Separation Logic in the Light of Substitutions

***Abstract*—The overall contribution of this paper is a new foundation of separation logic based on novel substitutions which yield weakest preconditions and strongest postconditions for all four types of the basic instructions of separation logic: look-up, mutation, allocation, and deallocation. Our substitutions describe the effect of the instructions compositionally in terms of the logical structure of a given pre/postcondition. As a consequence, in contrast to the standard axiomatization of separation logic, our substitutions do not generate additional complexity measured by the maximal depth of nested separating conjunctions and separating implications.**

I. Introduction

J.C. Reynolds [1], S.S. Ishtiaq and P.W. O'Hearn [2], and H. Yang [3] introduced separation logic for reasoning about invariant properties of heap manipulating program involving dynamic memory allocation by means of the following rule (also called the 'frame rule' [4]):

$$\frac{\{p\}\ S\ \{q\}}{\{p * r\}\ S\ \{q * r\}}$$

where the so-called separating conjunction (denoted by '$*$') allows to split the *heap*, that is, the set of allocated memory locations and their contents, into two disjoint parts which satisfy the corresponding conjuncts. Separation logic further features the basic heap assertion $(x \mapsto e)$, 'x points to e', which expresses that x denotes the single allocated memory location which stores the value of e. The above frame rule then can be used to generate incrementally a global description of the heap from assignment axioms which describe the local changes, e.g., the mutation axiom

$$\{\exists y(x \mapsto y)\}\ [x] := e\ \{x \mapsto e\}$$

where $[x] := e$ assigns the value of e to the location denoted by x. The result of this assignment is directly expressed by $(x \mapsto e)$, assuming that x denotes the single allocated location, as expressed by the precondition.

This overall bottom-up (or local) approach of separation logic contrasts with the overall top-down (or global) approach of Hoare logic, which allows to reduce a global specification incrementally to local verification conditions. In order to define a weakest precondition axiomatization of the basic instructions in separation logic, the separating implication $(p \mathrel{-\!\!*} q)$ has been introduced which, roughly, holds if every extension of the heap satisfies q, whenever p holds for the extension itself (separately). This implication allows for the following mutation axiom

$$\{(\exists y(x \mapsto y)) * ((x \mapsto e) \mathrel{-\!\!*} p)\}\ [x] := e\ \{p\}$$

which thus also supports backwards reasoning. Note that the introduction of both separating connectives increases the complexity of the generated precondition measured by the maximal depth of nested alternations between separating conjunction and separating implication (compared to that of the given postcondition). Repeated use of the axiom, e.g. in a sequential composition, thus generates increasingly complex preconditions because of the (implicit) second-order quantification over heaps. Note that already nested first-order quantifiers are a major difficulty in proof automation and interactive theorem proving. On the theoretical side, in the arithmetical hierarchy each quantifier alternation introduces a provably higher complexity class. On the practical side, quantifier elimination requires guessing the right witnesses over an infinite choice of terms that, in general, might not occur as sub-terms in the given formula. See for example recent papers on SMT solvers (e.g., [5], [6]).

We observe here an interesting analogy between the separating implication $((x \mapsto e) \mathrel{-\!\!*} p)$ and the corresponding formula $[[x] := e]p$ in dynamic logic [7]. A main difference though is that the basic modalities in dynamic logic can be reduced by corresponding substitutions, whereas for separation logic no such substitution have been defined. Quoting Reynolds [1]:

> ... expressions do not contain notations, such as $[-]$, that refer to the heap. It follows that none of the new heap manipulating [instructions] are instances of the simple assignment [instruction]. *In fact, they will not obey Hoare's inference rule for assignment.*

(Emphasis ours.) This seems to suggest that there are no such substitutions in separation logic.

The overall contribution of this paper is a new foundation of separation logic, based on substitutions which

- describes the effect of the basic heap manipulating instructions compositionally in terms of the logical structure of the given postcondition,
- provides a direct account of aliasing in terms of basic equational predicate logic, and
- does not generate additional complexity measured by the maximal depth of nested alternations between separating conjunction and implication.

We apply our substitutions in the definition of novel weakest precondition and strongest postcondition axiomatizations of the basic instructions in separation logic: look-up, mutation, allocation, and deallocation. A consequence of the latter point is that our axiomatizations have the so-called property of *grace-*

fulness[1], i.e. our generated weakest precondition/strongest postcondition with respect to a first-order post/precondition remains first-order, a property that all existing axiomatizations of separation logic, such as given by C. Bannister, P. Höfner and G. Klein [9], and M. Faisal Al Ameen and M. Tatsuta [10], lack. Moreover, our generated weakest precondition/strongest postconditions do not contain the separating implication when the given post/precondition also does not contain separating implication. Using the proof assistant Coq, we have formally verified the correctness of the substitution operations, and the basic soundness and completeness of our axiomatizations. All our results can be readily extended to a programming language involving (sequential) control structures such as loops.

II. Separation logic: syntax and semantics

We follow the presentation of separation logic in [1]. A heap[2] h is represented by a finitely-based *partial* function $\mathbb{Z} \rightharpoonup \mathbb{Z}$ and the domain of h is denoted by $dom(h)$. We write $h(n) = \bot$ if $n \notin dom(h)$. The heaps h, h' are disjoint iff $dom(h) \cap dom(h') = \emptyset$. 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, $dom(h) = dom(h_1) \cup dom(h_2)$, and $h(n) = h_i(n)$ if $n \in dom(h_i)$ 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 $V \to \mathbb{Z}$. We abstract from the syntax of arithmetic expressions, which we denote by e, and Boolean expressions, which we denote by b. By $var(e)$ (resp. $var(b)$) 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. 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 [1] 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 $n \in dom(h)$. More specifically, we have

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

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

Following [1], we have the following basic instructions: $x := e$ (simple assignment), $x := [e]$ (look-up), $[x] := e$ (mutation), $x := \textbf{cons}(e)$ (allocation), $\textbf{dispose}(x)$ (deallocation). Just like [2], "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 $\langle S, h, s\rangle \Rightarrow \textbf{fail}$ denotes a failing execution (e.g. due to access of a 'dangling pointer'). See Figure 1 for their semantics (and see Appendix, Figure 5, for the full syntax and semantics).

$\langle x := e, h, s\rangle \Rightarrow (h, s[x := s(e)])$,
$\langle x := [e], h, s\rangle \Rightarrow (h, s[x := h(s(e))])$ if $s(e) \in dom(h)$,
$\langle x := [e], h, s\rangle \Rightarrow \textbf{fail}$ if $s(e) \notin dom(h)$,
$\langle [x] := e, h, s\rangle \Rightarrow (h[s(x) := s(e)], s)$ if $s(x) \in dom(h)$,
$\langle [x] := e, h, s\rangle \Rightarrow \textbf{fail}$ if $s(x) \notin dom(h)$,
$\langle x := \textbf{cons}(e), h, s\rangle \Rightarrow (h[n := s(e)], s[x := n])$
  if $n \notin dom(h)$.
$\langle \textbf{dispose}(x), h, s\rangle \Rightarrow (h[s(x) := \bot], s)$ if $s(x) \in dom(h)$,
$\langle \textbf{dispose}(x), h, s\rangle \Rightarrow \textbf{fail}$ if $s(x) \notin dom(h)$.

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

$h, s \models b$ iff $s(b) = \textbf{true}$,
$h, s \models (e \hookrightarrow e')$ iff $s(e) \in dom(h)$ and $h(s(e)) = s(e')$,
$h, s \models (p \wedge q)$ iff $h, s \models p$ and $h, s \models q$,
$h, s \models (p \to q)$ 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 $h = h_1 \uplus h_2$,
$h, s \models (p \mathrel{-\!\!*} q)$ iff $h', s \models p$ implies $h'', s \models q$
  for all h', h'' such that $h'' = h \uplus h'$.

Fig. 2. Semantics of separation logic assertions.

We follow [2] in the definition of the syntax and semantics of the assertion language of separation logic, but we use a different atomic 'weak points to' formula (as in [11]):

$$p ::= b \mid (e \hookrightarrow e) \mid (p \wedge p) \mid (p \to p) \mid \forall x p \mid (p * p) \mid (p \mathrel{-\!\!*} p)$$

By $h, s \models p$ we denote the truth relation of (classical) separation logic (see Figure 2), and validity of p is denoted by $\models p$. We further define the following abbreviations: $\neg p$ denotes $(p \to \textbf{false})$, $(p \vee q)$ denotes $\neg(\neg p \wedge \neg q)$, $\exists x p$ denotes $\neg\forall x(\neg p)$, $(e \hookrightarrow -)$ denotes $\exists x(e \hookrightarrow x)$ for a fresh x, **emp** denotes $\forall x(x \not\hookrightarrow -)$, and $(e \mapsto e')$ denotes $(e \hookrightarrow e') \wedge (\forall x((x \hookrightarrow -) \to x = e))$ 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 $h, s \models \textbf{emp}$ iff $dom(h) = \emptyset$, and $h, s \models (e \mapsto e')$ iff $dom(h) = \{s(e)\}$ and $h(s(e)) = s(e')$. An assertion is *first-order* if its construction does not involve separating connectives.

The assertion $(e \hookrightarrow e')$ is implied by $(e \mapsto e')$, and to express the latter using the former requires the use of a separating conjunction (i.e. $(e \hookrightarrow e')$ is equivalent to $\textbf{true} * (e \mapsto e')$), whereas our definition of $(e \mapsto e')$ is first-order, and thus we use $(e \hookrightarrow e')$ as our atomic assertion.

A specification $\{p\}\ S\ \{q\}$ is a triple that consists of a precondition p, a program S, and a postcondition q. Specifications

[1]The term 'graceful', coined by J.C. Blanchette [8], 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.

[2]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.

$$\{p[x := e]\}\ x := e\ \{p\}$$

$$\{\exists y((e \hookrightarrow y) \land p[x := y])\}\ x := [e]\ \{p\}$$

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

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

$$\{(x \hookrightarrow -) \land p[\langle x\rangle := \bot]\}\ \textbf{dispose}(x)\ \{p\}$$

Fig. 3. Weakest precondition axioms (WP-SL), where y fresh and x does not occur in e in the case of $x := \textbf{cons}(e)$.

are interpreted in the sense of strong partial correctness, which ensures absence of explicit failure. Formally, following [1], the validity of a specification, denoted $\models \{p\}\ S\ \{q\}$, is defined as: if $h, s \models p$, then $\langle S, h, s\rangle \not\Rightarrow$ **fail** and also $\langle S, h, s\rangle \Rightarrow (h', s')$ implies $h', s' \models q$ for all h', s'.

III. Weakest precondition axiomatization

Figure 3 contains our novel weakest precondition axiomatization WP-SL. See the appendix, Figure 6 for the standard proof rules of Hoare [12], and Figure 7 for the existing weakest precondition axiomatization involving separating connectives [1], [9], [10]. For technical convenience only, we require in the axiom for $x := \textbf{cons}(e)$ that x does not appear in e (see Section VI to lift this restriction). Below we define and prove the correctness of the substitutions used in the axiomatization. We conclude this section with a soundness and completeness theorem, also discussing how we can extend our proof system with rules for a programming language involving the standard (sequential) control structures such as loops.

In the axiomatization of the basic assignment, $p[x := e]$ denotes the result of a standard substitution operation which simply replaces all free occurrences of the variable x in p by the expression e. The standard substitution operator on the new connectives are $(p * q)[x := e] = (p[x := e] * q[x := e])$ and $(p \mathrel{-\!\!*} q)[x := e] = (p[x := e] \mathrel{-\!\!*} q[x := e])$. We define $p[x := e]$ if and only if the variables of e do not also occur bound in p, thus our substitution operation avoids the capture of variables in e by a quantifier in p. By renaming bound variables of p, it is always possible to find an alphabetic variant for which the substitution is defined.

In the look-up axiom, the variable y is *fresh*, i.e. y does not occur (free or bound) in p, $y \notin var(e) \cup \{x\}$. Note that $[e]$ is not an expression of the assertion language, so we cannot replace x by $[e]$. Clearly, as already observed in [1], there is no need for introducing in the axiom of the look-up instruction separating conjunction and separating implication as in

$$\{\exists y((e \mapsto y) * ((e \mapsto y) \mathrel{-\!\!*} p[x := y]))\}\ x := [e]\ \{p\}$$

which increases the complexity of the precondition.

The following lemma is a straightforward extension of the substitution lemma for standard first-order logic.

Lemma 1 (Store substitution lemma)**.**
$h, s \models p[x := e]$ iff $h, s[x := s(e)] \models p$.

Proof. The proof proceeds by a straightforward induction on the structure of p, assuming the standard substitution lemma for expressions:

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

(note that e and e' are pure arithmetic expressions with no reference to the heap). □

Next we introduce a substitution for *heap updates*. In contrast to the mutation instruction, the pseudo heap update instruction $\langle x\rangle := e$ does not require that the updated location x is already allocated. Formally, we have the following semantics of the pseudo instruction $\langle x\rangle := e$:

$$\langle \langle x\rangle := e, h, s\rangle \Rightarrow (h[s(x) := s(e)], s)$$

which always successfully terminates, that is, it does not require that $s(x) \in dom(h)$. If $s(x) \notin dom(h)$ this transition thus involves extending the domain of the heap. As such the substitution operation corresponding to this instruction allows us to axiomatize both mutation and allocation instructions (note that $x := \textbf{cons}(e)$ does not require that x is allocated).

Definition 1 (Heap update substitution)**.** *We define $p[\langle x\rangle := e]$ recursively on p (assuming the variables of e and x do not occur bound in p).*
$b[\langle x\rangle := e] = b,$
$(e' \hookrightarrow e'')[\langle x\rangle := e] = (x = e' \land e'' = e) \lor (x \neq e' \land e' \hookrightarrow e''),$
$(p \land q)[\langle x\rangle := e] = p[\langle x\rangle := e] \land q[\langle x\rangle := e]$
and similar for $\rightarrow$ *and* $\lor$,
$(\forall y p)[\langle x\rangle := e] = \forall y(p[\langle x\rangle := e])$ *and similar for* $\exists$,
$(p * q)[\langle x\rangle := e] = (p[\langle x\rangle := e] * q') \lor (p' * q[\langle x\rangle := e])$
where $p' = p \land (x \not\hookrightarrow -)$ *and similarly* $q' = q \land (x \not\hookrightarrow -)$,
$(p \mathrel{-\!\!*} q)[\langle x\rangle := e] = p' \mathrel{-\!\!*} q[\langle x\rangle := e]$
where as above $p' = p \land (x \not\hookrightarrow -)$.

The general idea of the assertion $p[\langle x\rangle := e]$ is that it states the necessary and sufficient conditions for p to hold *after* executing the pseudo instruction $\langle x\rangle := e$. The different cases of this definition then can be informally explained 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 $b[\langle x\rangle := e] = b$. Since x is allocated *after* the heap update $\langle x\rangle := e$, we have that after the heap update **emp** does not hold and so $\textbf{emp}[\langle x\rangle := e] \equiv \textbf{false}$. 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''). Since after the heap update $\langle x\rangle := e$ at least x is allocated, for $e' \mapsto e''$ to hold[3] after the update we must have that *before* the update at most x is allocated, and so $(e' \mapsto e'')[\langle x\rangle := e] \equiv (\textbf{emp} \lor x \mapsto -) \land x = e' \land e'' = e$. The cases of the standard logical connectives are self-evident. Predicting whether $(p * q)$

[3]Note that in our case the assertions **emp** and $(e' \mapsto e'')$ are abbreviations, but it is also possible to directly define the substitution operator for these cases.

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. The semantics of $(p \mathrel{-\!\!*} q)$ 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 simple example we have the following instance of the mutation axiom (see Figure 3):

$$\begin{array}{c}\{(x \hookrightarrow -) \wedge ((y = x \wedge z = 0) \vee (y \neq x \wedge y \hookrightarrow z))\}\\ [x] := 0\\ \{y \hookrightarrow z\}\end{array} \qquad (1)$$

Compare this with the standard rule for backwards reasoning in [1] that gives the weakest precondition:

$$\{(x \mapsto -) * ((x \mapsto 0) \mathrel{-\!\!*} (y \hookrightarrow z))\}\ [x] := 0\ \{y \hookrightarrow z\} \quad (2)$$

This precondition introduces a nested application of separation conjunction and implication. Semantically this comes down to *nested alternating (second-order) quantifiers*! Consider a simple program that consists of three consecutive mutations. There, the standard rule semantically yields a precondition with a depth of six alternating quantifiers (without counting any that appear in the given postcondition). Hence, the most deeply nested variable may depend on the value of no less than 5 other variables! In contrast, calculating the weakest precondition with our new substitution introduced in Definition 1 yields no additional nesting of quantifiers and makes potential aliasing immediately explicit at the syntactic level.

Despite their different formulations, both preconditions of (1) and (2) are the *weakest* preconditions (we show this later in the paper), and so they must be equivalent. This example clearly shows the main difference between the two approaches: in standard separation logic the separating connectives are used to capture aliasing 'topologically', that is, e and e' denote different locations if the heap can be decomposed into two disjoint heaps containing the locations e and e', respectively (as expressed by $(e \hookrightarrow -) * (e' \hookrightarrow -)$). In our approach aliasing is captured directly by equational reasoning, that is, two expressions e and e' denote different locations if $e \neq e'$.

We can use our allocation axiom (see Figure 3) to derive

$$\{\textbf{emp}\}\ x := \textbf{cons}(e)\ \{\forall y((y \hookrightarrow -) \rightarrow y = x)\}.$$

First we calculate $(\forall y((y \hookrightarrow -) \rightarrow y = x))[\langle x\rangle := e]$, which (after an application of some trivial logical equivalences) gives $\forall y((y = x \vee (y \neq x \wedge y \hookrightarrow -)) \rightarrow y = x)$. This latter assertion can be further simplified to $\forall y(y = x \vee y \not\hookrightarrow -)$ which is clearly implied by **emp** (and so a final application of the consequence rule finishes the proof). Note that, also assuming $(x \not\hookrightarrow -)$, the above assertion is equivalent to **emp**.

An instance of the backwards allocation axiom in [1] is

$$\begin{array}{c}\{\forall x((x \mapsto e) \mathrel{-\!\!*} \forall y((y \hookrightarrow -) \rightarrow y = x)\}\\ x := \textbf{cons}(e)\\ \{\forall y((y \hookrightarrow) - \rightarrow y = x)\}.\end{array}$$

Clearly, to see that the resulting precondition is implied by the assertion **emp** we need to reason about disjoint heaps instead of basic first-order predicate reasoning.

In some cases the heap update substitution increases the size of the (input) formula. However, the sheer size of a formula obviously depends on the syntax and in general is not an adequate measure of the complexity of its semantics (in contrast to complexity measured by alternating quantifiers). But, more importantly, the heap update substitution reveals the necessary aliasing analysis: it makes explicit what is implicit using the separating connectives without introducing redundant information (this follows from the soundness and completeness theorem below).

We have the following lemma that shows the correctness of the substitution for heap updates.

Lemma 2 (Heap update substitution lemma)**.**
$h, s \models p[\langle x\rangle := e]$ *iff* $h[s(x) := s(e)], s \models p$.

Proof. The proof proceeds by induction on the structure of p. We present the following main cases.

- $b[\langle x\rangle := e] = b$: it suffices to observe that evaluation of b does not depend on the heap.
- $h, s \models (e' \hookrightarrow e'')[\langle x\rangle := e]$
  iff (definition substitution)
  $h, s \models (x = e' \wedge e'' = e) \vee (x \neq e' \wedge e' \hookrightarrow e'')$
  iff (semantics assertions)
  [if $s(x) = s(e')$ then $s(e) = s(e'')$ else $h(s(e')) = s(e'')$]
  iff (definition $h[s(x) := s(e)]$)
  $h[s(x) := s(e)](s(e')) = s(e'')$
  iff (semantics points-to)
  $h[s(x) := s(e)], s \models e' \hookrightarrow e''$.
- Let $p' = p \wedge (x \not\hookrightarrow -)$ and, similarly, $q' = q \wedge (x \not\hookrightarrow -)$.
  $h, s \models (p * q)[\langle x\rangle := e]$
  iff (definition substitution, semantics disjunction)
  $h, s \models p[\langle x\rangle := e] * q'$ or $h, s \models p' * q[\langle x\rangle := e]$.
  To prove the last equivalence, first let $h, s \models p[\langle x\rangle := e] * q'$ (the other case runs similarly). By the semantics separating conjunction, there exist h_1 and h_2 be such that $h = h_1 \uplus h_2$, $h_1, s \models p[\langle x\rangle := e]$, and $h_2, s \models q \wedge x \not\hookrightarrow -$. We then proceed as follows:
  $h_1, s \models p[\langle x\rangle := e]$ and $h_2, s \models q \wedge (x \not\hookrightarrow -)$
  iff (induction hypothesis)
  $h_1[s(x) := s(e)], s \models p$ and $h_2, s \models q \wedge (x \not\hookrightarrow -)$
  iff (semantics points-to)
  $h_1[s(x) := s(e)], s \models p$, $h_2, s \models q$, and $s(x) \notin dom(h_2)$.
  Since $s(x) \notin dom(h_2)$ it then follows that $h[s(x) := s(e)] = h_1[s(x) := s(e)] \uplus h_2$, and so $h[s(x) := s(e)], s \models p * q$, by the semantics of separating conjunction. Conversely, we have $h[s(x) := s(e)], s \models p * q$
  iff (semantics separating conjunction)
  $h_1, s \models p$ and $h_2, s \models q$, for some h_1, h_2 such that $h[s(x) := s(e)] = h_1 \uplus h_2$. So we have that either $s(x) \in dom(h_1)$ or $s(x) \in dom(h_2)$. Let $s(x) \in dom(h_1)$ (the other case runs similarly). Let $h_1' = h_1[s(x) := h(s(x))]$. It follows that $h = h_1' \uplus h_2$,

$h'_1[s(x) := s(e)], s \models p$, $h_2, s \models q$, and $s(x) \notin dom(h_2)$. From which we derive $h, s \models p[\langle x\rangle := e] * q'$ (as shown above). That is, $h, s \models (p * q)[\langle x\rangle := e]$, by definition of the substitution.

- $h, s \models (p \mathrel{-\!\!*} q)[\langle x\rangle := e]$
  iff (definition substitution)
  $h, s \models (p \wedge x \not\hookrightarrow -) \mathrel{-\!\!*} (q[\langle x\rangle := e])$
  iff (semantics separating implication)
  for every h' disjoint from h: if $h', s \models p \wedge x \not\hookrightarrow -$ then $h \uplus h', s \models q[\langle x\rangle := e]$
  iff (induction hypothesis)
  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 (since $s(x) \notin dom(h')$)
  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 (semantics separating implication)
  $h[s(x) := s(e)], s \models p \mathrel{-\!\!*} q$. □

Similar to what we have done for heap updates, we also introduce a substitution for the pseudo *heap clear* instruction. In contrast to the deallocation instruction, the instruction $\langle x\rangle := \bot$ does not require that the updated location x is already allocated. Formally, we have unconditionally $\langle \langle x\rangle := \bot, h, s\rangle \Rightarrow (h[s(x) := \bot], s)$.

Definition 2 (Heap clear substitution). *We define $p[\langle x\rangle := \bot]$ recursively on p (assuming that x does not occur bound in p).*

- $b[\langle x\rangle := \bot] = b$,
- $(e \hookrightarrow e')[\langle x\rangle := \bot] = x \neq e \wedge e \hookrightarrow e'$,
- $(p \wedge q)[\langle x\rangle := \bot] = p[\langle x\rangle := \bot] \wedge q[\langle x\rangle := \bot]$ *and similar for $\rightarrow$ and $\vee$,*
- $(\forall y p)[\langle x\rangle := \bot] = \forall y(p[\langle x\rangle := \bot])$ *and similar for $\exists$,*
- $(p * q)[\langle x\rangle := \bot] = (p[\langle x\rangle := \bot]) * (q[\langle x\rangle := \bot])$,
- $(p \mathrel{-\!\!*} q)[\langle x\rangle := \bot] = ((p \wedge x \not\hookrightarrow -) \mathrel{-\!\!*} q[\langle x\rangle := \bot]) \wedge \forall y(p[\langle x\rangle := y] \mathrel{-\!\!*} q[\langle x\rangle := y])$ *where y is fresh.*

The general idea of the assertion $p[\langle x\rangle := \bot]$ is that it states the necessary and sufficient conditions for p to hold *after* the execution of the pseudo instruction $\langle x\rangle := \bot$, which does not require that x is allocated. The different cases of this definition then can be informally explained as follows: Since $\langle x\rangle := \bot$ does not affect the store, and the evaluation of a Boolean condition b only depends on the store, we have that $b[\langle x\rangle := \bot] = b$. After executing $\langle x\rangle := \bot$ the heap is empty if and only if the heap before at most contains x, so $\mathbf{emp}[\langle x\rangle := \bot] \equiv \mathbf{emp} \vee (x \mapsto -)$. For $e \hookrightarrow e'$ to hold after executing $\langle x\rangle := \bot$, we must initially have that $x \neq e$ and $e \hookrightarrow e'$. For $e \mapsto e'$ to hold after executing $\langle x\rangle := \bot$, we must initially have that $x \neq e$, $e \hookrightarrow e'$, and that initially at most e and x are allocated: thus $(e \mapsto e')[\langle x\rangle := \bot] \equiv (e \hookrightarrow e') \wedge x \neq e \wedge \forall y((y \hookrightarrow -) \rightarrow (y = e \vee y = x))$. The cases of the standard logical connectives are self-evident. 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 $p \mathrel{-\!\!*} q$ after clearing x involves universal quantification over all disjoint heaps that do may contain x, whereas before executing $\langle x\rangle := \bot$ 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 $p \mathrel{-\!\!*} q$ in the context of heap updates $\langle x\rangle := y$ with *arbitrary* values y for the location x.

As a simple example, where $\forall y, z(y \not\hookrightarrow z)$ characterizes the empty heap **emp**, we have that $(\forall y, z(y \not\hookrightarrow z))[\langle x\rangle := \bot]$ reduces to $\forall y, z(\neg(y \neq x \wedge y \hookrightarrow z))$, which 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 $\langle x\rangle := \bot$ the heap is empty.

We have the following lemma that shows the correctness of the substitution for heap clear.

Lemma 3 (Heap clear substitution lemma).
$h, s \models p[\langle x\rangle := \bot]$ iff $h[s(x) := \bot], s \models p$.

Proof. The proof proceeds by induction on the structure of p. We treat the following cases:

- $b[\langle x\rangle := \bot] = b$, so, as above, it suffices to observe that the evaluation of b does not depend on the heap.
- $h, s \models (e \hookrightarrow e')[\langle x\rangle := \bot]$
  iff (definition substitution, semantics assertions)
  $h(s(e)) = s(e')$ and $s(x) \neq s(e)$
  iff (definition $h[\langle s(x)\rangle := \bot]$)
  $h[\langle s(x)\rangle := \bot](s(e)) = s(e')$
  iff (semantics points-to)
  $h[\langle s(x)\rangle := \bot], s \models e \hookrightarrow e'$.
- $h, s \models (p * q)[\langle x\rangle := \bot]$
  iff (definition substitution)
  $h, s \models p[\langle x\rangle := \bot] * q[\langle x\rangle := \bot]$
  iff (semantics separating conjunction)
  $h_1, s \models p[\langle x\rangle := \bot]$ and $h_2, s \models q[\langle x\rangle := \bot]$, for some h_1, h_2 such that $h = h_1 \uplus h_2$
  iff (induction hypotheses)
  $h_1[\langle s(x)\rangle := \bot], s \models p$ and $h_2[\langle s(x)\rangle := \bot], s \models q$, for some h_1, h_2 such that $h = h_1 \uplus h_2$
  iff (definition $h[\langle s(x)\rangle := \bot]$, see note below)
  $h_1, s \models p$ and $h_2, s \models q$, for some h_1, h_2 such that $h[\langle s(x)\rangle := \bot] = h_1 \uplus h_2$
  iff (definition separating conjunction)
  $h[\langle s(x)\rangle := \bot], s \models p * q$.
  Note: $h = h_1 \uplus h_2$ implies $h[\langle s(x)\rangle := \bot] = h_1[\langle s(x)\rangle := \bot] \uplus h_2[\langle s(x)\rangle := \bot]$, and, conversely, $h[\langle s(x)\rangle := \bot] = h_1 \uplus h_2$ implies there exists h'_1, h'_2 such that $h = h'_1 \uplus h'_2$ and $h_1 = h'_1[\langle s(x)\rangle := \bot]$ and $h_2 = h'_2[\langle s(x)\rangle := \bot]$.
- $h, s \models (p \mathrel{-\!\!*} q)[\langle x\rangle := \bot]$
  iff (definition substitution)
  $h, s \models ((p \wedge (x \not\hookrightarrow -)) \mathrel{-\!\!*} q[\langle x\rangle := \bot]) \wedge$
  $\forall y(p[\langle x\rangle := y] \mathrel{-\!\!*} q[\langle x\rangle := y])$
  iff (see below)
  $h[s(x) := \bot], s \models p \mathrel{-\!\!*} q$.
  (Here y is a fresh variable.) First we show that $h, s \models$

$((p \wedge x \not\hookrightarrow -) \mathrel{-\!\!*} q[\langle x\rangle := \bot])$ and $h, s \models \forall y(p[[x] := y] \mathrel{-\!\!*} q[[x] := y])$ implies $h[s(x) := \bot], s \models p \mathrel{-\!\!*} q$. Let h' be disjoint from $h[s(x) := \bot]$ and $h', s \models p$. We have to show that $h[s(x) := \bot] \uplus h', s \models q$. We distinguish the following two cases.

- First, let $s(x) \in \mathit{dom}(h')$. 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 above substitution lemma that $h'[s(x) := \bot], s' \models p[[x] := y]$. Since $h'[s(x) := \bot]$ and h are disjoint (which clearly follows from that h' and $h[s(x) := \bot]$ are disjoint), and since $h, s' \models p[[x] := y] \mathrel{-\!\!*} q[[x] := y]$, we have that $h \uplus (h'[s(x) := \bot]), s' \models q[[x] := y]$. Applying again the above substitution lemma, we obtain $(h \uplus (h'[s(x) := \bot]))[s(x) := s'(y)], s' \models q$. We then can conclude this case observing that y does not occur in q and that $h[s(x) := \bot] \uplus h' = (h \uplus (h'[s(x) := \bot]))[s(x) := s'(y)]$.
- Next, let $s(x) \notin \mathit{dom}(h')$. So h' and h are disjoint, and thus (since $h, s \models (p \wedge x \not\hookrightarrow -) \mathrel{-\!\!*} q[\langle x\rangle := \bot]$) we have $h \uplus h', s \models q[\langle x\rangle := \bot]$. From which we derive $(h \uplus h')[s(x) := \bot], s \models q$ by the induction hypothesis. We then can conclude this case by the observation that $h[s(x) := \bot] \uplus h' = (h \uplus h')[s(x) := \bot]$.

Conversely, assuming $h[s(x) := \bot], s \models p \mathrel{-\!\!*} q$, we first show that $h, s \models (p \wedge x \not\hookrightarrow -) \mathrel{-\!\!*} q[\langle x\rangle := \bot]$ and then $h, s \models \forall y(p[[x] := y] \mathrel{-\!\!*} q[[x] := y])$.

- Let h' be disjoint from h and $h', s \models p \wedge x \not\hookrightarrow -$. We have to show that $h \uplus h', s \models q[\langle x\rangle := \bot]$. Clearly, $h[s(x) := \bot]$ and h' are disjoint, and so $h[s(x) := \bot] \uplus h', s \models q$ from our assumption. By the induction hypothesis, we have $h \uplus h', s \models q[\langle x\rangle := \bot]$ iff $(h \uplus h')[s(x) := \bot], s \models q$. We then can conclude this case by the observation that $(h \uplus h')[s(x) := \bot] = h[s(x) := \bot] \uplus h'$, because $s(x) \notin \mathit{dom}(h')$.
- Let h' be disjoint from h and $s' = s[y := n]$, for some n such that $h', s' \models p[[x] := y]$. We have to show that $h \uplus h', s' \models q[[x] := y]$. By the above substitution lemma 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 $h[s(x) := \bot]$ are disjoint, we derive from the assumption $h[s(x) := \bot], s \models p \mathrel{-\!\!*} q$ that $h[s(x) := \bot] \uplus h'[s(x) := n], s \models q$. Again by the above substitution lemma, we have that $h \uplus h', s' \models q[[x] := y]$ 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 $(h \uplus h')[s(x) := n] = h[s(x) := \bot] \uplus h'[s(x) := n]$. □

Since it is always clear from a particular context which substitution lemma is applied, we simply refer to one of the three above lemmas as *the* substitution lemma.

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

$$\{p\}\ x := \mathbf{cons}(0); \mathbf{dispose}(x)\ \{p\}$$

where x does not occur in p. This program simulates the so-called random assignment [7]: the program terminates with a value of x that is chosen non-deterministically. First we apply the axiom for deallocation to obtain

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

Next, we apply the axiom for allocation to obtain (after some simplifications)

$$\begin{gathered}\{\forall x((x \not\hookrightarrow -) \to p[\langle x\rangle := \bot][\langle x\rangle := 0])\}\\ x := \mathbf{cons}(0)\\ \{(x \hookrightarrow -) \wedge p[\langle x\rangle := \bot]\}.\end{gathered}$$

We then apply the rule of consequence and sequential composition to derive our specification, for which we need to show that p implies the precondition so obtained. So let $h, s \models p$, and take an arbitrary $n \notin \mathit{dom}(h)$. We want to show that $h, s[x := n] \models p[\langle x\rangle := \bot][\langle x\rangle := 0]$. By the substitution lemma, it is sufficient to show $h[n := 0], s[x := n] \models p[\langle x\rangle := \bot]$. Again, by the substitution lemma, it is sufficient to show $h[n := 0][n := \bot], s[x := n] \models p$. The latter follows from $h, s \models p$, because $h = h[n := 0][n := \bot]$ since $n \notin \mathit{dom}(h)$ and we know that x does not occur in p.

Contrastingly, using the backwards deallocation and allocation axioms in [1] and sequential composition gives us:

$$\begin{gathered}\{\forall x((x \mapsto 0) \mathrel{-\!\!*} ((x \mapsto -) * p))\}\\ x := \mathbf{cons}(0); \mathbf{dispose}(x)\\ \{p\}.\end{gathered}$$

To derive our specification, we apply the consequence rule, and we need to show that p implies the precondition above. Let $h, s \models p$. Let n be arbitrary, we need to show $h, s[x := n] \models (x \mapsto 0) \mathrel{-\!\!*} ((x \mapsto -) * p)$. Take arbitrary h', h'' such that $h'' = h \uplus h'$ and $h', s[x := n] \models (x \mapsto 0)$. So $\mathit{dom}(h') = \{n\}$ and $n \notin \mathit{dom}(h)$. Now we have $h'', s[x := n] \models (x \mapsto -) * p$ since we can take h' and h as witnesses: $h', s[x := n] \models (x \mapsto -)$ holds, and $h, s[x := n] \models p$ (x does not occur in p).

We now show our main theorem.

Theorem 1 (Basic soundness and completeness WP-SL)**.** *For any basic instruction S, we have* $\models \{p\}\ S\ \{q\}$ *if and only if* $\{p\}\ S\ \{q\}$ *is derivable from the axioms in WP-SL (Figure 3) and (a single application of) the rule of consequence.*

Proof. It suffices to show that the axioms are valid and describe for each instruction weakest preconditions. We may assume, without loss of generality, that formulas satisfy the assumptions in such a way that the corresponding substitution operators applied on them are defined, i.e. by renaming bound variables into fresh variables.

a) Basic assignment:

- $\models \{q[x := e]\}\ x := e\ \{q\}$: standard.
- $\models \{p\}\ x := e\ \{q\}$ implies $\models p \to (q[x := e])$: standard.

b) Look-up:

- $\models \{\exists y((e \hookrightarrow y) \wedge (p[x := y]))\}\ x := [e]\ \{p\}$: Let $h, s \models \exists y((e \hookrightarrow y) \wedge (p[x := y]))$. So there exists n such that $h, s[y := n] \models (e \hookrightarrow y) \wedge (p[x := y])$, that is, $n = h(s(e))$ and $h, s[y := n] \models p[x := y]$. Since $s(e) \in \mathit{dom}(h)$, we have $\langle x := [e], h, s\rangle \Rightarrow (h, s[x := h(s(e))])$. Further (by the above substitution lemma), $h, s[y := n] \models p[x := y]$ if and only if $h, s[x := n] \models p$.
- $\models \{p\}\ x := [e]\ \{q\}$ implies $\models p \to \exists y((e \hookrightarrow y) \wedge (q[x := y]))$: Let $\models \{p\}\ x := [e]\ \{q\}$ and assume $h, s \models p$. So we have that $\langle x := [e], h, s\rangle \not\Rightarrow$ **fail**, that is, $s(e) \in \mathit{dom}(h)$. Let $n = h(s(e))$. We have $h, s[y := n] \models e \hookrightarrow y$ since y does not occur in e. We have that $h, s[x := n] \models q$, and so $h, s[y := n][x := n] \models q$ since y does not occur in q, and so $h, s[y := n] \models q[x := y]$ by the above substitution lemma. Summarizing, we have $h, s[y := n] \models (e \hookrightarrow y) \wedge (q[x := y])$, that is, $h, s \models \exists y((e \hookrightarrow y) \wedge (q[x := y]))$.

c) Mutation:

- $\models \{(x \hookrightarrow -) \wedge (p[\langle x\rangle := e])\}\ [x] := e\ \{p\}$: Let $h, s \models (x \hookrightarrow -) \wedge (p[\langle x\rangle := e])$. Since $h, s \models (x \hookrightarrow -)$ we have $s(x) \in \mathit{dom}(h)$ and thus $\langle [x] := e, h, s\rangle \Rightarrow (h[s(x) := s(e)], s)$. By the substitution lemma it follows from $h, s \models p[\langle x\rangle := e]$ that $h[s(x) := s(e)], s \models p$.
- $\models \{p\}\ [x] := e\ \{q\}$ implies $\models p \to ((x \hookrightarrow -) \wedge (q[\langle x\rangle := e]))$: Let $\models \{p\}\ [x] := e\ \{q\}$ and assume $h, s \models p$. We have $\langle [x] := e, h, s\rangle \Rightarrow (h[s(x) := s(e)], s)$. So we have $h, s \models x \hookrightarrow -$ since $\langle [x] := e, h, s\rangle \not\Rightarrow$ **fail**. We have that $h[s(x) := s(e)], s \models q$, and so $h, s \models q[\langle x\rangle := e]$ by the above substitution lemma. Hence we obtain our conclusion $h, s \models (x \hookrightarrow -) \wedge (q[\langle x\rangle := e])$.

d) Allocation:

- $\models \{\forall x((x \not\hookrightarrow -) \to (p[\langle x\rangle := e]))\}\ x := \mathbf{cons}(e)\ \{p\}$: Let $h, s \models \forall x((x \not\hookrightarrow -) \to (p[\langle x\rangle := e]))$ and $\langle x := \mathbf{cons}(e), h, s\rangle \Rightarrow (h[n := s(e)], s[x := n])$ for some $n \notin \mathit{dom}(h)$. We need to show $h[n := s(e)], s[x := n] \models p$. It follows from $h, s \models \forall x((x \not\hookrightarrow -) \to (p[\langle x\rangle := e]))$ that $h, s[x := n] \models p[\langle x\rangle := e]$, and so $h[s[x := n](x) := s[x := n](e)], s[x := n] \models p$ by the above substitution lemma. Our result follows from the facts that $s[x := n](x) = n$ and $s[x := n](e) = s(e)$ since x is assumed not to occur in e.
- $\models \{p\}\ x := \mathbf{cons}(e)\ \{q\}$ implies $\models p \to (\forall x((x \not\hookrightarrow -) \to (q[\langle x\rangle := e])))$: Let $h, s \models p$ and $h, s[x := n] \models \neg(x \hookrightarrow -)$, for some n. It follows that $n \notin \mathit{dom}(h)$. So we have $\langle x := \mathbf{cons}(e), h, s\rangle \Rightarrow (h[n := s(e)], s[x := n])$, and so it follows that we have $h[n := s(e)], s[x := n] \models q$ from $\models \{p\}\ x := \mathbf{cons}(e)\ \{q\}$. By the above substitution lemma, we derive $h, s[x := n] \models q[\langle x\rangle := e]$ (as above, here we use again that $s[x := n](x) = n$ and $s[x := n](e) = s(e)$).

e) Dispose:

- $\models \{(x \hookrightarrow -) \wedge p[\langle x\rangle := \bot]\}\ [x] := \bot\ \{p\}$: Let $h, s \models (x \hookrightarrow -) \wedge p[\langle x\rangle := \bot]$ and $\langle [x] := \bot, h, s\rangle \Rightarrow (h[s(x) := \bot], s)$. By the above substitution lemma we have that $h, s \models p[\langle x\rangle := \bot]$ implies $h[s(x) := \bot], s \models p$.
- $\models \{p\}\ [x] := \bot\ \{q\}$ implies $\models p \to ((x \hookrightarrow -) \wedge q[\langle x\rangle := \bot])$: Let $h, s \models p$. From $\models \{p\}\ [x] := \bot\ \{q\}$ we derive that $h, s \models x \hookrightarrow -$ and $h[s(x) := \bot], s \models q$, that is, $h, s \models q[\langle x\rangle := \bot]$ (by the above substitution lemma). ☐

The above basic soundness and completeness proof can be extended in a straightforward manner to a program logic for sequential while programs 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 (future work). The proof of soundness and relative completeness (in the sense of Cook) is standard, see e.g. [10], [13], [14], expect for the base cases which follow from Theorem 1. Note that, in the expressiveness theorem of the weakest liberal precondition of [10], one needs that the heap ranges over natural numbers instead of integers: Theorem 1 can be easily adapted to hold in such situation too.

IV. Completeness of local axioms

Reynolds introduced so-called local axioms [1] and shows how to derive from these local axioms a weakest precondition axiomatization of the basic instructions in separation logic, using the frame rule and the separating implication for expressing the weakest precondition. Here we illustrate the expressiveness of our approach in a direct completeness proof of the local axioms which uses substitutions for expressing the weakest precondition instead of the separating implication. In, for example, [4], it is already shown that the separating implication is not needed to prove completeness of the local axioms for basic assignments, look-up, allocation, and deallocation. However, to the best of our knowledge, such a completeness proof does not exist for the local mutation axiom

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

where, for simplicity, we assume x does not occur in e. 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 substitution can be used to describe such a frame. Let $\models \{p\}\ [x] := e\ \{q\}$ and r denote $\exists y(p[\langle x\rangle := y])$ 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 $h_2, s \models r$. Since $\models \{p\}\ [x] := e\ \{q\}$ we have that $s(x) \in \mathit{dom}(h)$. So we can introduce the split $h = h_1 \uplus h_2$ such that $h_1, s \models (x \mapsto -)$ and

$$\{p\}\ x := e\ \{\exists y(p[x := y] \wedge (x = e[x := y]))\}$$
$$\{p \wedge (e \hookrightarrow -)\}\ x := [e]\ \{\exists y(p[x := y] \wedge (x \hookrightarrow e[x := y]))\}$$
$$\{p \wedge (x \hookrightarrow -)\}\ [x] := e\ \{\exists y(p[\langle x\rangle := y]) \wedge (x \hookrightarrow e)\}$$
$$\{p\}\ x := \mathbf{cons}(e)\ \{(\exists y(p[x := y]))[\langle x\rangle := \bot] \wedge (x \hookrightarrow e)\}$$
$$\{p \wedge (x \hookrightarrow -)\}\ \mathbf{dispose}(x)\ \{\exists y(p[\langle x\rangle := y]) \wedge (x \not\hookrightarrow -)\}$$

Fig. 4. Strongest postcondition axioms of separation logic (SP-SL), where y is fresh everywhere and x does not occur in e in case of $x := \mathbf{cons}(e)$.

$h_2 = h[s(x) := \bot]$. By the above substitution lemma it then suffices to observe that $h_2, s[y := h(s(x))] \models p[\langle x\rangle := y]$ if and only if $h_2[s(x) := h(s(x))], s \models p$ (y does not appear in p), that is, $h, s \models p$. 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 $h_2, s[y := n] \models p[\langle x\rangle := y]$. By the above substitution lemma we have that $h_2, s[y := n] \models p[\langle x\rangle := y]$ 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 $h_2 = h[s(x) := \bot]$).

V. Strongest postcondition axiomatization

Before we discuss our strongest postcondition axiomatization of separation logic, 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 4 contains our novel strongest postcondition axiomatization SP-SL. See the appendix, Figure 6 for the standard proof rules of Hoare [12], and Figure 8 for the existing strongest postcondition axiomatization involving separating connectives [1], [9]. It is worthwhile to contrast, for example, the use of the heap clear substitution to express freshness in the strongest postcondition axiomatization of the allocation instruction with the following traditional axiom:

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

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 4:

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

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 substitution we thus obtain a basic assertion in predicate logic which provides an explicit but simple account of aliasing.

Theorem 2 (Basic soundness and completeness SP-CSL)**.** *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-CSL (Figure 4) 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 other instructions follow in a straightforward manner from the corresponding substitution lemmas).

- $\models \{p\}\ x := \mathbf{cons}(e)\ \{(\exists y(p[x := y]))[\langle x\rangle := \bot] \wedge x \hookrightarrow e\}$: Let $h, s \models p$. We have to show that $h[n := s(e)], s[x := n] \models (\exists y(p[x := y]))[\langle x\rangle := \bot] \wedge x \hookrightarrow e$, for $n \notin dom(h)$. By definition $h[n := s(e)], s[x := n] \models x \hookrightarrow e$. Further, by the semantics of existential quantification and the definition of the substitution $[\langle x\rangle := \bot]$, it suffices to show that $h[n := s(e)], s[x := n][y := s(x)] \models (p[x := y])[\langle x\rangle := \bot]$, that is (by the substitution lemma), $h[n := s(e)][n := \bot], s[x := n][y := s(x)] \models p[x := y]$. By the substitution lemma and since $h[n := s(e)][n := \bot] = h$, the latter is equivalent to $h, s \models p$ (also y does not appear in p), which holds by assumption.
- $\models \{p\}\ x := \mathbf{cons}(e)\ \{q\}$ implies $\models (\exists y(p[x := y])[\langle x\rangle := \bot] \wedge x \hookrightarrow e) \to q$: Let $h, s \models (\exists y(p[x := y]))[\langle x\rangle := \bot] \wedge x \hookrightarrow e$. We have to show that $h, s \models q$. By the correctness of the substitution $[\langle x\rangle := \bot]$ we derive $h[s(x) := \bot], s \models \exists y(p[x := y])$ from our assumption $h, s \models (\exists y(p[x := y]))[\langle x\rangle := \bot]$. Let $h[s(x) := \bot], s[y := n] \models p[x := y]$, for some n. It follows from the substitution lemma that $h[s(x) := \bot], s[x := n] \models p$ (as y does not appear in p). Since $s(x) \notin dom(h[s(x) := \bot])$, we have that $\langle x := \mathbf{cons}(e), h[s(x) := \bot], s[x := n]\rangle \Rightarrow (h[s(x) := s[x := n](e)], s)$. 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 $h, s \models q$. □

VI. Extensions

A straightforward extension concerns the 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 corresponding substitutions we derive the following axiom

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

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 $z := x;\ x := \mathbf{cons}(e[x := z])$ where z 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:

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

where y is fresh.

Reynolds introduced in [1] 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 $i = 1, \dots, n-1$, $\{m_1, \dots, m_n\} \cap \mathit{dom}(h) = \emptyset$, and, finally,

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

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 substitutions $[\langle e'_i\rangle := e_i]$, for $i = 1, \dots, n$. We abbreviate such a sequence of substitutions 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

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

VII. Recursive predicates

In this section we describe and 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 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 $r(e_1, \dots, e_n)[x := e]$ by $r(e_1[x := e], \dots, e_n[x := e])$. 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 $r(y)[x := 0]$ by $r(y[x := 0])$). Furthermore, modeling heap modifying instructions using substitutions also involves alias analysis and thus also requires a redefinition of the recursive definitions. Therefore, we introduce the following general method for applying substitutions to recursively defined predicates. Let σ denote a substitution. For every predicate $r(x_1, \dots, x_n)$ with defining formula p we introduce a *new* predicate $r_\sigma(x_1, \dots, x_n)$ defined by $p\sigma$, where applying σ to the basic assertion $r(t_1, \dots, t_n)$ yields $r_\sigma(t_1\sigma, \dots, t_n\sigma)$. The corresponding substitution lemma then extends to recursive predicates in a straightforward manner. Note that applying a heap update and heap clear substitution to an expression just simplifies to the expression itself, since evaluation of expressions does not depend on the heap.

We illustrate our theory of recursively defined predicates by the characteristic linked list example which we describe by the recursively defined *reachability* predicate

$$r(x, y) := x = y \vee \exists z(x \hookrightarrow z \wedge r(z, y))).$$

We prove

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

We model $\mathit{first} := \mathbf{cons}(\mathit{first})$ by $u := \mathit{first}; \mathit{first} := \mathbf{cons}(u)$, for some fresh variable u. Then we apply the allocation axiom to obtain

$$\begin{gathered}\{\forall \mathit{first}((\mathit{first} \not\hookrightarrow -) \rightarrow r'(\mathit{first}, y)\}\\ \mathit{first} := \mathbf{cons}(u)\\ \{r(\mathit{first}, y)\}.\end{gathered}$$

where

$$\begin{aligned} r'(x, y) := x = y &\vee (x = \mathit{first} \wedge r'(u, y)) \vee \\ &(x \neq \mathit{first} \wedge \exists z(x \hookrightarrow z \wedge r'(z, y))) \end{aligned}$$

is obtained by applying the substitution $[\langle \mathit{first}\rangle := u]$ (and some basic logical equivalences) to the definition of $r(x, y)$. Note that first and u are global variables.

In order to apply next the substitution $[u := \mathit{first}]$ to the above precondition of $\mathit{first} := \mathbf{cons}(u)$, we first have to rename the bound variable first to a fresh variable f. The consecutive application of these two simple substitutions gives rise to yet another predicate $r''(x, y)$ defined by

$$\begin{aligned} r''(x, y) := x = y &\vee (x = f \wedge r''(\mathit{first}, y)) \vee \\ &(x \neq f \wedge \exists z(x \hookrightarrow z \wedge r''(z, y))). \end{aligned}$$

Thus we obtain the weakest precondition

$$(\forall f((f \not\hookrightarrow -) \rightarrow r''(f, y))).$$

Let $p_r(x, y)$ and $p_{r''}(x, y)$ denote the defining formula of $r(x, y)$ and $r''(x, y)$, respectively. In order to show that $r(\mathit{first}, y)$ implies the above precondition, it suffices to show that $\forall x, y(p_r(x, y) \rightarrow p_{r''}(x, y))$, under the assumption that $\forall x, y(r(x, y) \rightarrow r''(x, y))$ and $f \not\hookrightarrow -$. Assume $p_r(x, y)$. Then $p_{r''}(x, y)$ trivially holds if $x = y$. So suppose that $x \neq y$. From $p_r(x, y)$ it follows that $\exists z(x \hookrightarrow z \wedge r(z, y))$. By assumption $r(z, y)$ implies $r''(z, y)$. Further $x \neq f$ (because by assumption $f \not\hookrightarrow -$). So we have that $x \neq f \wedge \exists z(x \hookrightarrow z \wedge r''(z, y))$, which trivially implies $p_{r''}(x, y)$.

It should be noted here that in general restricting to recursive definitions which do not involve the separating connectives (like the recursive definition of reachability above) allows for a standard least fixpoint semantics using Kleene's fixpoint theorem, and as such provides a formal justification of the kind of inductive reasoning as shown above. Further, note that the substitutions do not generate separating connectives, and as such do not affect the application of Kleene's fixpoint theorem.

Next we prove

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

where

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

Here we thus defined reachability in terms of the separating conjunction. In general, the separating connectives do *not* allow for an application of Kleene's fixpoint theorem because they do not satisfy the requirement of Scott-continuity. Even a sub-language without separating implication does not satisfy this requirement. For such a language only the Knaster–Tarski theorem applies and thus in general requires *transfinite* induction. However, since the above recursive definition of reachability also does not involve occurrences of negation it allows for inductive reasoning based on a least fixpoint semantics which is Scott-continuous.

To simplify the application of the substitution $[\langle \mathit{first} \rangle := u]$, we first unfold $r(\mathit{first}, y)$, obtaining

$$\mathit{first} = y \vee \exists z((\mathit{first} \mapsto z) * r(z, y)).$$

Note that $(\mathit{first} \mapsto z * r(z, y))[\langle \mathit{first} \rangle := u]$ reduces to a disjunction of $(\mathit{first} \mapsto z \wedge \mathit{first} \not\hookrightarrow -) * (r(z, y)[\langle \mathit{first} \rangle := u]))$ and $(\mathit{first} \mapsto z)[\langle \mathit{first} \rangle := u] * (r(z, y) \wedge \mathit{first} \not\hookrightarrow -)$. The first disjunct expresses that $r(z, y)$ holds in that part of the heap that is updated, under the assumption that *first* is not allocated in the disjoint part that satisfies $\mathit{first} \mapsto z$, which clearly is inconsistent and so this first disjunct reduces to **false**. So we do not need to introduce a new predicate. Applying the substitution $[\langle \mathit{first} \rangle := u]$ (and some basic logical equivalences) to the above assertion, thus results in

$$\mathit{first} = y \vee ((\mathbf{emp} \vee \mathit{first} \mapsto -) * (r(u, y) \wedge \mathit{first} \not\hookrightarrow -)).$$

Applying the allocation axiom and an application of the consequence rule (where we see that the separating conjunction can be simplified away), we obtain

$$\begin{gathered}\{\forall \mathit{first}((\mathit{first} \not\hookrightarrow -) \to (\mathit{first} = y \vee r(u, y)))\}\\ \mathit{first} := \mathbf{cons}(u)\\ \{r(\mathit{first}, y)\}.\end{gathered}$$

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

$$\begin{gathered}\{\forall f((f \not\hookrightarrow -) \to (f = y \vee r(u, y)))\}\\ \mathit{first} := \mathbf{cons}(u)\\ \{r(\mathit{first}, y)\}\end{gathered}$$

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

VIII. Formalization in Coq

The main motivation behind formalizing results in a proof assistant is to rigorously check hand-written proofs. We have formalized the proofs of the substitution lemmas corresponding to the heap update and heap clear substitutions, and the basic soundness/completeness of the weakest precondition axiomatization (WP-SL). For technical simplicity we restrict ourselves to the basic instructions, but it should be natural to extend the formalization of the completeness result to languages with **while**-statements, e.g. following [10].

For our formalization we used the dependently-typed calculus of inductive constructions as implemented by the Coq proof assistant. The source code of our formalization is accompanied with this paper as a digital artifact (which includes the files `Heap.v`, `Language.v`, `Classical.v`). Below we make some more detailed comments, relevant for those interested in the technical details of the formalization. The artifact consists of the following files:

- `Heap.v`: Provides an axiomatization of heaps as partial functions.
- `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 substitution operators (see Definitions 1 and 2). We formalize the basic instructions of our programming language (assignment, look-up, mutation, allocation, and deallocation) and the semantics of basic instructions. We finally formalize Hoare triples and the proof systems WP-SL and SP-SL (see Figures 3 and 4).
- `Classical.v`: Provides the classical semantics of assertions, and the strong partial correctness semantics of Hoare triples. Further it provides proofs of the substitution lemmas corresponding to our substitution operators (see Lemmas 1, 2 and 3). Finally, it provides proofs of the soundness and completeness of WP-SL and SP-SL (see Theorems 1 and 2).

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 have introduced an axiomatization of heaps and several of its operations (heap update, heap clear) and properties (domain, partition), and show there is a model that satisfies the axioms. In this model, we define heaps as partial functions (which are themselves modeled as a total function from integers to optional integers, the latter being the sum type consisting of the integers and a special bottom element). We show the extensionality of heaps, i.e. if two heaps assign the same values to the same locations we consider them (Leibniz) equal, by using the axiom for function extensionality. Of particular interest is that the heap axiomatization abstracts from the *cardinality* of the domain, that is, we allow for heaps with infinite domains. This is justified because for strong partial correctness (as used in this paper, which ensures absence of failures), the validity of pre/postconditions is not affected by the cardinality of heaps. In the case of a heap where all locations are allocated, that is, $\forall y(y \hookrightarrow -)$ holds, we have that $\{\forall y(y \hookrightarrow -)\}\ x := \textbf{cons}(e)\ \{\textbf{false}\}$ is true since there is no next state and it also does not lead to an explicit failure. In the case of heaps with finite domain we can derive this trivially because $\forall y(y \hookrightarrow -)$ is **false**. However, in the presence of heaps with an infinite domain we have that if $h, s \models \forall y(y \hookrightarrow -)$ then $h, s[x := n] \models (x \mapsto e) \mathrel{-\!\!*} \textbf{false}$ is vacuously true, for every n, because there is no h' disjoint from h which satisfies $h', s[x := n] \models x \mapsto e$. Note that the validity problem of the first-order assertion language based on finite heaps is not recursively enumerable [15]: restricting to finite heaps makes the logic non-compact, and finiteness itself cannot be expressed in first-order logic. Whether the validity problem of the full assertion language of separation logic, in the presence of first-order definable heaps with possibly infinite domains, is still recursively enumerable, is not known, as far as we know.

The assertion language and programming language is formalized using a mixed shallow/deep embedding. Boolean and arithmetic expressions use a shallow embedding and can be expressed directly using 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). The assertions are modeled using an inductive type, following closely the definitions in this paper for the substitution operators. We omitted the clauses for **emp** and $(e \mapsto e')$, since these could be defined as abbreviations.

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 [16]. In particular, we show that our satisfaction relation $h, s \models p$ is stable, i.e. $h, s \models \neg\neg p$ implies $h, s \models p$. This allows us to do classical reasoning on the image of the higher-order semantics of our assertions.

In particular, we are able to prove the equivalence of the preconditions of Equations (1) and (2) given in Section III. Note that this result already exceeds the capability of all the automated separation logic provers in the benchmark competition for separation logic [17]. In particular, only the CVC4-SL tool [18] supports the fragment of separation logic 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 [19]. In fact, the latest version, CVC5-SL, reports the same input as 'unknown', indicating that the tool is incomplete. The input given was: $(x \mapsto w) * ((x \mapsto 0) \mathrel{-\!\!*} ((y \mapsto z) * \textbf{true}))$ and $\neg((y = x \wedge z = 0) \vee (y \neq x \wedge ((y \mapsto z) * \textbf{true})))$.

IX. Conclusion and related work

To the best of our knowledge the theory presented in this paper provides the first systematic development of a weakest precondition axiomatization and strongest postcondition axiomatization of separation logic using substitution operators. Our axiomatizations have the property of *gracefulness*: for a provided first-order postcondition we generate a first-order weakest precondition, and for a provided first-order precondition we generate a first-order strongest postcondition. There is some related work in the literature. For example, [9] investigates weakest preconditions and strongest postconditions, obtained through a transformational approach. However, these are expressed using separating connectives (using septraction), and thus are not graceful. On the other hand, in [20] 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 [20] 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.

To summarize the main difference between our axiomatization of separation logic and the standard one is that *substitutions describe the effect of the instructions in terms of the logical structure of the given pre/postcondition*, whereas in the standard approach separating connectives are used to describe the effect of the instructions, *abstracting* from the logical structure of the given pre/postcondition. In other words: a first-order postcondition yields a resulting first-order weakest precondition with our substitutions. Clearly this is desirable when dealing with the complexity of proving the correctness of actual programs. For example, since our substitution operators do not introduce the separating connectives we can restrict to standard first-order logic when verifying standard first-order program specifications. As such, it is reasonable to expect that the use of substitutions (instead of the separating implication) in the generation of a weakest precondition may lead to a better integration of separation logic with existing verification tools, such as satisfiability/satisfiability modulo theories (SAT/SMT) solvers that lack support for separating connectives: local first-order program specifications can be proven correct using such tools, while the machinery of separation logic can be used for adapting local specifications to global specifications using the frame rule.

Our theory of substitutions generalizes to the axiomatization of the standard flow of control constructs (e.g., sequential composition, conditional, iteration statements and recursion), using the corresponding standard rules of Hoare logics. Moreover, our approach is robust: substitutions can also be applied to obtain a *strongest postcondition* calculus for the basic instructions (see Figure 4). In a follow-up paper we will describe how our approach can also be extended to intuitionistic separation logic. Intuitionistic separation logic is in fact the original version of separation logic [11], [15] where the basic assertions $e \mapsto e'$ and **emp** are omitted, and implication is defined by

$h, s \models p \to q$ if and only if $h', s \models p$ implies $h', s \models q$, for every h' such that $h \sqsubseteq h'$ (that is, $h'(n) = h(n)$, for $n \in dom(h)$). Consequently, we can no longer express that a location x is not allocated: thus the substitutions $p[\langle x\rangle := e]$ and $p[\langle x\rangle := \bot]$ are no longer correct in this semantics. Therefore, we need to define other substitution operators and prove their correctness.

Our artifact formalizes the syntax and semantics of programs and assertions of separation logic. We plan to further extend our formalization to support practical program verification, and investigate how to integrate our approach in Iris [21]: we will consider how our approach using substitution operators can also work for a shallow embedding of separation logic. Then the generated verification conditions require a proof of the validity of corresponding assertions in separation logic, which can be discharged by providing a proof directly in Coq. Further, we will investigate the application of our theory of substitutions to extensions of separation logics such as concurrent separation logic [22] and permission-based separation logic [23].

References

[1] J. C. Reynolds, "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*. IEEE Computer Society, 2002, pp. 55–74. [Online]. Available: https://doi.org/10.1109/LICS.2002.1029817

[2] S. S. Ishtiaq and P. W. O'Hearn, "Bi as an assertion language for mutable data structures," *SIGPLAN Not.*, vol. 36, no. 3, p. 14–26, jan 2001. [Online]. Available: https://doi.org/10.1145/373243.375719

[3] H. Yang, *Local reasoning for stateful programs*. University of Illinois at Urbana-Champaign, 2001.

[4] H. Yang and P. W. O'Hearn, "A semantic basis for local reasoning," in *International Conference on Foundations of Software Science and Computation Structures*. Springer, 2002, pp. 402–416.

[5] A. Reynolds, T. King, and V. Kuncak, "Solving quantified linear arithmetic by counterexample-guided instantiation," *Formal Methods Syst. Des.*, vol. 51, no. 3, pp. 500–532, 2017.

[6] A. Niemetz, M. Preiner, A. Reynolds, C. W. Barrett, and C. Tinelli, "Syntax-guided quantifier instantiation," in *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*, ser. Lecture Notes in Computer Science, J. F. Groote and K. G. Larsen, Eds., vol. 12652. Springer, 2021, pp. 145–163.

[7] D. Harel, *First-Order Dynamic Logic*, ser. Lecture Notes in Computer Science. Springer, 1979, vol. 68.

[8] P. Vukmirović, 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*, vol. 24, no. 1, pp. 67–87, 2022.

[9] C. Bannister, P. Höfner, and G. Klein, "Backwards and forwards with separation logic," in *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*, ser. Lecture Notes in Computer Science, J. Avigad and A. Mahboubi, Eds., vol. 10895. Springer, 2018, pp. 68–87.

[10] M. Faisal Al Ameen and M. Tatsuta, "Completeness for recursive procedures in separation logic," *Theoretical Computer Science*, vol. 631, pp. 73–96, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397516300329

[11] J. C. Reynolds, "Intuitionistic reasoning about shared mutable data structure," *Millennial perspectives in computer science*, vol. 2, no. 1, pp. 303–321, 2000.

[12] K. R. Apt and E.-R. Olderog, "Fifty years of hoare's logic," *Formal Aspects of Computing*, vol. 31, no. 6, pp. 751–807, 2019.

[13] M. Tatsuta and W.-N. Chin, "Completeness of separation logic with inductive definitions for program verification," in *International Conference on Software Engineering and Formal Methods*. Springer, 2014, pp. 20–34.

[14] R. Arthan, U. Martin, E. A. Mathiesen, and P. Oliva, "A general framework for sound and complete floyd-hoare logics," *ACM Transactions on Computational Logic (TOCL)*, vol. 11, no. 1, pp. 1–31, 2009.

[15] C. Calcagno, H. Yang, and P. W. O'Hearn, "Computability and complexity results for a spatial assertion language for data structures," in *FSTTCS 2001: Foundations of Software Technology and Theoretical Computer Science*, ser. Lecture Notes in Computer Science, R. Hariharan, V. Vinay, and M. Mukund, Eds. Springer Berlin Heidelberg, 2001, vol. 2245, pp. 108–119.

[16] R. O'Connor, "Classical mathematics for a constructive world," *Mathematical Structures in Computer Science*, vol. 21, no. 4, pp. 861–882, 2011.

[17] M. Sighireanu, 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*. Springer, 2019, pp. 116–132.

[18] A. Reynolds, 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*. Springer, 2016, pp. 244–261.

[19] A. Reynolds, Personal communication.

[20] A. Murali, L. Peña, C. Löding, and P. Madhusudan, "A first-order logic with frames," in *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*, ser. Lecture Notes in Computer Science, P. Müller, Ed., vol. 12075. Springer, 2020, pp. 515–543.

[21] R. Jung, 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*, vol. 28, 2018.

[22] S. Brookes and P. W. O'Hearn, "Concurrent separation logic," *ACM SIGLOG News*, vol. 3, no. 3, pp. 47–65, 2016.

[23] A. Amighi, C. Hurlin, M. Huisman, and C. Haack, "Permission-based separation logic for multithreaded java programs," *Logical Methods in Computer Science*, vol. 11, 2015.

APPENDIX

$S ::= x := [e] \mid [x] := e \mid x := \textbf{cons}(e) \mid \textbf{dispose}(x) \mid x := e \mid S; S \mid \textbf{if } b \textbf{ then } S \textbf{ else } S \textbf{ fi} \mid \textbf{while } b \textbf{ do } S \textbf{ od}$

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

Fig. 5. Syntax and semantics of heap manipulating programs.

$$\frac{\models p \to p' \quad \{p'\}\ S\ \{q'\} \quad \models q' \to q}{\{p\}\ S\ \{q\}}$$

$$\frac{\{p\}\ S_1\ \{r\} \quad \{r\}\ S_2\ \{q\}}{\{p\}\ S_1; S_2\ \{q\}}$$

$$\frac{\{p \wedge b\}\ S_1\ \{q\} \quad \{p \wedge \neg b\}\ S_2\ \{q\}}{\{p\}\ \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}\ \{q\}}$$

$$\frac{\{p \wedge b\}\ S\ \{p\}}{\{p\}\ \textbf{while } b \textbf{ do } S \textbf{ od}\ \{p \wedge \neg b\}}$$

Fig. 6. Hoare's standard proof rules.

$$\{p[x := e]\}\ x := e\ \{p\}$$
$$\{\exists y((e \hookrightarrow y) \wedge p[x := y])\}\ x := [e]\ \{p\}$$
$$\{(x \mapsto -) * ((x \mapsto e) \mathrel{-\!\!*} p)\}\ [x] := e\ \{p\}$$
$$\{\forall y((y \mapsto e) \mathrel{-\!\!*} p[x := y])\}\ x := \textbf{cons}(e)\ \{p\}$$
$$\{(x \mapsto -) * p\}\ \textbf{dispose}(x)\ \{p\}$$

Fig. 7. Global weakest precondition axiomatization (cf. [1], [9], [10]).

$$\{p\}\ x := e\ \{\exists y(p[x := y] \wedge x = e[x := y])\}$$
$$\{p \wedge (e \hookrightarrow -)\}\ x := [e]\ \{(e \mapsto x) * \neg((e \mapsto x) \mathrel{-\!\!*} \neg p)\}$$
$$\{p \wedge (x \hookrightarrow -)\}\ [x] := e\ \{(x \mapsto e) * \neg((x \mapsto -) \mathrel{-\!\!*} \neg p)\}$$
$$\{p\}\ x := \textbf{cons}(e)\ \{(x \mapsto e) * p\}$$
$$\{p \wedge (x \hookrightarrow -)\}\ \textbf{dispose}(x)\ \{\neg((x \mapsto -) \mathrel{-\!\!*} \neg p)\}$$

Fig. 8. Global strongest postcondition axiomatization (cf. [1], [9]), assuming x does not occur in e in the axioms for look-up, mutation, and allocation.