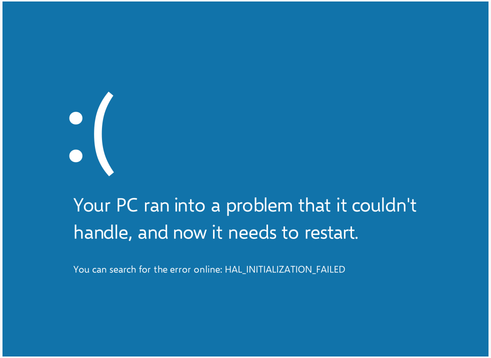
Formal Verification of Heap Space Bounds under Garbage Collection

Alexandre Moine
advised by Arthur Charguéraud and François Pottier

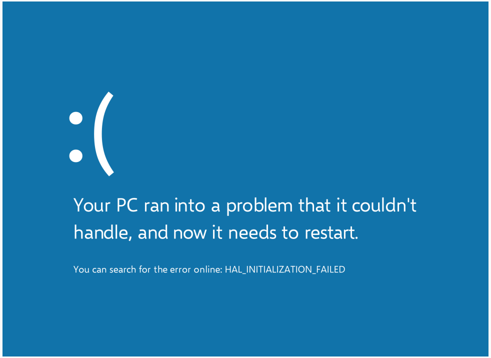
20/09/2024

Programs are everywhere... and bugs too!

Programs are everywhere... and bugs too!



Programs are everywhere... and bugs too!



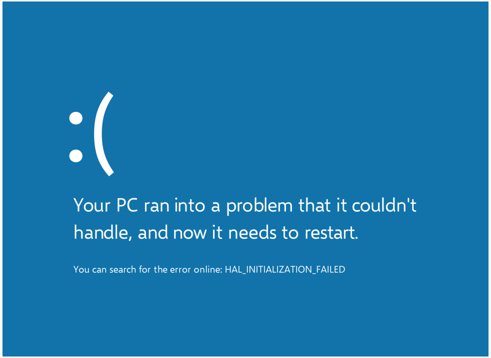

The New York Times

Chaos and Confusion: Tech Outage Causes Disruptions Worldwide

Airlines, hospitals and people's computers were affected after CrowdStrike, a cybersecurity company, sent out a flawed software update.

By Adam Satariano, Paul Mozur, Kate Conger and Sheera Frenkel
July 19, 2024

Programs are everywhere... and bugs too!



The New York Times

Chaos and Confusion: Tech Outage Causes Disruptions Worldwide

Airlines, hospitals and people's computers were affected after CrowdStrike, a cybersecurity company, sent out a flawed software update.

By Adam Satariano, Paul Mozur, Kate Conger and Sheera Frenkel
July 19, 2024

How to ensure that a program has no bugs?

Focusing on a Bug Category

Correctness	$\rightsquigarrow$	The program does not compute the correct result.
Security	$\rightsquigarrow$	The program allows a thief to steal private data.
Resource usage	$\rightsquigarrow$	The program uses more resources than expected.

Focusing on a Bug Category

Correctness	$\rightsquigarrow$	The program does not compute the correct result.
Security	$\rightsquigarrow$	The program allows a thief to steal private data.
Resource usage	$\rightsquigarrow$	The program uses more resources than expected.
Time usage	$\rightsquigarrow$	The program takes too much time to produce an answer.
Space usage	$\rightsquigarrow$	The program requires more memory than available and crashes.

Focusing on a Bug Category

Correctness	$\rightsquigarrow$	The program does not compute the correct result.
Security	$\rightsquigarrow$	The program allows a thief to steal private data.
Resource usage	$\rightsquigarrow$	The program uses more resources than expected.

Time usage	$\rightsquigarrow$	The program takes too much time to produce an answer.
Space usage	$\rightsquigarrow$	The program requires more memory than available and crashes.

Informal Central Question

How to bound the amount of memory required by a program?

A Reminder on Memory

- External memory for files
- RAM for runtime computations

A Reminder on Memory

- External memory for files
- RAM for runtime computations

The RAM is usually split in two parts:

- the stack for data whose lifetime does not exceed the one of the allocating function
- the heap for everything else

A Reminder on Memory

- External memory for files
- RAM for runtime computations

The RAM is usually split in two parts:

- the stack for data whose lifetime does not exceed the one of the allocating function
- the heap for everything else

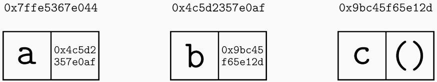
- The stack stores data following a strict discipline.
  $\rightsquigarrow$ Establishing stack space bounds is well-studied [Carbonneaux et al., 2014].
- The heap is under the control of the programmer.
  $\rightsquigarrow$ Establishing heap space bounds a subtle task!

The Heap

The heap is made of allocated blocks, each one having a particular location.

The Heap

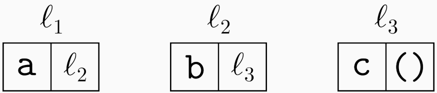
The heap is made of allocated blocks, each one having a particular location.



represents the linked list `[a;b;c]`

The Heap

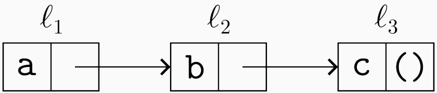
The heap is made of allocated blocks, each one having a particular location.



represents the linked list `[a;b;c]`

The Heap

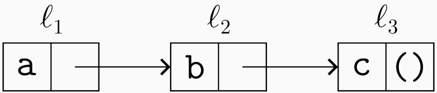
The heap is made of allocated blocks, each one having a particular location.



represents the linked list `[a;b;c]`

The Heap

The heap is made of allocated blocks, each one having a particular location.



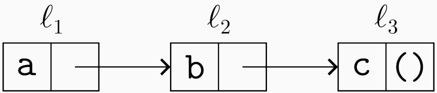
represents the linked list `[a;b;c]`

The program

- requests the allocation of a block (consuming free space),
- obtains the location of a fresh block,
- and can then write and read from it.

The Heap

The heap is made of allocated blocks, each one having a particular location.



represents the linked list `[a;b;c]`

The program

- requests the allocation of a block (consuming free space),
- obtains the location of a fresh block,
- and can then write and read from it.

Deallocation: with `free` / manual memory management vs. without `free` / garbage collection

Manual Memory Management

Languages such as C have explicit `free` operations.

Manual Memory Management

Languages such as C have explicit `free` operations.

```
char* arr = malloc(n);

free(arr)
```

Manual Memory Management

Languages such as C have explicit `free` operations.

```
char* arr = malloc(n);
// -n bytes of available space
free(arr)
// +n bytes of available space
```

- ✔ Known techniques allow for proving heap space bounds: a resource meter can track the available heap space.
- ✖ Manual memory deallocation is error-prone: memory leak, use-after-free, double-free, etc.

Garbage Collection

Languages such as Java and OCaml have implicit deallocation.

- A garbage collector (GC) runs together with the program.
- From time to time, the GC may deallocate unreachable blocks.

Garbage Collection

Languages such as Java and OCaml have implicit deallocation.

- A garbage collector (GC) runs together with the program.
- From time to time, the GC may deallocate unreachable blocks.

✔ Garbage collection simplifies the life of the programmer: no need to write `free`, no `free`-related bugs.

? It is not clear how to establish a heap space bound.

Garbage Collection

Languages such as Java and OCaml have implicit deallocation.

- A garbage collector (GC) runs together with the program.
- From time to time, the GC may deallocate unreachable blocks.

✔ Garbage collection simplifies the life of the programmer: no need to write `free`, no `free`-related bugs.

? It is not clear how to establish a heap space bound.

Central Question of my Thesis

How to establish heap space bounds in the presence of garbage collection?

Reachable Memory

```
let x = ref 42 in
let y = ref x in
x := 21;
let z = ref 84 in
y := z;
y
```

The heap

Reachable Memory

```
let x = ℓ_x in
let y = ref x in
x := 21;
let z = ref 84 in
y := z;
y
```

The heap

ℓ_x

$\boxed{42}$

Reachable Memory

```
let y = ref ℓx in
ℓx := 21;
let z = ref 84 in
y := z;
y
```

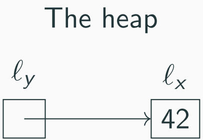
The heap

ℓ_x

$\boxed{42}$

Reachable Memory

```
ℓx := 21;
let z = ref 84 in
ℓy := z;
ℓy
```



Reachable Memory

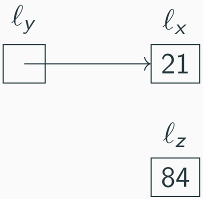
The heap


```
let z = ref 84 in
ℓy := z;
ℓy
```

Reachable Memory

ℓ_y := ℓ_z;
ℓ_y

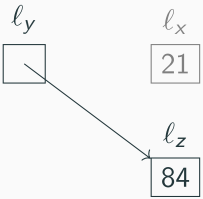
The heap



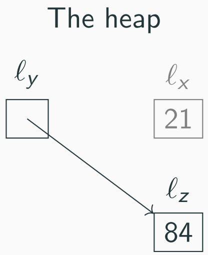
Reachable Memory

ℓ_y

The heap



Reachable Memory



ℓ_y

Definitions

- The roots are the locations occurring in the program that remains to execute.
- The set of reachable blocks is computed from the roots following heap paths.

A More Elaborated Example

```
let rec mapsucc (xs : int list) : int list =
  match xs with
  | [] -> []
  | y::ys -> (y+1)::(mapsucc ys)
```

What is the amount of free heap space needed by a call to `mapsucc xs`?

A More Elaborated Example

```
let rec mapsucc (xs : int list) : int list =
  match xs with
  | [] -> []
  | y::ys -> (y+1)::(mapsucc ys)
```

What is the amount of free heap space needed by a call to `mapsucc xs`?

The answer depends on the calling context!

A More Elaborated Example

```
let rec mapsucc (xs : int list) : int list =
  match xs with
  | [] -> []
  | y::ys -> (y+1)::(mapsucc ys)
```

What is the amount of free heap space needed by a call to `mapsucc xs`?

The answer depends on the calling context!

- If `xs` is reachable from the calling context: $O(\text{length}\,\text{xs})$

A More Elaborated Example

```
let rec mapsucc (xs : int list) : int list =
  match xs with
  | [] -> []
  | y::ys -> (y+1)::(mapsucc ys)
```

What is the amount of free heap space needed by a call to `mapsucc xs`?

The answer depends on the calling context!

- If `xs` is reachable from the calling context: $O(\text{length}\,\texttt{xs})$
- If `xs` is unreachable from the calling context: $O(1)$
  Cells from the input list can be freed as cells from the output list are allocated.

Goal: A Program Logic

We devise a variant of Separation Logic [O'Hearn et al., 2001, Reynolds, 2002].

$$\{\, \Phi \,\}\; t \;\{\, \lambda v.\; \Psi \,\}$$

Φ describes the heap before executing t. Ψ describes the heap after executing t.

Goal: A Program Logic

We devise a variant of Separation Logic [O'Hearn et al., 2001, Reynolds, 2002].

$$\{\, \Phi \,\}\; t \;\{\, \lambda v.\, \Psi \,\}$$

Φ describes the heap before executing t. Ψ describes the heap after executing t.

Standard reasoning rules:

$$\{\, \mathsf{True} \,\}\; \mathsf{alloc}\, 1 \;\{\, \lambda \ell.\, \ell \mapsto () \,\}$$

$$\{\, \ell \mapsto v \,\}\; \mathsf{load}\, \ell \;\{\, \lambda w.\, \ulcorner w = v \urcorner \;*\; \ell \mapsto v \,\} \qquad \{\, \ell \mapsto v \,\}\; \mathsf{store}\, \ell\, w \;\{\, \lambda ().\, \ell \mapsto w \,\}$$

Goal: A Program Logic

We devise a variant of Separation Logic [O'Hearn et al., 2001, Reynolds, 2002].

$$\{\,\Phi\,\}\; t \;\{\,\lambda v.\; \Psi\,\}$$

Φ describes the heap before executing t. Ψ describes the heap after executing t.

Standard reasoning rules:

$$\{\,\mathsf{True}\,\}\;\mathsf{alloc}\,1\;\{\,\lambda \ell.\; \ell \mapsto ()\,\}$$

$$\{\,\ell \mapsto v\,\}\;\mathsf{load}\,\ell\;\{\,\lambda w.\; \ulcorner w = v \urcorner \;\ast\; \ell \mapsto v\,\} \qquad \{\,\ell \mapsto v\,\}\;\mathsf{store}\,\ell\,w\;\{\,\lambda ().\; \ell \mapsto w\,\}$$

Can these rules be adapted to account for the available space under garbage collection?

Contribution

IrisFit, the first Separation Logic for verifying
heap space bounds in a high-level concurrent language equipped with a GC.

Contribution

IrisFit, the first Separation Logic for verifying
heap space bounds in a high-level concurrent language equipped with a GC.

Including:

- a formal account of unreachability and roots
- new realistic language constructs: protected sections and polling points, improving heap space bounds of lock-free data structures
- case studies and the soundness proof

Contribution

IrisFit, the first Separation Logic for verifying
heap space bounds in a high-level concurrent language equipped with a GC.

Including:

- a formal account of unreachability and roots
- new realistic language constructs: protected sections and polling points, improving heap space bounds of lock-free data structures
- case studies and the soundness proof

Reasoning rules, case studies and soundness are mechanized.

Contribution

IrisFit, the first Separation Logic for verifying
heap space bounds in a high-level concurrent language equipped with a GC.

Including:

Part I a formal account of unreachability and roots

Part II new realistic language constructs: protected sections and polling points, improving heap space bounds of lock-free data structures

- case studies and the soundness proof

Reasoning rules, case studies and soundness are mechanized.

Part I: Heap Space Bounds for Sequential Programs

Based on

A High-Level Separation Logic for Heap Space under Garbage Collection

[Moine, Charguéraud, and Pottier; POPL'23]

Space Credits

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

- A space credit represents one free memory word.
- Space credits are splittable: $\Diamond(n_1 + n_2) \equiv \Diamond n_1 * \Diamond n_2$
- Space credits are not duplicable: $\Diamond n \not\Longrightarrow \Diamond n * \Diamond n$

Space Credits

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

- A space credit represents one free memory word.
- Space credits are splittable: $\Diamond(n_1 + n_2) \equiv \Diamond n_1 \ast \Diamond n_2$
- Space credits are not duplicable: $\Diamond n \not\Rrightarrow \Diamond n \ast \Diamond n$

With manual memory management:

`alloc` consumes space credits

$$\{\, \Diamond 1 \,\}\ \mathsf{alloc}\, 1\ \{\, \lambda \ell.\ \ell \mapsto () \,\}$$

`free` produces space credits

$$\{\, \ell \mapsto v \,\}\ \mathsf{free}\, \ell\ \{\, \lambda().\ \Diamond 1 \,\}$$

With Garbage Collection: Where to Recover Space Credits?

With Garbage Collection: Where to Recover Space Credits?

Key Idea [Madiot and Pottier, 2022]

$\Diamond 1$ asserts that one memory word is free or can be freed by the GC.

With Garbage Collection: Where to Recover Space Credits?

Key Idea [Madiot and Pottier, 2022]

$\Diamond 1$ asserts that one memory word is free or can be freed by the GC.

Logical deallocation rule: $$(\,\ell \mapsto v \;*\; \text{“}\ell\text{ is unreachable”}\,) \;\Rrightarrow\; \Diamond 1$$

With Garbage Collection: Where to Recover Space Credits?

Key Idea [Madiot and Pottier, 2022]

$\Diamond 1$ asserts that one memory word is free or can be freed by the GC.

Logical deallocation rule: $$\left(\ell \mapsto v \;*\; \text{“}\ell\text{ is unreachable”}\right) \;\Rrightarrow\; \Diamond 1$$

Madiot and Pottier [2022] use a low-level, assembly-like, language with an explicit stack.

Motivating Question

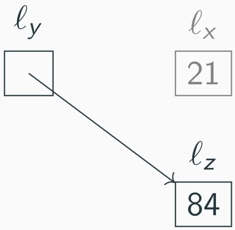
Can Madiot and Pottier's approach be scaled up to a high-level language?

How to Prove that "ℓ is unreachable"?

How to Prove that “ℓ is unreachable”?

The set of reachable locations is computed:

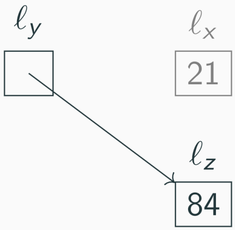
1. from the roots
2. following heap paths.



How to Prove that “ℓ is unreachable”?

The set of reachable locations is computed:

1. from the roots
2. following heap paths.



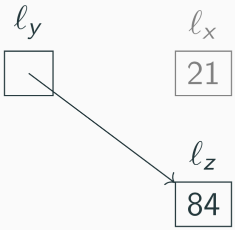
A location ℓ is unreachable if and only if:

1. ℓ is not a root
2. ℓ is not reachable by any reachable heap cell

How to Prove that "ℓ is unreachable"?

The set of reachable locations is computed:

1. from the roots
2. following heap paths.



A location ℓ is unreachable if and only if:

1. ℓ is not a root $\rightsquigarrow$ the pointed-by-thread assertion
2. ℓ is not reachable by any reachable heap cell $\rightsquigarrow$ the pointed-by-heap assertion

The Pointed-by-Heap Assertion

The pointed-by-heap assertion [Kassios and Kritikos, 2013, Madiot and Pottier, 2022]

$$\ell \hookleftarrow A$$

- A is a multiset of locations.
- $\ell \hookleftarrow A$ asserts that A is an over-approximation of the predecessors of ℓ.
- $\ell \hookleftarrow \emptyset$ asserts that ℓ has no predecessors.

The Pointed-by-Heap Assertion

The pointed-by-heap assertion [Kassios and Kritikos, 2013, Madiot and Pottier, 2022]

$$\ell \hookleftarrow A$$

- A is a multiset of locations.
- $\ell \hookleftarrow A$ asserts that A is an over-approximation of the reachable predecessors of ℓ.
- $\ell \hookleftarrow \emptyset$ asserts that ℓ has no reachable predecessors.

The Pointed-by-Heap Assertion

The pointed-by-heap assertion [Kassios and Kritikos, 2013, Madiot and Pottier, 2022]

$$\ell \hookleftarrow A$$

- A is a multiset of locations.
- $\ell \hookleftarrow A$ asserts that A is an over-approximation of the reachable predecessors of ℓ.
- $\ell \hookleftarrow \emptyset$ asserts that ℓ has no reachable predecessors.

We improve these assertions with fraction 0 and negative multisets. (Not shown here)

The Pointed-by-Thread Assertion

IrisFit features the pointed-by-thread assertion:

$$\ell \Lleftarrow \Pi$$

- Π is a set of thread identifiers. In a sequential setting, either $\{\pi\}$ or $\emptyset$.
- $\ell \Lleftarrow \{\pi\}$ asserts that ℓ may still be a root.
- $\ell \Lleftarrow \emptyset$ asserts that ℓ is not a root!

A Few Simplified Reasoning Rules

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

$$\{\, \Diamond 1 \,\}\ \mathsf{alloc}\ 1\ \{\, \lambda\ell.\ \ell \mapsto () \ * \ \ell \hookleftarrow \emptyset \ * \ \ell \Lleftarrow \{\pi\} \,\}$$

A Few Simplified Reasoning Rules

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

$$\{\,\Diamond 1\,\}\,\mathsf{alloc}\ 1\,\{\,\lambda\ell.\ \ell \mapsto () \ * \ \ell \leftarrow\!\!\dashv \emptyset \ * \ \ell \Leftarrow\!\!\dashv \{\pi\}\,\}$$

store updates pointed-by-heap assertions:

$$\{\,\ell \mapsto v \ * \ v \leftarrow\!\!\dashv \{+\ell\} \ * \ w \leftarrow\!\!\dashv \emptyset\,\}\,\mathsf{store}\,\ell\,w\,\{\,\lambda().\ \ell \mapsto w \ * \ v \leftarrow\!\!\dashv \emptyset \ * \ w \leftarrow\!\!\dashv \{+\ell\}\,\}$$

A Few Simplified Reasoning Rules

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

$$\{\,\Diamond 1\,\}\,\text{alloc}\ 1\,\{\,\lambda\ell.\ \ell \mapsto () \ * \ \ell \hookleftarrow \emptyset \ * \ \ell \Lleftarrow \{\pi\}\,\}$$

store updates pointed-by-heap assertions:

$$\{\,\ell \mapsto v \ * \ v \hookleftarrow \{+\ell\} \ * \ w \hookleftarrow \emptyset\,\}\,\text{store}\ \ell\ w\,\{\,\lambda().\ \ell \mapsto w \ * \ v \hookleftarrow \emptyset \ * \ w \hookleftarrow \{+\ell\}\,\}$$

load updates a pointed-by-thread assertion:

$$\{\,\ell \mapsto \ell' \ * \ \ell' \Lleftarrow \emptyset\,\}\,\text{load}\ \ell\,\{\,\lambda v.\ \ulcorner v = \ell' \urcorner \ * \ \ell \mapsto \ell' \ * \ \ell' \Lleftarrow \{\pi\}\,\}$$

Trimming and Logical Deallocation

When ℓ is not a root anymore, we can trim its pointed-by-thread assertion.

$$
\begin{array}{rl}
 & \{\, \ell \Lleftarrow \emptyset \;\; * \;\; \Phi \,\}\; t \;\{\, \Psi \,\} \;\; \land \;\; \ell \notin \mathit{roots}(t) \\
\Longrightarrow & \{\, \ell \Lleftarrow \{\pi\} \;\; * \;\; \Phi \,\}\; t \;\{\, \Psi \,\}
\end{array}
$$

Trimming and Logical Deallocation

When ℓ is not a root anymore, we can trim its pointed-by-thread assertion.

$$\begin{array}{rl} & \{\, \ell \Lleftarrow \emptyset \quad * \;\Phi \,\}\, t \,\{\, \Psi \,\} \;\wedge\; \ell \notin \mathit{roots}(t) \\ \Longrightarrow & \{\, \ell \Lleftarrow \{\pi\} \;*\; \Phi \,\}\, t \,\{\, \Psi \,\} \end{array}$$

For experts: this trimming rule requires a non-standard Let rule. (Not shown here)

Trimming and Logical Deallocation

When ℓ is not a root anymore, we can trim its pointed-by-thread assertion.

$$\begin{array}{rl} & \{\, \ell \Lleftarrow \emptyset \;\; * \;\; \Phi \,\}\; t \;\{\, \Psi \,\} \;\wedge\; \ell \notin \mathit{roots}(t) \\ \Longrightarrow & \{\, \ell \Lleftarrow \{\pi\} \;\; * \;\; \Phi \,\}\; t \;\{\, \Psi \,\} \end{array}$$

For experts: this trimming rule requires a non-standard Let rule. (Not shown here)

Unveiling our logical deallocation rule:

$$(\, \ell \mapsto v \;\; * \;\; \text{“}\ell \text{ is unreachable”} \,) \;\; \Rrightarrow \;\; \Diamond 1$$

Trimming and Logical Deallocation

When ℓ is not a root anymore, we can trim its pointed-by-thread assertion.

$$\begin{array}{rl} & \{\, \ell \Lleftarrow \emptyset \quad * \;\; \Phi \,\}\, t \,\{\, \Psi \,\} \;\; \wedge \;\; \ell \notin \mathit{roots}(t) \\ \Longrightarrow & \{\, \ell \Lleftarrow \{\pi\} \;\; * \;\; \Phi \,\}\, t \,\{\, \Psi \,\} \end{array}$$

For experts: this trimming rule requires a non-standard Let rule. (Not shown here)

Unveiling our logical deallocation rule:

$$(\ell \mapsto v \quad * \quad \ell \hookleftarrow \emptyset \quad * \quad \ell \Lleftarrow \emptyset) \quad \Rrightarrow \quad \Diamond 1$$

A Small Example

```
let x = ref 66 in
let y = ref x in
y := (!x / 2);
let z = ref 9 in
!z + !y
```

- Correctness: what is the result of this program?
- Heap space bound: how much memory does it need?

A Small Example

$\{\Diamond 2\}$

```
let x = ref 66 in
let y = ref x in
y := (!x / 2);
let z = ref 9 in
!z + !y
```

$\{\lambda v.\ulcorner v = 42 \urcorner * \Diamond 2\}$

- Correctness: what is the result of this program?
- Heap space bound: how much memory does it need?

A Small Example, Verified

$\{ \Diamond 2 \}$

```
let x = ref 66 in

let y = ref x in

y := (!x / 2);



let z = ref 9 in

!z + !y
```

$\{ \lambda v. \ulcorner v = 42 \urcorner \ast \Diamond 2 \}$

A Small Example, Verified

$\{ \Diamond 2 \}$

```
let x = ref 66 in
```

$\{ \Diamond 1 \ * \ \ell_x \mapsto 66 \ * \ \ell_x \hookleftarrow \emptyset \qquad * \ \ell_x \Lleftarrow \{\pi\} \}$

```
let y = ref x in

y := (!x / 2);




let z = ref 9 in

!z + !y
```

$\{ \lambda v. \ulcorner v = 42 \urcorner \ * \ \Diamond 2 \}$

A Small Example, Verified

$\{\,\Diamond 2\,\}$

```
let x = ref 66 in
```

$\{\,\Diamond 1 \;*\; \ell_x \mapsto 66 \;*\; \ell_x \hookleftarrow \emptyset \qquad\quad * \; \ell_x \Lleftarrow \{\pi\}\,\}$

```
let y = ref x in
```

$\{ \qquad\quad \ell_x \mapsto 66 \;*\; \ell_x \hookleftarrow \{+\ell_y\} \;*\; \ell_x \Lleftarrow \{\pi\} \;*\; \ell_y \mapsto \ell_x \;*\; \ell_y \hookleftarrow \emptyset \;*\; \ell_y \Lleftarrow \{\pi\}\,\}$

```
y := (!x / 2);
```

```
let z = ref 9 in
```

```
!z + !y
```

$\{\,\lambda v.\, \ulcorner v = 42 \urcorner \;*\; \Diamond 2\,\}$

A Small Example, Verified

$\{\, \Diamond 2 \,\}$

```
let x = ref 66 in
```

$\{\, \Diamond 1 \;*\; \ell_x \mapsto 66 \;*\; \ell_x \hookleftarrow \emptyset \;*\; \ell_x \Lleftarrow \{\pi\} \,\}$

```
let y = ref x in
```

$\{\, \ell_x \mapsto 66 \;*\; \ell_x \hookleftarrow \{+\ell_y\} \;*\; \ell_x \Lleftarrow \{\pi\} \;*\; \ell_y \mapsto \ell_x \;*\; \ell_y \hookleftarrow \emptyset \;*\; \ell_y \Lleftarrow \{\pi\} \,\}$

```
y := (!x / 2);
```

$\{\, \ell_x \mapsto 66 \;*\; \ell_x \hookleftarrow \emptyset \;*\; \ell_x \Lleftarrow \{\pi\} \;*\; \ell_y \mapsto 33 \;*\; \ell_y \hookleftarrow \emptyset \;*\; \ell_y \Lleftarrow \{\pi\} \,\}$

```
let z = ref 9 in
```

```
!z + !y
```

$\{\, \lambda v.\, \ulcorner v = 42 \urcorner \;*\; \Diamond 2 \,\}$

A Small Example, Verified

$\{\Diamond 2\}$

```
let x = ref 66 in
```

$\{\Diamond 1 \ * \ \ell_x \mapsto 66 \ * \ \ell_x \leftarrow\!\dashv \emptyset \ * \ \ell_x \Lleftarrow \{\pi\}\}$

```
let y = ref x in
```

$\{ \ell_x \mapsto 66 \ * \ \ell_x \leftarrow\!\dashv \{+\ell_y\} \ * \ \ell_x \Lleftarrow \{\pi\} \ * \ \ell_y \mapsto \ell_x \ * \ \ell_y \leftarrow\!\dashv \emptyset \ * \ \ell_y \Lleftarrow \{\pi\}\}$

```
y := (!x / 2);
```

$\{ \ell_x \mapsto 66 \ * \ \ell_x \leftarrow\!\dashv \emptyset \ * \ \ell_x \Lleftarrow \{\pi\} \ * \ \ell_y \mapsto 33 \ * \ \ell_y \leftarrow\!\dashv \emptyset \ * \ \ell_y \Lleftarrow \{\pi\}\}$

$\{ \ell_x \mapsto 66 \ * \ \ell_x \leftarrow\!\dashv \emptyset \ * \ \ell_x \Lleftarrow \emptyset \ * \ \ell_y \mapsto 33 \ * \ \ell_y \leftarrow\!\dashv \emptyset \ * \ \ell_y \Lleftarrow \{\pi\}\}$

```
let z = ref 9 in

!z + !y
```

$\{\lambda v. \ulcorner v = 42 \urcorner \ * \ \Diamond 2\}$

A Small Example, Verified

$\{\,\Diamond 2\,\}$

```
let x = ref 66 in
```

$\{\,\Diamond 1 \;*\; \ell_x \mapsto 66 \;*\; \ell_x \hookleftarrow \emptyset \;*\; \ell_x \Lleftarrow \{\pi\}\,\}$

```
let y = ref x in
```

$\{\,\ell_x \mapsto 66 \;*\; \ell_x \hookleftarrow \{+\ell_y\} \;*\; \ell_x \Lleftarrow \{\pi\} \;*\; \ell_y \mapsto \ell_x \;*\; \ell_y \hookleftarrow \emptyset \;*\; \ell_y \Lleftarrow \{\pi\}\,\}$

```
y := (!x / 2);
```

$\{\,\ell_x \mapsto 66 \;*\; \ell_x \hookleftarrow \emptyset \;*\; \ell_x \Lleftarrow \{\pi\} \;*\; \ell_y \mapsto 33 \;*\; \ell_y \hookleftarrow \emptyset \;*\; \ell_y \Lleftarrow \{\pi\}\,\}$

$\{\,\ell_x \mapsto 66 \;*\; \ell_x \hookleftarrow \emptyset \;*\; \ell_x \Lleftarrow \emptyset \;*\; \ell_y \mapsto 33 \;*\; \ell_y \hookleftarrow \emptyset \;*\; \ell_y \Lleftarrow \{\pi\}\,\}$

$\{\,\Diamond 1 \;*\; \ell_y \mapsto 33 \;*\; \ell_y \hookleftarrow \emptyset \;*\; \ell_y \Lleftarrow \{\pi\}\,\}$

```
let z = ref 9 in


!z + !y
```

$\{\,\lambda v.\, \ulcorner v = 42 \urcorner \;*\; \Diamond 2\,\}$

A Small Example, Verified

$\{\Diamond 2\}$

```
let x = ref 66 in
```

$\{\Diamond 1 \ast \ell_x \mapsto 66 \ast \ell_x \hookleftarrow \emptyset \ast \ell_x \Lleftarrow \{\pi\}\}$

```
let y = ref x in
```

$\{\ell_x \mapsto 66 \ast \ell_x \hookleftarrow \{+\ell_y\} \ast \ell_x \Lleftarrow \{\pi\} \ast \ell_y \mapsto \ell_x \ast \ell_y \hookleftarrow \emptyset \ast \ell_y \Lleftarrow \{\pi\}\}$

```
y := (!x / 2);
```

$\{\ell_x \mapsto 66 \ast \ell_x \hookleftarrow \emptyset \ast \ell_x \Lleftarrow \{\pi\} \ast \ell_y \mapsto 33 \ast \ell_y \hookleftarrow \emptyset \ast \ell_y \Lleftarrow \{\pi\}\}$

$\{\ell_x \mapsto 66 \ast \ell_x \hookleftarrow \emptyset \ast \ell_x \Lleftarrow \emptyset \ast \ell_y \mapsto 33 \ast \ell_y \hookleftarrow \emptyset \ast \ell_y \Lleftarrow \{\pi\}\}$

$\{\Diamond 1 \ast \ell_y \mapsto 33 \ast \ell_y \hookleftarrow \emptyset \ast \ell_y \Lleftarrow \{\pi\}\}$

```
let z = ref 9 in
```

$\{\ell_z \mapsto 9 \ast \ell_z \hookleftarrow \emptyset \ast \ell_z \Lleftarrow \{\pi\} \ast \ell_y \mapsto 33 \ast \ell_y \hookleftarrow \emptyset \ast \ell_y \Lleftarrow \{\pi\}\}$

```
!z + !y
```

$\{\lambda v. \ulcorner v = 42 \urcorner \ast \Diamond 2\}$

Two Specifications for mapsucc xs

- If `xs` is reachable from the calling context: $O(\mathsf{length}\,\mathsf{xs})$
- If `xs` is unreachable from the calling context: $O(1)$

Two Specifications for mapsucc xs

- If `xs` is reachable from the calling context: $O(\text{length}\,\text{xs})$
- If `xs` is unreachable from the calling context: $O(1)$

$$\left\{\begin{array}{c} \mathit{List}\ \ell_{xs}\ L \\ \Diamond(2 \times \text{length}\ L) \end{array}\right\}\ \mathit{mapsucc}\ \ell_{xs}\ \left\{\begin{array}{c} \mathit{List}\ \ell_{xs}\ L \\ \lambda \ell_{ys}.\ \mathit{List}\ \ell_{ys}\ (\mathit{map}\,(+1)\,L) \\ \ell_{ys} \leftarrowtail \emptyset\ *\ \ell_{ys} \Lleftarrow \{\pi\} \end{array}\right\}$$

Two Specifications for mapsucc xs

- If `xs` is reachable from the calling context: $O(\text{length}\,\text{xs})$
- If `xs` is unreachable from the calling context: $O(1)$

$$\left\{\begin{array}{c} \mathit{List}\ \ell_{xs}\ L \\ \Diamond(2 \times \text{length}\ L) \end{array}\right\}\ \mathit{mapsucc}\ \ell_{xs}\ \left\{\begin{array}{c} \mathit{List}\ \ell_{xs}\ L \\ \lambda \ell_{ys}.\ \ \mathit{List}\ \ell_{ys}\ (\mathit{map}\,(+1)\,L) \\ \ell_{ys} \hookleftarrow \emptyset\ *\ \ell_{ys} \Leftarrow \{\pi\} \end{array}\right\}$$

$$\left\{\begin{array}{c} \mathit{List}\ \ell_{xs}\ L \\ \ell_{xs} \hookleftarrow \emptyset\ *\ \ell_{xs} \Leftarrow \{\pi\} \end{array}\right\}\ \mathit{mapsucc}\ \ell_{xs}\ \left\{\lambda \ell_{ys}.\ \begin{array}{c} \mathit{List}\ \ell_{ys}\ (\mathit{map}\,(+1)\,L) \\ \ell_{ys} \hookleftarrow \emptyset\ *\ \ell_{ys} \Leftarrow \{\pi\} \end{array}\right\}$$

Soundness of IrisFit

Soundness Theorem

If $\{\, \Diamond S \,\}\ t\ \{\, \lambda\_.\mathrm{True} \,\}$ holds, then
the execution of the program t with a heap of size at least S

- cannot reach a stuck configuration, and
- cannot run out of memory.

Soundness of IrisFit

Soundness Theorem

If $\{\, \Diamond S \,\}\; t \;\{\, \lambda\_.\mathrm{True} \,\}$ holds, then

the execution of the program t with a heap of size at least S

- cannot reach a stuck configuration, and
- cannot run out of memory.

```
Lemma wp_adequacy (S:ℕ) (t t':tm) (σ:store) :
  locs t = ∅ →
  rtc step (t,∅) (t',σ) →
  (∀ `{!interpGS Σ},
      ⊢ ◊S -∗ wp t (λ _ ⇒ ⊤)) →
  not_stuck t' σ ∧ (live_heap_size (locs t') σ ≤ S).
```

Part II: Scaling up to Concurrency

Based on

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

[Moine, Charguéraud, and Pottier; submitted to TOPLAS]

Concurrency

- Modern computers are multi-core with shared memory.
- Threads execute concurrently and can be created dynamically.

Concurrency

- Modern computers are multi-core with shared memory.
- Threads execute concurrently and can be created dynamically.

```
fork (fun () -> g x); f x
```

Concurrency

- Modern computers are multi-core with shared memory.
- Threads execute concurrently and can be created dynamically.

```
f x   ||   g x
```

Concurrency

- Modern computers are multi-core with shared memory.
- Threads execute concurrently and can be created dynamically.

```
f x   ||   g x
```

Separation Logic scales seamlessly up to concurrency [O'Hearn, 2007, Jung et al., 2018].

Motivating Question

Can IrisFit be scaled up to concurrency?

How to Scale IrisFit up to Concurrency

Step 1: Annotate the triple with a ghost thread identifier π.

$$\{\,\Phi\,\}\ \pi : t\ \{\,\lambda v.\ \Psi\,\}$$

How to Scale IrisFit up to Concurrency

Step 1: Annotate the triple with a ghost thread identifier π.

$$\{\,\Phi\,\}\ \boxed{\pi} : t\ \{\,\lambda v.\ \Psi\,\}$$

Set of thread ids

Step 2: Unveil the power of the pointed-by-thread assertion: $\ell' \Lleftarrow \Pi$

$$\{\,\ell \mapsto \ell' \ * \ \boxed{\ell' \Lleftarrow \{\pi_1\}}\,\}\ \pi_2 : \mathsf{load}\,\ell\ \{\,\lambda v.\ \ulcorner v = \ell' \urcorner \ * \ \ell \mapsto \ell' \ * \ \boxed{\ell' \Lleftarrow \{\pi_1, \pi_2\}}\,\}$$

How to Scale IrisFit up to Concurrency

Step 1: Annotate the triple with a ghost thread identifier π.

$$\{\,\Phi\,\}\ \boxed{\pi} : t\ \{\,\lambda v.\ \Psi\,\}$$

Set of thread ids

Step 2: Unveil the power of the pointed-by-thread assertion: $\ell' \Lleftarrow \Pi$

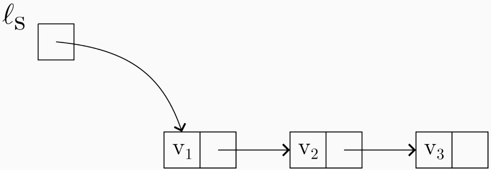
$$\{\,\ell \mapsto \ell' \ * \ \ell' \Lleftarrow \{\pi_1\}\,\}\ \pi_2 : \mathsf{load}\,\ell\ \{\,\lambda v.\ \ulcorner v = \ell' \urcorner \ * \ \ell \mapsto \ell' \ * \ \ell' \Lleftarrow \{\pi_1, \pi_2\}\,\}$$

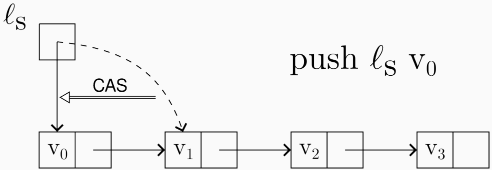
Step 3: Et voilà ?

- I verified some several concurrent programs: a lock, a concurrent counter, ...
- But certain lock-free data structures have an unexpected bound!

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

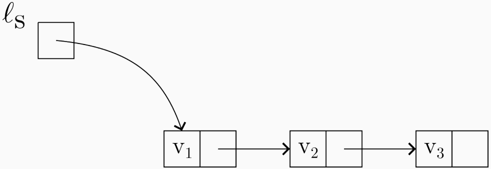
A linearizable lock-free stack, implemented as a reference on an immutable list.

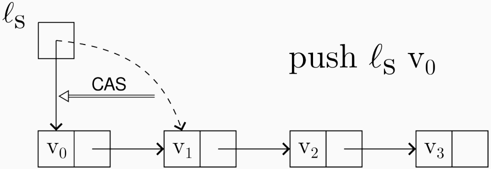




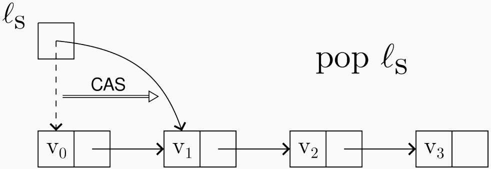
The Case of Lock-Free Data Structures: Treiber's Stack

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






```
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
```



Desired Specification of Treiber's Stack (for Unboxed Values)

$$\left\langle \frac{\Diamond 2}{\forall L.\ \mathsf{stack}\ \ell_s\ L} \right\rangle \pi : \mathsf{push}\ \ell_s\ v \left\langle \frac{\lambda().\qquad \mathsf{True}}{\mathsf{stack}\ \ell_s\ (v :: L)} \right\rangle$$

Desired Specification of Treiber's Stack (for Unboxed Values)

$$\left\langle \frac{\Diamond 2}{\forall L.\ \mathsf{stack}\ \ell_s\ L} \right\rangle \pi : \mathsf{push}\ \ell_s\ v \left\langle \frac{\lambda().\qquad \mathsf{True}}{\mathsf{stack}\ \ell_s\ (v :: L)} \right\rangle$$

$$\left\langle \frac{\mathsf{True}}{\forall v\ L.\ \mathsf{stack}\ \ell_s\ (v :: L)} \right\rangle \pi : \mathsf{pop}\ \ell_s \left\langle \frac{\lambda w.\qquad \ulcorner w = v \urcorner}{\mathsf{stack}\ \ell_s\ L\ \ast\ \Diamond 2} \right\rangle$$

Desired Specification of Treiber's Stack (for Unboxed Values)

$$\left\langle \frac{\Diamond 2}{\forall L.\ \text{stack}\ \ell_s\ L} \right\rangle \pi : \text{push}\ \ell_s\ v \left\langle \frac{\lambda().\qquad \text{True}}{\text{stack}\ \ell_s\ (v :: L)} \right\rangle$$

$$\left\langle \frac{\text{True}}{\forall v\ L.\ \text{stack}\ \ell_s\ (v :: L)} \right\rangle \pi : \text{pop}\ \ell_s \left\langle \frac{\lambda w.\qquad \ulcorner w = v \urcorner}{\text{stack}\ \ell_s\ L\ \ast\ \Diamond 2} \right\rangle$$

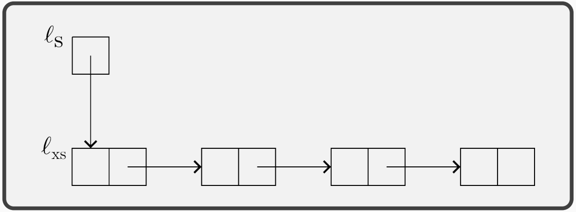
⚠ pop's specification is false: some interleavings invalidate it. ⚠

A Problematic Interleaving for pop

pop ℓ_s

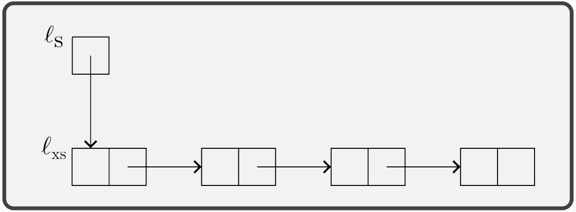
||

pop ℓ_s;
pop ℓ_s;
pop ℓ_s;
a_big_alloc ()



A Problematic Interleaving for pop

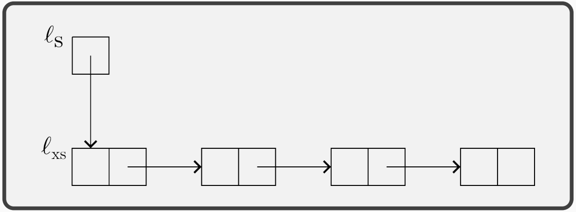
```
let xs = !ℓs in        ||  pop ℓs;
match xs with          ||  pop ℓs;
...                    ||  pop ℓs;
                       ||  a_big_alloc ()
```



A Problematic Interleaving for pop

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

```
pop ℓs;
pop ℓs;
pop ℓs;
a_big_alloc ()
```

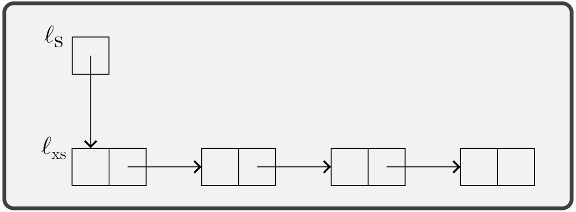


A Problematic Interleaving for pop

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

||

```
pop ℓs;
pop ℓs;
pop ℓs;
a_big_alloc ()
```

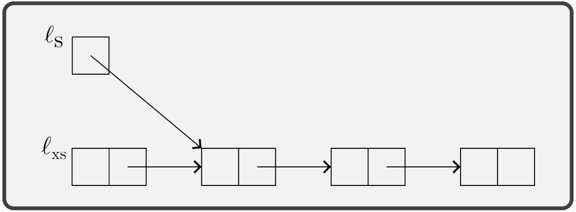


A Problematic Interleaving for pop

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

||

```
pop ℓs;
pop ℓs;
pop ℓs;
a_big_alloc ()
```

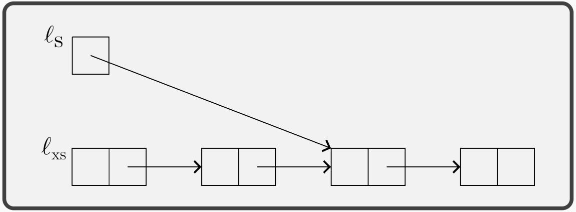


A Problematic Interleaving for pop

let xs = ℓ_{xs} in
match xs with
...

||

pop ℓ_s;
pop ℓ_s;
pop ℓ_s;
a_big_alloc ()

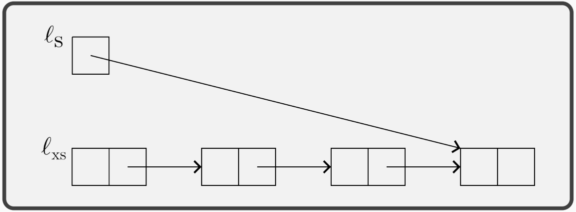


A Problematic Interleaving for pop

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

||

```
pop ℓs;
pop ℓs;
pop ℓs;
a_big_alloc ()
```



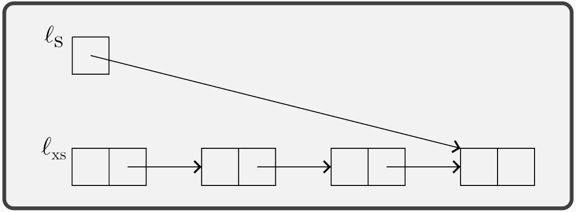
- The sleeping thread maintains reachable the popped-off cells.
- If the GC runs at this point, it cannot free these cells, and the allocation will fail.

A Problematic Interleaving for pop

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

||

```
pop ℓs;
pop ℓs;
pop ℓs;
a_big_alloc ()
```



- The sleeping thread maintains reachable the popped-off cells.
- If the GC runs at this point, it cannot free these cells, and the allocation will fail.

Motivating Question

Can new language constructs be devised to prevent these interleavings?

Protected Sections

```
let xs = ℓ_xs in                                           ||  pop ℓ_s;
match xs with                                              ||  pop ℓ_s;
| nil -> assert false                                      ||  pop ℓ_s;
| y::ys -> if CAS s xs ys then y else pop s                ||  a_big_alloc ()
```

- The location ℓ_{xs} is a root for small number of instructions.
- The allocation should wait for these instructions to complete, until ℓ_{xs} is released.

Protected Sections

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

```
|| pop ℓs;
   pop ℓs;
   pop ℓs;
   a_big_alloc ()
```

- The location ℓ_{xs} is a root for small number of instructions.
- The allocation should wait for these instructions to complete, until ℓ_{xs} is released.

```
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)
```

$\rightsquigarrow$ The location ℓ_{xs} is a temporary root: it is a root only inside a protected section.

Protected Sections, in Details

- Each thread is either outside or inside a protected section.
- The GC runs only when every thread is outside protected sections.

Protected Sections, in Details

- Each thread is either outside or inside a protected section.
- The GC runs only when every thread is outside protected sections.

Protected sections forbid the problematic interleaving:

- An allocation that would exceed the bound S waits for the GC to run.
- The GC waits for protected sections to end, releasing their temporary roots.

Protected Sections, in Details

- Each thread is either outside or inside a protected section.
- The GC runs only when every thread is outside protected sections.

Protected sections forbid the problematic interleaving:

- An allocation that would exceed the bound S waits for the GC to run.
- The GC waits for protected sections to end, releasing their temporary roots.

Constraints inside protected sections:

- No allocation.
- No divergence.
- No nesting.

Protected Sections, in Details

- Each thread is either outside or inside a protected section.
- The GC runs only when every thread is outside protected sections.

Protected sections forbid the problematic interleaving:

- An allocation that would exceed the bound S waits for the GC to run.
- The GC waits for protected sections to end, releasing their temporary roots.

Constraints inside protected sections:

- No allocation.
- No divergence.
- No nesting.

$$\left\langle \frac{\mathrm{True}}{\forall v\, L.\ \mathrm{stack}\ \ell_s\ (v :: L)} \right\rangle \pi : \mathrm{pop}\ \ell_s \left\langle \frac{\lambda w.\ \ \ulcorner w = v \urcorner}{\mathrm{stack}\ \ell_s\ L\ \ast\ \Diamond 2} \right\rangle$$

Key Idea: Logical Deallocation of Temporary Roots

Key Idea: Logical Deallocation of Temporary Roots

New assertion inside π T forming an escape-hatch to the pointed-by-thread discipline.

The set T is for temporary roots: T must be empty when the protected section ends.

Key Idea: Logical Deallocation of Temporary Roots

New assertion inside π T forming an escape-hatch to the pointed-by-thread discipline.
The set T is for temporary roots: T must be empty when the protected section ends.

Thread π_0	Thread π_1
`enter ();`	`enter ();`
`let xs = !`ℓ_s` in`	`let xs = !`ℓ_s` in`
`...`	`...`
	`CAS s xs ys`
`exit ();`	
`...`	`...`

Key Idea: Logical Deallocation of Temporary Roots

New assertion inside π T forming an escape-hatch to the pointed-by-thread discipline.

The set T is for temporary roots: T must be empty when the protected section ends.

Thread π_0	Thread π_1
`enter ();`	`enter ();`
$\{ \boxed{\ell_s \mapsto \ell_{xs}} * \text{inside}\,\pi_0\,\emptyset \}$	
`let xs = !`ℓ_s `in`	`let xs = !`ℓ_s `in`
$\{ \boxed{\ell_s \mapsto \ell_{xs}} * \text{inside}\,\pi_0\,\{\ell_{xs}\} \}$	
...	...
$\{ \text{inside}\,\pi_0\,\{\ell_{xs}\} \}$	
$\{ \text{inside}\,\pi_0\,\emptyset \}$	`CAS s xs ys`
`exit ();`	
...	...

Key Idea: Logical Deallocation of Temporary Roots

New assertion inside π T forming an escape-hatch to the pointed-by-thread discipline.
The set T is for temporary roots: T must be empty when the protected section ends.

Thread π_0

```
enter ();
```
$\{ \boxed{\ell_s \mapsto \ell_{xs}} * \text{inside}\,\pi_0\,\emptyset \}$
```
let xs = !ℓs in
```
$\{ \boxed{\ell_s \mapsto \ell_{xs}} * \text{inside}\,\pi_0\,\{\ell_{xs}\} \}$

...

$\{ \text{inside}\,\pi_0\,\{\ell_{xs}\} \}$

$\{ \text{inside}\,\pi_0\,\emptyset \}$
```
exit ();
```
...

Thread π_1

```
enter ();
```
$\{ \boxed{\ell_s \mapsto \ell_{xs} * \ell_{xs} \hookleftarrow \{+\ell_s\} * \ell_{xs} \Lleftarrow \emptyset} * \text{inside}\,\pi_1\,\emptyset \}$
```
let xs = !ℓs in
```
$\{ \boxed{\ell_s \mapsto \ell_{xs} * \ell_{xs} \hookleftarrow \{+\ell_s\} * \ell_{xs} \Lleftarrow \emptyset} * \text{inside}\,\pi_1\,\{\ell_{xs}\} \}$

...

$\{ \boxed{\ell_s \mapsto \ell_{xs} * \ell_{xs} \hookleftarrow \{+\ell_s\} * \ell_{xs} \Lleftarrow \emptyset} * \text{inside}\,\pi_1\,\{\ell_{xs}\} \}$
```
CAS s xs ys
```
$\{ \boxed{\ell_s \mapsto \ldots} * \ell_{xs} \hookleftarrow \emptyset * \ell_{xs} \Lleftarrow \emptyset * \text{inside}\,\pi_1\,\{\ell_{xs}\} \}$

$\{ \Diamond 1 * \text{inside}\,\pi_1\,\{\ell_{xs}\} \}$

...

Polling for Liveness

Due to protected sections, a thread may wait forever for the GC.

```
while true do                      ||
  enter (); ...; exit ()           ||     a_big_alloc ()
done                               ||
```

Polling for Liveness

Due to protected sections, a thread may wait forever for the GC.

```
while true do
  enter (); ...; exit ()
done
```

||

```
a_big_alloc ()
```

- New construct: polling points.
- A thread facing a polling point stops its execution until no thread requires the GC.

```
while true do
  enter (); ...; exit ();
  poll ()
done
```

||

```
a_big_alloc ()
```

Polling for Liveness

Due to protected sections, a thread may wait forever for the GC.

```
while true do
  enter (); ...; exit ()        ||    a_big_alloc ()
done
```

- New construct: polling points.
- A thread facing a polling point stops its execution until no thread requires the GC.

```
while true do
  enter (); ...; exit ();
  poll ()                        ||    a_big_alloc ()
done
```

An automatic approach guaranteeing liveness: a polling point in every loop.

Back to Reality

- Protected sections and polling points are inspired by safe points.
- Safe points are used internally by OCaml to implement a stop-the-world GC.
- Issue of safe points: they delimit protected sections and act as polling points.

Back to Reality

- Protected sections and polling points are inspired by safe points.
- Safe points are used internally by OCaml to implement a stop-the-world GC.
- Issue of safe points: they delimit protected sections and act as polling points.

Proposal

Ask the programmer to be explicit about protected sections;
let the compiler insert polling points.

There is More

In the manuscript:

- reasoning about closures
- case studies
- soundness proof
- simplified reasoning when no deallocation is needed
- logical deallocation of cyclic data structures

There is More

In the manuscript:

- reasoning about closures
- case studies
- soundness proof
- simplified reasoning when no deallocation is needed
- logical deallocation of cyclic data structures

Making use of the presented ideas, I participated in other projects:

POPL'24 *DisLog: A Separation Logic for Disentanglement* [Moine, Westrick, and Balzer]
a logic for "disentanglement", a reachability property of parallel programs.

ICFP'24 *Snapshottable Stores* [Allain, Clément, Moine, and Scherer]
verifying a data structure with non-trivial reachability arguments.

Future Work

Connections with related works

- Integration with verified compilers
  CakeML [Gómez-Londoño et al., 2020]
- Safe Memory Reclamation (SMR)
  Space consumption [Jung et al., 2023]
- Foundation for type systems
  AARA [Hoffmann and Jost, 2022]

Practical applications

- Protected sections for OCaml
- Control over polling points position

Theoretical extensions

- More advanced case studies
  Harris's list [Harris, 2001]
- Weak pointers and ephemerons

How to Establish Heap Space Bounds in the Presence of Garbage Collection?

How to Establish Heap Space Bounds in the Presence of Garbage Collection?

IrisFit, the first Separation Logic for verifying
heap space bounds in a high-level concurrent language equipped with a GC

How to Establish Heap Space Bounds in the Presence of Garbage Collection?

IrisFit, the first Separation Logic for verifying
heap space bounds in a high-level concurrent language equipped with a GC

Key ingredients:

- space credits to keep track of available heap space
- pointed-by-heap and pointed-by-thread assertions to prove unreachability
- protected sections to improve heap space bounds of lock-free data structures
- polling points to recover liveness

Reasoning rules, case studies and soundness are mechanized.

How to Establish Heap Space Bounds in the Presence of Garbage Collection?

IrisFit, the first Separation Logic for verifying
heap space bounds in a high-level concurrent language equipped with a GC

Key ingredients:

- space credits to keep track of available heap space
- pointed-by-heap and pointed-by-thread assertions to prove unreachability
- protected sections to improve heap space bounds of lock-free data structures
- polling points to recover liveness

Reasoning rules, case studies and soundness are mechanized.

Thank you for your attention!

Backup Slides

The Bind Problem and its Solution

Trimming is unsound with the standard $\textsc{Let}$:
what if a location $\ell \in (roots(t_2) \setminus roots(t_1))$?

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

✖ One could leak $\ell \Lleftarrow \{\pi\}$ in Φ.

- Unveiling fractions: $\ell \Lleftarrow_{(p_1+p_2)} (\Pi_1 \cup \Pi_2) \;\equiv\; \ell \Lleftarrow_{p_1} \Pi_1 \;*\; \ell \Lleftarrow_{p_2} \Pi_2$
- Only logical deallocation requires full fraction 1.
- The $\textsc{Let}$ rule withhold a fraction of the pointed-by-thread assertion.

$$\frac{roots(t_2) = \{\ell\} \qquad \{\,\Phi\,\}\; t_1 \;\{\,\Psi'\,\} \qquad \forall v.\; \{\,\ell \Lleftarrow_p \{\pi\} \;*\; \Psi'\; v\,\}\; [v/x]t_2 \;\{\,\Psi\,\}}{\{\,\ell \Lleftarrow_p \{\pi\} \;*\; \Phi\,\}\; \mathrm{let}\; x = t_1 \;\mathrm{in}\; t_2 \;\{\,\Psi\,\}}$$

Cycles

We handle cycles following the approach of Madiot and Pottier [2022].

$$\text{True} \mathrel{-\!\!*} \emptyset \,\text{☁}^{0}\, P$$

$$\begin{array}{r} D \,\text{☁}^{n}\, P \\ \ell \mapsto \vec{v} \;*\; \ell \leftarrowtail A \;*\; \ell \Lleftarrow \emptyset \end{array} \mathrel{-\!\!*} (\{\ell\} \cup D) \,\text{☁}^{n+size(\vec{v})}\, P \quad \text{if } A \subseteq P$$

$$D \,\text{☁}^{n}\, D \Rrightarrow \Diamond n \;*\; \big(\mathop{\Huge *}_{\ell \in D} \dagger\ell\big) \qquad \text{if } D \cap roots(t) = \emptyset$$

Closures

Functions with an environment are usually compiled down to closures.

A closure is	a heap allocated block	pointing to the environment's values.
Closure allocation	consumes space credits	and updates pointed-by assertions.

We encode closures as derived constructions using closure conversion:

- closure creation and call are not in the syntax,
- but we provide macros implementing them,
- and provide reasoning rules about these macros!

The True Pointed-by-Heap Assertion

$$\ell \mapsfrom_q A$$

(signed) multiset

$\in [0, 1]$

- $\ell \mapsfrom_1 A$ asserts that A is an over-approximation of the reachable predecessors of ℓ.
- $\ell \mapsfrom_1 \emptyset$ asserts that ℓ is unreachable from the heap.

$$\ell \mapsfrom_1 \{+\ell_1; +\ell_2\} \quad \mathrel{-\!\!*} \quad \ell \mapsfrom_{\frac{1}{2}} \{+\ell_1\} * \ell \mapsfrom_{\frac{1}{2}} \{+\ell_2\}$$

$$\ell \mapsfrom_{\frac{1}{2}} \{+\ell_1\} * \ell \mapsfrom_0 \{-\ell_1\} \quad \mathrel{-\!\!*} \quad \ell \mapsfrom_{\frac{1}{2}} (\{+\ell_1\} \uplus \{-\ell_1\})$$

Main invariant: if $\ell \mapsfrom_0 A$ then A must contain only negative elements.

References i

Clément Allain, Basile Clément, Alexandre Moine, and Gabriel Scherer. Snapshottable stores. Proc. ACM Program. Lang., 8(ICFP), aug 2024. doi: 10.1145/3674637. URL https://doi.org/10.1145/3674637.

Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. End-to-end verification of stack-space bounds for C programs. In Programming Language Design and Implementation (PLDI), pages 270–281, June 2014. URL http://flint.cs.yale.edu/flint/publications/veristack.pdf.

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.

Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In Jennifer Welch, editor, Distributed Computing, pages 300–314, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-45414-4.

Jan Hoffmann and Steffen Jost. Two decades of automatic amortized resource analysis. Mathematical Structures in Computer Science, 32(6):729–759, 2022. doi: 10.1017/S0960129521000487.

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.

Jaehwang Jung, Janggun Lee, Jaemin Choi, Jaewoo Kim, Sunho Park, and Jeehoon Kang. Modular verification of safe memory reclamation in concurrent separation logic. Proc. ACM Program. Lang., 7(OOPSLA2), oct 2023. doi: 10.1145/3622827. URL https://doi.org/10.1145/3622827.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. Journal of Functional Programming, 28:e20, 2018. URL https://people.mpi-sws.org/~dreyer/papers/iris-ground-up/paper.pdf.

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.

References v

Alexandre Moine, Arthur Charguéraud, and François Pottier. Will it fit? Verifying heap space bounds of concurrent programs under garbage collection with separation logic. Submitted, September 2024a. URL http://cambium.inria.fr/~amoine/publications/moine-chargueraud-pottier-24.pdf.

Alexandre Moine, Sam Westrick, and Stephanie Balzer. Dislog: A separation logic for disentanglement. Proc. ACM Program. Lang., 8(POPL), jan 2024b. doi: 10.1145/3632853. URL https://doi.org/10.1145/3632853.

Peter W. O'Hearn. Resources, concurrency and local reasoning. Theoretical Computer Science, 375(1–3):271–307, May 2007. URL http://www.cs.ucl.ac.uk/staff/p.ohearn/papers/concurrency.pdf.

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Computer Science Logic, volume 2142 of Lecture Notes in Computer Science, pages 1–19. Springer, September 2001. URL http://www0.cs.ucl.ac.uk/staff/p.ohearn/papers/localreasoning.pdf.

John C. Reynolds. Separation logic: A logic for shared mutable data structures. In Logic in Computer Science (LICS), pages 55–74, 2002. URL http://www.cs.cmu.edu/~jcr/seplogic.pdf.