François Pottier
Jean-Marie Madiot

A Separation Logic for Heap Space under GC
We wish to verify a program’s heap space usage,

• using separation logic,
• by viewing heap space as a resource...
Idea 1. Following Hofmann (1999), let $\diamond 1$ represent one space credit. Allocation consumes credits; deallocation produces credits.

\[
\{ \diamond \text{size}(b) \} \quad x := \text{alloc}(b) \quad \{ x \rightarrow b \} \\
\{ x \rightarrow b \} \quad \text{free}(x) \quad \{ \diamond \text{size}(b) \}
\]

A function’s space requirement is visible in its specification.
In the presence of GC, what Happens?

In the presence of GC,

• deallocation becomes *implicit*,
• so we lose the ability to recover space credits while reasoning.
Idea 2. Switch to a *logical deallocation* operation:

\[ x \mapsto b \implies \Diamond \text{size}(b) \]

A ghost update $\Rightarrow$ *consumes* an assertion and *produces* an assertion.

This marries

- *manual reasoning* about memory at verification time
- *automatic management* of memory at runtime.
Is logical deallocation sound?

\[ x \leadsto b \implies \Diamond \text{size}(b) \]

It does have a few good properties: *no double-free, no use-after-free.*

Because \( x \leadsto b \) is consumed,

- a block cannot be logically deallocated twice;
- a block cannot be accessed after it has been logically deallocated.
Unfortunately, logical deallocation in this form is *not sound*.

Introducing logical deallocation creates a distinction between

- the *logical heap* that the programmer keeps in mind,
- the *physical heap* that exists at runtime.
The following situation is problematic.

The programmer has logically deallocated a block and obtained 3, but this block is reachable and cannot be reclaimed by the GC.

We have 3 space credits but no free space in the physical heap!
To avoid this problem, we must restrict logical deallocation:

- A reachable block must not be deallocated.

In the contrapositive,

- A block should be logically deallocatable only if it is unreachable,
- so the GC can reclaim this block,
- so the logical and physical heaps remain synchronized.
The logical and physical heaps *coincide on their reachable fragments*. 

```
logical heap

physical heap
```
How do we restrict logical deallocation?

- We want to disallow deallocating a *reachable* block,
- but Separation Logic lets us reason about *ownership*.
- Reachability is a *nonlocal* property.
Idea 3. Following Kassios and Kritikos (2013),

- we *keep track of the predecessors* of every block.
- If a block has no predecessor, *then* it is unreachable,
- therefore it can be logically deallocated.
In addition to *points-to*, we use *pointed-by* assertions:

\[
\begin{align*}
&\text{points-to} \\
&\quad l \rightarrow b \\
&\quad \ast \\
\end{align*}
\]

\[
\begin{align*}
&\text{pointed-by} \\
&\quad l \leftarrow l \\
\end{align*}
\]

permission to read/route the block at \( l \)

permission to add/remove pointers to \( l \)

permission to deallocate if \( L = \emptyset \)
We get a sound logical deallocation axiom:

\[ x \mapsto b \ast x \leftarrow \emptyset \implies \Diamond \text{size}(b) \]

This axiom deallocates one block.

There is also a *bulk logical deallocation* axiom.
Dealing with Roots

We want the pointers *from the stack(s) to the heap* to be explicit,

- so the operational semantics views them as GC *roots*,
- so our predecessor-tracking logic keeps track of them.

**Idea 4.** Use a low-level calculus where *stack cells* are explicit.
1 A Glimpse of SpaceLang
2 A Glimpse of the Reasoning Rules
3 Specification of a Stack
4 Conclusion
SpaceLang is imperative. An *instruction* *i* does not return a value.

- **skip**
- **i; i**
- **if *q* then *i* else *i**
- ***q(\overline{o})***
- ***q = v***
- ***q = *q***

- **no-op**
- **sequencing**
- **conditional**
- **procedure call**
- **constant load**
- **move**

The operands of every instruction are stack cells *q*.

There is *no heap deallocation* instruction.
A small-step operational semantics, with a few unique features:

- *Garbage collection* takes place before every reduction step.
- The GC *roots* are the stack cells.
- Heap allocation *fails* if the heap size exceeds a fixed limit $S$. 
Roadmap

1. A Glimpse of SpaceLang
2. A Glimpse of the Reasoning Rules
3. Specification of a Stack
4. Conclusion
Heap allocation consumes space credits.

\[
\text{ALLOC} \begin{cases} 
\diamond \text{size}()^n \\
\ell \mapsto \langle v \rangle \\
v \leftarrow_q L 
\end{cases}
\]
\[\ast s = \text{alloc } n\]
\[
\exists \ell. 
\begin{cases} 
\ell \mapsto ()^n \\
\ell \leftarrow \{ s \} \\
s \mapsto \langle \ell \rangle \\
v \leftarrow_q L \setminus \{ s \}
\end{cases}
\]

Points-to and pointed-by assertions for the new location appear.

One pointer to the value \( v \) is deleted.
Reasoning about a heap store involves some administration...

One pointer to $v$ is deleted; one pointer to $v'$ is created.
Logical deallocation of a block is a *ghost operation*:

- $l \rightarrow_1 \overline{v} \ast l \leftarrow_1 L \ast \text{dom}(L) \subseteq \{l\} \Rightarrow l \parallel \{l\} \ast \text{size}(\overline{v})$
- Knowledge of all antecedents
- Ownership of the block
- Location now dead
- No antecedent (but self)
Theorem (Soundness)

If \{\diamond S\} i \{\text{True}\} holds, then, executing i in an empty store cannot lead to a situation where a thread is stuck.

If, under a precondition of $S$ space credits, the code can be verified, then its live heap space cannot exceed $S$.

This holds regardless of the value of $S$ (the heap size limit). Furthermore, the reasoning rules are independent of $S$.

The rules allow compositional reasoning about space.
1. A Glimpse of SpaceLang
2. A Glimpse of the Reasoning Rules
3. Specification of a Stack
4. Conclusion
The user may define *custom* (simplified) predecessor tracking disciplines. For example, sometimes, *counting* predecessors is enough.

\[ v \leftarrow n \triangleq \exists L. (v \leftarrow_1 L \star |L| = n) \]

Edge addition and deletion increment and decrement \( n \).
Creating a stack *consumes 4 space credits.*

\[
\begin{align*}
\{ f \mapsto \langle \text{create} \rangle \} \quad & \star f(\text{stack}) \\
\{ \text{stack} \mapsto \langle () \rangle \} \quad & \exists \ell. \quad \text{stack} \mapsto \langle \ell \rangle \\
\diamond 4 \quad & \text{isStack } \ell{} [\] \star \ell \leftarrow 1
\end{align*}
\]

We get unique ownership of the stack and *we have the sole pointer* to it.
Pushing consumes 4 space credits.

\[
\begin{cases}
  f \mapsto \langle \text{push} \rangle \\
  stack \mapsto \langle \ell \rangle \\
  elem \mapsto \langle v \rangle \\
  \diamond 4 \ast \text{isStack} \ell \ vs \\
  v \leftarrow n
\end{cases}
\]

\[
\begin{cases}
  f \mapsto \langle \text{push} \rangle \\
  stack \mapsto \langle \ell \rangle \\
  elem \mapsto \langle v \rangle \\
  \text{isStack} \ell (v :: vs) \\
  v \leftarrow n + 1
\end{cases}
\]

The value \( v \) receives one more antecedent.
Popping frees up 4 space credits.

\[
\begin{align*}
  f & \mapsto \langle \text{pop} \rangle \\
  \text{stack} & \mapsto \langle \ell \rangle \\
  \text{elem} & \mapsto \langle () \rangle \\
  \text{isStack} \, \ell \, (v :: vs) & \\
  v & \leftarrow n
\end{align*}
\]

\[
\begin{align*}
  f & \mapsto \langle \text{pop} \rangle \\
  \text{stack} & \mapsto \langle \ell \rangle \\
  \text{elem} & \mapsto \langle v \rangle \\
  \diamond 4 & \ast \text{isStack} \, \ell \, vs \\
  v & \leftarrow n
\end{align*}
\]

The number of predecessors of \( v \) is unchanged, because the out-parameter \( \text{elem} \) receives a pointer to it.
Logically deallocation of the stack, a *ghost operation*, is part of the API. It requires proving that the stack has *zero predecessors*.

\[
\begin{align*}
\text{isStack } \ell \text{ vs } \star \ell & \leftarrow 0 \\
\star v & \leftarrow n \\
(\nu, n) \in vns
\end{align*}
\begin{align*}
\Rightarrow l
\begin{align*}
\diamond (4 + 4 \times |vs|) \\
\star v & \leftarrow n - (v \$ vs) \\
(\nu, n) \in vns
\end{align*}
\end{align*}
\]

It frees up *a linear number of space credits*. 
Roadmap

1. A Glimpse of SpaceLang
2. A Glimpse of the Reasoning Rules
3. Specification of a Stack
4. Conclusion
A sound logic to reason about heap space usage in the presence of GC.

Our main insights:

- Allocation consumes space credits $\diamond n$.
- *Logical deallocation*, a ghost operation, produces space credits.
- Logical dellocation requires *predecessor tracking*, which we perform via *pointed-by assertions* $v \leftarrow L$. 

Currently, predecessor tracking requires *heavy bookkeeping*. We are investigating

- a more flexible *deferred* logical edge deletion mechanism;
- coarse-grained predecessor tracking based on *islands*;
- *simpler / more automated* tracking of *roots*;
- reasoning directly about call-by-value *\( \lambda \)-calculus*. 
Roadmap

5 Syntax, Semantics of SpaceLang

6 Reasoning Rules of SL

7 Specification of List Copy
Values, Blocks, Stores

Memory locations: $\ell, c, r, s \in \mathcal{L}$.

Values include constants, memory locations, and closed procedures:

$$v ::= () \mid k \mid \ell \mid \lambda \vec{x}.i$$

Memory blocks include heap tuples, stack cells, and deallocated blocks:

$$b ::= \vec{v} \mid \langle v \rangle \mid \emptyset$$

A store maps locations to blocks, encompassing the heap and stack(s). The size of a block:

$$\text{size}(\vec{v}) = 1 + |\vec{v}| \quad \text{size}(\langle v \rangle) = \text{size}(\emptyset) = 0$$

The size of the store is the sum of the sizes of all blocks.
A reference is a variable or a (stack) location and denotes a stack cell.

$$\varnothing ::= x \mid c$$

SpaceLang uses call-by-reference.

A variable denotes a closed reference, *not* a closed value as is usual.

The operational semantics involves substitutions $[c/x]$.

This preserves the property that *the code never points to the heap*.

The roots of the garbage collection process are *the stack cells*.
SpaceLang is imperative. An *instruction* \( i \) does not return a value.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Example</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>no-op</td>
<td>( _ )</td>
<td></td>
</tr>
<tr>
<td>( i; i )</td>
<td>sequencing</td>
<td>( i )</td>
<td></td>
</tr>
<tr>
<td>if ( _ ) then ( i ) else ( i )</td>
<td>conditional</td>
<td>( _ ) ( i ) ( _ )</td>
<td></td>
</tr>
<tr>
<td>( _ ) ( _ )</td>
<td>procedure call</td>
<td>( _ ) ( _ )</td>
<td></td>
</tr>
<tr>
<td>( _ ) ( _ )</td>
<td>constant load</td>
<td>( _ ) ( _ )</td>
<td></td>
</tr>
<tr>
<td>( _ ) ( _ )</td>
<td>move</td>
<td>( _ ) ( _ )</td>
<td></td>
</tr>
</tbody>
</table>

The operands of every instruction are stack cells (\( \_ \)).

There is no deallocation instruction for heap blocks.
Operational Semantics: Heap Allocation

We fix a *maximum heap size* $S$.

Heap allocation *fails* if the heap size exceeds $S$.

\[
\text{STEPALLOC} \quad 
\begin{align*}
\sigma' &= [\ell += (\_)^n]\sigma \\
\text{size}(\sigma') &\leq S \\
\sigma'' &= \langle s := \ell \rangle \sigma' \\
\ast s &= \text{alloc } n / \sigma \rightarrow \text{skip } / \sigma''
\end{align*}
\]

$S$ is a parameter of the operational semantics,

but the reasoning rules of SL$\diamond$ are independent of $S$. 
The dynamic semantics of stack allocation is in **three steps**:

**StepAllocaEntry**

\[
\sigma' = [c \leftarrow \langle()\rangle] \sigma
\]

\[
\text{alloca } x \text{ in } i / \sigma \rightarrow \text{alloca } c \text{ in } [c/x]i / \sigma'
\]

**StepAllocaExit**

\[
\sigma(c) = \langle v \rangle \quad \sigma' = [c := \emptyset] \sigma
\]

\[
\text{alloca } c \text{ in skip} / \sigma \rightarrow \text{skip} / \sigma'
\]

Evaluation contexts: \( K ::= [] | K; i | \text{alloca } c \text{ in } K \).
To complete the definition of the operational semantics,

- allow *garbage collection* before every reduction step.

  \( \sigma \trianglerighteq \sigma' \) holds if

  - the stores \( \sigma \) and \( \sigma' \) have the same domain;
  - for every \( \ell \) in this domain,
    either \( \sigma'(\ell) = \sigma(\ell) \), or \( \ell \) is unreachable in \( \sigma \) and \( \sigma'(\ell) = \bowtie \).

- allow *thread interleavings* (comes for free with Iris).
Complete Operational Semantics

\[
\text{StepSeqSkip} \quad \text{skip; } i / \sigma \rightarrow i / \sigma
\]

\[
\text{StepIf} \quad \sigma(r) = \langle k \rangle \quad \text{if } *r \text{ then } i_1 \text{ else } i_2 / \sigma \rightarrow k \neq 0 ? i_1 : i_2 / \sigma
\]

\[
\text{StepCall} \quad \sigma(r) = \langle \lambda \vec{x}.i \rangle \quad |\vec{x}| = |\vec{s}|
\quad *r(\vec{s}) / \sigma \rightarrow [\vec{s}/\vec{x}]i / \sigma
\]

\[
\text{StepConst} \quad \sigma' = \langle s := v \rangle \sigma \quad \text{pointers}(v) = \emptyset
\quad *s = v / \sigma \rightarrow \text{skip} / \sigma'
\]

\[
\text{StepMove} \quad \sigma(r) = \langle v \rangle \quad \sigma' = \langle s := v \rangle \sigma
\quad *s = *r / \sigma \rightarrow \text{skip} / \sigma'
\]

\[
\text{StepAlloc} \quad \sigma' = [\ell := (\ell^0)] \sigma \quad \text{size}(\sigma') \leq S
\quad \sigma'' = \langle s := \ell \rangle \sigma'
\quad *s = \text{alloc } n / \sigma \rightarrow \text{skip} / \sigma''
\]

\[
\text{StepLoad} \quad \sigma(r) = \langle \ell \rangle \quad \sigma(\ell) = \vec{v} \quad 0 \leq o < |\vec{v}|
\quad \vec{v}(o) = v \quad \sigma' = \langle s := v \rangle \sigma
\quad *s = [*r + o] / \sigma \rightarrow \text{skip} / \sigma'
\]

\[
\text{StepStore} \quad \sigma(r) = \langle v \rangle \quad \sigma(s) = \langle \ell \rangle \quad \sigma(\ell) = \vec{v}
\quad 0 \leq o < |\vec{v}| \quad \sigma' = [\ell := [o := v] \vec{v}] \sigma
\quad [*s + o] = *r / \sigma \rightarrow \text{skip} / \sigma'
\]

\[
\text{StepLocEq} \quad \sigma(r_1) = \langle \ell_1 \rangle \quad \sigma(r_2) = \langle \ell_2 \rangle
\quad \sigma' = \langle s := (\ell_1 = \ell_2 \iff 1 : 0) \rangle \sigma
\quad *s = (*r_1 == *r_2) / \sigma \rightarrow \text{skip} / \sigma'
\]

\[
\text{StepAllocaEntry} \quad \sigma' = [c := \langle () \rangle] \sigma
\quad \text{alloc } x \text{ in } i / \sigma \rightarrow \text{alloc } c \text{ in } [c/x]i / \sigma'
\]

\[
\text{StepAllocaExit} \quad \sigma(c) = \langle v \rangle \quad \sigma' = [c := \emptyset] \sigma
\quad \text{alloc } c \text{ in skip} / \sigma \rightarrow \text{skip} / \sigma'
\]

\[
\text{StepFork} \quad \sigma(r) = \langle v \rangle \quad \sigma' = [r := ()][c := \langle v \rangle] \sigma
\quad \text{fork } *r \text{ as } x \text{ in } i / \sigma \rightarrow \text{skip} / \sigma'
\quad \text{spawning alloc } c \text{ in } [c/x]i
\]

\[
\text{StepContext} \quad i / \sigma \rightarrow i' / \sigma'
\quad K[i] / \sigma \rightarrow K[i'] / \sigma'
\quad \text{spawning } i
\]
Roadmap

5 Syntax, Semantics of SpaceLang

6 Reasoning Rules of SL

7 Specification of List Copy
Heap allocation *consumes space credits.*

\[
\textbf{Alloc} \left\{ \begin{array}{l}
\diamond \text{size}((()^n)} \\
 s \mapsto \langle v \rangle \\
v \leftarrow_q L
\end{array} \right\} \star s = \text{alloc } n
\]

Points-to and pointed-by assertions for the new location appear.

One pointer to the value \(v\) is \textit{deleted}. (This aspect is optional.)
Writing a heap cell is simple... but involves some administration.

One pointer to \( \nu \) is deleted; one pointer to \( \nu' \) is \textit{created}. 
A points-to assertion for the new stack cell exists throughout its lifetime.

No pointed-by assertion is provided. (A design choice.)

- No pointers (from the heap or stack) to the stack.
Logical deallocation of a block is a *ghost operation*:

\[ l \rightarrow_1 \vec{v} \star l \leftarrow_1 L \star \text{dom}(L) \subseteq \{l\} \ \Rightarrow \ hd \{l\} \star \text{size}(\vec{v}) \]

- Knowledge of all antecedents
- Location now dead
- Ownership of the block
- No antecedent (but self)
Deferred Predecessor Deletion

Deletion of deallocated predecessors can be *deferred*:

\[ v \leftarrow_{q} L \ast \frac{\perp D}{\frac{\text{dead locations}}{\text{antecedents}}} \Rightarrow v \leftarrow_{q} L' \]

\[ \text{if } \text{dom}(L \setminus L') \subseteq D \]

A key rule: if \( L' \) is empty, then \( v \) becomes eligible for deallocation.
A group that is *closed under predecessors* can be deallocated at once:

$$D \overset{m}{\equiv} D \implies I \vdash D \ast \Diamond N$$

- size of cloud
- footprint is $D$
- all antecedents are in $D$
- $D$ now dead
- logically reclaimed space!

The rules for constructing a “cloud” (omitted) are straightforward.
Points-to and pointed-by assertions can be *split* and *joined*.

\[ l \mapsto_{q_1+q_2} b \equiv l \mapsto_{q_1} b \ast l \mapsto_{q_2} b \]

\[ \nu \leftarrow_{q_1+q_2} l_1 \cup l_2 \equiv \nu \leftarrow_{q_1} l_1 \ast \nu \leftarrow_{q_2} l_2 \]

\[ \nu \leftarrow_q l \ast \nu \leftarrow_q l' \quad \text{if } l \subseteq l' \]

Pointed-by assertions are *covariant*.

Points-to and pointed-by assertions can be *confronted*.
Space credits can be *split* and *joined*.

\[
\text{True} \implies_{H} \blacklozenge 0 \\
\blacklozenge (m_1 + m_2) \implies_{H} \blacklozenge m_1 \ast \blacklozenge m_2
\]
5 Syntax, Semantics of SpaceLang

6 Reasoning Rules of SL

7 Specification of List Copy
Each cell owns the next cell and possesses *the sole pointer* to it.

\[
\text{isList } \ell \; [\,] \; \overset{\triangle}{=} \; \ell \mapsto [0] \\
\text{isList } \ell \; (\; v :: \; vs \; ) \; \overset{\triangle}{=} \; \exists \ell'. \; \ell \mapsto [1; \; v; \; \ell'] \star \ell' \leftarrow 1 \star \text{isList } \ell' \; vs
\]

Let’s now have a look at *list copy* and its spec. (Fasten seatbelts!)
List Copy in SpaceLang

\[ \text{copy} \triangleq \lambda(self, dst, src). \]

\[
\begin{align*}
\text{alloca tag in } & *tag = [*src + 0]; \\
\text{if } & *tag \text{ then} \\
& \text{alloca head in } *head = [*src + 1]; \\
& \text{alloca tail in } *tail = [*src + 2]; \\
& *src = (); \\
& \text{alloca } dst' \text{ in } *self(self, dst', tail); \\
& *dst = \text{alloc 3;} \\
& [*dst + 0] = *tag; \\
& [*dst + 1] = *head; \\
& [*dst + 2] = *dst'
\end{align*}
\]

\[
\begin{align*}
\text{else} \\
& *src = (); \\
& *dst = \text{alloc 1;} \\
& [*dst + 0] = *tag
\end{align*}
\]

– read the list’s tag
– if this is a cons cell, then
– read the list’s head
– read the list’s tail
– clobber this root
– copy the list’s tail
– allocate a new cons cell
– and initialize it

– this must be a nil cell
– clobber this root
– allocate a new nil cell
– and initialize it
The case $m = 1$, where we have the sole pointer to the list, is special.

$$\begin{align*}
\{ & f \mapsto \langle \text{copy} \rangle \star dst \mapsto \langle () \rangle \star src \mapsto \langle l \rangle \\
& \text{isList } l \ vs \star \ l \leftarrow m \\
& m = 1 ? \emptyset 0 : \emptyset (2 + 4 \times |vs|) \\
& \forall v \in vs. \exists n. (v, n) \in vns \\
& \star_{(v,n) \in vns} v \leftarrow n
\} \equiv \text{need no space or linear space}
\end{align*}$$

$$\star f(f, dst, src)$$

$$\begin{align*}
\exists l'. \left\{ & f \mapsto \langle \text{copy} \rangle \star dst \mapsto \langle l' \rangle \star src \mapsto \langle () \rangle \\
& m = 1 ? \text{True : (isList } l \ vs \star \ l \leftarrow m - 1) \\
& \text{isList } l' \ vs \star \ l' \leftarrow 1 \\
& \star_{(v,n) \in vns} v \leftarrow n + (m = 1 ? 0 : v \ $ vs)
\right\} \equiv \text{orig. list is deallocated or preserved}
\end{align*}$$

Each element receives zero new antecedent or a number of new antecedents.