On Separation Logic

The overall contribution of this paper is a new logical foundation of separation logic based on a theory of substitutions which yield both weakest preconditions and strongest postconditions for all four types of basic instructions of separation logic: look-up, mutation, allocation, and dispose.

In contrast to the standard axiomatization in separation logic these substitutions do not generate additional complexity measured by the maximal depth of nested separating conjunctions and separating implications, but provide a direct account of aliasing in terms of basic equational predicate logic. Instead of using heap operations to describe the effect of the instructions, *abstracting* from the logical structure of the given pre/postcondition, we develop a new theory of substitutions for separation logic which describe the effect of the instructions compositionally in terms of the logical structure of the given pre/postcondition.

ACM Reference Format:
. 2022. On Separation Logic . 1, 1 (July 2022), 26 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Program logics. The basic assignment instructions in imperative programming languages describe suitable abstractions of memory updates, as outlined by the (modified) Harvard computer architecture. To reason about such updates, Hoare and Dijkstra introduced a logical view of programs as *predicate transformers*. Following Turing and Floyd, this logical interpretation of programs is based on the use of *assertions* for the logical description of the memory contents.

In program logics, such as Hoare logic, correctness specifications provide a formal relation between a program (the 'how') and its associated pre- and postcondition (the 'what'). This formal relation is axiomatically captured by proof rules which support *compositional* reasoning about various programming constructs. At the leaves of the resulting proof trees are instances of the basic assignment axiom: $\{p[x := t]\}\ x := t\ \{p\}$, where $p[x := t]$ denotes the result of *substituting* the expression t for the free occurrences of x in p. This axiom scheme thus describes assignments as predicate transformers which transform the postcondition in a *weakest precondition* by the operation of substitution.

Harel introduced dynamic logic in [Har79] which generalizes this predicate transformer view of programs by incorporating in the logic of assertions programs as *modalities*. For any statement S of the programming language the assertion $[S]q$ then by definition describes its weakest precondition: whenever $\{p\}\ S\ \{q\}$ holds then $p \to [S]q$ holds in dynamic logic. The proof rules of Hoare logic are then formalized in dynamic logic in terms of these modalities. The above assignment axiom translates to $([x := t]p) \leftrightarrow (p[x := t])$, where p does not contain modalities. Thus in the end, this axiom and the proof rules for the different programming constructs reduce the correctness of a program to verification conditions in the underlying standard predicate logic. This in fact is the ultimate goal of any program logic: to reduce the correctness of a program to a finite number of verification conditions in predicate logic which describe the memory contents, e.g., the relations between the values of the program variables.

Author's address:

XXXX-XXXX/2022/7-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Aliasing. In programming languages arrays and the dynamic allocation of memory locations give rise to *aliasing*, that is, syntactically different expressions which refer to the same memory location. Clearly, the above assignment axiom breaks down for array assignments. For example, to compute the weakest precondition of an array assignment $a[x] := e$, it ignores aliases of the expression $a[x]$, that is, expressions $a[y]$, where the (integer) value of y equals that of x. Gries [Gri81] extended the assignment axiom to arrays by introducing arrays in the assertion language as first-class citizens which denote *functions*. The assertion language further features *expressions* like $a[x] := e$ which denote the corresponding update of the function represented by a. We then have the following axiom for array assignments. $\{p[a := (a[x] := e)]\}\ a[x] := e\ \{p\}$, where the substitution $[a := (a[x] := e)]$ simply replaces every occurrence of the variable a by $(a[x] := e)$. The phenomenon of aliasing is then captured in the logic of assertions by axioms such as $(a[x] := e)(a[y]) =$ **if** $y = x$ **then** e **else** $a[y]$ **fi**. Alternatively, we can introduce a substitution $[a[x] := e]$ which takes into account possible aliases of $a[x]$, e.g., it transforms an expression $a[y]$ by the above conditional expression **if** $y = x$ **then** e **else** $a[y]$ **fi** (see for example [AdBO09b]).

The above approaches for reasoning about arrays, essentially can also be applied to programming constructs which describe the dynamic allocation of memory locations (e.g., pointers in C or object structures in Java). See for example the assertion language underlying the proof system for a basic object-oriented language presented by Abadi and Leino in [AL03], which features explicit references to the heap (the set of allocated objects and their local states) as a function in the assertion language.

Invariance. A crucial (and usually, complicating) aspect of reasoning about program correctness is reasoning about *invariance*, that is, what properties are *not* affected by the program. In Hoare logic one reasons about invariant properties by means of auxiliary rules, the so-called *adaptation* rules. These rules are used to adapt a given correctness formula, for example by adding to the pre- and postcondition an invariant as in the following invariance rule:

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

where the statement S does not assign the variables that occur free in formula r. Such an invariance rule is clearly of limited use in the presence of aliasing. A main challenge thus is to generalize the invariance rule.

Reynolds [Rey02] introduced separation logic for reasoning about invariant properties of dynamic memory allocation by means of the following rule (also called the 'frame rule'):

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

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 assertions **emp** and $x \mapsto e$, which hold if the set of allocated memory locations is empty and 'x points to e', that is, 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 which is directly expressed by $x \mapsto e$, assuming that x denotes the single allocated location, as expressed by the precondition.

This bottom-up approach of separation logic contrasts with the top-down approach of Hoare logic, which (as already remarked above) has as goal to reduce a global specification incrementally to local verification conditions. In order to support a top-down approach in separation logic, a separating implication $p \mathrel{-\!\!*} q$ has been introduced which, roughly, holds if every extension of

the heap satisfies q, if 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 above use of the separating implication increases the complexity of the generated precondition measured by the maximal depth of nested separating conjunction and separating implication connectives (compared to that of the given postcondition). Repeated use of the axiom, e.g. in a sequential composition, thus generates increasingly complex preconditions. Furthermore, the undecidability of the propositional subset of separation logic [BK14] and the non-recursive enumerability of the valid assertions of the first-order language restricted to the 'points to' predicate [CYO01] further complicates reasoning about such generated preconditions.

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. 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 has been defined. Quoting Reynolds: "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." [Rey02] This suggests that there are no such substitutions in separation logic.

Contributions. The overall contribution of this paper is a new theory of substitutions for separation logic which

- describes the effect of the basic heap manipulating instructions compositionally in terms of the logical structure of the given pre/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 separating conjunction/implication connectives.

This result justifies the view that the main purpose of separation logic is to support modular reasoning about unbounded heaps, as formalized by the frame rule, which abstracts from the internal structure of programs. Using separation logic for reasoning about the internal structure, as given by the control flow of the basic instructions, however complicates reasoning about aliasing which is captured only indirectly by the specification of properties of disjoint heaps.

Moreover, if local specifications are expressed without using separating conjunction/implication, our substitution operators do not introduce these connectives when generating verification conditions. As such, this result thus may lead to a better integration of separation logic with existing program verification tools, such as Satisfiability/Satisfiability Modulo Theories (SAT/SMT) solvers, that lack support for separating connectives: local specifications can be proven correct supported by such tools, and the machinery of separation logic can be used for adapting local specifications to global specifications by the application of the frame rule.

More specifically, we have the following main technical results:

- We develop a sound and complete *weakest precondition* axiomatization of the basic heap manipulating instructions in the standard classical interpretation of separation logic (Figure 3). This axiomatization is based on the standard notion of substitution (for the axiomatization of the look-up instruction), and two new substitution operations for the axiomatization of mutation and dispose instructions, respectively. The allocation instruction is axiomatized as a particular mutation instruction.
- We illustrate the expressiveness of our approach in a completeness proof of the local axioms (Figure 4) which also provides a forwards orientation like in the standard postcondition axiomatization of Hoare logic.

Basic assignment $\langle x := e, h, s\rangle \Rightarrow (h, s[x := s(e)])$,
Look-up $\langle x := [e], h, s\rangle \Rightarrow (h, s[x := h(s(e))])$ if $s(e) \in dom(h)$,
$\langle x := [e], h, s\rangle \Rightarrow$ **fail** if $s(e) \notin dom(h)$,
Mutation $\langle [x] := e, h, s\rangle \Rightarrow (h[s(x) := s(e)], s)$ if $s(x) \in dom(h)$,
$\langle [x] := e, h, s\rangle \Rightarrow$ **fail** if $s(x) \notin dom(h)$,
Allocation $\langle x := \mathbf{new}(e), h, s\rangle \Rightarrow (h[n := s(e)], s[x := n])$ if $n \notin dom(h)$.
Dispose $\langle [x] := \bot, h, s\rangle \Rightarrow (h[s(x) := \bot], s)$ if $s(x) \in dom(h)$,
$\langle [x] := \bot, h, s\rangle \Rightarrow$ **fail** if $s(x) \notin dom(h)$

Fig. 1. Operational semantics of basic (heap-manipulating) instructions of our programming language

- We further apply our theory in the definition of a sound and complete *strongest postcondition* axiomatization of the basic heap manipulating instructions (Figure 6).
- Finally, we have formally verified using the proof assistant Coq the correctness of the substitutions, and the soundness and completeness results of the novel weakest precondition and strongest postcondition axiomatizations.

2 SEPARATION LOGIC: SYNTAX AND SEMANTICS

We follow the presentation of separation logic in [Rey02]. A heap[1] h is represented by a *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, denoted by $h\#h$, iff $dom(h) \cap dom(h') = \emptyset$. A heap h is partitioned in h_1, 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 $i \in \{1, 2\}$. Further, for any two disjoint heaps h_1, h_2 there always exists a unique heap h such that $h = h_1 \uplus h_2$.

Let V denote 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 integer 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 integer 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 [Rey02] expressions thus do not refer to the heap. It is important to observe here that this is a crucial choice underlying the design of separation logic. 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 $h[n := \bot]$. Similarly, we have $dom(h[n := \bot]) = dom(h) \setminus \{n\}$.

We define the operational semantics of the basic instructions of our programming language (see Figure 1). We focus on the basic instructions below only for technical convenience: one may extend the semantics for programming constructs such as sequential composition, conditional, and while statements.

[1] 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_1, h_2 are all heaps.

$h, s \models b$ iff $s(b) = \mathbf{true}$,
$h, s \models \mathbf{emp}$ iff $dom(h) = \emptyset$,
$h, s \models (e \mapsto e')$ iff $dom(h) = \{s(e)\}$ and $h(s(e)) = s(e')$,
$h, s \models (e \hookrightarrow e')$ iff $s(e) \in dom(h)$ and $h(s(e)) = s(e')$,
$h, s \models (p \wedge q)$ iff $h, s \models p$ and $h, s \models q$,
$h, s \models (p \vee q)$ iff $h, s \models p$ or $h, s \models q$,
$h, s \models (p \rightarrow q)$ iff $h, s \models p$ implies $h, s \models q$,
$h, s \models \exists x p$ iff $h, s[x := n] \models p$ for some n,
$h, s \models \forall x p$ 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 partition $h = h_1 \uplus h_2$,
$h, s \models (p \mathrel{-\!\!*} q)$ iff for all h' disjoint from h: $h', s \models p$ implies $h \uplus h', s \models q$.

Fig. 2. Classical semantics of separation logic assertions (CSL) with respect to a heap h and store s

Note the distinction between the **fail** configuration and the indeterminacy of allocation when the heap is full, which is of importance when giving semantics to correctness specifications. For notational convenience only we restrict to mutation instructions $[x] := e$ and dispose instructions $[x] := \bot$. In Subsection 3.1 we show how for example mutation instructions like $[e'] := e$ can be axiomatized in terms of the two assignments $x := e'$ and $[x] := e$, where x is a fresh variable.

The assertion language of separation logic (used to formalize the pre- and postconditions of statements) is inductively defined:

$$p, q ::= b \mid \mathbf{emp} \mid (e \mapsto e') \mid (e \hookrightarrow e') \mid (p \wedge q) \mid (p \vee q) \mid (p \rightarrow q) \mid \exists x p \mid \forall x p \mid (p * q) \mid (p \mathrel{-\!\!*} q)$$

We further have the usual abbreviations, e.g., $\neg p$ denotes the assertion $(p \rightarrow \mathbf{false})$. Compound assertions are constructed by the standard logical connectives of first-order predicate logic, the separating conjunction $(p * q)$, and the separating implication $(p \mathrel{-\!\!*} q)$. We may drop matching parentheses if doing so would not give rise to ambiguity.

By $h, s \models p$ we denote the truth definition of *classic* separation logic (CSL). See Figure 2 for its inductive definition. Validity of an assertion p is denoted by $\models p$. The assertions **emp**, $e \mapsto e'$, and $e \hookrightarrow e'$ are the basic heap assertions. The assertion **emp** states that the heap is empty, the assertion $(e \mapsto e')$ states that the value of e' is stored at the location e and e is the only allocated location, and the assertion $(e \hookrightarrow e')$ asserts that the value of e' is stored at the location e. By $(e \mapsto -)$ we denote the assertion $\exists y (e \mapsto y)$ for some fresh $y \notin var(e)$, which thus states that exactly the location e is allocated. Similarly, by $(e \hookrightarrow -)$ we denote $\exists y (e \hookrightarrow y)$ for some fresh $y \notin var(e)$, which states that at least the location e is allocated. The assertion $\neg(e \hookrightarrow -)$ is abbreviated to $(e \not\hookrightarrow -)$. Similarly, $e \not\hookrightarrow e'$ abbreviates $\neg(e \hookrightarrow e')$. The assertion **emp** can be equivalently expressed by $\forall x (x \not\hookrightarrow -)$. Further, $(e \hookrightarrow e')$ can be expressed by $(e \mapsto e') * \mathbf{true}$. Conversely $(e \mapsto e')$ can be expressed by the conjunction of $(e \hookrightarrow e')$ and $\forall x((x \hookrightarrow -) \rightarrow x = e)$ for some fresh $x \notin var(e)$. The assertion $(e \hookrightarrow e')$ is implied by $(e \mapsto e')$, and as such $(e \hookrightarrow e')$ can be viewed as the more basic assertion.

It is worthwhile to observe here that there exists a straightforward formalization of the above semantics of assertions in an extension of the assertion language with second-order quantification over partial unary functions. However, to avoid the need for explicit reference to the 'undefined', we represent these partial functions by binary relations (which as such satisfy the functional property $\forall x, y, z (r(x, y) \wedge r(x, z) \rightarrow y = z)$). For any assertion p as defined above we define inductively the second-order assertion $p(r)$ as follows (restricting without loss of generality to basic heap assertions $e \hookrightarrow e'$):

Basic assignment $\{p[x := e]\}\ x := e\ \{p\}$
Look-up $\{\exists y((e \hookrightarrow y) \wedge p[x := y])\}\ x := [e]\ \{p\}$ where y is fresh
Mutation $\{(x \hookrightarrow -) \wedge p[\langle x \rangle := e]\}\ [x] := e\ \{p\}$
Allocation $\{\forall x((x \not\hookrightarrow -) \rightarrow p[\langle x \rangle := e])\}\ x := \mathbf{new}(e)\ \{p\}$
Dispose $\{(x \hookrightarrow -) \wedge p[\langle x \rangle := \bot]\}\ [x] := \bot\ \{p\}$

Fig. 3. WP-CSL

Definition 1 (Logical formalization semantics assertions).

- $b(r) = b$,
- $(e \hookrightarrow e')(r) = r(e, e')$,
- $(p \wedge q)(r) = p(r) \wedge q(r)$, *similarly for* $\vee, \rightarrow$
- $(\forall x(p))(r) = \forall x(p(r))$, *similarly for* $\exists x$
- $(p * q)(r) = \exists r_1, r_2(r = r_1 \uplus r_2 \wedge p(r_1) \wedge q(r_2)))$,
- $(p \mathrel{-\!\!*} q)(r) = \forall r''((r'' = r \uplus r' \wedge p(r')) \rightarrow q(r''))$.

where $r'' = r \uplus r'$ *abbreviates that* r'' *is the disjoint union of the heaps represented by* r *and* r'.

We denote by $s \models p(r)$ the standard truth definition, where the store s additionally interprets the second-order binary variables, e.g., $s(\hookrightarrow)$ represents the current heap. We skip the details of the standard truth definition of $s \models p(r)$ The following proposition states the correctness of this translation.

Proposition 1 (Correctness second-order translation). *For any assertion* p *in separation logic, we have* $h, s \models p$ *if and only if* $s[r := graph(h)] \models p(r)$, *where* $graph(h)$ *denotes the binary relation representing the graph of partial unary function* h.

In particular we thus have $h, s \models p$ if and only if $s \models p(\hookrightarrow)$. The proof of this proposition proceeds by a straightforward induction on p.

A correctness specification $\{p\}\ S\ \{q\}$ is a triple that consists of a precondition p, a program S, and a postcondition q. We consider correctness specifications in the sense of strong partial correctness, which ensures absence of explicit failure. Formally, following [Rey02], the validity of a correctness specification, denoted $\models \{p\}\ S\ \{q\}$, is defined as: if $h, s \models p$, then $\langle S, h, s\rangle \Rightarrow \mathbf{fail}$ and also $\langle S, h, s\rangle \Rightarrow (h', s')$ implies $h', s' \models q$ for all h', s'.

3 WEAKEST PRECONDITION AXIOMATIZATION

Figure 3 contains a weakest precondition axiomatization (WP-CSL) of the basic instructions in classic separation logic. For technical convenience only, we require in the axiomatization of $x := \mathbf{new}(e)$ that x does not appear in e (see Subsection 3.1 for how to remove 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.

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 also observed in [Rey02], there is no need for introducing in the axiomatization 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:

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

For example, we have
$h, s \models (p * q)[x := e]$
iff (definition substitution)
$h, s \models (p[x := e]) * (q[x := e])$
iff (semantics for separating conjunction)
$h_1, s \models p[x := e]$ and $h_2, s \models q[x := e]$, for some h_1, h_2 with $h_1 \uplus h_2 = h$
iff (induction hypothesis)
$h_1, s[x := s(e)] \models p$ and $h_2, s[x := s(e)] \models q$ for some h_1, h_2 with $h_1 \uplus h_2 = h$
iff (semantics for separating conjunction)
$h, s[x := s(e)] \models p * q$. □

Next we introduce a substitution for *heap updates*. In contrast to the mutation instruction, the pseudo 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 $\langle x \rangle := e$ interpreted as a heap update:

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

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 $x := \mathbf{new}(e)$ does not require that x is allocated).

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* the heap update $\langle x \rangle := e$. 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 $\mathbf{emp}[\langle x \rangle := e] = \mathbf{false}$. For $e' \mapsto e''$ to hold after the update we must have that *before* the update at most x is allocated (formally expressed by $(\mathbf{emp} \vee x \mapsto -)$), and $x = e'$ and $e'' = e$. 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 $x = e'$. The cases of the standard logical connectives are self-evident. 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. 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.

Definition 2 (Substitution for heap updates). *We define $p[\langle x\rangle := e]$ recursively on p (assuming that the variables of e and x do not occur bound in p).*

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

As a simple example we have the following instance of the above backwards mutation axiom:

$$\{(x \hookrightarrow -) \wedge ((y = x \wedge z = 0) \vee (y \neq x \wedge y \hookrightarrow z))\}\ [x] := 0\ \{y \hookrightarrow z\}$$

Compare this with the standard backwards rule that yields the weakest precondition:

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

This precondition introduces a nested application of separation conjunction and implication. Semantically this comes down to *nested alternating quantifiers*! In particular, we have the following second-order formulation of the meaning of the above precondition (applying the translation in Definition 1):

$$\exists r_1, r_2(\hookrightarrow = r_1 \uplus r_2 \wedge \exists y(r_1(x, y)) \wedge \forall r'((r' \# r_2 \wedge r'(x, 0)) \rightarrow \exists r''(r'' = r' \uplus r_2 \wedge r''(y, z)))$$

Note that the $\forall r'$ subformula (with the postcondition), arising from the separating implication, is nested inside the $\exists r_1, r_2$ top-level formula arising from the separating conjunction. Reasoning about validity of assertions involving nested quantification is notoriously hard: typically one has to find suitable instantiations of the quantified variables (here, heaps) and these instantiations may depend on (the instantiations of) all less deeply nested variables. Consider a simple program that consists of three consecutive mutations. Thus, 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 2 yields no additional nesting of quantifiers and makes potential aliasing immediately explicit at the syntactic level.

In the above backwards axiomatization of the allocation instruction $x := \mathbf{new}(e)$ we assume the variables of e and x do not occur bound in p, but also that x does not occur in e. How to reduce the case in which x does occur in e to this one is discussed in the next section below. We show how to use this axiom to derive

$$\{\mathbf{emp}\}\ x := \mathbf{new}(e)\ \{\forall y((y \hookrightarrow -) \rightarrow y = x)\}$$

First we calculate $\forall y(y \hookrightarrow -) \rightarrow y = x))[\langle x\rangle := e]$, which gives $\forall y(((y = x \wedge y \hookrightarrow e) \vee (y \neq x \wedge y \hookrightarrow -)) \rightarrow y = x)$. Under the assumption that $(x \not\hookrightarrow -)$ this reduces further to $\forall y(y \not\hookrightarrow -)$, that is, to **emp**. Instantiating the backwards allocation axiom of separation logic we obtain $\{\forall x(x \mapsto e \mathrel{-\!\!*} \forall y((y \hookrightarrow -) \rightarrow y = x)\}\ x := \mathbf{new}(e)\ \{\forall y(y \hookrightarrow - \rightarrow y = x)\}$. 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.

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.

- $h, s \models b[\langle x\rangle := e]$
  iff (semantics boolean expressions: they do not depend on the heap)
  $s(b[\langle x\rangle := e]) = \textbf{true}$
  iff (Definition 2, substitution for boolean expressions)
  $s(b) = \textbf{true}$
  iff (semantics boolean expressions)
  $h[s(x) := s(e)], s \models b$.
- $h, s \models \textbf{emp}[\langle x\rangle := e]$
  iff (definition substitution)
  $h, s \models \textbf{false}$
  iff (semantics assertions: no heap/state pair satisfies **false**)
  $h[s(x) := s(e)], s \models \textbf{false}$
  iff ($dom(h[s(x) := s(e)]) \neq \emptyset$)
  $h[s(x) := s(e)], s \models \textbf{emp}$
  For the last step, observe that in separation logic there are no undefined integer expressions e such that $s(e) = \bot$. Hence $s(x) \in dom(h[s(x) := s(e)])$, thus $h[s(x) := s(e)], s \not\models \textbf{emp}$.
- $h, s \models (e' \mapsto e'')[\langle x\rangle := e]$
  iff (definition substitution for points-to)
  $h, s \models (\textbf{emp} \vee x \mapsto -) \wedge x = e' \wedge e'' = e$
  iff (semantics assertions)
  $dom(h) \subseteq \{s(x)\}$, $s(x) = s(e')$ and $s(e) = s(e'')$
  iff (definition $h[s(x) := s(e)]$)
  $dom(h[s(x) := s(e)]) = \{s(x)\}$ and $s(x) = s(e')$, and $s(e) = s(e'')$
  iff (definition $h[s(x) := s(e)]$)
  $dom(h[s(x) := s(e)]) = \{s(x)\}$, $h[s(x) := s(e)](s(e')) = s(e'')$
  iff (semantics points-to)
  $h[s(x) := s(e)], s \models e' \mapsto e''$.
- $h, s \models (e' \hookrightarrow e'')[\langle x\rangle := e]$
  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 $h[s(x) := s(e)]$)
  $s(e') \in dom(h[s(x) := s(e)])$ and $h[s(x) := s(e)](s(e')) = s(e'')$.
  iff (semantics points-to)
  $h[s(x) := s(e)], s \models e' \hookrightarrow e''$.
- $h, s \models (p \wedge q)[\langle x\rangle := e]$
  iff (definition substitution)
  $h, s \models (p[\langle x\rangle := e]) \wedge (q[\langle x\rangle := e])$
  iff (semantics conjunction)
  $h, s \models p[\langle x\rangle := e]$ and $h, s \models q[\langle x\rangle := e]$
  iff (induction hypothesis)
  $h[s(x) := s(e)], s \models p$ and $h[s(x) := s(e)], s \models q$

iff (semantics conjunction)
$h[s(x) := s(e)], s \models p \wedge q$.

- $h, s \models (\exists y p)[\langle x \rangle := e]$
  iff (definition substitution)
  $h, s \models \exists y (p[\langle x \rangle := e])$
  iff (semantics existential quantification)
  $h, s[y := n] \models p[\langle x \rangle := e]$, for some n
  iff (induction hypothesis)
  $h[s(x) := s(e)], s[y := n] \models p$, for some n
  iff (semantics existential quantification)
  $h[s(x) := s(e)], s \models \exists y(p)$.
  Note that y is distinct from x, and y does not appear in e (otherwise one can rename the bound variable), and so $s[y := n](x) = s(x)$ and $s[y := n](e) = s(e)$.
- Let $p' = p \wedge x \not\mapsto -$ and, similarly, $q' = q \wedge x \not\mapsto -$.
  $h, s \models (p * q)[\langle x \rangle := e]$
  iff (definition substitution)
  $h, s \models (p[\langle x \rangle := e] * q') \vee (p' * q[\langle x \rangle := e])$
  iff (semantics disjunction)
  $h, s \models p[\langle x \rangle := e] * q'$ or $h, s \models p' * q[\langle x \rangle := e]$.
  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 $h_2, s \models q \wedge x \not\mapsto -$. We then proceed as follows:
  $h_1, s \models p[\langle x \rangle := e]$ and $h_2, s \models q \wedge x \not\mapsto -$
  iff (induction hypothesis)
  $h_1[s(x) := s(e)], s \models p$ and $h_2, s \models q \wedge x \not\mapsto -$
  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)
  $h, s \models (p \wedge x \not\mapsto -) \mathrel{-\!\!*} (q[\langle x \rangle := e])$
  iff (semantics separating implication)
  for every h' disjoint from h: if $h', s \models p \wedge x \not\mapsto -$ then $h \uplus h', s \models q[\langle x \rangle := e]$
  iff (induction hypothesis)
  for every h' disjoint from h: if $h', s \models p \wedge x \not\mapsto -$ 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 *heap clear*. In contrast to the dispose instruction, the pseudo instruction $\langle x \rangle := \bot$ does not require that the updated location x is already allocated. Formally, we have $\langle \langle x \rangle := \bot, h, s \rangle \Rightarrow (h[s(x) := \bot], s)$, unconditionally.

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. 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. For $e \hookrightarrow e'$ to hold after executing $\langle x \rangle := \bot$, we must initially have that $x \neq e$ and $e \hookrightarrow e'$. The cases of the standard logical connectives are self-evident. 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 with *arbitrary* values for the location x.

Definition 3 (Substitution for heap clear). *We define $p[\langle x \rangle := \bot]$ recursively on p (assuming that x does not occur bound in p).*

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

As a simple example, we have that $(\forall y, z(y \not\hookrightarrow z))[\langle x \rangle := \bot]$, where $\forall y, z(y \not\hookrightarrow z)$ characterizes the empty heap **emp**, 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.
- $h, s \models \mathbf{emp}[\langle x \rangle := \bot]$
  iff (definition substitution)
  $h, s \models \mathbf{emp} \vee x \mapsto -$
  iff (semantics assertions)
  $dom(h) \subseteq \{s(x)\}$
  iff (basic set-theory, definition $h[\langle s(x) \rangle := \bot]$)
  $dom(h[\langle s(x) \rangle := \bot]) = \emptyset$

iff (semantics assertions)
$h[\langle s(x)\rangle := \bot], s \models \mathbf{emp}$.

- $h, s \models (e \mapsto e')[\langle x\rangle := \bot]$
  iff (definition substitution)
  $h, s \models e \hookrightarrow e' \wedge x \neq e \wedge \forall y((y \hookrightarrow -) \rightarrow (y = e \vee y = x))$
  iff (semantics assertions)
  $h(s(e)) = s(e')$, $s(x) \neq s(e)$, and $\{s(e)\} \subseteq dom(h) \subseteq \{s(x), s(e)\}$
  iff (basic set-theory, definition $h[\langle s(x)\rangle := \bot]$)
  $h[\langle s(x)\rangle := \bot](s(e)) = s(e')$ and $dom(h[\langle s(x)\rangle := \bot]) = \{s(e)\}$
  iff (semantics assertions)
  $h[\langle s(x)\rangle := \bot], s \models e \mapsto e$.
- $h, s \models (e \hookrightarrow e')[\langle x\rangle := \bot]$
  iff (definition substitution)
  $h, s \models (e \hookrightarrow e') \wedge x \neq e$
  iff (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)
  $h, s \models ((p \wedge x \not\hookrightarrow -) \mathrel{-\!\!*} q[\langle x\rangle := \bot]) \wedge \forall y(p[\langle x\rangle := y] \mathrel{-\!\!*} q[\langle x\rangle := y])$
  where y is a fresh variable
  iff (see below)
  $h[s(x) :=\bot], s \models p \mathrel{-\!\!*} q$.
  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 $s(x) \in dom(h')$. 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 $s(x) \notin dom(h')$. 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 $s(x) \notin dom(h')$.
- 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.

In the following theorem, we restrict ourselves to allocation instructions of the form $x := \mathbf{new}(e)$ where x does not occur in e. This restriction is for technical convenience only, see Section 3.1 how our result can be extended to the general case.

Theorem 1 (Soundness and completeness WP-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 WP-CSL (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.

Basic assignment.

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

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 $s(e) \in dom(h)$, 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 $\models p \rightarrow \exists y((e \hookrightarrow y) \wedge (q[x := y]))$:
  Let $\models \{p\}\ x := [e]\ \{q\}$ and assume $h, s \models p$. So we have that $\langle x := [e], h, s\rangle \nRightarrow \textbf{fail}$, that is, $s(e) \in dom(h)$. 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, $h, s \models \exists y((e \hookrightarrow y) \wedge (q[x := y]))$.

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 $s(x) \in dom(h)$ 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 $\models p \rightarrow ((x \hookrightarrow -) \wedge (q[\langle x\rangle := e]))$:
  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 $\langle [x] := e, h, s\rangle \nRightarrow \textbf{fail}$. 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 $h, s \models (x \hookrightarrow -) \wedge (q[\langle x\rangle := e])$.

Allocation.

- $\models \{\forall x((x \not\hookrightarrow -) \rightarrow (p[\langle x\rangle := e]))\}\ x := \textbf{new}(e)\ \{p\}$:
  Let $h, s \models \forall x((x \not\hookrightarrow -) \rightarrow (p[\langle x\rangle := e]))$ and $\langle x := \textbf{new}(e), h, s\rangle \Rightarrow (h[n := s(e)], s[x := n])$ for some $n \notin dom(h)$. We need to show $h[n := s(e)], s[x := n] \models p$. It follows from $h, s \models \forall x((x \not\hookrightarrow -) \rightarrow (p[\langle x\rangle := e]))$ 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.
- $\models \{p\}\ x := \textbf{new}(e)\ \{q\}$ implies $\models p \rightarrow (\forall x(x \not\hookrightarrow - \rightarrow (q[\langle x\rangle := e])))$:
  Let $h, s \models p$ and $h, s[x := n] \models \neg(x \hookrightarrow -)$, for some n. It follows that $n \notin dom(h)$. So we have $\langle x := \textbf{new}(e), h, s\rangle \Rightarrow (h[n := s(e)], s[x := n])$, and so it follows that we have $h[n := s(e)], s[x := n] \models q$ from $\models \{p\}\ x := \textbf{new}(e)\ \{q\}$. 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 $s[x := n](e) = s(e)$).

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 completeness proof can be extended in a straightforward manner to a Hoare logic for sequential programs which does not require the frame rule, or any other adaptation rule besides the consequence rule (see for example [AdBO09a]). In Section 6 we discuss the use of the frame rule in reasoning about recursive procedures.

3.1 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. As already observed above, 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 $x := \mathbf{new}(e)$ in the case where x does occur in e, since the instruction $x := \mathbf{new}(e)$ can be simulated by $z := x;\ x := \mathbf{new}(e[x := z])$ where z is a fresh variable. Applying the (standard) sequential composition rule and the axiom for basic assignments, it is straightforward to derive the following generalized backwards allocation axiom:

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

where y is fresh and the variables of e and x do not occur bound in p.

Reynolds introduced in [Rey02] 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 $\bar{e} = e_1, \ldots, e_n$, $\bar{m} = m_1, \ldots, m_n$, $m_{i+1} = m_i + 1$, for $i = 1, \ldots, n-1$, $\{m_1, \ldots, m_n\} \cap dom(h) = \emptyset$, and, finally,

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

Let $\bar{e}'$ denote a sequence of expressions $e'_1, \ldots e'_n$ such that e'_1 denotes the variable x and e'_i denotes the expression $x + (i-1)$, for $i = 2, \ldots, n$. The storage of the values of $e_1, \ldots, e_n$ then can be modeled by a sequence of substitutions $[[e'_i] := e_i]$, for $i = 1, \ldots, n$. We abbreviate such a sequence of substitutions by $[[\bar{e}'] := \bar{e}]$. 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

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

3.2 On symbolic execution

We briefly describe an application of our weakest precondition axiomatization to the generation of path-conditions by symbolic execution. In our setting a path-condition is an assertion on the initial heap and store which enforces the execution of a given path of the program. We follow the approach of [dBB21] by applying our weakest precondition axiomatization to symbolic traces consisting of the basic instructions and Boolean expressions, indicating which branch was taken of

an if- or while-statement. Such traces can be easily obtained by a straightforward unfolding of the program. We will define the path condition of a symbolic trace sing using a weakest precondition axiomatization (such as the one we presented above).

Definition 4 (Path condition). *Let We define the path condition $path(\rho)$ in CSL of a symbolic execution trace inductively by (here ϵ denotes the empty path):*

$$\begin{array}{ll} path(\epsilon) & = \mathbf{true} \\ path(b \cdot \rho) & = path(\rho) \wedge b \\ path(s \cdot \rho) & = wp(s, path(\rho)) \text{ where } s \text{ is a basic statement (see Figure 3).} \end{array}$$

Note that the resulting path-condition does not involve the connectives of separating conjunction/implication.

Extending in a straightforward manner the transition relation $\Rightarrow$ to configurations $\langle \rho, h, s \rangle$, we have the following theorem, where $\langle \rho, h, s \rangle \Downarrow$ indicates that there exists a successful terminating computation starting from $\langle \rho, h, s \rangle$.

Theorem 2 (Soundness and completeness symbolic execution). *We have that*

$$\langle \rho, h, s \rangle \Downarrow \text{ if and only if } h, s \models path(\rho)$$

Proof. The proof proceeds by a straightforward induction on ρ, using the soundness and completeness of the weakest precondition axiomatization. □

Note that for a given trace ρ we can actually verify in separation logic that $\{p\}\ \rho\ \{q\}$ holds by validating the implication $(p \wedge path(\rho)) \rightarrow q$. So that we cover all executions of ρ, instead of executing ρ on a set of inputs.

How do the novel substitutions compare to the traditional weakest preconditions (taken from e.g. [Rey02])? As an example, consider the program

$$[0] := 0; x := [1]; \mathbf{if}\ x > 0\ \mathbf{then}\ m := x\ \mathbf{else}\ m := 0\ \mathbf{fi}.$$

This program first assigns 0 to the heap at location 0, and calculates and then stores the maximum value of the heap at location 0 and 1. Let us find the path condition in case this maximum value is at location 1 (i.e. the $x > 0$ branch is taken). This entails calculating the path condition of the symbolic trace $[0] := 0 \cdot x := [1] \cdot x > 0 \cdot m := x$:

$$\begin{array}{ll} & path([0] := 0 \cdot x := [1] \cdot x > 0 \cdot m := x) \\ = & path(x := [1] \cdot x > 0 \cdot m := x)[[0] := 0] \wedge 0 \hookrightarrow - \\ = & \exists y(path(x > 0 \cdot m := x)[x := y] \wedge 1 \hookrightarrow y)[[0] := 0] \wedge 0 \hookrightarrow - \\ = & \exists y((path(m := x \cdot \epsilon) \wedge x > 0)[x := y] \wedge 1 \hookrightarrow y)[[0] := 0] \wedge 0 \hookrightarrow - \\ = & \exists y((path(\epsilon)[m := x] \wedge x > 0)[x := y] \wedge 1 \hookrightarrow y)[[0] := 0] \wedge 0 \hookrightarrow - \\ = & \exists y((\mathbf{true} \wedge x > 0)[x := y] \wedge 1 \hookrightarrow y)[[0] := 0] \wedge 0 \hookrightarrow - \\ = & \exists y(y > 0 \wedge 1 \hookrightarrow y)[[0] := 0] \wedge 0 \hookrightarrow - \\ = & \exists y(y > 0 \wedge ((0 = 1 \wedge y = 0) \vee (0 \neq 1 \wedge 1 \hookrightarrow y))) \wedge 0 \hookrightarrow - \\ = & \exists y(y > 0 \wedge 1 \hookrightarrow y) \wedge 0 \hookrightarrow - \end{array}$$

In the first five lines, we reduced the symbolic path one instruction at a time using the path condition definition and the weakest precondition from Figure 3. In the last 4 lines, the novel substitutions (together with some basic arithmetic and logical simplifications) came in handy to calculate the desired path condition. The resulting path condition expresses that the heap at location 0 must be allocated, and at location 1 it must store a positive value. This is indeed exactly the precondition that is needed to ensure that the desired execution path from the symbolic trace is taken and the program does not fail.

Basic assignment $\{\mathbf{true}\}\ x := e\ \{x = e\}$
Look-up $\{e \hookrightarrow -\}\ x := [e]\ \{e \hookrightarrow x\}$
Mutation $\{x \mapsto -\}\ [x] := e\ \{x \mapsto e\}$
Allocation $\{\mathbf{emp}\}\ x := \mathbf{new}(e)\ \{x \mapsto e\}$
Dispose $\{x \mapsto -\}\ [x] :=\bot\ \{\mathbf{emp}\}$

Fig. 4. Local axioms in CSL

Next, we compute the path condition with the traditional weakest precondition definitions of separation logic:

$$\begin{aligned}
& path([0] := 0 \cdot x := [1] \cdot x > 0 \cdot m := x) \\
= \ & wp([0] := 0, path(x := [1] \cdot x > 0 \cdot m := x)) \\
= \ & wp([0] := 0, wp(x := [1], path(x > 0 \cdot m := x))) \\
= \ & wp([0] := 0, wp(x := [1], x > 0 \wedge path(m := x))) \\
= \ & wp([0] := 0, wp(x := [1], x > 0 \wedge wp(m := x, path(\epsilon)))) \\
= \ & wp([0] := 0, wp(x := [1], x > 0 \wedge wp(m := x, \mathbf{true}))) \\
= \ & wp([0] := 0, wp(x := [1], x > 0 \wedge \mathbf{true})) \\
= \ & wp([0] := 0, \exists y(1 \hookrightarrow y \wedge y > 0)) \\
= \ & 0 \mapsto - * ((0 \mapsto 0) \mathrel{-\!\!*} \exists y(1 \hookrightarrow y \wedge y > 0))
\end{aligned}$$

The classical weakest preconditions for separation logic thus yields path conditions that are considerably more intricate than the simple path condition resulting from our substitutions. The difference in complexity increases especially for symbolic paths with multiple mutation or allocation operations.

4 STRONGEST POSTCONDITION AXIOMATIZATION

Before we discuss strongest postcondition axiomatizations of CSL, it should be noted that in general the semantics of Hoare 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).

Reynolds in [Rey02] distinguishes between local and global axioms. Some of the global axioms provide a backwards (weakest precondition) axiomatization. We illustrate here the expressiveness of the $[\langle x \rangle := e]$ substitution in a completeness proof of the local proof axioms in Figure 4 which also provides a forwards orientation like in the standard postcondition axiomatization of Hoare logics. Here we assume without loss of generality that in the instructions $x := e$, $[x] := e$, and $x := \mathbf{new}(e)$, the variable x does not occur in e (note that for example $x := e$ can be simulated by $z := x; x := (e[x := z])$).

The following completeness result assumes the invariance rule for the basic assignment and look-up instruction:

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

where S is a basic assignment $x := e$ or a look-up instruction $x := [e]$, and r is an assertion which does not contain (free) occurrences of x. It is straightforward to check that this restricted invariance rule is sound (note that it is not sound for the mutation and allocation instructions). Of particular

Mutation $\{p \wedge x \hookrightarrow -\}\ [x] := e\ \{(x \mapsto e) * \exists y((x \mapsto y) \mathrel{-\!\!*} p)\}$
Dispose $\{p \wedge x \hookrightarrow -\}\ [x] := \bot\ \{(x \not\hookrightarrow -) \wedge \exists y((x \mapsto y) \mathrel{-\!\!*} p)\}$

Fig. 5. Global strongest postconditions mutation/dispose instruction

interest is that below we use the heap update substitution $[\langle x \rangle := e]$ to construct a so-called frame in the application of the frame rule for both the mutation and dispose instruction.

Theorem 3 (Completeness local axioms CSL). *For any basic instruction S, if $\models \{p\}\ S\ \{q\}$ then $\{p\}\ S\ \{q\}$ is derivable from the corresponding axiom in Figure 4 by the frame rule, the consequence rule, and the invariance rule for basic assignment and look-up.*

Proof. Let $\models \{p\}\ S\ \{q\}$.

Basic assignment. By the invariance rule for basic assignments, we first derive

$$\{\mathbf{true} \wedge \exists x(p)\}\ x := e\ \{x = e \wedge \exists x(p)\}$$

Clearly, p implies $\exists x(p)$. Let $h, s \models x = e$, that is, $s(x) = s(e)$, and $h, s[x := n] \models p$, for some n. From the assumption $\models \{p\}\ x := e\ \{q\}$ we then derive $h, s[x := s[x := n](e)] \models q$, that is, $h, s \models q$ (since $s[x := n](e) = s(e) = s(x)$).

Look-up. By the restricted invariance rule, we first derive

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

Since $\models \{p\}\ x := [e]\ \{q\}$, we have that p implies $e \hookrightarrow -$, and so p implies $\exists x(p) \wedge (e \hookrightarrow -)$. On the other hand, let $h(s(e)) = s(x)$ and $h, s' \models p$, where $s' = s[x := n]$, for some n. From the assumption $\models \{p\}\ x := [e]\ \{q\}$ we then derive $h, s[x := h(s'(e))] \models q$, that is, $h, s \models q$ (since x does not occur in e and $h(s(e)) = s(x)$, we have that $s[x := h(s'(e))] = s[x := h(s(e))] = s$).

Mutation. Let p' denote $\exists y(p[\langle x \rangle := y])$. By the frame rule, we first derive

$$\{(x \mapsto -) * p'\}\ [x] := e\ \{(x \mapsto e) * p'\}$$

Let $h, s \models p$. We show that $h, s \models (x \mapsto -) * p'$: Since $\models \{p\}\ [x] := e\ \{q\}$ we have that $s(x) \in dom(h)$. So we can introduce the split $h = h_1 \uplus h_2$ such that $h_1, s \models x \mapsto -$ and $h_2 = h[s(x) := \bot]$. 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 $(x \mapsto e) * p'$ implies q: Let $h, s \models (x \mapsto e) * p'$. So there exists a split $h = h_1 \uplus h_2$ such that $h_1, s \models x \mapsto e$ and $h_2, s \models p'$. Let n be such that $h_2, s[y := n] \models p[[x] := y]$. By the above substitution lemma we have that $h_2, s[y := n] \models p[\langle x \rangle := y]$ if and only if $h_2[s(x) := n], s \models p$ (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 $h_2 = h[s(x) := \bot]$).

Allocation. By the frame rule, we first derive

$$\{\mathbf{emp} * \exists x(p)\}\ x := \mathbf{new}(e)\ \{(x \mapsto e) * \exists x(p)\}$$

Clearly, p implies $\mathbf{emp} * \exists x(p)$. On the other hand, let $h, s \models (x \mapsto e) * \exists x(p)$. So there exists a split $h = h_1 \uplus h_2$ such that $h_1, s \models x \mapsto e$ and $h_2, s[x := n] \models p$, for some n. Since $\models \{p\}\ x := \mathbf{new}(e)\ \{q\}$, we derive that $h_2[s(x) := s[x := n](e)], s \models q$, that is, $h, s \models q$ (note that $s(x) \notin dom(h_2)$ and, since x does not appear in e, we have $s[x := n](e) = s(e)$, and thus $h = h_2[s(x) := s(e)]$).

Basic assignment $\{p\}\ x := e\ \{\exists y(p[x := y] \wedge (e[x := y] = x))\}$ where y is fresh
Look-up $\{p \wedge e \hookrightarrow -\}\ x := [e]\ \{\exists y(p[x := y] \wedge (e[x := y] \hookrightarrow x))\}$ where y is fresh
Mutation $\{p \wedge x \hookrightarrow -\}\ [x] := e\ \{\exists y(p[\langle x\rangle := y]) \wedge x \hookrightarrow e\}$ where y is fresh
Allocation $\{p\}\ x := \mathbf{new}(e)\ \{(\exists y(p[x := y]))[\langle x\rangle := \bot] \wedge x \hookrightarrow e\}$
where y is fresh and x does not occur in e
Dispose $\{p \wedge x \hookrightarrow -\}\ [x] := \bot\ \{\exists y(p[\langle x\rangle := y]) \wedge x \not\hookrightarrow -\}$ where y is fresh

Fig. 6. SP-CSL

Dispose. Let p' denote $(x \not\hookrightarrow -) \wedge \exists y(p[\langle x\rangle := y])$. By the frame rule, we first derive

$$\{(x \mapsto -) * p'\}\ [x] := \bot\ \{\mathbf{emp} * p'\}$$

See above (mutation) for the kind of argument that establishes that p implies $(x \mapsto -) * p'$. On the other hand, Let $h, s \models \mathbf{emp} * p'$, that is, $h, s \models p'$, and so by the above substitution lemma, we have $h[s(x) := n], s \models p$, for some n (again, y does not appear in p). Since $\{p\}\ [x] := \bot\ \{q\}$, we derive $h[s(x) := \bot], s \models q$, that is, $h, s \models q$, since $h, s \models x \not\hookrightarrow -$. □

To the best of our knowledge there are no global axioms in separation logic which provide a strongest postcondition axiomatization of all the basic instructions, including the mutation and dispose instruction. Figure 5 provides such an axiomatization for these instructions using the separating connectives.

Proposition 2. *For any mutation and dispose instruction S we have that $\{p\}\ S\ \{q\}$ is valid if and only if $\{p\}\ S\ \{q\}$ is derivable from the axioms in Figure 5 and (a single application of) the rule of consequence.*

Proof. Omitted (follows in a straightforward manner from the semantics of separation conjunction/implication). □

For the standard interpretation of separation logic Figure 6 provides an alternative strongest postcondition axiomatization of the basic instructions (denoted by SP-CSL), using instead the substitutions corresponding to heap update and heap clear.

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 (in Figure 6) with the following traditional axiom:

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

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 6:

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

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

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

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 4 (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 6) 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 above axiomatization of the other instructions follow in a straightforward manner from the corresponding substitution lemmas).

- $\models \{p\}\ x := \mathbf{new}(e)\ \{(\exists y(p[x := y]))[\langle x\rangle := \bot] \wedge x \hookrightarrow e\}$:
  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 of $[\langle x\rangle := \bot]$),
  $$h[n := s(e)][n := \bot], s[x := n][y := s(x)] \models p[x := y].$$
  By the substitution lemma and since $h[n := s(e)] = h[n := \bot] = h$, the latter is equivalent to $h, s \models p$ (also y does not appear in p), which holds by assumption.
- $\models \{p\}\ x := \mathbf{new}(e)\ \{q\}$ implies $\models (\exists y(p[x := y])[\langle x\rangle :=\bot] \wedge x \hookrightarrow e) \rightarrow q$:
  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
  $$\langle x := \mathbf{new}(e), h[s(x) :=\bot], s[x := n]\rangle \Rightarrow (h[s(x) := s[x := n](e)], s).$$
  Since we can assume without loss of generality that x does not occur in e (as argued in Subsection 3.1) 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 $\{p\}\ x := \mathbf{new}(e)\ \{q\}$ then we conclude that $h, s \models q$. □

5 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 correctness of the heap update and clear substitutions, the soundness/completeness of both the weakest precondition (WP-CSL) and the strongest postcondition axiomatizations (SP-CSL). For this formalization we used the dependently-typed calculus of inductive constructions as implemented by the Coq theorem prover. 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 shall make some more detailed comments, relevant for those interested in the technical details of the formalization. Here are some of the main characteristics of this formalization:

- `Heap.v`: Provides an axiomatization of heaps as partial functions.

- `Language.v`: Provides a shallow embedding of Boolean expressions and integer expressions, and a deep embedding of our assertion language, on which we inductively define the substitution operators (see Definition 2 and Definition 3). Further it provides a syntactic representation of the basic instructions of our programming language (assignment, look-up, mutation, allocation, and dispose) and the semantics of basic instructions (see Figure 1). Finally, it includes a syntactic representation of Hoare triples, and the proof systems WP-CSL (see Figure 3) and SP-CSL (see Figure 6).
- `Classical.v`: Provides the classical semantics of assertions (see Figure 2 and Definition 1), and the strong partial correctness semantics of Hoare triples. Further it provides proofs of the substitution lemmas corresponding to our substitution operators (see Lemma 1, Lemma 2 and Lemma 3). Finally, it provides proofs of the soundness and completeness of WP-CSL and SP-CSL (see Theorem 1 and Theorem 4).

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 $\{\forall y(y \hookrightarrow -)\}\ x := \mathbf{new}(e)\ \{\mathbf{false}\}$ 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 $h, s[x := n] \models (x \mapsto e) \mathrel{-\!\!*} \mathbf{false}$ is vacuously true, for every n, because there is no h' disjoint from h which satisfies $h', s[x := n] \models x \mapsto e$. With respect to total correctness in the presence of heaps with infinite domain we only need to require that the precondition of an allocation instruction implies $\exists y(y \not\hookrightarrow -)$. Thus, the presence of heaps with infinite domain allows for a standard first-order assertion language. In contrast, the validity problem of the first-order assertion language based on finite heaps is not recursively enumerable [CYO01], because finiteness itself cannot be expressed in first-order logic.

The assertion language and programming language is formalized using a mixed shallow/deep embedding. Boolean and integer expressions are expressed directly using a Coq term of the appropriate type (with a coincidence condition assumed, that the value of expressions depends 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 O'Connor [O'C11]. 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.

6 CONCLUSION AND RELATED WORK

In this paper we have shown that both a weakest precondition and a strongest postcondition axiomatization of the basic instructions of look-up, mutation, allocation, and dispose in separation logic does not require the introduction of separating conjunction/implication (or any other separating connectives). This generalizes to the axiomatization of the standard flow of control constructs (e.g., sequential composition, conditional and iteration statements), using the corresponding rules of Hoare logics. To the best of our knowledge this is the first systematic development of such axiomatizations of the basic instructions of separation logic. For example, [BHK18] investigates the transformation of specifications $\{p * r\}\ S\ \{q * r\}$, for any statement S, into a weakest precondition/strongest postcondition format, using, besides separating conjunction/implication, separating *co-implication* to generate a strongest postcondition format.

One of the main advantages of our axiomatization using substitutions is that it does not increase the complexity of the pre/postconditions measured by the maximal depth of nested connectives of separating conjunction/implication. To summarize the main difference between our axiomatization of the basic instructions 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 heap operations are used to describe the effect of the instructions, *abstracting* from the logical structure of the given pre/postcondition. In other words: a simple postcondition in first-order predicate logic yields a resulting simple weakest precondition with our substitutions (and vice versa, a simple precondition in first-order predicate logic yields a resulting simple strongest postcondition with our substitutions). Clearly this is desirable when dealing with the complexity of proving the correctness of actual programs. We plan to further develop our Coq mechanization to serve as the basis for a theorem prover for program correctness. This basically requires extending the formalization to the standard Hoare logics of sequential programs (including recursion, discussed below). For the basic instructions the resulting tool will thus automate the application of the substitutions in the generation of the verification conditions.

As discussed in the introduction, one of the main challenges in modular reasoning about heaps is in the specification and verification of invariant properties, since the usual adaptation rules of Hoare logics are not applicable, which are, for example, crucial in reasoning about recursive procedures. *Our overall conclusion is that the main strength of separation logic is in the use of the frame rule in modular reasoning about heaps, abstracting from implementation details.* Such reasoning is formalized by the concept of *modular completeness* which is introduced in [ZHL+96], and, roughly, requires to show that if $\models \{p\}\ S\ \{q\}$ implies $\models \{p'\}\ S\ \{q'\}$ then $\{p'\}\ S\ \{q'\}$ can be derived from $\{p\}\ S\ \{q\}$, abstracting from the implementation of S. In the thesis [Yan01] a proof of *modular* completeness of SL is given, using the frame rule and the expressive power of the separating connectives.

This beautiful result however contrasts sharply with the Gorelick-style completeness proof of a separation logic for recursive procedures which operate on heaps as given in [AT16]. The seminal completeness proof by Gorelick [Gor75] requires 'freezing' initial states by a finite set of fresh variables. This is not directly possible for heaps. Therefore the completeness proof in [AT16] is based on an (arithmetic) encoding of the semantics of the programming language and the assertion language to express a notion of strongest postcondition which involves a variable x_h which encodes the initial heap. This allows to apply the standard adaptation rules of Hoare logic (instead of the frame rule) to the standard most general correctness specification

$$\{\bar{x} = \bar{z}\}\ P\ \{SP(\bar{x} = \bar{z}, P)\}$$

to derive any valid specification $\{p\}\ P\ \{q\}$. Here $\bar{x} = \bar{z}$ abbreviates the assertion $\bigwedge_{i=1}^{n} x_i = z_i$, $x_1, \ldots, x_n$ being the program variables and $z_1, \ldots, z_n$ are fresh variables used to 'freeze' the initial

values of the program variables. This is achieved by applying the encoding of the assertion language to the assertion $p[\bar{x} = \bar{z}]$ to obtain a *pure* assertion $\epsilon(p[\bar{x} := \bar{z}])$, where for any assertion q we have that $h, s \models q$ if and only if $s[x_h := \epsilon(h)] \models \epsilon(q)$ ($\epsilon(h)$ denotes the encoding of the heap h). Since the invariance rule is sound for pure assertions which do not refer to the program variables and do not depend on the heap, one then derives

$$\{\bar{x} = \bar{z} \wedge \epsilon(p[\bar{x} := \bar{z}])\}\ P\ \{SP(\bar{x} = \bar{z}, P) \wedge \epsilon(p[\bar{x} := \bar{z}])\}$$

From this we then can derive by the consequence rule and the substitution rule the given valid specification $\{p\}\ P\ \{q\}$. Summarizing, in [AT16] separation logic is only used in the axiomatization of the basic instructions, where it is in fact not needed (and complicates reasoning), and not used where actually it is needed (and simplifies reasoning).

Further, following the approach [AT16] to the extreme, one could also specify and verify programs in a standard Hoare logic that supports arithmetic directly in terms of their encoding. The point of course is that instead in general one wants to reason about programs at the abstraction level of the programming language (and not for example in terms of the generated machine code, or in terms of some arithmetic encoding).

Instead of exposing the arithmetic encoding used in the logical expression of a notion of strongest postcondition, we can exploit the expressive power of the extended assertion language introduced in Section 2 to formalize logically the semantics of assertions in SL. This extension allows to capture a heap by the assertion $heap(r)$:

$$\forall y, z(r(y, z) \leftrightarrow y \hookrightarrow z)$$

It follows that $h, s \models heap(r)$ if and only if $s(r) = graph(h)$, where $graph(h)$ denotes the binary relation representing the graph of the partial unary function h. Let $init$ denote the assertion $heap(r) \wedge \bar{x} = \bar{z}$, where, as introduced above, $\bar{x} = \bar{z}$ 'freezes' the initial values of the program variables $\bar{x}$. We have the following *most general correctness specification*:

$$\{init\}\ P\ \{SP(init, P)\}$$

where $SP(init, P)$ denotes the strongest postcondition: $h, s \models SP(init, P)$ if and only if there exists a pair (h', s') such that $h', s' \models init$ and (h, s) can be reached as a final state from (h', s') by a computation of P. To derive any valid specification $\{p\}\ P\ \{q\}$ from the above most general correctness specification, using this standard definition of the strongest postcondition, we apply the invariance rule to derive

$$\{init \wedge p'(r)\}\ P\ \{SP(init, P) \wedge (p[\bar{x} := \bar{z}])(r)\}$$

For the definition of the pure assertion $(p[\bar{x} := \bar{z}])(r)$ see Definition 1. According to Proposition 1 this assertion states that $p[\bar{x} : 0\bar{z}]$ holds in the heap represented by r. Assuming $\models \{p\}\ P\ \{q\}$, one can show that $SP(init, P) \wedge (p[\bar{x} := \bar{z}])(r)$ implies q. So we obtain by the consequence rule and the generalized elimination rule

$$\{\exists r, \bar{z}(\bar{x} = \bar{z} \wedge (p[\bar{x} := \bar{z}])(r))\}\ P\ \{q\}$$

Since p implies the above precondition, an application of the consequence rule concludes the completeness proof.

We next show that in fact the expressive power of the separating connectives allows for a completeness proof in a *first-order* version of the assertion language extended with binary (arithmetic) relations (as used above). Let $\models \{p\}\ P\ \{q\}$ and $\{init\}\ P\ \{SP(init, P)\}$ be the most general correctness specification as defined above. By the frame rule we derive

$$\{init * p'\}\ P\ \{SP(init, P) * p'\}$$

where p'' denotes the assertion

$$\mathbf{emp} \wedge (\mathit{heap}(r) \mathrel{-\!\!*} (p[\bar{x} := \bar{z}])).$$

This assertion in fact expresses the second-order assertion $(p[\bar{x} := \bar{z}])(r)$ used in the above completeness proof: It follows from the semantics of the assertion language and Proposition 1 that $\emptyset, s \models p'$ if and only if $s \models (p[\bar{x} := \bar{z}])(r)$:
$\emptyset, s \models p'$
iff (semantics assertions)
$h, s \models \mathit{heap}(r)$ implies $h, s \models p[\bar{x} := \bar{z}]$, for every heap h
iff (definition $\mathit{heap}(r)$)
$s(r) = \mathit{graph}(h)$ implies $h, s \models p[\bar{x} := \bar{z}]$, for every heap h
iff (Proposition 1)
$s(r) = \mathit{graph}(h)$ implies $s[r := \mathit{graph}(h)] \models (p[\bar{x} := \bar{z}])(r)$, for every heap h
iff
$s \models (p[\bar{x} := \bar{z}])(r)$.
We first show that $(\mathit{SP}(\mathrm{init}, P) * p') \rightarrow q$: Let $h, s \models \mathit{SP}(\mathit{init}, P) * p'$. So $h, s \models \mathit{SP}(\mathit{init}, P)$ and $\emptyset, s \models \mathit{heap}(r) \mathrel{-\!\!*} (p[\bar{x} := \bar{z}])$. By definition of SP there exist h' and s' such that $h', s' \models \mathit{init}$ and (h, s) can be reached as a final state from (h', s') by a computation of P. From $\emptyset, s \models \mathit{heap}(r) \mathrel{-\!\!*} (p[\bar{x} := \bar{z}])$ we derive that $h', s \models p[\bar{x} := \bar{z}]$ (note that $h', s \models \mathit{heap}(r)$, because $h', s', \models \mathit{heap}(r)$ and $s'(r) = s(r)$, since r is not affected by the computation of P). So we have that $h', s' \models p$ (follows from the substitution lemma and $s'(x_i) = s'(z_i) = s(z_i)$, for $i = 1, \ldots, n$). Since by assumption $\models \{p\}\ P\ \{q\}$ we conclude that $h, s \models q$. So by the consequence rule, we derive

$$\{\mathit{init} * p'\}\ P\ \{q\}$$

It is not difficult to show that the assertion $\mathit{init} \wedge p$ implies $\mathit{init} * p'$. Another application of the consequence rule then gives

$$\{\mathit{init} \wedge p\}\ P\ \{q\}$$

We can remove the information about $\bar{z}$ in the precondition by substituting $\bar{z}$ for $\bar{x}$ (this transforms $\bar{x} = \bar{z}$ to **true**). We can remove the information about r (as expressed by $\mathit{heap}(r)$) by existentially quantifying r in the precondition. However, we can avoid this second-order quantification by instead substituting $\hookrightarrow$ for r (so that also $\mathit{heap}(r)$ reduces to **true**). It should be noted here that this substitution is restricted to assertions where r does not appear in the context of the separating connectives (because the semantics of $\hookrightarrow$ is context-sensitive).

The above completeness proof shows the expressive power of the separating connectives in reasoning about invariant properties of heaps in the first-order assertion language extended with variables ranging over binary (arithmetic) relations. As far as we know it remains an open problem whether there exists a completeness proof for a separation logic for recursive procedures using the frame rule instead of the invariance rule and based on the first-order assertion language with only the binary relation $\hookrightarrow$. On the other hand, this is a rather theoretical problem because in a first-order language one cannot express general heap properties like reachability, for that one needs second-order quantification of binary relations or recursive predicates (though, it is unclear how to 'freeze' the initial heap using recursive predicates).

We conclude with a brief discussion of intuitionistic separation logic (ISL). Intuitionistic separation logic (ISL) is the original version of separation logic [Rey00, IO01] where the basic assertions $e \mapsto e'$ and **emp** are omitted, and implication is redefined by $h, s \models p \rightarrow q$ 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 $n \in \mathit{dom}(h)$. Consequently, $(x \not\hookrightarrow -)$, which is is an abbreviation for $(x \hookrightarrow -) \rightarrow \mathbf{false}$, is intuitionistically equivalent to **false**. That is, we cannot express directly that a location x is not allocated anymore. Therefore we

need to redefine the substitutions $p[\langle x\rangle := e]$ and $p[\langle x\rangle := \bot]$. For the heap update substitution we only need to redefine the clauses for separation conjunction and implication as follows:

- $(p * q)[\langle x\rangle := e] = (x \hookrightarrow -) \rightarrow ((p[\langle x\rangle := e] \wedge x \hookrightarrow -) * q) \vee (p * (q[\langle x\rangle := e] \wedge x \hookrightarrow -))$
- $(p \mathrel{-\!\!*} q)[\langle x\rangle := e] = (x \hookrightarrow -) \rightarrow (p \mathrel{-\!\!*} (q[\langle x\rangle := e])$

For the heap clear substitution we only need to redefine the clause for implication along the lines of the definition of $(p \mathrel{-\!\!*} q)[\langle x\rangle := \bot]$:

- $(p \rightarrow q)[\langle x\rangle := \bot] = (p[\langle x\rangle := \bot] \rightarrow q[\langle x\rangle := \bot]]) \wedge \forall y(p[\langle x\rangle := y] \rightarrow q[\langle x\rangle := y])$ where y is a fresh variable.

The proofs of the correctness of these substitutions require the monotonicity property: for any assertion p (of ISL) $h, s \models p$ implies $h', s \models p$, where $h \sqsubseteq h'$. For example, in order to show that $h[s(x) := s(e)], s \models p * q$ implies

$$h, s \models (x \hookrightarrow -) \rightarrow ((p[\langle x\rangle := e] \wedge x \hookrightarrow -) * q) \vee (p * (q[\langle x\rangle := e] \wedge x \hookrightarrow -)),$$

let $h[s(x) := s(e)] = h_1 \uplus h_2$ such that $h_1, s \models p$ and $h_2, s \models q$. Without loss of generality, we assume that $s(x) \in dom(h_1)$ (so that $h_1(s(x)) = s(e)$). Further , let h' be such that $s(x) \in dom(h')$ and $h \sqsubseteq h'$. We show that $h', s \models p[\langle x\rangle := e] * q$. Let h'_1 be such that $dom(h'_1) = dom(h') \setminus dom(h_2)$ and $h'_1(n) = h'(n)$, for $n \in dom(h'_1)$. Note that thus $h'_1(s(x)) = s(e)$. It follows that $h' = h'_1 \uplus h_2$ and $h_1 \sqsubseteq h'_1$. It thus suffices to show that $h'_1, s \models p[\langle x\rangle := e]$. Since $h_1, s \models p$, we derive $h'_1, s \models p$ by monotonicity. Further, by the induction hypothesis, we have $h'_1, s \models p[\langle x\rangle := e]$ if and only if $h'_1[s(x) := s(e)], s \models p$, that is, $h'_1, s \models p$ (note that, as above, $h'_1[s(x) := s(e)] = h'_1$).

Given these new definitions of the heap update and the heap clear substitution the axioms for the basic assignment, look-up, mutation, and dispose instruction also can be shown to be sound and complete for ISL. We only need to redefine the allocation axiom as follows:

$$\{\forall x((x \hookrightarrow -) \vee p[\langle x\rangle := e])\}\ x := \mathbf{new}(e)\ \{p\}$$

This works because of the decidability of the left disjunct: in ISL $h, s \models \forall x((x \hookrightarrow -) \vee p[\langle x\rangle := e])$ if and only if $n \notin dom(h)$ implies $h, s[x := n] \models p[\langle x\rangle := e]$, for all n. As such, freshness of x is described logically in exactly the same manner as in the precondition of the allocation axiom for CSL (Figure 3).

Further, given these new definitions of substitution for heap update and heap clear, the strongest postcondition axiomatization for CSL is also sound and complete for ISL.

REFERENCES

[AdBO09a] Krzysztof R. Apt, Frank de Boer, and Ernst-Rdiger Olderog. *Verification of Sequential and Concurrent Programs.* Springer Publishing Company, Incorporated, 3rd edition, 2009.

[AdBO09b] Krzysztof R. Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs.* Texts in Computer Science. Springer, 2009.

[AL03] Martín Abadi and K. Rustan M. Leino. A logic of object-oriented programs. In Nachum Dershowitz, editor, *Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, volume 2772 of *Lecture Notes in Computer Science*, pages 11–41. Springer, 2003.

[AT16] Mahmudul Faisal Al Ameen and Makoto Tatsuta. Completeness for recursive procedures in separation logic. *Theor. Comput. Sci.*, 631:73–96, 2016.

[BHK18] Callum Bannister, Peter Höfner, and Gerwin Klein. Backwards and forwards with separation logic. In Jeremy Avigad and Assia 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.

[BK14] James Brotherston and Max I. Kanovich. Undecidability of propositional separation logic and its neighbours. *J. ACM*, 61(2):14, 2014.

[CYO01] Cristiano Calcagno, Hongseok Yang, and Peter W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In Ramesh Hariharan, V. Vinay, and Madhavan Mukund, editors, *FSTTCS*

2001: Foundations of Software Technology and Theoretical Computer Science, volume 2245 of *Lecture Notes in Computer Science*, pages 108–119. Springer Berlin Heidelberg, 2001.

[dBB21] Frank S. de Boer and Marcello M. Bonsangue. Symbolic execution formally explained. *Formal Aspects Comput.*, 33(4-5):617–636, 2021.

[Gor75] G. A. Gorelick. A complete axiomatic system for proving assertions about recursive and non-recursive programs. Master's thesis, University of Toronto, 1975. Available at https://archive.org/details/ACompleteAxiomaticSystemForProvingAssertionsAboutRecursiveAnd.

[Gri81] David Gries. *The Science of Programming*. Texts and Monographs in Computer Science. Springer, 1981.

[Har79] David Harel. *First-Order Dynamic Logic*, volume 68 of *Lecture Notes in Computer Science*. Springer, 1979.

[IO01] Samin Ishtiaq and Peter W. O'Hearn. Bi as an assertion language for mutable data structures. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 14–26, 2001.

[O'C11] Russell O'Connor. Classical mathematics for a constructive world. *Mathematical Structures in Computer Science*, 21(4):861–882, 2011.

[Rey00] John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In J. Davies, B. Roscoe, and J. Woodcock, editors, *Millennial Perspectives in Computer Science*, Cornerstones of Computing, pages 303–321. Macmillan Education UK, 2000.

[Rey02] John 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*, pages 55–74. IEEE Computer Society, 2002.

[Yan01] Hongseok Yang. *Local Reasoning for Stateful Programs*. PhD thesis, USA, 2001.

[ZHL+96] Job Zwiers, Ulrich Hannemann, Yassine Lakhnech, Willem P. de Roever, and Frank A. Stomp. Modular completeness: Integrating the reuse of specified software in top-down program development. volume 1051 of *Lecture Notes in Computer Science*, pages 595–608. Springer, 1996.