dashv_1 \{\pi\} \,\}$$

Reading a location ℓ' from the heap makes it a root in this thread:

$$\frac{0 \leq i < |\vec{v}| \qquad \vec{v}(i) = \ell'}{\{\, \ell \mapsto \vec{v} \;\ast\; \ell' \Leftarrow\!\!\dashv_p \emptyset \,\}\ \pi \colon \ell[i] \,\{\, \lambda w.\ \ulcorner w = \ell' \urcorner \;\ast\; \ell \mapsto \vec{v} \;\ast\; \ell' \Leftarrow\!\!\dashv_p \{\pi\} \,\}}$$

Spawning a new thread transfers some roots to it. Here is a (simplified) rule:

$$\frac{locs(t) = \{\ell\} \qquad \forall \pi'.\ \{\, \ell \Leftarrow\!\!\dashv_p \{\pi'\} \;\ast\; \Phi \,\}\ \pi' \colon t \,\{\, \lambda ().\ \mathsf{True} \,\}}{\{\, \ell \Leftarrow\!\!\dashv_p \{\pi\} \;\ast\; \Phi \,\}\ \pi \colon \mathsf{fork}\ t \,\{\, \lambda ().\ \mathsf{True} \,\}}$$

Some Reasoning Rules: Trimming and Logical Deallocation

Once ℓ is no longer a root in thread π, pointed-by-thread assertions can be trimmed:

$$\dfrac{\ell \notin \mathit{locs}(t) \qquad \{\, \ell \Lleftarrow_p \emptyset \;\; * \;\; \Phi \,\}\, \pi : t \,\{\, \Psi \,\}}{\{\, \ell \Lleftarrow_p \{\pi\} \;\; * \;\; \Phi \,\}\, \pi : t \,\{\, \Psi \,\}}\ \textsc{Trim}$$

Some Reasoning Rules: Trimming and Logical Deallocation

Once ℓ is no longer a root in thread π, pointed-by-thread assertions can be trimmed :

$$\dfrac{\ell \notin locs(t) \qquad \{\, \ell \Lleftarrow_p \emptyset \;*\; \Phi \,\}\, \pi\colon t \,\{\, \Psi \,\}}{\{\, \ell \Lleftarrow_p \{\pi\} \;*\; \Phi \,\}\, \pi\colon t \,\{\, \Psi \,\}}\ \textsc{Trim}$$

The logical deallocation rule can now be shown:

$$\ell \mapsto [v_1, \ldots, v_n] \;*\; \underbrace{\boxed{\ell \leftarrow_1 \emptyset}}_{\ell \text{ has no predecessors in the heap}} \;*\; \overbrace{\boxed{\ell \Lleftarrow_1 \emptyset}}^{\ell \text{ is not a root}} \;\Rrightarrow\; \ell \mapsto [v_1, \ldots, v_n] \;*\; \Diamond n \;*\; \underbrace{\dagger \ell}_{\text{deallocation witness}}$$

The points-to assertion is not consumed.

The Adequacy Theorem

A configuration is stuck if

- even after garbage collection has taken place,
- some thread has not terminated and cannot make progress.

The Adequacy Theorem

A configuration is stuck if

- even after garbage collection has taken place,
- some thread has not terminated and cannot make progress.

In particular, a memory allocation request that cannot be satisfied is stuck.

The Adequacy Theorem

A configuration is stuck if

- even after garbage collection has taken place,
- some thread has not terminated and cannot make progress.

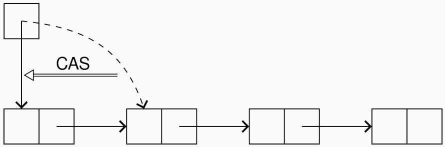
In particular, a memory allocation request that cannot be satisfied is stuck.

Adequacy Theorem

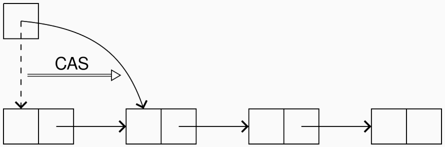
If $\{\,\Diamond S\,\}\,\pi\colon t\,\{\,\lambda\_.\ \mathrm{True}\,\}$ holds, then under heap limit S, t cannot become stuck.

The Case of Lock-Free Data Structures: Treiber's Stack

A linearizable lock-free stack, implemented as a reference on an immutable list.



push



pop

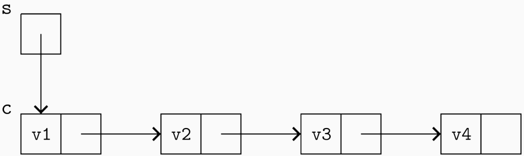
The Case of Lock-Free Data Structures: Treiber's Stack

A linearizable lock-free stack, implemented as a reference on an immutable list.

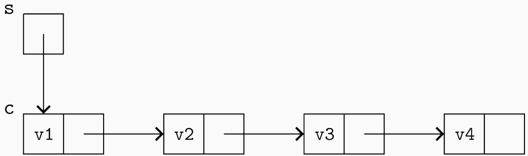
push pop

```
let rec pop s =
  let xs = !s in
  match xs with
  | nil -> assert false
  | y::ys -> if CAS s xs ys then y else pop s
```

A Problematic Interleaving for pop



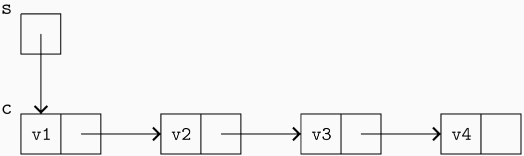
A Problematic Interleaving for pop




```
pop s
```

```
pop s;
pop s;
pop s;
...
```

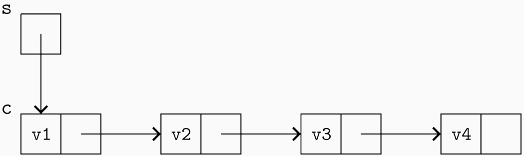
A Problematic Interleaving for pop




```
let xs = !s in
match xs with
...
```

```
pop s;
pop s;
pop s;
...
```

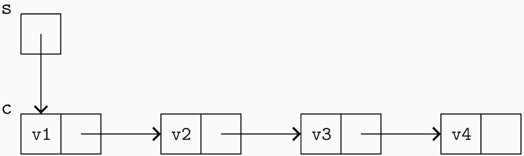
A Problematic Interleaving for pop




```
let xs = c in
match xs with
...
```

```
pop s;
pop s;
pop s;
...
```

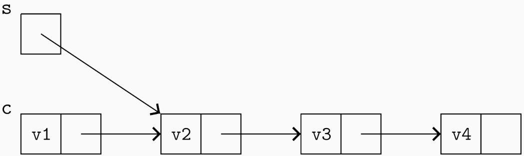
A Problematic Interleaving for pop




```
let xs = c in
match xs with
...
```


```
pop s;
pop s;
pop s;
...
```

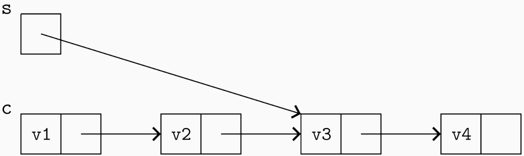
A Problematic Interleaving for pop




```
let xs = c in
match xs with
...
```


```
pop s;
pop s;
pop s;
...
```

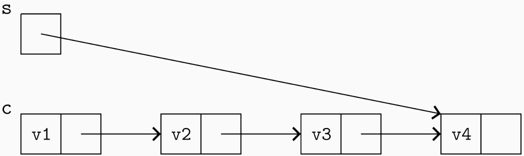
A Problematic Interleaving for pop




```
let xs = c in
match xs with
...
```


```
pop s;
pop s;
pop s;
...
```

A Problematic Interleaving for pop




```
let xs = c in
match xs with
...
```


```
pop s;
pop s;
pop s;
...
```

- The sleeping thread holds the root c, therefore keeps the whole list reachable.
- We cannot prove that pop produces space credits!
- **Key idea**: c is a temporary root. Do not allow the GC to run while it is held.

LambdaFit

Our calculus, LambdaFit, has several novel features:

- protected sections, where the GC is disabled.
  - the GC can run only if all threads are outside protected sections
  - inside protected sections, allocation and divergence are forbidden
- polling points, which cannot be passed if any thread is waiting for the GC to run;
- there is a heap size limit S,
- and a memory allocation request blocks until it can be satisfied.

Temporary Roots

Inside a critical section, a root can be temporary – unobservable by the GC.

Temporary Roots

Inside a critical section, a root can be temporary – unobservable by the GC.

Our reasoning rules take advantage of protected sections:

Key Idea

In the logical deallocation rule, ignore temporary roots.

Treiber's Stack, Fixed: pop

```
let rec pop s =
  enter ();
  let xs = !s in
  match xs with
  | nil -> assert false
  | y::ys ->
    if CAS s xs ys then (exit (); y) else (exit (); pop s)
```

- The variable `xs` is a root inside a protected section only.
- The GC can never observe this root.
- The problematic scenario disappears.

Inside and Outside Assertions

- Two new assertions: $\mathsf{outside}\,\pi$ and $\mathsf{inside}\,\pi\ T$.
- The parameter T is the set of temporary roots.

$$\{\,\mathsf{outside}\,\pi\,\}\ \pi : \mathsf{enter}\,()\ \{\,\lambda().\ \mathsf{inside}\,\pi\,\emptyset\,\} \qquad \{\,\mathsf{inside}\,\pi\,\emptyset\,\}\ \pi : \mathsf{exit}\,()\ \{\,\lambda().\ \mathsf{outside}\,\pi\,\}$$

Inside and Outside Assertions

- Two new assertions: $\mathsf{outside}\,\pi$ and $\mathsf{inside}\,\pi\ T$.
- The parameter T is the set of temporary roots.

$$\{\,\mathsf{outside}\,\pi\,\}\ \pi : \mathsf{enter}\,()\ \{\,\lambda().\ \mathsf{inside}\,\pi\,\emptyset\,\} \qquad \{\,\mathsf{inside}\,\pi\,\emptyset\,\}\ \pi : \mathsf{exit}\,()\ \{\,\lambda().\ \mathsf{outside}\,\pi\,\}$$

The assertion $\mathsf{inside}\,\pi\ T$ forms an escape hatch to the pointed-by-thread discipline.

$$\begin{array}{lcll}
\mathsf{inside}\,\pi\ T\ *\ \ell \Lleftarrow_p \{\pi\} & \Rrightarrow & \mathsf{inside}\,\pi\,(T \cup \{\ell\})\ *\ \ell \Lleftarrow_p \emptyset & \textsc{AddTemporary} \\
\mathsf{inside}\,\pi\ T\ *\ \ell \Lleftarrow_p \emptyset & \Rrightarrow & \mathsf{inside}\,\pi\,(T \setminus \{\ell\})\ *\ \ell \Lleftarrow_p \{\pi\} & \textsc{RemTemporary}
\end{array}$$

The Big Picture

$$\frac{\{\,\mathsf{inside}\,\pi\,(\,T \cap \mathit{locs}(t))\;\;*\;\;\Phi\,\}\,\pi\!: t\,\{\,\Psi\,\}}{\{\,\mathsf{inside}\,\pi\;T\;\;*\;\;\Phi\,\}\,\pi\!: t\,\{\,\Psi\,\}}\ \textsc{TrimInside}$$

The Big Picture

$$\frac{\{\,\mathsf{inside}\,\pi\,(T \cap \mathit{locs}(t)) \ \ast\ \Phi\,\}\ \pi : t\ \{\,\Psi\,\}}{\{\,\mathsf{inside}\,\pi\ T \ \ast\ \Phi\,\}\ \pi : t\ \{\,\Psi\,\}}\ \textsc{TrimInside}$$

- All of the previous rules remain sound, including logical deallocation.
- Therefore logical deallocation is not impeded by temporary roots.

Treiber's Stack, Fixed: push

```
let rec push s v =
  let xs = new_cell () in
  set_data xs v;
  enter ();
  let ys = !s in
  set_tail xs ys;
  if compare_and_swap s ys xs
  then exit ()
  else (exit (); push s v)
```

Treiber's Stack, Fixed: push

```
1 let rec push s v =
2   let xs = new_cell () in
3   set_data xs v;
4   enter ();
5   let ys = !s in
6   set_tail xs ys;
7   if compare_and_swap s ys xs
8   then exit ()
9   else (exit (); push s v)
```

- What if this thread falls asleep at the start of line 6 or 7?

Treiber's Stack, Fixed: push

```
1 let rec push s v =
2   let xs = new_cell () in
3   set_data xs v;
4   enter ();
5   let ys = !s in
6   set_tail xs ys;
7   if compare_and_swap s ys xs
8   then exit ()
9   else (exit (); push s v)
```

- What if this thread falls asleep at the start of line 6 or 7?
- Some other thread may successfully pop `ys`.

Treiber's Stack, Fixed: push

```
1 let rec push s v =
2   let xs = new_cell () in
3   set_data xs v;
4   enter ();
5   let ys = !s in
6   set_tail xs ys;
7   if compare_and_swap s ys xs
8   then exit ()
9   else (exit (); push s v)
```

- What if this thread falls asleep at the start of line 6 or 7?
- Some other thread may successfully pop ys.
- That thread must logically deallocate ys,

Treiber's Stack, Fixed: push

```
1 let rec push s v =
2   let xs = new_cell () in
3   set_data xs v;
4   enter ();
5   let ys = !s in
6   set_tail xs ys;
7   if compare_and_swap s ys xs
8   then exit ()
9   else (exit (); push s v)
```

- What if this thread falls asleep at the start of line 6 or 7?
- Some other thread may successfully pop `ys`.
- That thread must logically deallocate `ys`,
- and to do so, must also logically deallocate `xs`.

Treiber's Stack, Fixed: push

```
1 let rec push s v =
2   let xs = new_cell () in
3   set_data xs v;
4   enter ();
5   let ys = !s in
6   set_tail xs ys;
7   if compare_and_swap s ys xs
8   then exit ()
9   else (exit (); push s v)
```

- What if this thread falls asleep at the start of line 6 or 7?
- Some other thread may successfully pop `ys`.
- That thread must logically deallocate `ys`,
- and to do so, must also logically deallocate `xs`.
- This is possible, as `ys` is a temporary root.

Treiber's Stack, Fixed: push

```
1 let rec push s v =
2   let xs = new_cell () in
3   set_data xs v;
4   enter ();
5   let ys = !s in
6   set_tail xs ys;
7   if compare_and_swap s ys xs
8   then exit ()
9   else (exit (); push s v)
```

- What if this thread falls asleep at the start of line 6 or 7?
- Some other thread may successfully pop `ys`.
- That thread must logically deallocate `ys`,
- and to do so, must also logically deallocate `xs`.
- This is possible, as `ys` is a temporary root.
- Once this thread wakes up, lines 6 or 7 access a logically deallocated block!
- This is possible, as logical deallocation does not consume the points-to assertion.

Specification of Treiber's Stack

$$\frac{\{\ \Diamond 2\ \}}{\dfrac{\langle\ \forall L.\ \mathsf{stack}\ s\ L\ \rangle \quad \pi : \mathsf{push}\ s\ v \quad \langle\ \mathsf{stack}\ s\ (v :: L)\ \rangle}{\{\ \lambda().\ \mathsf{True}\ \}}}$$

Specification of Treiber's Stack

Private precondition $\longrightarrow$

$$\frac{\{\ \Diamond 2\ \}}{\dfrac{\langle\ \forall L.\ \mathsf{stack}\ s\ L\ \rangle \quad \pi : \mathsf{push}\ s\ v \quad \langle\ \mathsf{stack}\ s\ (v :: L)\ \rangle}{\{\ \lambda().\ \mathsf{True}\ \}}}$$

Specification of Treiber's Stack

Private precondition

Public precondition

$$\frac{\{\ \Diamond 2\ \}}{\dfrac{\begin{array}{c}\langle\ \forall L.\ \mathsf{stack}\ s\ L\ \rangle \\ \pi : \mathsf{push}\ s\ v \\ \langle\ \mathsf{stack}\ s\ (v :: L)\ \rangle\end{array}}{\{\ \lambda().\ \mathsf{True}\ \}}}$$

Specification of Treiber's Stack

Private precondition

Public precondition

Public postcondition

$$\frac{\{\ \Diamond 2\ \}}{\dfrac{\langle\ \forall L.\ \mathsf{stack}\ s\ L\ \rangle \quad \pi : \mathsf{push}\ s\ v \quad \langle\ \mathsf{stack}\ s\ (v :: L)\ \rangle}{\{\ \lambda().\ \mathsf{True}\ \}}}$$

Specification of Treiber's Stack

Private precondition

Public precondition

Public postcondition

Private postcondition

$$\dfrac{\dfrac{\{\ \Diamond 2\ \}}{\langle\ \forall L.\ \mathsf{stack}\ s\ L\ \rangle}}{} \quad \pi : \mathsf{push}\ s\ v \quad \dfrac{\langle\ \mathsf{stack}\ s\ (v :: L)\ \rangle}{\{\ \lambda().\ \mathsf{True}\ \}}$$

Specification of Treiber's Stack

Private precondition → Public precondition ← Public postcondition ← Private postcondition →

$$\cfrac{\{\, \Diamond 2 \,\}}{\begin{array}{c}\langle\, \forall L.\ \mathsf{stack}\ s\ L \,\rangle \\ \pi : \mathsf{push}\ s\ v \\ \langle\, \mathsf{stack}\ s\ (v :: L) \,\rangle\end{array}}\over\{\, \lambda().\ \mathsf{True} \,\}$$

$$\cfrac{\{\, \mathsf{True} \,\}}{\begin{array}{c}\langle\, \forall v\ L.\ \mathsf{stack}\ s\ (v :: L) \,\rangle \\ \pi : \mathsf{pop}\ s \\ \langle\, \mathsf{stack}\ s\ L \,\rangle\end{array}}\over\{\, \lambda\, v'.\ \ulcorner v' = v \urcorner \ * \ \Diamond 2 \,\}$$

Polling Point Insertion

Polling points have no effect on reasoning.

$$\{\,\text{outside}\,\pi\,\}\;\pi\!:\text{poll}\,()\;\{\,\lambda().\;\text{outside}\,\pi\,\}$$

They can be automatically inserted by the compiler outside of protected sections.

If enough polling points are inserted,
then at all times,
every thread must reach a polling point in bounded time.

Safety and Liveness Theorems

Under the assumption that the program is verified:

$$\{\, \Diamond S \;\ast\; \text{outside}\,\pi \,\}\, \pi \colon t \,\{\, \lambda\_.\ \text{outside}\,\pi \,\}$$

Safety and Liveness Theorems

Under the assumption that the program is verified:

$$\{ \Diamond S \;*\; \mathsf{outside}\,\pi \}\ \pi : t\ \{ \lambda\_.\ \mathsf{outside}\,\pi \}$$

Then, at all times,

Safety

Every thread has terminated, is blocked on allocation, or can make progress.

Safety and Liveness Theorems

Under the assumption that the program is verified:

$$\{ \Diamond S \ * \ \text{outside}\, \pi \} \, \pi : t \, \{ \lambda\_. \, \text{outside}\, \pi \}$$

Then, at all times,

Safety

Every thread has terminated, is blocked on allocation, or can make progress.

and assuming that enough polling points have been inserted, at all times,

Liveness

Every blocked thread is eventually unblocked.

Enough for Today!

Under review (journal paper):

Will it Fit? Verifying Heap Space Bounds of Concurrent Programs under Garbage Collection with Separation Logic

See also Alexandre's dissertation (upcoming).

Thank you for your attention!

alexandre.moine [at] inria.fr

https://cambium.inria.fr/~amoine/

References i

Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. Theoretical Computer Science, 103(2):235–271, 1992. URL https://www2.ccs.neu.edu/racket/pubs/tcs92-fh.pdf.

Martin Hofmann. Linear types and non-size-increasing polynomial time computation. In Logic in Computer Science (LICS), pages 464–473, July 1999. URL https://doi.org/10.1109/LICS.1999.782641.

Ioannis T. Kassios and Eleftherios Kritikos. A discipline for program verification based on backpointers and its use in observational disjointness. In European Symposium on Programming (ESOP), volume 7792 of Lecture Notes in Computer Science, pages 149–168. Springer, March 2013. URL https://doi.org/10.1007/978-3-642-37036-6_10.

Jean-Marie Madiot and François Pottier. A separation logic for heap space under garbage collection. Proceedings of the ACM on Programming Languages, 6(POPL), January 2022. URL http://cambium.inria.fr/~fpottier/publis/madiot-pottier-diamonds-2022.pdf.

Alexandre Moine, Arthur Charguéraud, and François Pottier. A high-level separation logic for heap space under garbage collection. Proc. ACM Program. Lang., 7(POPL), jan 2023. doi: 10.1145/3571218. URL https://doi.org/10.1145/3571218.