François Pottier
Jean-Marie Madiot

SL◇

A Separation Logic for Heap Space under GC

Reasoning about Heap Space

How can we establish *formal (verified)* bounds
on a program's *heap space* usage?

We wish to

- work in the setting of a *program logic*,
- view heap space as a *resource*.

Reasoning about Heap Space, without GC

Following Hofmann (1999, 2000), let $\diamond 1$ represent one *space credit*.

Allocation consumes credits; deallocation produces credits.

$$\{\ \diamond\, size(b)\ \}\quad x := alloc(b)\quad \{\ x \mapsto b\ \}$$

$$\{\ x \mapsto b\ \}\quad free(x)\quad \{\ \diamond\, size(b)\ \}$$

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:

$$x \mapsto b \quad \Rrightarrow \quad \Diamond\, size(b)$$

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 *practical?* Is it *sound?*

Practicality?

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*.

Is logical deallocation sound?

$$x \mapsto b \quad \Rrightarrow \quad \Diamond \mathit{size}(b)$$

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.

Soundness?

Unfortunately, logical deallocation in this form is *not sound*.

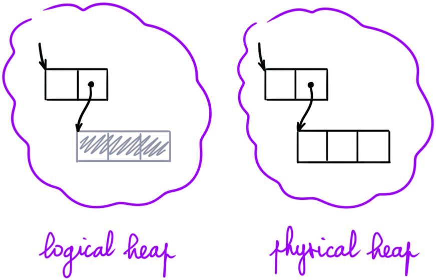
$$x \mapsto \ell \quad \cancel{\Rrightarrow} \quad \Diamond \mathit{size}(\ell)$$

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 $\diamond 3$,



but this block is *reachable* and cannot be reclaimed by the GC.

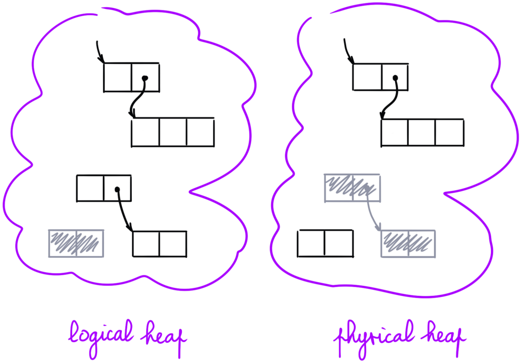
We have 3 space credits but *no free space* in the physical heap!

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*.

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



So, $\diamond k$ implies k free words *exist* in the logical heap
implies k free words *can be created* in the physical heap.

Problem Solved? Not Yet

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:

$$\underbrace{\overset{\text{points-to}}{\ell \mapsto b}}_{\substack{\text{permission to read/write} \\ \text{the block at } \ell}} \;\ast\; \underbrace{\overset{\text{pointed-by}}{\ell \leftarrow\!\!\dashv L}}_{\substack{\text{permission to add/remove} \\ \text{pointers to } \ell}}$$

permission to deallocate
if $L = \emptyset$

We get a sound logical deallocation axiom, for a single block:

$$x \mapsto b \ast x \hookleftarrow \emptyset \Rrightarrow \Diamond \mathit{size}(b)$$

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.

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:

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

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

$$\mathit{size}(\vec{v}) = 1 + |\vec{v}| \qquad \mathit{size}(\langle v \rangle) = \mathit{size}(\lightning) = 0$$

The size of the store is the sum of the sizes of all blocks.

Call-by-Reference and GC Roots

A *reference* is a variable or a (stack) location and denotes a *stack cell*.

$$\varrho ::= x \mid c$$

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	*no-op*	$*\varrho = \text{alloc } n$	*heap allocation*
$i; i$	*sequencing*	$*\varrho = [*\varrho + o]$	*heap load*
if $*\varrho$ then i else i	*conditional*	$[*\varrho + o] = *\varrho$	*heap store*
$*\varrho(\vec{\varrho})$	*procedure call*	$*\varrho = (*\varrho == *\varrho)$	*address comparison*
$*\varrho = v$	*constant load*	alloca x in i	*stack allocation*
$*\varrho = *\varrho$	*move*	alloca c in i	*active stack cell*
		fork $*\varrho$ as x in i	*thread creation*

The operands of every instruction are stack cells (ϱ).

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.

$$\textsc{StepAlloc}\quad \frac{\sigma' = [\ell \mathrel{+}= ()^n]\sigma \qquad \mathit{size}(\sigma') \leq S \qquad \sigma'' = \langle s := \ell \rangle \sigma'}{*s = \mathsf{alloc}\ n \ /\ \sigma \longrightarrow \mathsf{skip}\ /\ \sigma''}$$

S is a parameter of the operational semantics,

but the reasoning rules of SL$\diamond$ are independent of S.

Operational Semantics: Stack Allocation

The dynamic semantics of stack allocation is in *three steps*:

$$\textsc{StepAllocaEntry}\quad \frac{\sigma' = [c \mathrel{+}= \langle()\rangle]\sigma}{\mathsf{alloca}\ x\ \mathsf{in}\ i\ /\ \sigma \longrightarrow \mathsf{alloca}\ c\ \mathsf{in}\ [c/x]i\ /\ \sigma'}$$

$$\textsc{StepAllocaExit}\quad \frac{\sigma(c) = \langle v\rangle \qquad \sigma' = [c := \lightning]\sigma}{\mathsf{alloca}\ c\ \mathsf{in}\ \mathsf{skip}\ /\ \sigma \longrightarrow \mathsf{skip}\ /\ \sigma'}$$

Evaluation contexts: $K ::= [] \mid K; i \mid \mathsf{alloca}\ c\ \mathsf{in}\ K$.

To complete the definition of the operational semantics,

- allow *garbage collection* before every reduction step.

  σ 🗑 σ' holds if

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

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

Heap allocation *consumes space credits*.

new assertions

diamonds consumed

$$\textsc{Alloc}\quad \left\{\begin{array}{l} \Diamond size(()^n) \\ s \mapsto \langle v \rangle \\ v \hookleftarrow_q L \end{array}\right\} \ {*s} = \text{alloc } n \ \left\{ \exists \ell. \begin{array}{l} \ell \mapsto ()^n \\ \ell \hookleftarrow \{s\} \\ s \mapsto \langle \ell \rangle \\ v \hookleftarrow_q L \setminus \{s\} \end{array} \right\}$$

updated stack cell

edge deletion

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.

$$\textsc{Store}\quad \frac{\vec{v}(o) = v}{\left\{\begin{array}{c} s \mapsto \langle \ell \rangle \\ r \mapsto \langle v' \rangle \\ \ell \mapsto_1 \vec{v} \\ v \hookleftarrow_q L \\ v' \hookleftarrow_{q'} L' \end{array}\right\} [*s + o] = *r \left\{\begin{array}{c} s \mapsto \langle \ell \rangle \\ r \mapsto \langle v' \rangle \\ \ell \mapsto_1 [o := v']\vec{v} \\ v \hookleftarrow_q L \setminus \{\ell\} \\ v' \hookleftarrow_{q'} L' \uplus \{\ell\} \end{array}\right\}}$$

operands

updated heap cell

edge deletion

edge addition

One pointer to v is deleted; one pointer to v' is *created*.

A points-to assertion for the new stack cell exists throughout its lifetime.

$$\dfrac{\forall c.\ \{\Phi \star \overbrace{c \mapsto \langle()\rangle}^{\text{stack cell appears}}\}\ [c/x]i\ \{\overbrace{c \mapsto \langle()\rangle}^{\text{stack cell disappears}} \star \Psi\}}{\{\Phi\}\ \mathsf{alloca}\, x\, \mathsf{in}\, i\ \{\Psi\}}\ \textsc{Alloca}$$

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:*

$$\underbrace{\ell \mapsto_1 \vec{v}}_{\text{ownership of the block}} \star \overbrace{\ell \leftarrow\!\dashv_1 L}^{\text{knowledge of all antecedents}} \star \underbrace{dom(L) \subseteq \{\ell\}}_{\text{no antecedent (but self)}} \quad \Rrightarrow_I \quad \overbrace{\dagger\dagger\{\ell\}}^{\text{location now dead}} \star \underbrace{\diamond size(\vec{v})}_{\text{credit!}}$$

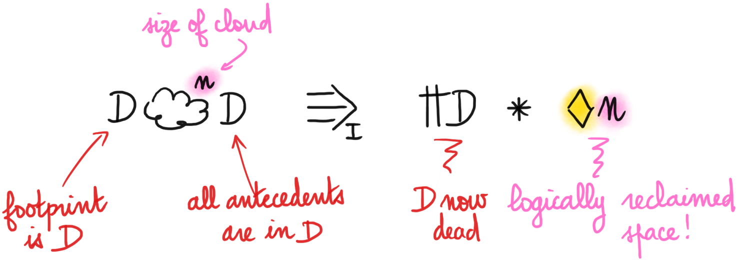
Deletion of deallocated predecessors can be *deferred:*

$$v \leftarrow\!\!\!\dashv_q L \ast \underbrace{\dagger\!\dagger D}_{\text{dead locations}} \Rrightarrow_{I} v \leftarrow\!\!\!\dashv_q L' \quad \text{removed from antecedents}$$

$$\text{if } \mathrm{dom}(L \setminus L') \subseteq D$$

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 *split* and *joined*.

$$\ell \mapsto_{q_1+q_2} b \;\equiv\; \ell \mapsto_{q_1} b \ast \ell \mapsto_{q_2} b$$

$$v \hookleftarrow_{q_1+q_2} L_1 \uplus L_2 \;\equiv\; v \hookleftarrow_{q_1} L_1 \ast v \hookleftarrow_{q_2} L_2$$

$$v \hookleftarrow_{q} L \;\twoheadrightarrow\; v \hookleftarrow_{q} L' \qquad \text{if } L \subseteq L'$$

$$\ell \mapsto_q b \ast \ell' \hookleftarrow_1 L \;\Rrightarrow_{\mathcal{I}}\; \ell \mapsto_q b \ast \ell' \hookleftarrow_1 L \ast \ulcorner \ell' \,\$\, \mathit{pointers}(b) \leq \ell \,\$\, L \urcorner$$

Pointed-by assertions are *covariant*.

Points-to and pointed-by assertions can be *confronted*.

Space credits can be *split* and *joined*.

$$\mathrm{True} \Rrightarrow_{\mathrm{I}} \Diamond 0$$

$$\Diamond(n_1 + n_2) \Rrightarrow_{\mathrm{I}} \Diamond n_1 * \Diamond n_2$$

Theorem (Soundness)

If $\{\diamond S\}\ i\ \{True\}$ *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.

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 SL$\diamond$?

Example: Ghost Reference Counting

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

$$v \hookleftarrow n \quad \triangleq \quad \exists L.\, (v \hookleftarrow_1 L \,\star\, |L| = n)$$

Edge addition / deletion increment / decrement n.

Creating a stack *consumes 4 space credits*.

$$\left\{\begin{matrix} f \mapsto \langle \mathit{create} \rangle \\ \mathit{stack} \mapsto \langle () \rangle \\ \diamond 4 \end{matrix}\right\} \ast f(\mathit{stack}) \ \left\{ \exists \ell. \begin{matrix} f \mapsto \langle \mathit{create} \rangle \\ \mathit{stack} \mapsto \langle \ell \rangle \\ \mathit{isStack}\ \ell\ []\ \star\ \ell \leftarrow\!\!\shortmid 1 \end{matrix}\right\}$$

We get unique ownership of the stack and *we have the sole pointer* to it.

Pushing

Pushing *consumes 4 space credits*.

$$\left\{\begin{array}{c} f \mapsto \langle push\rangle \\ stack \mapsto \langle \ell \rangle \\ elem \mapsto \langle v \rangle \\ \diamond 4 \star isStack\ \ell\ vs \\ v \hookleftarrow n \end{array}\right\} * f(stack, elem) \left\{\begin{array}{c} f \mapsto \langle push\rangle \\ stack \mapsto \langle \ell \rangle \\ elem \mapsto \langle v \rangle \\ isStack\ \ell\ (v :: vs) \\ v \hookleftarrow n+1 \end{array}\right\}$$

The value v receives *one more antecedent*.

Popping *frees up 4 space credits*.

$$\left\{\begin{array}{c} f \mapsto \langle pop \rangle \\ stack \mapsto \langle \ell \rangle \\ elem \mapsto \langle () \rangle \\ isStack\ \ell\ (v :: vs) \\ v \leftarrowtail n \end{array}\right\} * f(stack, elem) \left\{\begin{array}{c} f \mapsto \langle pop \rangle \\ stack \mapsto \langle \ell \rangle \\ elem \mapsto \langle v \rangle \\ \diamond 4 \star isStack\ \ell\ vs \\ v \leftarrowtail n \end{array}\right\}$$

The number of antecedents of v is unchanged, as *elem* points to it.

Logically deallocating the entire stack is a *ghost operation*.

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

$$\left\{\begin{matrix} \mathit{isStack}\ \ell\ \mathit{vs}\ \star\ \ell \leftarrow\!\shortmid 0 \\ \mathop{\ast}\limits_{(v,n)\in \mathit{vns}} v \leftarrow\!\shortmid n \end{matrix}\right\} \Rrightarrow_I \left\{\begin{matrix} \diamond(4 + 4 \times |\mathit{vs}|) \\ \mathop{\ast}\limits_{(v,n)\in \mathit{vns}} v \leftarrow\!\shortmid n - (v\ \$\ \mathit{vs}) \end{matrix}\right\}$$

The ghost reference counters of the stack elements are decremented.

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

$$
\begin{array}{rcl}
\mathit{isList}\;\ell\;[] & \triangleq & \ell \mapsto [0] \\
\mathit{isList}\;\ell\;(v :: vs) & \triangleq & \exists \ell'.\, \ell \mapsto [1; v; \ell'] \star \ell' \hookleftarrow 1 \star \mathit{isList}\;\ell'\;vs
\end{array}
$$

Let's now have a look at *list copy* and its spec. (Fasten seatbelts!)

$copy \triangleq \lambda(self, dst, src).$	
$\quad$ alloca tag in $*tag = [*src + 0]$;	– read the list's tag
$\quad$ if $*tag$ then	– if this is a cons cell, then
$\qquad$ alloca $head$ in $*head = [*src + 1]$;	– read the list's head
$\qquad$ alloca $tail$ in $*tail = [*src + 2]$;	– read the list's tail
$\qquad *src = ()$;	– clobber this root
$\qquad$ alloca dst' in $*self(self, dst', tail)$;	– copy the list's tail
$\qquad *dst =$ alloc 3;	– allocate a new cons cell
$\qquad [*dst + 0] = *tag$;	– and initialize it
$\qquad [*dst + 1] = *head$;	
$\qquad [*dst + 2] = *dst'$	
$\quad$ else	– this must be a nil cell
$\qquad *src = ()$;	– clobber this root
$\qquad *dst =$ alloc 1;	– allocate a new nil cell
$\qquad [*dst + 0] = *tag$	– and initialize it

The case $m = 1$, where we have *the sole pointer* to the list, is special.

$$\dfrac{\left\{\begin{array}{c} f \mapsto \langle copy \rangle \;\star\; dst \mapsto \langle () \rangle \;\star\; src \mapsto \langle \ell \rangle \\ isList\ \ell\ vs \;\star\; \ell \hookleftarrow m \\ m = 1\ ?\ \diamond 0 : \diamond(2 + 4 \times |vs|) \\ \forall v \in vs.\ \exists n.\ (v, n) \in vns \\ \ast_{(v,n) \in vns}\ v \hookleftarrow n \end{array}\right\}}{\dfrac{\ast f(f, dst, src)}{\exists \ell'.\left\{\begin{array}{c} f \mapsto \langle copy \rangle \;\star\; dst \mapsto \langle \ell' \rangle \;\star\; src \mapsto \langle () \rangle \\ m = 1\ ?\ True : (isList\ \ell\ vs \;\star\; \ell \hookleftarrow m - 1) \\ isList\ \ell'\ vs \;\star\; \ell' \hookleftarrow 1 \\ \ast_{(v,n) \in vns}\ v \hookleftarrow n + (m = 1\ ?\ 0 : \underbrace{v \$ vs}) \end{array}\right\}}}$$

need no space

or linear space

orig. list is deallocated

or preserved

each element receives zero new antecedent

or a number of new antecedents

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

- Allocation consumes *space credits* $\diamond n$.
- *Logical deallocation* is a ghost operation.
- Logical dellocation requires *predecessor tracking* $v \leftarrowtail L$.

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$\diamond$ directly to call-by-value λ-calculus.

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

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

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

A Bit of Controversy about OCaml

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

```
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…

A Bit of Controversy about OCaml

But the OCaml compiler transforms the code roughly as follows:

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

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

Thus, `t` remains live while `walk t.1` is executed.

Every *left subtree remains live* until it has been entirely traversed.

Reasoning about space at this level
requires a *precise definition* of where each variable is a root.

StepSeqSkip
$$\text{skip}; i \;/\; \sigma \longrightarrow i \;/\; \sigma$$

StepIf
$$\frac{\sigma(r) = \langle k \rangle}{\text{if } {*r} \text{ then } i_1 \text{ else } i_2 \;/\; \sigma \longrightarrow k \neq 0 \;?\; i_1 : i_2 \;/\; \sigma}$$

StepCall
$$\frac{\sigma(r) = \langle \lambda \vec{x}.i \rangle \qquad |\vec{x}| = |\vec{s}|}{{*r}(\vec{s}) \;/\; \sigma \longrightarrow [\vec{s}/\vec{x}]i \;/\; \sigma}$$

StepConst
$$\frac{\sigma' = \langle s := v \rangle \sigma \qquad \mathit{pointers}(v) = \emptyset}{{*s} = v \;/\; \sigma \longrightarrow \text{skip} \;/\; \sigma'}$$

StepMove
$$\frac{\sigma(r) = \langle v \rangle \qquad \sigma' = \langle s := v \rangle \sigma}{{*s} = {*r} \;/\; \sigma \longrightarrow \text{skip} \;/\; \sigma'}$$

StepAlloc
$$\frac{\sigma' = [\ell \mathrel{+}= ()^n]\sigma \qquad \mathit{size}(\sigma') \leq S \qquad \sigma'' = \langle s := \ell \rangle \sigma'}{{*s} = \text{alloc } n \;/\; \sigma \longrightarrow \text{skip} \;/\; \sigma''}$$

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

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

StepLocEq
$$\frac{\sigma(r_1) = \langle \ell_1 \rangle \qquad \sigma(r_2) = \langle \ell_2 \rangle \qquad \sigma' = \langle s := (\ell_1 = \ell_2 \;?\; 1 : 0) \rangle \sigma}{{*s} = ({*r_1} == {*r_2}) \;/\; \sigma \longrightarrow \text{skip} \;/\; \sigma'}$$

StepAllocaEntry
$$\frac{\sigma' = [c \mathrel{+}= \langle () \rangle]\sigma}{\text{alloca } x \text{ in } i \;/\; \sigma \longrightarrow \text{alloca } c \text{ in } [c/x]i \;/\; \sigma'}$$

StepAllocaExit
$$\frac{\sigma(c) = \langle v \rangle \qquad \sigma' = [c := \lightning]\sigma}{\text{alloca } c \text{ in skip} \;/\; \sigma \longrightarrow \text{skip} \;/\; \sigma'}$$

StepFork
$$\frac{\sigma(r) = \langle v \rangle \qquad \sigma' = [r := ()][c \mathrel{+}= \langle v \rangle]\sigma}{\text{fork } {*r} \text{ as } x \text{ in } i \;/\; \sigma \longrightarrow \text{skip} \;/\; \sigma' \;\; \mathit{spawning} \text{ alloca } c \text{ in } [c/x]i}$$

StepContext
$$\frac{i \;/\; \sigma \longrightarrow i' \;/\; \sigma' \;\; \mathit{spawning}\ \vec{i}}{K[i] \;/\; \sigma \longrightarrow K[i'] \;/\; \sigma' \;\; \mathit{spawning}\ \vec{i}}$$