

Dynamic Separation Logic

Frank S. de Boer^{a,b,1} Hans-Dieter A. Hiep^{a,b,2} Stijn de Gouw^{c,3}

^a *Leiden Institute for Advanced Computer Science (LIACS)
Leiden University
Leiden, the Netherlands*

^b *Computer Security group
Centrum Wiskunde & Informatica (CWI)
Amsterdam, the Netherlands*

^c *Department of Computer Science
Open University (OU)
Heerlen, the Netherlands*

Abstract

This paper introduces a dynamic logic extension of separation logic. The assertion language of separation logic is extended with modalities for the five types of the basic instructions of separation logic: simple assignment, look-up, mutation, allocation, and de-allocation. The main novelty of the resulting dynamic logic is that it allows to combine different approaches to resolving these modalities. One such approach is based on the standard weakest precondition calculus of separation logic. The other approach introduced in this paper provides a novel alternative formalization in the proposed dynamic logic extension of separation logic. The soundness and completeness of this axiomatization has been formalized in the Coq theorem prover.

Keywords: Separation logic, dynamic logic, weakest precondition, sequential programs.

1 Introduction

This paper describes a study into the expressive power of separation logic (SL, for short) with regard to the formalization of *weakest preconditions* [7]. To this end, we introduce a novel dynamic logic extension of SL, which we abbreviate by DSL (for Dynamic Separation Logic).

SL [19] extends Hoare logic for the specification and verification of heap manipulating programs in terms of pre- and postconditions. The assertion language of SL features the basic heap assertion $(x \mapsto e)$, ‘ x points to e ’, which expresses that the variable x denotes the single allocated memory location which stores the value of the expression e . The so-called separating conjunction $(p * q)$ allows to split the heap, that is, the set of allocated memory locations and their contents, into two disjoint parts one of which satisfies the conjunct p and the other satisfies q . The separating implication $(p \multimap q)$, roughly, holds if every extension of the heap satisfies q , whenever p holds for the extension itself (separately). For an

¹ Email: frb@cwi.nl

² Email: hdh@cwi.nl

³ Email: sdg@ou.nl

introduction to SL and an extensive survey of the literature, intended for a broad audience, see the paper by A. Charguéraud [5].

Dynamic logic [9] generalizes Hoare logics by introducing for each statement of the underlying programming language a corresponding modality, so that the formula $[S]p$ expresses the weakest precondition of the statement S with respect to the postcondition p . Informally, $[S]p$ is valid if every terminating computation establishes p . In this paper we extend the assertion language of SL with *modalities* for the five types of the basic instructions of SL: simple assignment, look-up, mutation, allocation, and de-allocation. For any such basic instruction S , we then can introduce in the Hoare logic the axiom

$$\{[S]p\} S \{p\}$$

which is trivially sound and complete by definition of $[S]p$. In case S is a simple assignment $x := e$ and p is an assertion in standard SL, we can resolve the weakest precondition $[S]p$, as in first-order dynamic logic, simply by *substituting* every free occurrence of x in p by the expression e .⁴ In SL we can resolve $[S]p$, for any other basic instruction S , by a formula with a hole $C_S(\cdot)$ in SL itself, such that $C_S(p)$ is equivalent to $[S]p$. For example, the assertion

$$(\exists y(x \mapsto y)) * ((x \mapsto e) \multimap p)$$

states that the heap can be split in a sub-heap which consists of a single memory cell denoted by x such that p holds for every extension of the other part with a single memory cell denoted by x and which contains the value of e . It follows that this assertion is equivalent to $[[x] := e]p$, where the *mutation* instruction $[x] := e$ assigns the value of the expression e to the heap location denoted by the variable x .

The main contribution of this paper is a complementary approach to resolving $[S]p$, for any basic instruction. In this approach we obtain an alternative characterization of the weakest precondition $[S]p$ by a novel axiomatization of the modalities in DSL. This axiomatization allows for a characterization of $[S]p$ *compositionally* in terms of the syntactical structure of p .

O’Hearn, Reynolds, and Yang introduced local axioms [15] and show how to derive from these local axioms a weakest precondition axiomatization of the basic instructions in SL, using the frame rule and the separating implication for expressing the weakest precondition. However, the separating implication is actually not needed to prove completeness of the local axioms for simple assignments, look-up, allocation, and de-allocation. We illustrate the expressiveness of DSL by extending this result to the local mutation axiom. We further illustrate the expressiveness of DSL by a novel *strongest postcondition* axiomatization.

Using the proof assistant Coq, we have formally verified the soundness and completeness proofs of the axiomatization of the DSL modalities. All our results can be readily extended to a programming language involving (sequential) control structures such as loops.

Acknowledgments

The authors are grateful for the constructive feedback provided by the anonymous referees.

2 Syntax and semantics

We follow the presentation of SL in [19]. A heap⁵ h is represented by a (finitely-based) *partial* function $\mathbb{Z} \rightarrow \mathbb{Z}$ and the domain of h is denoted by $\text{dom}(h)$. We write $h(n) = \perp$ if $n \notin \text{dom}(h)$. The heaps h, h' are disjoint iff $\text{dom}(h) \cap \text{dom}(h') = \emptyset$. A heap h is partitioned in h_1 and h_2 , denoted by $h = h_1 \uplus h_2$, iff h_1 and h_2 are disjoint, $\text{dom}(h) = \text{dom}(h_1) \cup \text{dom}(h_2)$, and $h(n) = h_i(n)$ if $n \in \text{dom}(h_i)$ for $i \in \{1, 2\}$.

V denotes a countably infinite set of integer variables, with typical element x . A store s is a total function $V \rightarrow \mathbb{Z}$. We abstract from the syntax of arithmetic expressions e , and Boolean expressions b . By $\text{var}(e)$ (resp. $\text{var}(b)$) we denote the finite set of variables that occur in e (resp. b). We have the Boolean constants **true** and **false**, and $(e_1 = e_2)$ is a Boolean expression given arithmetic expressions e_1 and e_2 .

⁴ After suitable renaming of the bound variables in p such that no variable of e gets bound.

⁵ All italicized variables are typical meta-variables, and we use primes and subscripts for other meta-variables of the same type, e.g. h, h', h'', h_1, h_2 are all heaps.

$$\begin{aligned}
\langle x := e, h, s \rangle &\Rightarrow (h, s[x := s(e)]), \\
\langle x := [e], h, s \rangle &\Rightarrow (h, s[x := h(s(e))]) \text{ if } s(e) \in \text{dom}(h), \\
\langle x := [e], h, s \rangle &\Rightarrow \mathbf{fail} \text{ if } s(e) \notin \text{dom}(h), \\
\langle [x] := e, h, s \rangle &\Rightarrow (h[s(x) := s(e)], s) \text{ if } s(x) \in \text{dom}(h), \\
\langle [x] := e, h, s \rangle &\Rightarrow \mathbf{fail} \text{ if } s(x) \notin \text{dom}(h), \\
\langle x := \mathbf{cons}(e), h, s \rangle &\Rightarrow (h[n := s(e)], s[x := n]) \text{ where } n \notin \text{dom}(h). \\
\langle \mathbf{dispose}(x), h, s \rangle &\Rightarrow (h[s(x) := \perp], s) \text{ if } s(x) \in \text{dom}(h), \\
\langle \mathbf{dispose}(x), h, s \rangle &\Rightarrow \mathbf{fail} \text{ if } s(x) \notin \text{dom}(h).
\end{aligned}$$

Fig. 1. Semantics of basic instructions of heap manipulating programs.

By $s(e)$ we denote the integer value of e in s , and by $s(b)$ we denote the Boolean value of b in s . Following [19] expressions thus do not refer to the heap. By $s[x := v]$ and $h[n := v]$ we denote the result of updating the value of the variable x and the location n , respectively. The definition of $h[n := v]$ does not require that $n \in \text{dom}(h)$. More specifically, we have

$$h[n := v](m) = \begin{cases} v & \text{if } n = m \\ h(m) & \text{otherwise} \end{cases}$$

Thus, $\text{dom}(h[n := v]) = \text{dom}(h) \cup \{n\}$. For heaps we also define the clearing of a location, denoted by $h[n := \perp]$. We have $h[n := \perp](m) = \perp$ if $n = m$, and $h[n := \perp](m) = h(m)$ otherwise. Similarly, we have $\text{dom}(h[n := \perp]) = \text{dom}(h) \setminus \{n\}$.

Following [19], we have the following basic instructions: $x := e$ (simple assignment), $x := [e]$ (look-up), $[x] := e$ (mutation), $x := \mathbf{cons}(e)$ (allocation), $\mathbf{dispose}(x)$ (de-allocation). Just like [10],

We will not give a full syntax of [statements], as the treatment of conditionals and looping statements is standard. Instead, we will concentrate on assignment statements, which is where the main novelty of the approach lies.

The successful execution of any basic instruction S is denoted by $\langle S, h, s \rangle \Rightarrow (h', s')$, whereas $\langle S, h, s \rangle \Rightarrow \mathbf{fail}$ denotes a failing execution (e.g. due to access of a ‘dangling pointer’). See Figure 1 for their semantics (and see Appendix, Figure A.1, for the full syntax and semantics).

We follow [10] in the definition of the syntax and semantics of the assertion language of SL but we use a different atomic ‘weak points to’ formula (as in [18] and [6]). In DSL we have additionally a modality for each statement S , which has highest binding priority.

$$p, q ::= b \mid (e \hookrightarrow e') \mid (p \rightarrow q) \mid (\forall x p) \mid (p * q) \mid (p \multimap q) \mid [S]p$$

By $h, s \models p$ we denote the truth relation of classical SL, see Figure 2. Validity of p is denoted by $\models p$. Semantics of DSL extends the semantics of SL by giving semantics to the modality, expressing the weakest precondition. We further have the usual abbreviations: $\neg p$ denotes $(p \rightarrow \mathbf{false})$, $(p \vee q)$ denotes $(\neg p \rightarrow q)$ (negation has binding priority over implication), $p \equiv q$ denotes $(p \rightarrow q) \wedge (q \rightarrow p)$, $(\exists x p)$ denotes $\neg(\forall x(\neg p))$ and note that x is bound in p . By logical connective we mean the connectives $\neg, \wedge, \vee, \rightarrow, \forall, \exists$, and by separating connective we mean $*$ and \multimap . Further, $(e \hookrightarrow -)$ denotes $\exists x(e \hookrightarrow x)$ for a fresh x , \mathbf{emp} denotes $\forall x(x \not\hookrightarrow -)$, and $(e \mapsto e')$ denotes $(e \hookrightarrow e') \wedge (\forall x((x \hookrightarrow -) \rightarrow x = e))$ for a fresh x . We use $\not\hookrightarrow$ and \neq as negations of the predicate as usual, and in particular $(e \not\hookrightarrow -)$ is $\neg\exists x(e \hookrightarrow x)$. We may drop matching parentheses if doing so would not give rise to ambiguity. Note that $h, s \models \mathbf{emp}$ iff $\text{dom}(h) = \emptyset$, and $h, s \models (e \mapsto e')$ iff $\text{dom}(h) = \{s(e)\}$ and $h(s(e)) = s(e')$. An assertion is *first-order* if its construction does not involve separating connectives or modalities.

The assertion $(e \hookrightarrow e')$ is implied by $(e \mapsto e')$, and to express the latter using the former requires the use of separating connectives (i.e. $(e \hookrightarrow e')$ is equivalent to $\mathbf{true} * (e \mapsto e')$), whereas our definition of $(e \mapsto e')$ requires only logical connectives, and thus we use $(e \hookrightarrow e')$ as atomic formula.

A specification $\{p\} S \{q\}$ is a triple that consists of a precondition p , a program S , and a postcondition q . Specifications are interpreted in the sense of strong partial correctness, which ensures absence of explicit

$h, s \models b$ iff $s(b) = \mathbf{true}$,
 $h, s \models (e \hookrightarrow e')$ iff $s(e) \in \text{dom}(h)$ and $h(s(e)) = s(e')$,
 $h, s \models (p \wedge q)$ iff $h, s \models p$ and $h, s \models q$,
 $h, s \models (p \rightarrow q)$ iff $h, s \models p$ implies $h, s \models q$,
 $h, s \models (\forall x p)$ iff $h, s[x := n] \models p$ for all n ,
 $h, s \models (p * q)$ iff $h_1, s \models p$ and $h_2, s \models q$ for some h_1, h_2 such that $h = h_1 \uplus h_2$,
 $h, s \models (p \multimap q)$ iff $h', s \models p$ implies $h'', s \models q$ for all h', h'' such that $h'' = h \uplus h'$,
 $h, s \models [S]p$ iff $\langle S, h, s \rangle \not\Rightarrow \mathbf{fail}$ and $h', s' \models p$ for all h', s' such that $\langle S, h, s \rangle \Rightarrow (h', s')$.

Fig. 2. Semantics of Dynamic Separation Logic.

failure. Formally, following [19], the validity of a specification, denoted $\models \{p\} S \{q\}$, is defined as: if $h, s \models p$, then $\langle S, h, s \rangle \not\Rightarrow \mathbf{fail}$ and also $\langle S, h, s \rangle \Rightarrow (h', s')$ implies $h', s' \models q$ for all h', s' .

Note that we have that $\models \{[S]q\} S \{q\}$ holds, and $\models \{p\} S \{q\}$ implies $\models p \rightarrow [S]q$, that is, $[S]q$ is the weakest precondition of statement S and postcondition q .

3 A sound and complete axiomatization of DSL

In dynamic logic axioms are introduced to simplify formulas in which modalities occur. For example, we have the following basic equivalences **E1-3** for simple assignments.

Lemma 3.1 (Basic equivalences) *Let S denote a simple assignment $x := e$ and \circ denote a (binary) logical or separating connective.*

$$[S]\mathbf{false} \equiv \mathbf{false} \quad (\mathbf{E1})$$

$$[S](p \circ q) \equiv [S]p \circ [S]q \quad (\mathbf{E2})$$

$$[S](\forall y p) \equiv \forall y([S]p) \quad (\mathbf{E3})$$

In **E3** we assume that y does not appear in S , neither in the left-hand-side of the assignment S nor in its right-hand-side.

The proofs of these equivalences proceed by a straightforward induction on the structure of p , where the base cases of Boolean expressions and the weak points to predicate are handled by a straightforward extension of the *substitution lemma* for standard first-order logic. By $b[e/x]$ we denote the result of replacing every occurrence of x in the Boolean expression b by the expression e (and similar for arithmetic expressions).

Lemma 3.2 (Substitution lemma)

$$[x := e]b \equiv b[e/x] \quad [x := e](e' \hookrightarrow e'') \equiv (e'[e/x] \hookrightarrow e''[e/x]) \quad (\mathbf{E4})$$

Proof. This lemma follows from the semantics of simple assignment modality and the substitution lemma of first-order expressions: $s(e'[e/x]) = s[x := s(e)](e')$. Note that expressions do not refer to the heap. \square

The above equivalences **E1-3** do not hold in general for the other basic instructions. For example, we have $[x := [e]]\mathbf{false} \equiv \neg(e \hookrightarrow -)$. On the other hand, $[x := \mathbf{cons}(0)]\mathbf{false} \equiv \mathbf{false}$, but $[x := \mathbf{cons}(0)](x \neq 0)$ is not equivalent to $\neg([x := \mathbf{cons}(0)](x = 0))$, because $[x := \mathbf{cons}(0)](x \neq 0)$ is equivalent to $(0 \hookrightarrow -)$ (‘zero is allocated’), whereas $\neg([x := \mathbf{cons}(0)](x = 0))$ expresses that $(n \not\hookrightarrow -)$, for some $n \neq 0$ (which holds for any finite heap).

The above equivalences **E1-3**, with **E2** restricted to the (standard) logical connectives, *do* hold for the *pseudo* instructions $\langle x \rangle := e$, a so-called *heap update*, and $\langle x \rangle := \perp$, a so-called *heap clear*. These pseudo instructions are defined by the transitions

$$\langle \langle x \rangle := e, h, s \rangle \Rightarrow (h[s(x) := s(e)], s) \quad \text{and} \quad \langle \langle x \rangle := \perp, h, s \rangle \Rightarrow (h[s(x) := \perp], s)$$

In contrast to the mutation and de-allocation instructions, these pseudo-instructions do not require that $s(x) \in \text{dom}(h)$, e.g., if $s(x) \notin \text{dom}(h)$ then the heap update $\langle x \rangle := e$ extends the domain of the heap, whereas $[x] := e$ leads to failure in that case. From a practical viewpoint, the heap update and heap clear pseudo-instructions are ‘lower level’ instructions, e.g. in processors that implement virtual memory (where an operating system allocates memory on the fly whenever a program performs a write to a virtual address that is not allocated), and on top of these instructions efficient memory allocation algorithms are implemented, e.g. malloc and free in C. In the following lemma we give an axiomatization in DSL of the basic SL instructions in terms of simple assignments and these two pseudo-instructions. For comparison we also give the standard SL axiomatization [19,8,3].

Lemma 3.3 (Axioms basic instructions)

$$[x := [e]]p \equiv \exists y((e \hookrightarrow y) \wedge [x := y]p), \quad (\mathbf{E5})$$

$$[[x] := e]p \equiv \begin{cases} (x \hookrightarrow -) \wedge [\langle x \rangle := e]p \\ (x \mapsto -) * ((x \mapsto e) \multimap p) \end{cases} \quad (\mathbf{E6})$$

$$[x := \mathbf{cons}(e)]p \equiv \begin{cases} \forall x((x \not\mapsto -) \rightarrow [\langle x \rangle := e]p) \\ \forall x((x \mapsto e) \multimap p) \end{cases} \quad (\mathbf{E7})$$

$$[\mathbf{dispose}(x)]p \equiv \begin{cases} (x \hookrightarrow -) \wedge [\langle x \rangle := \perp]p \\ (x \mapsto -) * p \end{cases} \quad (\mathbf{E8})$$

Note that $[x := y]p$ in **E5** reduces to $p[y/x]$ by **E1-4**. For technical convenience only, we require in the axioms for $x := \mathbf{cons}(e)$ that x does not appear in e (see Section 5 to lift this restriction).

In the sequel **E5-8** refer to the corresponding DSL equivalences. The proofs of these equivalences are straightforward (consist simply of expanding the semantics of the involved modalities) and therefore omitted.

We have the following SL axiomatization of the heap update and heap clear pseudo-instructions.

$$\begin{aligned} [\langle x \rangle := e]p &\equiv ((x \mapsto -) * ((x \mapsto e) \multimap p)) \vee ((x \not\mapsto -) \wedge ((x \mapsto e) \multimap p)) \\ [\langle x \rangle := \perp]p &\equiv ((x \mapsto -) * p) \vee ((x \not\mapsto -) \wedge p) \end{aligned}$$

This axiomatization thus requires a case distinction between whether or not x is allocated.

For the complementary approach, we want to resolve the modalities for the heap update and heap clear instructions compositionally in terms of p . What thus remains for a complete axiomatization is a characterization of $[S]b$, $[S](e \hookrightarrow e')$, $[S](p * q)$, and $[S](p \multimap q)$, where S denotes one of the two pseudo-instructions. Lemma 3.4 provides an axiomatization in DSL of a heap update.

Lemma 3.4 (Heap update) *We have the following equivalences for the heap update modality.*

$$[\langle x \rangle := e]b \equiv b, \quad (\mathbf{E9})$$

$$[\langle x \rangle := e](e' \hookrightarrow e'') \equiv (x = e' \wedge e'' = e) \vee (x \neq e' \wedge e' \hookrightarrow e''), \quad (\mathbf{E10})$$

$$[\langle x \rangle := e](p * q) \equiv ([\langle x \rangle := e]p * q') \vee (p' * [\langle x \rangle := e]q), \quad (\mathbf{E11})$$

$$[\langle x \rangle := e](p \multimap q) \equiv p' \multimap [\langle x \rangle := e]q, \quad (\mathbf{E12})$$

where p' abbreviates $p \wedge (x \not\mapsto -)$ and, similarly, q' abbreviates $q \wedge (x \not\mapsto -)$.

These equivalences we can informally explain as follows. Since the heap update $\langle x \rangle := e$ does not affect the store, and the evaluation of a Boolean condition b only depends on the store, we have that $([\langle x \rangle := e]b) \equiv b$.

Predicting whether $(e' \hookrightarrow e'')$ holds after $\langle x \rangle := e$, we only need to make a distinction between whether x and e' are aliases, that is, whether they denote the same location, which is simply expressed by $x = e'$. If $x = e'$ then $e'' = e$ should hold, otherwise $(e' \hookrightarrow e'')$ (note again, that $\langle x \rangle := e$ does not affect the values of the expressions e, e' and e''). As a basic example, we compute

$$\begin{aligned}
[\langle x \rangle := e](y \hookrightarrow -) &\equiv (\text{definition } y \hookrightarrow -) \\
[\langle x \rangle := e]\exists z(y \hookrightarrow z) &\equiv \mathbf{(E3)} \\
\exists z[\langle x \rangle := e](y \hookrightarrow z) &\equiv \mathbf{(E10)} \\
\exists z((y = x \wedge e = z) \vee (y \neq x \wedge (y \hookrightarrow z))) &\equiv (\text{semantics SL}) \\
y \neq x \rightarrow (y \hookrightarrow -) &
\end{aligned}$$

We use this derived equivalence in the following example:

$$\begin{aligned}
[\langle x \rangle := e](y \mapsto -) &\equiv (\text{definition } y \mapsto -) \\
[\langle x \rangle := e]((y \hookrightarrow -) \wedge \forall z((z \hookrightarrow -) \rightarrow z = y)) &\equiv \mathbf{(E2, E3, E9)} \\
[\langle x \rangle := e](y \hookrightarrow -) \wedge \forall z([\langle x \rangle := e](z \hookrightarrow -) \rightarrow z = y) &\equiv (\text{see above}) \\
(y \neq x \rightarrow (y \hookrightarrow -)) \wedge \forall z((z \neq x \rightarrow (z \hookrightarrow -)) \rightarrow z = y) &\equiv (\text{semantics SL}) \\
y = x \wedge (\mathbf{emp} \vee (x \mapsto -)) &
\end{aligned}$$

Predicting whether $(p * q)$ holds after the heap update $\langle x \rangle := e$, we need to distinguish between whether p or q holds for the sub-heap that contains the (updated) location x . Since we do not assume that x is already allocated, we instead distinguish between whether p or q holds initially for the sub-heap that does *not* contain the updated location x . As a simple example, we compute

$$\begin{aligned}
[\langle x \rangle := e](\mathbf{true} * (x \mapsto -)) &\equiv \mathbf{(E9, E11)} \\
(\mathbf{true} * ((x \mapsto -) \wedge (x \not\mapsto -))) \vee ((x \not\mapsto -) * [\langle x \rangle := e](x \mapsto -)) &\equiv (\text{see above}) \\
(\mathbf{true} * ((x \mapsto -) \wedge (x \not\mapsto -))) \vee ((x \not\mapsto -) * (\mathbf{emp} \vee (x \mapsto -))) &\equiv (\text{semantics SL}) \\
(\mathbf{true} * \mathbf{false}) \vee ((x \not\mapsto -) * (\mathbf{emp} \vee (x \mapsto -))) &\equiv (\text{semantics SL}) \\
\mathbf{true} &
\end{aligned}$$

Note that this coincides with the above calculation of $[\langle x \rangle := e](y \hookrightarrow -)$, which also reduces to \mathbf{true} , instantiating y by x .

The semantics of $(p * q)$ after the heap update $\langle x \rangle := e$ involves universal quantification over all disjoint heaps that do not contain x (because after the heap update x is allocated). Therefore we simply add the condition that x is not allocated to p , and apply the heap update to q . As a very basic example, we compute

$$\begin{aligned}
[\langle x \rangle := 0]((y \hookrightarrow 1) \multimap (y \hookrightarrow 1)) &\equiv \mathbf{(E12)} \\
((y \mapsto 1) \wedge (x \not\mapsto -)) \multimap [\langle x \rangle := 0](y \hookrightarrow 1) &\equiv \mathbf{(E10)} \\
((y \mapsto 1) \wedge (x \not\mapsto -)) \multimap ((y = x \wedge 0 = 1) \vee (y \neq x \wedge y \hookrightarrow 1)) &\equiv (\text{semantics SL}) \\
\mathbf{true} &
\end{aligned}$$

Note that $(y \hookrightarrow 1) \multimap (y \hookrightarrow 1) \equiv \mathbf{true}$ and $[\langle x \rangle := 0]\mathbf{true} \equiv \mathbf{true}$.

Proof of Lemma 3.4.

E9 $h, s \models [\langle x \rangle := e]b$

iff (semantics heap update modality)

$h[s(x) := s(e)], s \models b$

iff (b does not depend on the heap)

$h, s \models b$

E10 $h, s \models [\langle x \rangle := e](e' \hookrightarrow e'')$

iff (semantics heap update modality)

$h[s(x) := s(e)], s \models e' \hookrightarrow e''$

iff (semantics points-to)

$h[s(x) := s(e)](s(e')) = s(e'')$

iff (definition $h[s(x) := s(e)]$)

if $s(x) = s(e')$ then $s(e) = s(e'')$ else $h(s(e')) = s(e'')$

iff (semantics assertions)

$h, s \models (x = e' \wedge e'' = e) \vee (x \neq e' \wedge e' \hookrightarrow e'')$

E11 $h, s \models [\langle x \rangle := e](p * q)$

iff (semantics heap update modality)

$h[s(x) := s(e)], s \models p * q$.

From here we proceed as follows. By the semantics of separating conjunction, there exist h_1 and h_2 such that $h[s(x) := s(e)] = h_1 \uplus h_2$, $h_1, s \models p$, and $h_2, s \models q$. Let $s(x) \in \text{dom}(h_1)$ (the other case runs similarly). So $h[s(x) := s(e)] = h_1 \uplus h_2$ implies $h_1(s(x)) = s(e)$ and $h = h_1[s(x) := h(x)] \uplus h_2$. By the semantics of the heap update modality, $h_1(s(x)) = s(e)$ and $h_1, s \models p$ implies $h_1[s(x) := h(x)], s \models [\langle x \rangle := e]p$. Since $s(x) \notin \text{dom}(h_2)$, we have $h_2, s \models q \wedge x \not\hookrightarrow -$. By the semantics of separation conjunction we conclude that $h, s \models [\langle x \rangle := e]p * q'$ (q' denotes $q \wedge x \not\hookrightarrow -$).

In the other direction, from $h, s \models [\langle x \rangle := e]p * q'$ (the other case runs similarly) we derive that there exist h_1 and h_2 such that $h = h_1 \uplus h_2$, $h_1, s \models [\langle x \rangle := e]p$ and $h_2, s \models q'$. By the semantics of the heap update modality it follows that $h_1[s(x) := s(e)], s \models p$. Since $s(x) \notin \text{dom}(h_2)$, we have that $h[s(x) := s(e)] = h_1[s(x) := s(e)] \uplus h_2$, and so $h[s(x) := s(e)], s \models p * q$, that is, $h, s \models [\langle x \rangle := e](p * q)$.

E12 $h, s \models [\langle x \rangle := e](p \multimap q)$

iff (semantics of heap update modality)

$h[s(x) := s(e)], s \models p \multimap q$

iff (semantics separating implication)

for every h' disjoint from $h[s(x) := s(e)]$: if $h', s \models p$ then $h[s(x) := s(e)] \uplus h', s \models q$

iff (since $s(x) \notin \text{dom}(h')$)

for every h' disjoint from h : if $h', s \models p \wedge x \not\hookrightarrow -$ then $(h \uplus h')[s(x) := s(e)], s \models q$

iff (semantics of heap update modality)

for every h' disjoint from h : if $h', s \models p \wedge x \not\hookrightarrow -$ then $h \uplus h', s \models [s(x) := s(e)]q$

iff (semantics separating implication)

$h, s \models (p \wedge x \not\hookrightarrow -) \multimap [\langle x \rangle := e]q$. □

The equivalences for the heap clear modality in the following lemma can be informally explained as follows: Since $\langle x \rangle := \perp$ does not affect the store, and the evaluation of a Boolean condition b only depends on the store, we have that $[\langle x \rangle := \perp]b = b$. For $e \hookrightarrow e'$ to hold after executing $\langle x \rangle := \perp$, we must initially have that $x \neq e$ and $e \hookrightarrow e'$. As a simple example, we have that $\forall y, z (y \not\hookrightarrow z)$ characterizes the empty heap. It follows that $[\langle x \rangle := \perp](\forall y, z (y \not\hookrightarrow z))$ is equivalent to $\forall y, z (\neg(y \neq x \wedge y \hookrightarrow z))$. The latter first-order formula is equivalent to $\forall y, z (y = x \vee y \not\hookrightarrow z)$. This assertion thus states that the domain consists at most of the location x , which indeed ensures that after $\langle x \rangle := \perp$ the heap is empty. To ensure that $p * q$ holds after clearing x it suffices to show that the initial heap can be split such that both p and q hold in their respective sub-heaps with x cleared. The semantics of $p \multimap q$ after clearing x involves universal quantification over all disjoint heaps that do may contain x , whereas before executing $\langle x \rangle := \perp$ it involves universal quantification over all disjoint heaps that do *not* contain x , in case x is allocated initially. To formalize in the initial configuration universal quantification over all disjoint heaps we distinguish between all disjoint heaps that do not contain x and *simulate* all disjoint heaps that contain x by interpreting both

p and q in $p \multimap q$ in the context of heap updates $\langle x \rangle := y$ with *arbitrary* values y for the location x . As a very basic example, consider $[\langle x \rangle := \perp]((x \hookrightarrow 0) \multimap (x \hookrightarrow 0))$, which should be equivalent to **true**. The left conjunct $((x \hookrightarrow 0) \wedge (x \not\hookrightarrow -)) \multimap [\langle x \rangle := \perp](x \hookrightarrow 0)$ of the resulting formula after applying **E16** is equivalent to **true** (because $(x \hookrightarrow 0) \wedge (x \not\hookrightarrow -)$ is equivalent to **false**). We compute the second conjunct (in the application of **E10** we omitted some trivial reasoning steps):

$$\begin{aligned} \forall y([\langle x \rangle := y](x \hookrightarrow 0) \multimap [\langle x \rangle := y](x \hookrightarrow 0)) &\equiv \text{(E10)} \\ \forall y(y = 0 \multimap y = 0) &\equiv \text{(semantics SL)} \\ \text{true} & \end{aligned}$$

Lemma 3.5 (Heap clear) *We have the following equivalences for the heap clear modality.*

$$\begin{aligned} [\langle x \rangle := \perp]b &\equiv b, & \text{(E13)} \\ [\langle x \rangle := \perp](e \hookrightarrow e') &\equiv (x \neq e) \wedge (e \hookrightarrow e'), & \text{(E14)} \\ [\langle x \rangle := \perp](p * q) &\equiv [\langle x \rangle := \perp]p * [\langle x \rangle := \perp]q, & \text{(E15)} \\ [\langle x \rangle := \perp](p \multimap q) &\equiv ((p \wedge x \not\hookrightarrow -) \multimap [\langle x \rangle := \perp]q) \wedge \forall y([\langle x \rangle := y]p \multimap [\langle x \rangle := y]q), & \text{(E16)} \end{aligned}$$

where y is fresh.

Proof. Here we go.

E13 $[\langle x \rangle := \perp]b \equiv b$. As above, it suffices to observe that the evaluation of b does not depend on the heap.

E14 $h, s \models [\langle x \rangle := \perp](e \hookrightarrow e')$
iff (semantics heap clear modality)
 $h[\langle s(x) \rangle := \perp], s \models e \hookrightarrow e'$
iff (semantics points-to)
 $s(e) \in \text{dom}(h[\langle s(x) \rangle := \perp])$ and $h[\langle s(x) \rangle := \perp](s(e)) = h(s(e)) = s(e')$
iff (semantics assertions)
 $h, s \models x \neq e \wedge e \hookrightarrow e'$

E15 $h, s \models [\langle x \rangle := \perp](p * q)$
iff (semantics heap clear modality)
 $h[\langle s(x) \rangle := \perp], s \models p * q$
iff (semantics separating conjunction)
 $h_1, s \models p$ and $h_2, s \models q$, for some h_1, h_2 such that $h[\langle s(x) \rangle := \perp] = h_1 \uplus h_2$
iff (semantics heap clear modality)
 $h_1, s \models [\langle x \rangle := \perp]p$ and $h_2, s \models [\langle x \rangle := \perp]q$, for some h_1, h_2 such that $h = h_1 \uplus h_2$.
Note: $h = h_1 \uplus h_2$ implies $h[\langle s(x) \rangle := \perp] = h_1[\langle s(x) \rangle := \perp] \uplus h_2[\langle s(x) \rangle := \perp]$, and, conversely, $h[\langle s(x) \rangle := \perp] = h_1 \uplus h_2$ implies there exists h'_1, h'_2 such that $h = h'_1 \uplus h'_2$ and $h_1 = h'_1[\langle s(x) \rangle := \perp]$ and $h_2 = h'_2[\langle s(x) \rangle := \perp]$.

E16 $h, s \models [\langle x \rangle := \perp](p \multimap q)$
iff (semantics heap clear modality)
 $h[s(x) := \perp], s \models p \multimap q$.
From here we proceed as follows. First we show that $h, s \models ((p \wedge x \not\hookrightarrow -) \multimap [\langle x \rangle := \perp]q)$ and $h, s \models \forall y([\langle x \rangle := y]p \multimap [\langle x \rangle := y]q)$ implies $h[s(x) := \perp], s \models p \multimap q$. Let h' be disjoint from $h[s(x) := \perp]$ and $h', s \models p$. We have to show that $h[s(x) := \perp] \uplus h', s \models q$. We distinguish the following two cases.

- First, let $s(x) \in \text{dom}(h')$. We then introduce $s' = s[y := h'(s(x))]$. We have $h', s' \models p$ (since y does not occur in p), so it follows by the semantics of the heap update modality that $h'[s(x) := \perp], s' \models [\langle x \rangle := y]p$. Since $h'[s(x) := \perp]$ and h are disjoint (which clearly follows from that h' and $h[s(x) := \perp]$ are disjoint), and since $h, s' \models [\langle x \rangle := y]p \multimap [\langle x \rangle := y]q$, we have that $h \uplus (h'[s(x) := \perp]), s' \models [\langle x \rangle := y]q$. Applying again the semantics of the heap update modality, we obtain $(h \uplus (h'[s(x) := \perp]))[s(x) :=$

$s'(y), s' \models q$. We then can conclude this case observing that y does not occur in q and that $h[s(x) := \perp] \uplus h' = (h \uplus (h'[s(x) := \perp]))[s(x) := s'(y)]$.

- Next, let $s(x) \notin \text{dom}(h')$. So h' and h are disjoint, and thus (since $h, s \models (p \wedge x \not\rightarrow -) \multimap [\langle x \rangle := \perp]q$) we have $h \uplus h', s \models [\langle x \rangle := \perp]q$. From which we derive $(h \uplus h')[s(x) := \perp], s \models q$ by the induction hypothesis. We then can conclude this case by the observation that $h[s(x) := \perp] \uplus h' = (h \uplus h')[s(x) := \perp]$.

Conversely, assuming $h[s(x) := \perp], s \models p \multimap q$, we first show that $h, s \models (p \wedge x \not\rightarrow -) \multimap [\langle x \rangle := \perp]q$ and then $h, s \models \forall y([\langle x \rangle := y]p \multimap [\langle x \rangle := y]q)$.

- Let h' be disjoint from h and $h', s \models p \wedge x \not\rightarrow -$. We have to show that $h \uplus h', s \models [\langle x \rangle := \perp]q$, that is, $(h \uplus h')[s(x) := \perp], s \models q$ (by the semantics of the heap clear update). Clearly, $h[s(x) := \perp]$ and h' are disjoint, and so $h[s(x) := \perp] \uplus h', s \models q$ follows from our assumption. We then can conclude this case by the observation that $(h \uplus h')[s(x) := \perp] = h[s(x) := \perp] \uplus h'$, because $s(x) \notin \text{dom}(h')$.
- Let h' be disjoint from h and $s' = s[y := n]$, for some n such that $h', s' \models [\langle x \rangle := y]p$. We have to show that $h \uplus h', s' \models [\langle x \rangle := y]q$. By the semantics of the heap update modality it follows that $h'[s(x) := n], s' \models p$, that is, $h'[s(x) := n], s \models p$ (since y does not occur in p). Since $h'[s(x) := n]$ and $h[s(x) := \perp]$ are disjoint, we derive from the assumption $h[s(x) := \perp], s \models p \multimap q$ that $h[s(x) := \perp] \uplus h'[s(x) := n], s \models q$. Again by the semantics of the heap update modality we have that $h \uplus h', s' \models [\langle x \rangle := y]q$ iff $(h \uplus h')[s(x) := n], s' \models q$ (that is, $(h \uplus h')[s(x) := n], s \models q$, because y does not occur in q). We then can conclude this case by the observation that $(h \uplus h')[s(x) := n] = h[s(x) := \perp] \uplus h'[s(x) := n]$. \square

We denote by \mathbf{E} the *rewrite system* obtained from the equivalences **E1-16** by orienting these equivalences from left to right, e.g., equivalence **E1** is turned into a rewrite rule $[S]\mathbf{false} \Rightarrow \mathbf{false}$. The following theorem states that the rewrite system \mathbf{E} is complete, that is, confluent and strongly normalizing. Its proof is straightforward (using standard techniques) and therefore omitted.

Theorem 3.6 (Completeness of \mathbf{E})

- **Normal form.** *Every standard formula p of SL is in normal form (which means that it cannot be reduced by the rewrite system \mathbf{E}).*
- **Local confluence.** *For any two reductions $p \Rightarrow q_1$ and $p \Rightarrow q_2$ (p a formula of DSL) there exists a DSL formula q such that $q_1 \Rightarrow q$ and $q_2 \Rightarrow q$.*
- **Termination.** *There does not exist an infinite chain of reductions $p_1 \Rightarrow p_2 \Rightarrow p_3 \cdots$.*

We now show an example of the interplay between the modalities for heap update and heap clear. We want to derive

$$\{\forall x((x \not\rightarrow -) \rightarrow p)\} x := \mathbf{cons}(0); \mathbf{dispose}(x) \{p\}$$

where statement $x := \mathbf{cons}(0); \mathbf{dispose}(x)$ simulates the so-called random assignment [9]: the program terminates with a value of x that is chosen non-deterministically. First we apply the axiom **E8** for deallocation to obtain

$$\{(x \hookrightarrow -) \wedge [\langle x \rangle := \perp]p\} \mathbf{dispose}(x) \{p\}.$$

Next, we apply the axiom **E8** for allocation to obtain

$$\begin{aligned} &\{\forall x((x \not\rightarrow -) \rightarrow [\langle x \rangle := 0]((x \hookrightarrow -) \wedge [\langle x \rangle := \perp]p))\} \\ &\quad x := \mathbf{cons}(0) \\ &\{(x \hookrightarrow -) \wedge p[\langle x \rangle := \perp]\}. \end{aligned}$$

Applying **E10** (after pushing the heap update modality inside), followed by some basic first-order reason-

ing, we can reduce $[\langle x \rangle := 0](\exists y(x \hookrightarrow y))$ to true. So we obtain

$$\begin{aligned} & \{\forall x((x \not\hookrightarrow -) \rightarrow [\langle x \rangle := 0][\langle x \rangle := \perp]p)\} \\ & \quad x := \mathbf{cons}(0) \\ & \{(x \hookrightarrow -) \wedge p[\langle x \rangle := \perp]\}. \end{aligned}$$

In order to proceed we formalize the interplay between the modalities for heap update and heap clear by the following general equivalence:

$$[\langle x \rangle := e][\langle x \rangle := \perp]p \equiv [\langle x \rangle := \perp]p$$

We then complete the proof by applying the sequential composition rule and consequence rule, using the above equivalence and the following axiomatization of the heap clear modality:

$$(x \not\hookrightarrow -) \wedge [\langle x \rangle := \perp]p \equiv (x \not\hookrightarrow -) \wedge p$$

The above axiomatization can be extended in the standard manner to a program logic for sequential while programs, see [9], which does not require the frame rule, nor any other adaptation rule besides the consequence rule. For recursive programs however one does need more adaptation rules: a further discussion about the use of the frame rule in a completeness proof for recursive programs is outside the scope of this paper.

4 Expressiveness DSL

In this section, we illustrate the expressiveness of DSL in a completeness proof of the local mutation axiom and a novel strongest postcondition axiomatization.

4.1 Completeness local axioms

We consider the completeness of the following local mutation axiom (completeness of the local axioms for the other standard basic instructions have already been established, as observed in the Introduction)

$$\{x \mapsto -\} [x] := e \{x \mapsto e\}$$

The proof itself does not make use of the separating implication.

Theorem 4.1 (Completeness local mutation axiom) *If $\models \{p\} [x] := e \{q\}$ then $\{p\} [x] := e \{q\}$ is derivable using the local mutation axiom, frame rule, and consequence rule.*

Proof. The problem here is how to compute a ‘frame’ r for a given valid specification $\{p\} [x] := e \{q\}$ so that p implies $(x \mapsto -) * r$ and $(x \mapsto e) * r$ implies q . We show here how the heap update modality can be used to describe such a frame. Let $\models \{p\} [x] := e \{q\}$ and r denote $\exists y([\langle x \rangle := y]p)$ for some fresh y . By the local axiom and the frame rule, we first derive

$$\{(x \mapsto -) * r\} [x] := e \{(x \mapsto e) * r\}.$$

Let $h, s \models p$. To prove that $h, s \models (x \mapsto -) * r$, it suffices to show that there exists a split $h = h_1 \uplus h_2$ such that $h_1, s \models (x \mapsto -)$ and $h_2, s[y := n] \models [\langle x \rangle := y]p$, for some n . Since $\models \{p\} [x] := e \{q\}$ we have that $s(x) \in \text{dom}(h)$. So we can introduce the split $h = h_1 \uplus h_2$ such that $h_1, s \models (x \mapsto -)$ and $h_2 = h[s(x) := \perp]$. By the semantics of the heap update modality it then suffices to observe that $h_2, s[y := h(s(x))] \models [\langle x \rangle := y]p$ if and only if $h_2[s(x) := h(s(x))], s \models p$ (y does not appear in p), that is, $h, s \models p$.

$$\begin{aligned}
& \{p\} x := e \{ \exists y ([x := y]p \wedge (x = e[y/x])) \} \\
& \{p \wedge (e \hookrightarrow -)\} x := [e] \{ \exists y ([x := y]p \wedge (x \hookrightarrow e[y/x])) \} \\
& \{p \wedge (x \hookrightarrow -)\} [x] := e \{ \exists y ([\langle x \rangle := y]p \wedge (x \hookrightarrow e)) \} \\
& \{p\} x := \mathbf{cons}(e) \{ [\langle x \rangle := \perp] (\exists x p) \wedge (x \hookrightarrow e) \} \\
& \{p \wedge (x \hookrightarrow -)\} \mathbf{dispose}(x) \{ \exists y ([\langle x \rangle := y]p \wedge (x \not\hookrightarrow -)) \}
\end{aligned}$$

Fig. 3. Strongest postcondition axioms of separation logic (SP-DSL), where y is fresh everywhere and x does not occur in e in case of $x := \mathbf{cons}(e)$.

On the other hand, we have that $(x \mapsto e) * r$ implies q : Let $h, s \models (x \mapsto e) * r$. So there exists a split $h = h_1 \uplus h_2$ such that $h_1, s \models x \mapsto e$ and $h_2, s \models r$. Let n be such that $h_2, s[y := n] \models [\langle x \rangle := y]p$. By the semantics of the heap update modality again we have that $h_2, s[y := n] \models [\langle x \rangle := y]p$ if and only if $h_2[s(x) := n], s \models p$ (here y does not appear in p). Since $\models \{p\} [x] := e \{q\}$ it then follows that $h_2[s(x) := s(e)], s \models q$, that is, $h, s \models q$ (note that $h = h_2[s(x) := s(e)]$ because $h(s(x)) = s(e)$ and $h_2 = h[s(x) := \perp]$). \square

4.2 Strongest postcondition axiomatization

Before we discuss a novel strongest postcondition axiomatization using the modalities of DSL, it should be noted that in general the semantics of program logics which require absence of certain failures gives rise to an asymmetry between weakest preconditions and strongest postconditions: For any statement S and postcondition q we have that $\models \{\mathbf{false}\} S \{q\}$. However, for any precondition p which does not exclude failures, there does not exist *any* postcondition q such that $\models \{p\} S \{q\}$. We solve this by simply requiring that the given precondition does not give rise to failures (see below).

Figure 3 contains our novel strongest postcondition axiomatization SP-DSL, where the main novelty is in the use of the heap update and heap clear modalities in the axiomatization of the mutation, allocation, and de-allocation instruction. It is worthwhile to contrast, for example, the use of the heap clear modality to express freshness in the strongest postcondition axiomatization of the allocation instruction with the following traditional axiom (assuming that x does not occur free in p):

$$\{p\} x := \mathbf{cons}(e) \{p * (x \mapsto e)\}$$

where freshness is enforced by the introduction of the separating conjunction (which as such increases the complexity of the postcondition). More specifically, we have the following instance of the allocation axiom in Figure 3 (also making use of that x does not appear in the precondition)

$$\{y \hookrightarrow 0\} x := \mathbf{cons}(1) \{ [\langle x \rangle := \perp] (y \hookrightarrow 0) \wedge (x \hookrightarrow 1) \}$$

Applying **E14** we obtain

$$\{y \hookrightarrow 0\} x := \mathbf{cons}(1) \{y \neq x \wedge (y \hookrightarrow 0) \wedge (x \hookrightarrow 1)\}$$

On the other hand, instantiating the above traditional axiom we obtain

$$\{y \hookrightarrow 0\} x := \mathbf{cons}(1) \{(y \hookrightarrow 0) * (x \mapsto 1)\}$$

which is implicit and needs unraveling the semantics of separating conjunction. Using the heap clear modality we thus obtain a basic assertion in predicate logic which provides an explicit but simple account of aliasing.

Theorem 4.2 (Soundness and completeness SP-DSL) *For any basic instruction S , we have $\models \{p\} S \{q\}$ if and only if $\{p\} S \{q\}$ is derivable from the axioms in SP-DSL (Figure 3) and (a single application of) the rule of consequence.*

Proof. We showcase the soundness and completeness of the strongest postcondition axiomatization of allocation (soundness and completeness of the strongest postconditions for the mutation and de-allocation instructions follow in a straightforward manner from the semantics of the heap update modality).

- $\models \{p\} x := \mathbf{cons}(e) \{[\langle x \rangle := \perp](\exists y([x := y]p)) \wedge x \hookrightarrow e\}$:
Let $h, s \models p$. We have to show that $h[n := s(e)], s[x := n] \models [\langle x \rangle := \perp](\exists y([x := y]p)) \wedge x \hookrightarrow e$, for $n \notin \text{dom}(h)$. By definition $h[n := s(e)], s[x := n] \models x \hookrightarrow e$. By the semantics of the heap clear modality and existential quantification, it then suffices to show that $h[n := \perp], s[x := n][y := s(x)] \models [x := y]p$, which by the semantics of the simple assignment modality boils down to $h, s[y := s(x)] \models p$ (note that $n \notin \text{dom}(h)$, that is, $h, s \models p$ (y does not appear in p), which holds by assumption).
- $\models \{p\} x := \mathbf{cons}(e) \{q\}$ implies
 $\models ([\langle x \rangle := \perp](\exists y(p[x := y])) \wedge x \hookrightarrow e) \rightarrow q$:
Let $h, s \models [\langle x \rangle := \perp](\exists y([x := y]p)) \wedge x \hookrightarrow e$. We have to show that $h, s \models q$. By the semantics of the heap clear modality we derive from the above assumption that $h[s(x) := \perp], s \models \exists y(p[x := y])$. Let $h[s(x) := \perp], s[y := n] \models p[x := y]$, for some n . It follows from the semantics of the simple assignment modality that $h[s(x) := \perp], s[x := n] \models p$ (y does not appear in p). Since $s(x) \notin \text{dom}(h[s(x) := \perp])$, we have that $\langle x := \mathbf{cons}(e), h[s(x) := \perp], s[x := n] \rangle \Rightarrow (h[s(x) := s[x := n](e)], s)$. Since we can assume without loss of generality that x does not occur in e we have that $s[x := n](e) = s(e)$, and so from the assumption that $h, s \models x \hookrightarrow e$ we derive that $h[s(x) := s[x := n](e)] = h$. From $\{p\} x := \mathbf{cons}(e) \{q\}$ then we conclude that $h, s \models q$. □

5 Extensions

A straightforward extension concerns the general mutation instruction $[e] := e'$, which allows the use of an arbitrary arithmetic expression e to denote the updated location. We can simulate this by the statement $x := e; [x] := e'$, where x is a fresh variable. Applying the modalities we derive the following axiom

$$\{(e \hookrightarrow -) \wedge [x := e][\langle x \rangle := e']p\} [e] := e' \{p\}$$

where x is a fresh variable.

Another straightforward extension concerns the allocation $x := \mathbf{cons}(e)$ in the case where x does occur in e . The instruction $x := \mathbf{cons}(e)$ can be simulated by $y := x; y := \mathbf{cons}(e[y/x])$ where y is a fresh variable. Applying the sequential composition rule and the axiom for basic assignments, it is straightforward to derive the following generalized backwards allocation axiom:

$$\{\forall y((y \not\rightarrow -) \rightarrow [y := x][\langle y \rangle := e[y/x]]p)\} x := \mathbf{cons}(e) \{p\}$$

where y is fresh.

Reynolds introduced in [19] the allocation instruction $x := \mathbf{cons}(\bar{e})$, which allocates a consecutive part of the memory for storing the values of \bar{e} : its semantics is described by

$$\langle x := \mathbf{cons}(\bar{e}), h, s \rangle \Rightarrow (h[\bar{m} := s(\bar{e})], s[x := m_1])$$

where $\bar{e} = e_1, \dots, e_n$, $\bar{m} = m_1, \dots, m_n$, $m_{i+1} = m_i + 1$, for $i = 1, \dots, n - 1$, $\{m_1, \dots, m_n\} \cap \text{dom}(h) = \emptyset$, and, finally,

$$h[\bar{m} := s(\bar{e})](k) = \begin{cases} h(k) & \text{if } k \notin \{m_1, \dots, m_n\} \\ s(e_i) & \text{if } k = m_i \text{ for some } i = 1, \dots, n. \end{cases}$$

Let \bar{e}' denote a sequence of expressions e'_1, \dots, e'_n such that e'_1 denotes the variable x and e'_i denotes the expression $x + (i - 1)$, for $i = 2, \dots, n$. The storage of the values of e_1, \dots, e_n then can be modeled by a sequence of heap update modalities $[\langle e'_i \rangle := e_i]$, for $i = 1, \dots, n$. We abbreviate such a sequence by $[\langle \bar{e}' \rangle := \bar{e}]$. Assuming that x does not occur in one of the expressions \bar{e} (this restriction can be lifted as described above), we have the following generalization of the above backwards allocation axiom

$$\{\forall x(\left(\bigwedge_{i=1}^n (e'_i \not\rightarrow -)\right) \rightarrow [\langle \bar{e}' \rangle := \bar{e}]p)\} x := \mathbf{cons}(\bar{e}) \{p\}$$

Recursive predicates

Next we illustrate the extension of our approach to recursive predicates for reasoning about a linked list. Assuming a set of user-defined predicates $r(x_1, \dots, x_n)$ of arity n , we introduce corresponding basic assertions $r(e_1, \dots, e_n)$ which are interpreted by (the least fixed point of) a system of recursive predicate definitions $r(x_1, \dots, x_n) := p$, where the user-defined predicates only occur positively in p .

If for any recursive definition $r(x_1, \dots, x_n) := p$ only the formal parameters x_1, \dots, x_n occur free in p , we can simply define $[x := e]r(e_1, \dots, e_n)$ by $r(e_1[e/x], \dots, e_n[e/x])$. However, allowing global variables in recursive predicate definitions does affect the interpretation of these definitions. As a very simple example, given $r(y) := x = 1$, clearly $\{r(y)\} x := 0 \{r(y)\}$ is invalid (and so we cannot simply define $[x := 0]r(y)$ by $r(y[0/x])$). Furthermore, substituting the parameters of r clearly does not make sense for modalities with heap modifications (such as mutation, allocation, etc.): as subformulas may depend on the heap, these may require alias analysis *in the definition of r* .

We illustrate how our dynamic logic works with recursively defined predicates on a characteristic linked list example. In particular, let r be the recursively defined *reachability* predicate

$$r(x, y) := x = y \vee \exists z((x \mapsto z) * r(z, y)).$$

We shall prove $\{r(\mathit{first}, y)\} \mathit{first} := \mathbf{cons}(\mathit{first}) \{r(\mathit{first}, y)\}$. To do so, we model $\mathit{first} := \mathbf{cons}(\mathit{first})$ by $u := \mathit{first}; \mathit{first} := \mathbf{cons}(u)$, for some fresh variable u . Thus it is sufficient to show

$$\{r(\mathit{first}, y)\} u := \mathit{first}; \mathit{first} := \mathbf{cons}(u) \{r(\mathit{first}, y)\}.$$

We first calculate the weakest precondition of the last assignment: $[\mathit{first} := \mathbf{cons}(u)]r(\mathit{first}, y)$. Using equivalence (E7) of Lemma 3.3 we obtain $\forall \mathit{first}((\mathit{first} \not\rightarrow -) \rightarrow [\langle \mathit{first} \rangle := u]r(\mathit{first}, y))$.

Next, we simplify the modal subformula $[\langle \mathit{first} \rangle := u]r(\mathit{first}, y)$ we first unfold the definition of r , obtaining $\mathit{first} = y \vee \exists z((\mathit{first} \mapsto z) * r(z, y))$. By Lemma 3.4 (E11), $[\langle \mathit{first} \rangle := u](\mathit{first} \mapsto z * r(z, y))$ reduces to the disjunction of $(\mathit{first} \mapsto z \wedge \mathit{first} \not\rightarrow -) * [\langle \mathit{first} \rangle := u]r(z, y)$ and $[\langle \mathit{first} \rangle := u](\mathit{first} \mapsto z) * (r(z, y) \wedge \mathit{first} \not\rightarrow -)$. In the first disjunct, the left-hand side of the separating conjunction asserts that first is allocated (and points to z) and that simultaneously first is not allocated. This clearly is false in every heap, so that whole disjunct reduces to **false**. Simplifying the second disjunct (reducing the modality with equivalence (E10) of Lemma 3.4) and applying standard logical equivalences, yields that the whole subformula is equivalent to

$$\mathit{first} = y \vee (r(u, y) \wedge (\mathit{first} \not\rightarrow -)).$$

Applying the allocation axiom and an application of the consequence rule, we obtain

$$\begin{aligned} &\{\forall \mathit{first}((\mathit{first} \not\rightarrow -) \rightarrow (\mathit{first} = y \vee r(u, y)))\} \\ &\quad \mathit{first} := \mathbf{cons}(u) \\ &\quad \{r(\mathit{first}, y)\}. \end{aligned}$$

Renaming *first* by the fresh variable *f* does not affect *r*, so

$$\begin{aligned} & \{\forall f((f \not\leftrightarrow -) \rightarrow (f = y \vee r(u, y)))\} \\ & \text{first} := \mathbf{cons}(u) \\ & \{r(\text{first}, y)\} \end{aligned}$$

can be derived. Also substituting *u* for *first* does not affect the definition of *r*. It then suffices to observe that $r(\text{first}, y)$ (trivially) implies $\forall f((f \not\leftrightarrow -) \rightarrow (f = y \vee r(\text{first}, y)))$.

6 Formalization in Coq

The main motivation behind formalizing results in a proof assistant is to rigorously check hand-written proofs. For our formalization we used the dependently-typed calculus of inductive constructions as implemented by the Coq proof assistant. We have used no axioms other than the axiom of function extensionality (for every two functions *f, g* we have that $f = g$ if $f(x) = g(x)$ for all *x*). This means that we work with an underlying intuitionistic logic: we have not used the axiom of excluded middle for reasoning classically about propositions. However, the decidable propositions (propositions *P* for which the excluded middle $P \vee \neg P$ can be proven) allow for a limited form of classical reasoning.

We formalize the basic instructions of our programming language (assignment, look-up, mutation, allocation, and deallocation) and the semantics of basic instructions. For Boolean and arithmetic expressions we use a shallow embedding, so that those expressions can be directly given as a Coq term of the appropriate type (with a coincidence condition assumed, i.e. that values of expressions depend only on finitely many variables of the store).

There are two approaches in formalizing the semantics of assertions: shallow and deep embedding. We have taken both approaches. In the first approach, the shallow embedding of assertions, we define assertions of DSL by their extension of satisfiability (i.e. the set of heap and store pairs in which the assertion is satisfied), that must satisfy a coincidence condition (assertions depend only on finitely many variables of the store) and a stability condition (see below). The definition of the modality operator follows from the semantics of programs, which includes basic control structures such as the **while**-loop. In the second approach, the deep embedding of assertions, assertions are modeled using an inductive type and we explicitly introduce two meta-operations on assertions that capture the heap update and heap clear modality. We have omitted the clauses for **emp** and $(e \mapsto e')$, since these could be defined as abbreviations, and we restrict to the basic instructions.

In the deep embedding we have no constructor corresponding to the program modality $[S]p$. Instead, two meta-operations denoted $p[\langle x \rangle = e]$ and $p[\langle x \rangle := \perp]$ are defined recursively on the structure of *p*. Crucially, we formalized and proven the following lemmas (the details are almost the same as showing the equivalences hold in the shallow embedding, Lemmas 3.4 and 3.5):

Lemma 6.1 (Heap update substitution lemma) $h, s \models p[\langle x \rangle := e]$ iff $h[s(x) := s(e)], s \models p$.

Lemma 6.2 (Heap clear substitution lemma) $h, s \models p[\langle x \rangle := \perp]$ iff $h[s(x) := \perp], s \models p$.

By also formalizing a deep embedding, we show that the modality operator can be defined entirely on the meta-level by introducing meta-operations on formulas that are recursively defined by the structure of assertions: this captures Theorem 3.6. The shallow embedding, on the other hand, is easier to show that our approach can be readily extended to complex programs including **while**-loops.

In both approaches, the semantics of assertions is classical, although we work in an intuitionistic meta-logic. We do this by employing a double negation translation, following the set-up by R. O’Connor [14]. In particular, we have that our satisfaction relation $h, s \models p$ is stable, i.e. $\neg\neg(h, s \models p)$ implies $h, s \models p$. This allows us to do classical reasoning on the image of the higher-order semantics of our assertions.

The source code of our formalization is accompanied with this paper as a digital artifact (which includes the files `shallow/Language.v` and `shallow/Proof.v`, and the files `deep/Heap.v`, `deep/Language.v`,

deep/Classical.v). The artifact consists of the following files:

- `shallow/Language.v`: Provides a shallow embedding of Boolean expressions and arithmetic expressions, and a shallow embedding of our assertion language, as presented in the prequel.
- `shallow/Proof.v`: Provides proof of the equivalences (**E1-16**), and additionally standard equivalences for modalities involving complex programs.
- `deep/Heap.v`: Provides an axiomatization of heaps as partial functions.
- `deep/Language.v`: Provides a shallow embedding of Boolean expressions and arithmetic expressions, and a deep embedding of our assertion language, on which we inductively define the meta operations of heap update and heap clear. We finally formalize Hoare triples and proof systems using weakest precondition and strongest postcondition axioms for the basic instructions.
- `deep/Classical.v`: Provides the classical semantics of assertions, and the strong partial correctness semantics of Hoare triples. Further it provides proofs of substitution lemmas corresponding to our meta-operators. Finally, it provides proofs of the soundness and completeness of the aforementioned proof systems.

7 Conclusion and related work

To the best of our knowledge no other works exist that study dynamic logic extensions of SL. We have shown how we can combine the standard programming logics in SL with a new DSL axiomatization of both weakest preconditions and strongest postconditions. These new axiomatizations in DSL have the so-called property of *gracefulness*:⁶ any first-order postcondition gives rise to a first-order weakest precondition (for any basic instruction). A property that existing axiomatizations of SL, such as given by C. Bannister, P. Höfner and G. Klein [3], and M. Faisal Al Ameen and M. Tatsuta [8], lack. (See also [21].) As a simple example, in our approach $[[x] := 0](y \leftrightarrow z)$ can be resolved to the first-order formula

$$(x \leftrightarrow -) \wedge ((y = x \wedge z = 0) \vee (y \neq x \wedge y \leftrightarrow z))$$

by applying the above equivalences **E6** and **E10**. The standard rule for backwards reasoning in [19] however gives the weakest precondition:

$$(x \mapsto -) * ((x \mapsto 0) -* (y \leftrightarrow z))$$

Despite their different formulations, both formulas characterize $[[x] := 0](y \leftrightarrow z)$, and thus must be equivalent. In fact, the equivalence has been proven in our Coq formalization (Section 6). Surprisingly, this however exceeds the capability of all the automated SL provers in the benchmark competition for SL [20]. In particular, only the CVC4-SL tool [17] supports the fragment of SL that includes the separating implication connective. However, from our own experiments with that tool, we found that it produces an incorrect counter-example and reported this as a bug to one of the maintainers of the project [16]. In fact, the latest version, CVC5-SL, reports the same input as ‘unknown’, indicating that the tool is incomplete. Furthermore, we have investigated whether the equivalence of these formulas can be proven in an interactive tool for reasoning about SL: the Iris project [11]. However, also in that system it is not possible to show the equivalence of these assertions, at least not without adding additional axioms to its underlying model [12]. On the other hand, the equivalence between the above two formulas can be expressed in quantifier-free separation logic, for which a complete axiomatization of all valid formulas has been given in [6].

In general, the calculation of $[S]p$ by means of a compositional analysis of p , in contrast with the standard approach, does not generate additional *nesting* of the separating connectives. On the other

⁶ The term ‘graceful’, coined by J.C. Blanchette [22], comes from higher-order automated theorem proving where it means that a higher-order prover does not perform significantly worse on first-order problems than existing first-order provers that lack the ability to reason about higher-order problems.

hand, the compositional analysis generates a case distinction in the definitions of $[\langle x \rangle := e](p * q)$ and $[\langle x \rangle := \perp](p -* q)$. How the combined application of the two approaches works in practice needs to be further investigated. Such an investigation will also involve the use of the modalities for the basic instructions in the generation of the verification conditions of a program (as is done for example in the KeY tool [1] for the verification of Java programs), which allows to *postpone* and *optimize* their actual application. For example, the equivalence

$$[x := e][\langle y \rangle := e']p \equiv [\langle y \rangle := e'[e/x]][x := e]p$$

allows to resolve the simple assignment modality by ‘pushing it inside’.

Other works that investigate weakest preconditions in SL are briefly discussed below. For example, [3] investigates both weakest preconditions and strongest postconditions in SL, also obtained through a transformational approach. However, the transformation uses other separating connectives (like *septraction*), and thus is not graceful. On the other hand, in [13] an alternative logic is introduced which, instead of the separating connectives, extends standard first-order logic with an operator $Sp(p)$ which captures the parts of the heap the (first-order) formula p depends on. Thus also [13] goes beyond first-order, and is not graceful. But the main motivation of that work coincides with ours: avoiding unnecessary reasoning about the separating connectives.

Our artifact formalizes the syntax and semantics of programs and assertions of SL. We plan to further extend our formalization to support practical program verification, and investigate how to integrate our approach in Iris [11]: we will consider how DSL can also work for a shallow embedding of SL. Then the generated verification conditions require a proof of the validity of corresponding assertions in SL, which can be discharged by providing a proof directly in Coq. Further, we will investigate the application of DSL to concurrent SL [4] and permission-based SL [2].

References

- [1] Ahrendt, W., B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt and M. Ulbrich, editors, *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*, Springer (2016), ISBN 978-3-319-49811-9.
<https://doi.org/10.1007/978-3-319-49812-6>
- [2] Amighi, A., C. Hurlin, M. Huisman and C. Haack, *Permission-based separation logic for multithreaded java programs*, *Logical Methods in Computer Science* **11** (2015).
[https://doi.org/10.2168/LMCS-11\(1:2\)2015](https://doi.org/10.2168/LMCS-11(1:2)2015)
- [3] Bannister, C., P. Höfner and G. Klein, *Backwards and forwards with separation logic*, in: J. Avigad and A. Mahboubi, editors, *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10895 of *Lecture Notes in Computer Science*, pages 68–87, Springer (2018).
https://doi.org/10.1007/978-3-319-94821-8_38
- [4] Brookes, S. and P. W. O’Hearn, *Concurrent separation logic*, *ACM SIGLOG News* **3**, pages 47–65 (2016).
<https://dl.acm.org/doi/10.1145/2984450.2984457>
- [5] Charguéraud, A., *Separation logic for sequential programs (functional pearl)*, *Proc. ACM Program. Lang.* **4** (2020).
<https://doi.org/10.1145/3408998>
- [6] Demri, S., É. Lozes and A. Mansutti, *A complete axiomatisation for quantifier-free separation logic*, *Log. Methods Comput. Sci.* **17** (2021).
[https://doi.org/10.46298/lmcs-17\(3:17\)2021](https://doi.org/10.46298/lmcs-17(3:17)2021)
- [7] Dijkstra, E., *A Discipline of Programming*, Prentice-Hall (1976), ISBN 978-0132158718.
- [8] Faisal Al Ameen, M. and M. Tatsuta, *Completeness for recursive procedures in separation logic*, *Theoretical Computer Science* **631**, pages 73–96 (2016), ISSN 0304-3975.
<https://doi.org/https://doi.org/10.1016/j.tcs.2016.04.004>
- [9] Harel, D., *First-Order Dynamic Logic*, volume 68 of *Lecture Notes in Computer Science*, Springer (1979), ISBN: 978-3-540-09237-7.

- [10] Ishtiaq, S. S. and P. W. O’Hearn, *BI as an assertion language for mutable data structures*, SIGPLAN Not. **36**, page 14–26 (2001), ISSN 0362-1340.
<https://doi.org/10.1145/373243.375719>
- [11] Jung, R., R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal and D. Dreyer, *Iris from the ground up: A modular foundation for higher-order concurrent separation logic*, Journal of Functional Programming **28** (2018).
<https://doi.org/10.1017/S0956796818000151>
- [12] Krebbers, R., Personal communication.
- [13] Murali, A., L. Peña, C. Löding and P. Madhusudan, *A first-order logic with frames*, in: P. Müller, editor, *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, volume 12075 of *Lecture Notes in Computer Science*, pages 515–543, Springer (2020).
<https://doi.org/10.1145/3583057>
- [14] O’Connor, R., *Classical mathematics for a constructive world*, Mathematical Structures in Computer Science **21**, pages 861–882 (2011).
<https://doi.org/10.1017/S0960129511000132>
- [15] O’Hearn, P. W., J. C. Reynolds and H. Yang, *Local reasoning about programs that alter data structures*, in: L. Fribourg, editor, *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19, Springer (2001).
https://doi.org/10.1007/3-540-44802-0_1
- [16] Reynolds, A., Personal communication.
- [17] Reynolds, A., R. Iosif, C. Serban and T. King, *A decision procedure for separation logic in smt*, in: *International Symposium on Automated Technology for Verification and Analysis*, pages 244–261, Springer (2016).
https://doi.org/10.1007/978-3-319-46520-3_16
- [18] Reynolds, J. C., *Intuitionistic reasoning about shared mutable data structure*, Millennial perspectives in computer science **2**, pages 303–321 (2000).
<https://citeseer.ist.psu.edu/viewdoc/download;jsessionid=9391C33E31002EA5799A75C653135CEA?doi=10.1.1.11.5999&rep=rep1>
- [19] Reynolds, J. C., *Separation logic: A logic for shared mutable data structures*, in: *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74, IEEE Computer Society (2002).
<https://doi.org/10.1109/LICS.2002.1029817>
- [20] Sighireanu, M., J. A. Navarro Pérez, A. Rybalchenko, N. Gorogiannis, R. Iosif, A. Reynolds, C. Serban, J. Katelaan, C. Matheja, T. Noll *et al.*, *Sl-comp: competition of solvers for separation logic*, in: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 116–132, Springer (2019).
https://doi.org/10.1007/978-3-030-17502-3_8
- [21] Tatsuta, M., W.-N. Chin and M. F. Al Ameen, *Completeness and expressiveness of pointer program verification by separation logic*, Information and Computation **267**, pages 1–27 (2019), ISSN 0890-5401.
<https://doi.org/https://doi.org/10.1016/j.ic.2019.03.002>
- [22] Vukmirović, P., J. Blanchette, S. Cruanes and S. Schulz, *Extending a brainiac prover to lambda-free higher-order logic*, International Journal on Software Tools for Technology Transfer **24**, pages 67–87 (2022).
<https://doi.org/10.1007/s10009-021-00639-7>

A Appendix

$$\begin{aligned}
S ::= & x := [e] \mid [x] := e \mid x := \mathbf{cons}(e) \mid \mathbf{dispose}(x) \mid \\
& x := e \mid S; S \mid \mathbf{if } b \mathbf{ then } S \mathbf{ else } S \mathbf{ fi} \mid \mathbf{while } b \mathbf{ do } S \mathbf{ od} \\
\langle x := e, h, s \rangle \Rightarrow & (h, s[x := s(e)]), \\
\langle x := [e], h, s \rangle \Rightarrow & (h, s[x := h(s(e))]) \text{ if } s(e) \in \mathit{dom}(h), \\
\langle x := [e], h, s \rangle \Rightarrow & \mathbf{fail} \text{ if } s(e) \notin \mathit{dom}(h), \\
\langle [x] := e, h, s \rangle \Rightarrow & (h[s(x) := s(e)], s) \text{ if } s(x) \in \mathit{dom}(h), \\
\langle [x] := e, h, s \rangle \Rightarrow & \mathbf{fail} \text{ if } s(x) \notin \mathit{dom}(h), \\
\langle x := \mathbf{cons}(e), h, s \rangle \Rightarrow & (h[n := s(e)], s[x := n]) \text{ where } n \notin \mathit{dom}(h). \\
\langle \mathbf{dispose}(x), h, s \rangle \Rightarrow & (h[s(x) := \perp], s) \text{ if } s(x) \in \mathit{dom}(h), \\
\langle \mathbf{dispose}(x), h, s \rangle \Rightarrow & \mathbf{fail} \text{ if } s(x) \notin \mathit{dom}(h), \\
\langle S_1; S_2, h, s \rangle \Rightarrow o & \text{ if } \langle S_1, h, s \rangle \Rightarrow (h', s') \text{ and } \langle S_2, h', s' \rangle \Rightarrow o, \\
\langle S_1; S_2, h, s \rangle \Rightarrow & \mathbf{fail} \text{ if } \langle S_1, h, s \rangle \Rightarrow \mathbf{fail}, \\
\langle \mathbf{if } b \mathbf{ then } S_1 \mathbf{ else } S_2 \mathbf{ fi}, h, s \rangle \Rightarrow o & \text{ if } s(b) = \mathbf{true} \text{ and } \langle S_1, h, s \rangle \Rightarrow o, \\
\langle \mathbf{if } b \mathbf{ then } S_1 \mathbf{ else } S_2 \mathbf{ fi}, h, s \rangle \Rightarrow o & \text{ if } s(b) = \mathbf{false} \text{ and } \langle S_2, h, s \rangle \Rightarrow o, \\
\langle \mathbf{while } b \mathbf{ do } S \mathbf{ od}, h, s \rangle \Rightarrow o & \text{ if } s(b) = \mathbf{true}, \langle S, h, s \rangle \Rightarrow (h', s'), \text{ and } \langle \mathbf{while } b \mathbf{ do } S \mathbf{ od}, h', s' \rangle \Rightarrow o, \\
\langle \mathbf{while } b \mathbf{ do } S \mathbf{ od}, h, s \rangle \Rightarrow & \mathbf{fail} \text{ if } s(b) = \mathbf{true} \text{ and } \langle S, h, s \rangle \Rightarrow \mathbf{fail}, \\
\langle \mathbf{while } b \mathbf{ do } S \mathbf{ od}, h, s \rangle \Rightarrow & (h, s) \text{ if } s(b) = \mathbf{false}.
\end{aligned}$$

Fig. A.1. Syntax and semantics of heap manipulating programs.

$$\begin{array}{c}
\frac{\frac{\frac{\frac{\frac{\frac{\vdash p \rightarrow p' \quad \{p'\} S \{q'\} \quad \vdash q' \rightarrow q}{\{p\} S \{q\}}}{\{p\} S_1 \{r\} \quad \{r\} S_2 \{q\}}}{\{p\} S_1; S_2 \{q\}}}{\{p \wedge b\} S_1 \{q\} \quad \{p \wedge \neg b\} S_2 \{q\}}}{\{p\} \mathbf{if } b \mathbf{ then } S_1 \mathbf{ else } S_2 \mathbf{ fi} \{q\}}}{\{p \wedge b\} S \{p\}}}{\{p\} \mathbf{while } b \mathbf{ do } S \mathbf{ od} \{p \wedge \neg b\}}
\end{array}$$

Fig. A.2. Hoare's standard proof rules.

$$\begin{aligned}
& \{p\} x := e \{ \exists y (p[x := y] \wedge x = e[x := y]) \} \\
& \{p \wedge (e \hookrightarrow -)\} x := [e] \{ (e \mapsto x) * \neg((e \mapsto x) -* \neg p) \} \\
& \{p \wedge (x \hookrightarrow -)\} [x] := e \{ (x \mapsto e) * \neg((x \mapsto e) -* \neg p) \} \\
& \{p\} x := \mathbf{cons}(e) \{ (x \mapsto e) * p \} \\
& \{p \wedge (x \hookrightarrow -)\} \mathbf{dispose}(x) \{ \neg((x \mapsto -) -* \neg p) \}
\end{aligned}$$

Fig. A.3. Global strongest postcondition axiomatization (cf. [19,3]), assuming x does not occur in e in the axioms for look-up, mutation, and allocation.

This figure "xy-picdiag.png" is available in "png" format from:

<http://arxiv.org/ps/2309.08962v2>