François Pottier
Jean-Marie Madiot

SL◊

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.

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

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

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

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.

A Ghost Deallocation Operation?

Idea 2. Switch to a *logical deallocation* operation:

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

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

This marries

- *manual reasoning* about memory at verification time
- *automatic management* of memory at runtime.

Soundness?

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*.

Because $x \mapsto b$ is consumed,

- 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*.

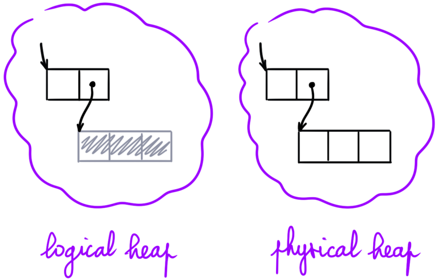
$$x \mapsto \ell \;\not\Rrightarrow\; \Diamond\, \mathit{size}(\ell)$$

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!

To avoid this problem, we must *restrict logical deallocation*:

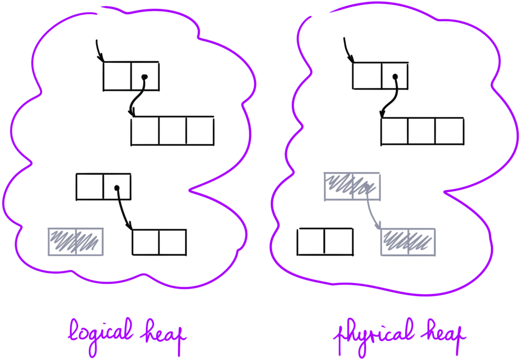
- A reachable block must not be deallocated.

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 Desired Global Invariant

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



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.

Points-To and Pointed-By Assertions

In addition to *points-to*, we use *pointed-by* assertions:

points-to pointed-by

$$\ell \mapsto b \quad * \quad \ell \mapsfrom L$$

permission to read/write the block at ℓ

permission to add/remove pointers to ℓ

permission to deallocate if $L = \emptyset$

We get a sound logical deallocation axiom:

$$x \mapsto b \ast x \mapsfrom \emptyset \Rrightarrow \Diamond\mathrm{size}(b)$$

This axiom deallocates one block.

There is also a *bulk logical deallocation* axiom.

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.

SpaceLang is imperative. An *instruction* i does not return a value.

skip	*no-op*	$*\varrho = \mathsf{alloc}\ n$	*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*	fork $*\varrho$ as x in i	*thread creation*

The operands of every instruction are stack cells ϱ.

There is *no heap deallocation* instruction.

Operational Semantics of SpaceLang

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.

Heap allocation *consumes space credits*.

ALLOC

diamonds consumed

new assertions

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

updated stack cell

edge deletion

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

One pointer to the value v is *deleted*.

Reasoning about a heap store 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*.

Logical deallocation of a block is a *ghost operation:*

knowledge of all antecedents

location now dead

$$\ell \mapsto_1 \vec{v} \star \ell \leftarrow\!\!\dashv_1 L \star dom(L) \subseteq \{\ell\} \quad \Rrightarrow_I \quad \dagger\!\!\dagger\{\ell\} \star \diamond size(\vec{v})$$

ownership of the block

no antecedent (but self)

credit!

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, 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.

Roadmap

The user may define *custom* (simplified) predecessor tracking disciplines.

For example, sometimes, *counting* predecessors is enough.

$$v \leftarrowtail n \quad \triangleq \quad \exists L.\, (v \leftarrowtail_1 L \;\star\; |L| = n)$$

Edge addition and deletion increment and decrement n.

Creating a stack *consumes 4 space credits*.

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

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

SL◇

Pushing *consumes 4 space credits*.

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

The value v receives *one more antecedent*.

Popping *frees up 4 space credits*.

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

The number of predecessors of v is unchanged,
because the out-parameter *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*.

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

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

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* $v \leftarrowtail L$.

Limitations and Future Work

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 *λ-calculus*.

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:

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

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.

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.

We fix a *maximum heap size* S.

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

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

S is a parameter of the operational semantics,

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

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

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

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

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).

Complete Operational Semantics

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

Heap allocation *consumes space credits*.

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

new assertions

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.)

Heap Store

Writing a heap cell is simple... but involves some administration.

$$\textsc{Store}\quad \frac{\vec{v}(o) = v}{\begin{Bmatrix} s \mapsto \langle \ell \rangle \\ r \mapsto \langle v' \rangle \\ \ell \mapsto_1 \vec{v} \\ v \longmapsfrom_q L \\ v' \longmapsfrom_{q'} L' \end{Bmatrix} [*s + o] = *r \begin{Bmatrix} s \mapsto \langle \ell \rangle \\ r \mapsto \langle v' \rangle \\ \ell \mapsto_1 [o := v']\vec{v} \\ v \longmapsfrom_q L \setminus \{\ell\} \\ v' \longmapsfrom_{q'} L' \uplus \{\ell\} \end{Bmatrix}}$$

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.

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

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:*

knowledge of all antecedents

location now dead

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

ownership of the block

no antecedent (but self)

credit!

Deletion of deallocated predecessors can be *deferred:*

$$v \longleftarrow\!\!\dashv_q L \;*\; \dagger D \;\Rrightarrow_{I}\; v \longleftarrow\!\!\dashv_q L'$$

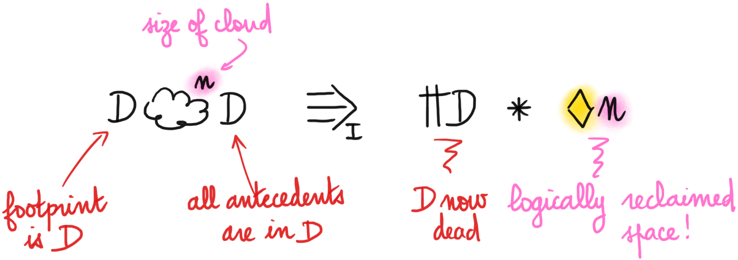
removed from antecedents

dead locations

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*.

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

$$v \leftarrow_{q_1+q_2} L_1 \uplus L_2 \quad\equiv\quad v \leftarrow_{q_1} L_1 \ast v \leftarrow_{q_2} L_2$$

$$v \leftarrow_{q} L \quad -\!\!\ast \quad v \leftarrow_{q} L' \qquad \text{if } L \subseteq L'$$

$$\ell \mapsto_{q} b \ast \ell' \leftarrow_{1} L \Rrightarrow_{\mathcal{I}} \ell \mapsto_{q} b \ast \ell' \leftarrow_{1} L \ast \ulcorner \ell' \mathbin{\$} \mathit{pointers}(b) \leq \ell \mathbin{\$} L \urcorner$$

Pointed-by assertions are *covariant*.

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

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

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

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

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

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

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

need no space
or linear space

orig. list is
deallocated
or preserved

each element receives
zero new antecedent
or a number of new antecedents

SL◇