François Pottier
Jean-Marie Madiot

A Separation Logic for Heap Space under GC
Reasoning about Heap Space

How can we establish \textit{formal (verified)} bounds on a program’s \textit{heap space} usage?

We wish to

- work in the setting of a \textit{program logic},
- view heap space as a \textit{resource}.
Following Hofmann (1999, 2000), let $\Diamond 1$ represent one space credit. Allocation consumes credits; deallocation produces credits.

\[
\begin{align*}
\{ \Diamond \text{size}(b) \} & \quad x := \text{alloc}(b) \quad \{ x \mapsto b \} \\
\{ x \mapsto b \} & \quad \text{free}(x) \quad \{ \Diamond \text{size}(b) \}
\end{align*}
\]

A function’s space requirement is visible in its specification.

End of talk...?
In the presence of GC, what Happens?

Garbage collection can offer superior *simplicity, safety, performance.*

In the presence of GC,

- deallocation becomes *implicit,*
- so we lose the ability to recover space credits while reasoning.
A Ghost Deallocation Operation?

It is tempting to switch to a *logical deallocation* operation:

\[ x \mapsto b \iff \diamond \text{size}(b) \]

This would marry

- *manual reasoning* about memory at verification time
- *automatic management* of memory at runtime.
At least two questions spring to mind:

Is this approach \textit{practical}? Is it \textit{sound}?
A pitfall would be to get *the worst of both worlds*:

- mental burden of manual reasoning about memory deallocation,
- performance issues sometimes caused by GC.

Yet we can strive to get the *best* of each:

- simplicity and possibly superior performance afforded by GC,
- reasoning at a suitable level of abstraction: e.g., via *bulk logical deallocation*. 
Soundness?

Is logical deallocation sound?

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

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

- 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 \textit{not sound}. 

Introducing logical deallocation creates a distinction between

\begin{itemize}
  \item the \textit{logical heap} that the programmer keeps in mind,
  \item the \textit{physical heap} that exists at runtime.
\end{itemize}
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!
Restricting Logical Deallocation

To avoid this problem, we want to restrict logical deallocation.

- A block should be logically deallocated only if it is unreachable,
- which guarantees that the GC can reclaim this block,
- so the logical and physical heaps remain synchronized.
A Global Invariant

The logical and physical heaps *coincide on their reachable fragments*.

So, $\diamondsuit k$ implies $k$ free words *exist* in the logical heap
implies $k$ free words *can be created* in the physical heap.
The outstanding problem is, how do we restrict logical deallocation?

- We want to disallow deallocating a reachable block,
- but Separation Logic lets us reason about ownership.
- Ownership and reachability are unrelated!
- Furthermore, reachability is a nonlocal property.

Not requiring reachability reasoning is a strength of traditional SL.
A Solution: Predecessor Tracking

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:

- *points-to*: 
  - $l \rightarrow b$  
  - Permission to read/route the block at $l$

- *pointed-by*: 
  - $l \leftarrow L$  
  - Permission to add/remove pointers to $l$

- Permission to deallocate if $L = \emptyset$
We get a sound logical deallocation axiom, for a single block:

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

This leads to a calculus where *stack cells* are explicit
and *a variable denotes an address* on the stack.
Roadmap

1. Syntax, Semantics of SpaceLang
2. Reasoning Rules of SL
3. Ghost Reference Counting
4. Examples of Specifications
5. Conclusion
Values, Blocks, Stores

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

Values include constants, memory locations, and \textit{closed procedures}:

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

Memory blocks include \textit{heap tuples}, \textit{stack cells}, and deallocated blocks:

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

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

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

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*.

$$\rho ::= x \mid c$$

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.

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

### Instruction Types

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

### Stack Operations

- **$\varrho = \text{alloc } n$**
- **$\varrho = [*\varrho + o]$**
- **$[\varrho + o] = \varrho$**
- **$\varrho = (\varrho == \varrho)$**
- **alloca $x$ in $i$**
- **alloca $c$ in $i$**
- **fork $\varrho$ as $x$ in $i$**

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

There is no deallocation instruction for heap blocks.
We fix a *maximum heap size* $S$.

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

$$\text{StepAlloc} \quad \sigma' = [\ell + = ()^n] \sigma$$

$$\text{size}(\sigma') \leq S \quad \sigma'' = \langle s := \ell \rangle \sigma'$$

$$*s = \text{alloc } n / \sigma \longrightarrow \text{skip } / \sigma''$$

$S$ 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 \textit{three steps}:

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

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

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

- allow *garbage collection* before every reduction step.

\[ \sigma \triangleright \sigma' \text{ holds if} \]

- 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 \( \sigma' (\ell) = \# \).

- allow *thread interleavings* (comes for free with Iris).
Roadmap

1 Syntax, Semantics of SpaceLang
2 Reasoning Rules of SL
3 Ghost Reference Counting
4 Examples of Specifications
5 Conclusion
Heap allocation *consumes space credits.*

\[
\text{Alloc} \left\{ \begin{array}{l}
\Diamond \text{size}(\cdot)^n \\
 s \mapsto \langle v \rangle \\
 v \leftarrow_q L
\end{array} \right. \quad \ast s = \text{alloc } n
\]

\[
\exists \ell. \left\{ \begin{array}{l}
\ell \mapsto (\cdot)^n \\
\ell \leftarrow \{ s \} \\
 s \mapsto \langle \ell \rangle \\
 v \leftarrow_q L \setminus \{ s \}
\end{array} \right.
\]

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

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

\[
\begin{array}{c}
\text{Heap Store} \\
\text{Writing a heap cell is simple... but involves some administration.}
\end{array}
\]

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

\[
\text{ALLOC}\ \{\Phi \star c \mapsto \langle () \rangle \} \ [c/x] i \ \{c \mapsto \langle () \rangle \star \Psi\} \\
\{\Phi\} \text{alloca } x \text{ in } i \ \{\Psi\}
\]

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*:

\[
\begin{align*}
\ell \rightarrow_1 \tilde{\nu} & \quad \star \quad \ell \leftarrow_1 L & \quad \star \quad \text{knowledge of all antecedents} \\
\text{ownership of the block} & \quad \text{no antecedent (but self)} & \\
\implies \quad \text{location now dead} & \\
\hfill \quad \text{credit!} & \\
\end{align*}
\]
Deletion of deallocated predecessors can be deferred:

\[ v \leftarrow_{q} L \ast \mathbb{H} D \Rightarrow \frac{\text{removed from antecedents}}{\text{dead locations}} \text{ if } \text{dom}(L \setminus L') \subseteq D \]

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:

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

\[
\begin{align*}
    l \rightarrow_{q_1+q_2} b & \equiv l \rightarrow_{q_1} b \ast l \rightarrow_{q_2} b \\
    \nu \leftarrow_{q_1+q_2} l_1 \uplus l_2 & \equiv \nu \leftarrow_{q_1} l_1 \ast \nu \leftarrow_{q_2} l_2 \\
    \nu \leftarrow_{q} l & \rightarrow* \nu \leftarrow_{q} l' & \text{if } l \subseteq l' \\
    l \rightarrow_{q} b \ast l' \leftarrow_{1} l & \equiv l \rightarrow_{q} b \ast l' \leftarrow_{1} l \ast \left( l' \text{ pointer } (b) \leq l \text{ pointer } l' \right)
\end{align*}
\]

Pointed-by assertions are \textit{covariant}.

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

\[
\begin{align*}
\text{True} & \implies_{H} \Diamond 0 \\
\Diamond (m_1 + m_2) & \implies_{H} \Diamond m_1 * \Diamond m_2
\end{align*}
\]
**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 the code is verified under \( S \) space credits, then its heap space usage cannot exceed \( S \).

This guarantee holds for every \( S \).

The reasoning rules are independent of \( S \).

The rules allow compositional reasoning about space.
Roadmap

1. Syntax, Semantics of SpaceLang
2. Reasoning Rules of SL
3. Ghost Reference Counting
4. Examples of Specifications
5. Conclusion
Choice of a Predecessor Tracking Discipline

Keeping track of a *multiset* of predecessors can be heavy. Sometimes

- *counting* predecessors is enough,
- or recording what *regions* the predecessors inhabit is enough.

Can *high-level predecessor tracking disciplines* be defined on top of SLdiamond?
Example: Ghost Reference Counting

The simplified pointed-by assertion $v \leftarrow n$ counts predecessors:

$$v \leftarrow n \triangleq \exists L. (v \leftarrow 1 L \ast |L| = n)$$

Edge addition / deletion increment / decrement $n$. 
Roadmap

1. Syntax, Semantics of SpaceLang
2. Reasoning Rules of SL
3. Ghost Reference Counting
4. Examples of Specifications
   - A Stack
   - List Copy
5. Conclusion
Examples of Specifications
A Stack
List Copy
Creating a stack *consumes 4 space credits*.

\[
\begin{align*}
  f & \mapsto \langle \text{create} \rangle \\
  \text{stack} & \mapsto \langle () \rangle
\end{align*}
\]

\[\diamond 4\]

\[\star f(\text{stack})\]

\[
\begin{align*}
  \exists \ell. & \quad stack \mapsto \langle \ell \rangle \\
  \text{isStack} \ \ell \ [\] & \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*.

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

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

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 antecedents of \( v \) is unchanged, as \( \text{elem} \) points to it.
Logically deallocating the entire stack is a *ghost operation*. It frees up *a linear number of space credits*.

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

The ghost reference counters of the stack elements are decremented.
Examples of Specifications

A Stack

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

\[
\text{isList } \ell \ [\] \triangleq \ell \mapsto [0] \\
\text{isList } \ell \ (v :: vs) \triangleq \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

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

\[
\text{alloc} \ \text{tag} \text{ in } *\text{tag} = [*\text{src} + 0];
\]

\[
\text{if } *\text{tag} \text{ then }
\]

\[
\text{alloc} \ \text{head} \text{ in } *\text{head} = [*\text{src} + 1];
\]

\[
\text{alloc} \ \text{tail} \text{ in } *\text{tail} = [*\text{src} + 2];
\]

\[
*\text{src} = ();
\]

\[
\text{alloc} \ \text{dst}' \text{ in } \text{self}(\text{self}, \text{dst}', \text{tail});
\]

\[
*\text{dst} = \text{alloc} \ 3;
\]

\[
[*\text{dst} + 0] = *\text{tag};
\]

\[
[*\text{dst} + 1] = *\text{head};
\]

\[
[*\text{dst} + 2] = *\text{dst}'
\]

\[
\text{else }
\]

\[
*\text{src} = ();
\]

\[
*\text{dst} = \text{alloc} \ 1;
\]

\[
[*\text{dst} + 0] = *\text{tag}
\]

\[
\begin{align*}
& \quad \text{– read the list’s tag} \\
& \quad \text{– if this is a cons cell, then} \\
& \quad \text{– read the list’s head} \\
& \quad \text{– read the list’s tail} \\
& \quad \text{– clobber this root} \\
& \quad \text{– copy the list’s tail} \\
& \quad \text{– allocate a new cons cell} \\
& \quad \text{– and initialize it} \\
& \quad \text{– this must be a nil cell} \\
& \quad \text{– clobber this root} \\
& \quad \text{– allocate a new nil cell} \\
& \quad \text{– and initialize it}
\end{align*}
\]
The case $m = 1$, where we have the sole pointer to the list, is special.

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

\[
f(f, dst, src) =
\begin{align*}
\{ & f \mapsto \langle \text{copy} \rangle \ \ast \ dst \mapsto \langle \ell' \rangle \ \ast \ src \mapsto \langle () \rangle \\
& \quad m = 1 \ ? \ True : (\text{isList } \ell \ vs \ \ast \ \ell \ \leftarrow m - 1) \\
& \quad \text{isList } \ell' \ vs \ \ast \ \ell' \ \leftarrow 1 \\
& \quad *_{(v, n)\in vns} \ v \ \leftarrow n + (m = 1 \ ? \ 0 : v \ \$_{vs}$)
\} \equiv \text{orig. list is deallocated or preserved}
\end{align*}
\]
Roadmap

1. Syntax, Semantics of SpaceLang
2. Reasoning Rules of SL
3. Ghost Reference Counting
4. Examples of Specifications
5. Conclusion
Summary of Contributions

A sound logic to reason about space usage in the presence of GC.

- Allocation consumes *space credits* ♦ n.
- *Logical deallocation* is a ghost operation.
- Logical dellocation requires *predecessor tracking* ν ← L.
Future Work

Predecessor tracking still requires *too much administration*. We are investigating

- *deferred* edge deletion;
- *automated or simplified* tracking of *roots*;
- predecessor tracking based on *regions*;
- notions of *single-entry-point* regions.

We would also like to adapt SL◊ directly to call-by-value λ-calculus.
A Bit of Controversy about OCaml

During this traversal, *which part of the tree is live?*

```ocaml
type tree = Leaf | Node of tree * tree

let rec walk t =
  match t with
    | Leaf       -> ()
    | Node (t1, t2) -> walk t1; walk t2
```

It could (should?) be the subtrees that have not yet been traversed, because \( t_2 \) remains live while \( \text{walk } t_1 \) is executed...
During this traversal, *which part of the tree is live?*

```ocaml
type tree = Leaf | Node of tree * tree

let rec walk t =
  match t with
  | Leaf            -> ()
  | Node (t1, t2)   -> walk t1; walk t2
```

It could (should?) be *the subtrees that have not yet been traversed*,

because t2 remains live while `walk t1` is executed...
But the OCaml compiler transforms the code roughly as follows:

```ocaml
type tree = Leaf | Node of tree * tree

let rec walk t =
  match t with
  | Leaf -> ()
  | Node (_, _) -> walk t.1 ; walk t.2
```

Thus, \textit{t} remains live while \textit{walk t.1} is executed.

Every \textit{left subtree remains live} until it has been entirely traversed.

Reasoning about space at this level requires a \textit{precise definition} of where each variable is a root.