Extending Separation Logic with Fixpoints and Postponed Substitution*

Élodie-Jane Sims

École Polytechnique **, 91128 Palaiseau, France
Elodie-Jane.Sims@polytechnique.fr

Abstract. We are interested in separation-logic-based static analysis of programs that use shared mutable data structures. In this paper, we introduce backward and forward analysis for a separation logic called $BI^{\mu\nu}$, an extension of separation logic [Ishtiaq & O'Hearn, POPL'01], to which we add fixpoint connectives and postponed substitution. This allows us to express recursive definitions within the logic as well as the axiomatic semantics of `while` statements.

Keywords: separation logic, fixpoint, wlp, sp, abstract interpretation

1 Introduction

In this paper we address the problem of doing static analysis of programs [2] which use shared mutable data structures. The final goal of our work is to detect errors in a program (problems of dereferencing, aliasing, etc.) or to prove that a program is correct (with respect to these problems) in an automatic way. John Reynolds, Peter O'Hearn and others have developed [7, 10] an extension of Hoare logic called separation logic (also known as BI logic) that permits reasoning about such programs. The classical definition of predicates on abstract data structures is extended by introducing a "separating conjunction", denoted $*$, which asserts that its sub-formulae hold for disjoint parts of the heap, and a closely related "separating implication", denoted $-\!\!*$. This extension permits the concise and flexible description of structures with controlled sharing.

We extend this logic with fixpoint connectives to define recursive properties and to express the axiomatic semantics of a `while` statement. We present forward and backward analyses (*sp (strongest postcondition)*, *wlp (weakest liberal precondition)* expressed for all statements and all formulae).

Organization of the paper The structure of the paper goes as follow: In Sect. 2 we describe the command language we analyze and in Sect. 3 we present our

* extended version of [12].

** CNRS Doctoral Thesis Fellowship. Doctoral Thesis in collaboration with Kansas State University.

logic $BI^{\mu\nu}$. In Sect. 4, we provide a backward analysis with $BI^{\mu\nu}$ in terms of "weakest liberal preconditions". We express the *wlp* for the composition, `if`$-$`then`$-$`else` and `while` commands. In Sect. 5, we provide a forward analysis with $BI^{\mu\nu}$ in terms of "strongest postconditions". In Sect. 6, we discuss another possibility for adding fixpoints to separation logic.

Background Hoare logic [6] and Dijkstra-style weakest-precondition logics [5] are well known. It is also well known that these logics disallow *aliasing*, that is, the logics require that each program variable names a distinct storage location. Therefore, it is difficult to reason about programs that manipulate pointers or heap storage.

Through a series of papers [**?**,7, 10], Reynolds and O'Hearn have addressed this foundationally difficult issue. Their key insight is that a command executes within a *region* of heap storage: they write

$$s, h \models \phi$$

to denote that property ϕ holds true within heap subregion h and local-variable stack s. One could also say that a formula describes some properties of the memories it represents. For example, ϕ might be:

`emp` means that the heap is empty

$E \mapsto a, b$ means that there is exactly one cell in the heap, the one containing the values of a and b and that E points to it.

$E \hookrightarrow a, b$ is the same except that the heap can contain additional cells

With the assistance of a new connective, the "separating conjunction", denoted $*$, Reynolds and O'Hearn write

$$s, h_1 \cdot h_2 \models \phi_1 * \phi_2$$

to assert that both ϕ_1 and ϕ_2 hold but use *disjoint* heap subregions to justify their truth — there is no aliasing between the variables mentioned in ϕ_1 and ϕ_2. For example, consider the two cases below.

If $s = \begin{bmatrix} x \to l_1 \\ y \to l_2 \end{bmatrix}$, $h = \begin{bmatrix} l_1 \to \langle 3, 4\rangle \\ l_2 \to \langle 1, 2\rangle \end{bmatrix}$ then $s, h \models (x \mapsto 3, 4) * (y \mapsto 1, 2)$ or also $s, h \models (x \hookrightarrow 3, 4)$ but $s, h \not\models (x \mapsto 3, 4)$.

If $s = \begin{bmatrix} x \to l_1 \\ y \to l_1 \end{bmatrix}$, $h = [l_1 \to \langle 3, 4\rangle]$ then $s, h \models (x \mapsto 3, 4) \wedge (y \mapsto 3, 4)$ but $s, h \not\models (x \mapsto 3, 4) * (y \mapsto 3, 4)$.

Adjoint to the separating conjunction is a "separating implication":

$$s, h \models \phi_1 \mathrel{-\!\!*} \phi_2$$

asserts, "if heap region h is augmented by h' such that $s, h' \models \phi_1$, then $s, h \cdot h' \models \phi_2$". For example: if $s = \begin{bmatrix} x \to l_1 \\ y \to l_2 \end{bmatrix}$, $h = [l_1 \to \langle 3, 4\rangle]$ then $s, h \models (y \mapsto 1, 2) \mathrel{-\!\!*} ((x \mapsto 3, 4) * (y \mapsto 1, 2))$.

Istiaq and O'Hearn [7] showed how to add the separating connectives to a classical logic, producing a *bunched implication logic* (*BI* or *separation logic*) in which Hoare-logic-style reasoning can be conducted on while-programs that manipulate temporary-variable stacks and heaps.

A Hoare triple, $\{\phi_1\}C\{\phi_2\}$, uses assertions ϕ_i, written in separation logic; the semantics of the triple is stated with respect to a stack-heap storage model.

Finally, there is an additional inference rule, the *frame rule*, which formalizes compositional reasoning based on disjoint heap regions:

$$\frac{\{\phi_1\}C\{\phi_2\}}{\{\phi_1 * \phi'\}C\{\phi_2 * \phi'\}}$$

where ϕ''s variables are not modified by C.

The reader interested in the *set-of-inference-rules* approach for separation logic is invited to read [7], and [17] for details on the frame rule. The rules could also be found in a survey on separation logics [10]. We do not present the set of rules in this paper.

Our contribution Istiaq and O'Hearn's efforts were impressive but incomplete: the weakest-precondition and strongest postcondition semantics for their while-language were absent, because these require *recursively defined* assertions, which were not in their reach.

The primary accomplishment of this paper is to add least- and greatest-fixed-point operators to separation logic, so that pre- and post-condition semantics for the while-language can be wholly expressed within the logic. As a pleasant consequence, it becomes possible to formalize recursively defined properties on inductively (and co-inductively) defined data structures, e.g.,

$$\begin{array}{l}\texttt{nonCircularList}(x) = \\ \quad \mu X_v.\ (x = \texttt{nil}) \vee \exists x_1, x_2.(\texttt{isval}(x_1) \wedge (x \mapsto x_1, x_2\ *\ X_v[x_2/x]))\end{array}$$

asserts that x is a linear, non-circular list ($\texttt{isval}(x_1)$ insures that x_1 is a value, this predicate is defined later).

The addition of the recursion operators comes with a price: the usual definition of syntactic substitution and the classic substitution laws become more complex; the reasons are related both to the semantics of stack- and heap-storage as well as the inclusion of the recursion operators; details are given later in the paper.

2 Commands and basic domains

We consider a simple "while"-language with Lisp-like expressions for accessing and creating `cons` cells.

2.1 Command syntax

The commands we consider are as follows.

$$
\begin{aligned}
C &::= x := E \mid x := E.i \mid E.i := E' \mid x := \texttt{cons}(E_1, E_2) \mid \texttt{dispose}(E) \\
&\quad \mid C_1; C_2 \mid \texttt{if } E \texttt{ then } C_1 \texttt{ else } C_2 \mid \texttt{skip} \mid \texttt{while } E \texttt{ do } C_1 \\
i &::= 1 \mid 2 \\
E &::= x \mid n \mid \texttt{nil} \mid \mathit{True} \mid \mathit{False} \mid E_1 \ \mathit{op} \ E_2
\end{aligned}
$$

An expression can denote an integer, an atom, or a heap-location. Here x is a variable in *Var*, n an integer and *op* is an operator in $(\mathit{Val} \times \mathit{Val}) \to \mathit{Val}$ such as $+ : (\mathit{Int} \times \mathit{Int}) \to \mathit{Int}$, $\vee : (\mathit{Bool} \times \mathit{Bool}) \to \mathit{Bool}$ (for *Var* and *Val*, see Sect. 2.2).

The second and third assignment statements read and update the heap, respectively. The fourth creates a new cons cell in the heap, and places a pointer to it in x.

Notice that in our language we do not handle two dereferencings in a simple statement (no $x.i.j$, no $x.i := y.j$); this restriction is for simplicity and does not limit the expressivity of the language, requiring merely the addition of intermediate variables.

2.2 Semantic domains

$$
\begin{aligned}
\mathit{Val} &= \mathit{Int} \cup \mathit{Bool} \cup \mathit{Atoms} \cup \mathit{Loc} \\
S &= \mathit{Var} \rightharpoonup \mathit{Val} \\
H &= \mathit{Loc} \rightharpoonup \mathit{Val} \times \mathit{Val}
\end{aligned}
$$

Here, $\mathit{Loc} = \{l_1, l_2, ...\}$ is an infinite set of locations, $\mathit{Var} = \{x, y, ...\}$ is an infinite set of variables, $\mathit{Atoms} = \{\texttt{nil}, a, ...\}$ is a set of atoms, and $\rightharpoonup$ is for partial functions. We call an element $s \in S$ a stack, and $h \in H$ a heap. We also call the pair $(s, h) \in S \times H$ a *state*.

We use $\mathit{dom}(h)$ to denote the domain of definition of a heap $h \in H$, and $\mathit{dom}(s)$ to denote the domain of a stack $s \in S$. Notice that we allow $\mathit{dom}(h)$ to be infinite.

An expression is interpreted as a heap-independent value: $[\![E]\!]^s \in \mathit{Val}$. For example, $[\![x]\!]^s = s(x)$, $[\![n]\!]^s = n$, $[\![\texttt{true}]\!]^s = \mathit{true}$, $[\![E_1 + E_2]\!]^s = [\![E_1]\!]^s + [\![E_2]\!]^s$.

Since domain S allows partial functions, $[\![\ ]\!]^s$ is also partial. Thus $[\![E_1 = E_2]\!]^s$ means $[\![E_1]\!]^s$ and $[\![E_2]\!]^s$ are defined and equal. From here on, when we write a formula of the form $\cdots [\![E]\!]^s \cdots$, we are also asserting that $[\![E]\!]^s$ is defined.

2.3 Small-step semantics

The semantics of statements, C, are given small-step semantics defined by the relation $\rightsquigarrow$ on configurations. The configurations include triples C, s, h and terminal configurations s, h for $s \in S$ and $h \in H$. The rules are given in Fig. 1.

In the rules, we use r for elements of $\mathit{Val} \times \mathit{Val}$; $\pi_i r$ with $i \in \{1, 2\}$ for the first or second projection; $(r|i \to v)$ for the pair like r except that the i'th component is replaced with v; and $[s \mid x \to v]$ for the stack like s except that it maps x to v, $(h - l)$ for $h_{\restriction_{\mathit{dom}(h) \setminus \{l\}}}$.

The location l in the `cons` case is not specified uniquely, so a new location is chosen non-deterministically.
Let the set of error configurations be: $\Omega = \{C, s, h \mid \nexists K.\ C, s, h \rightsquigarrow K\}$.
We say that:

- "C, s, h is safe" if and only if $\forall K.\ (C, s, h \rightsquigarrow^* K \ \Rightarrow\ K \notin \Omega)$
- "C, s, h is stuck" if and only if $C, s, h \in \Omega$

For instance, an error state can be reached by an attempt to dereference `nil` or an integer. Note also that the semantics allows dangling references, as in stack $[x \to l]$ with empty heap $[]$.

The definition of safety is formulated with partial correctness in mind: with loops, C, s, h could fail to converge to a terminal configuration but not get stuck.

We define the weakest liberal precondition in the operational domain:

Definition 1. *For* $\Delta \subseteq S \times H$, $wlp_o(\Delta, C) = \{s, h \mid (C, s, h \rightsquigarrow^* s', h' \Rightarrow s', h' \in \Delta) \wedge C, s, h \text{ is safe}\}$

We define the strongest postcondition similarly:

Definition 2. $sp_o(\Delta, C) = \{s', h' \mid \exists s, h \in \Delta.\ C, s, h \rightsquigarrow^* s', h'\}$

$$\frac{[\![E]\!]^s = v}{x := E, s, h \rightsquigarrow [s|x \to v], h} \qquad \frac{[\![E]\!]^s = l \quad h(l) = r}{x := E.i, s, h \rightsquigarrow [s|x \to \pi_i r], h}$$

$$\frac{[\![E]\!]^s = l \quad h(l) = r \quad [\![E']\!]^s = v'}{E.i = E', s, h \rightsquigarrow s, [h|l \to (r|i \to v')]} \qquad \frac{l \in dom(h) \quad [\![E]\!]^s = l}{\texttt{dispose}(E), s, h \rightsquigarrow s, (h - l)}$$

$$\frac{l \in Loc \quad l \notin dom(h) \quad [\![E_1]\!]^s = v_1, [\![E_2]\!]^s = v_2}{x := \texttt{cons}(E_1, E_2), s, h \rightsquigarrow [s|x \to l], [h|l \to \langle v_1, v_2 \rangle]}$$

$$\frac{C_1, s, h \rightsquigarrow C', s', h'}{C_1; C_2, s, h \rightsquigarrow C'; C_2, s', h'} \qquad \frac{C_1, s, h \rightsquigarrow s', h'}{C_1; C_2, s, h \rightsquigarrow C_2, s', h'} \qquad \frac{}{\texttt{skip}, s, h \rightsquigarrow s, h}$$

$$\frac{[\![E]\!]^s = True}{\texttt{if } E \texttt{ then } C_1 \texttt{ else } C_2, s, h \rightsquigarrow C_1, s, h} \qquad \frac{[\![E]\!]^s = False}{\texttt{if } E \texttt{ then } C_1 \texttt{ else } C_2, s, h \rightsquigarrow C_2, s, h}$$

$$\frac{[\![E]\!]^s = False}{\texttt{while } E \texttt{ do } C, s, h \rightsquigarrow s, h} \qquad \frac{[\![E]\!]^s = True}{\texttt{while } E \texttt{ do } C, s, h \rightsquigarrow C; \texttt{while } E \texttt{ do } C, s, h}$$

Fig. 1. Operational small-step semantics of the commands

3 $BI^{\mu\nu}$

In this section, we present the logic $BI^{\mu\nu}$. It is designed to describe properties of the state. Typically, for analysis it will be used in Hoare triples of the form $\{P\}C\{Q\}$ with P and Q formulae of the logic and C a command.

We present in Sect. 3.1 the syntax of the logic and in Sect. 3.2 its formal semantics. In Sect. 3.3, we give the definition of a *true triple* $\{P\}C\{Q\}$. In Sect. 3.4, we discuss the additions to separation logic (fixpoints and postponed substitution).

3.1 Syntax of formulae

$P,Q,R ::= E = E'$	Equality	$\mid E \mapsto E_1, E_2$	Points to
$\mid$ `false`	Falsity	$\mid P \Rightarrow Q$	Classical Imp.
$\mid \exists x.P$	Existential Quant.	$\mid$ `emp`	Empty Heap
$\mid P * Q$	Spatial Conj.	$\mid P \mathrel{-\!\!*} Q$	Spatial Imp.
$\mid X_v$	Formula Variable	$\mid P\,\wr E/x\wr$	Postponed Substitution
$\mid \nu X_v.P$	Greatest Fixpoint	$\mid \mu X_v.P$	Least Fixpoint

Fig. 2. Syntax of formulae

We have an infinite set of variables, Var_v, used for the variables bound by μ and ν and disjoint from the set Var. They range over sets of states, the others (x,y,...) are variables which range over values. For emphasis, uppercase variables subscripted by $_v$ are used to define recursive formulae. We use the term "closed" for the usual notion of closure of variables in Var (closed by $\exists$ or $\forall$) and the term "v-closed" for closure of variables in Var_v (v-closed by μ or ν).

Our additions to Reynolds and O'Hearn's separation logic are the fixpoint operators $\mu X_v.\ P$ and $\nu X_v.\ P$ and the substitution construction $P\,\wr E/x\wr$.

We can define various other connectives as usual, rather than taking them as primitives:

$$\neg P \triangleq P \Rightarrow \texttt{false} \qquad \texttt{true} \triangleq \neg(\texttt{false})$$
$$P \vee Q \triangleq (\neg P) \Rightarrow Q \qquad P \wedge Q \triangleq \neg(\neg P \vee \neg Q)$$
$$\forall x.P \triangleq \neg(\exists x.\neg P) \qquad E \hookrightarrow a, b \triangleq \texttt{true} * (E \mapsto a, b)$$
$$x = E.i \triangleq \exists x_1, x_2.\ (E \hookrightarrow x_1, x_2) \wedge (x = x_i)$$

We could have only one fixpoint connective in the syntax, since the usual equivalences, $\mu X_v.\ P \equiv \neg\nu X_v.\neg(P\{\neg X_v/X_v\})$ and $\nu X_v.\ P \equiv \neg\mu X_v.\neg(P\{\neg X_v/X_v\})$, hold.

The set $FV(P)$ of free variables of a formula is defined as usual. The set $Var(P)$ of variables of a formula is defined as usual with $Var(P\ \wr E/x\wr) = Var(P) \cup Var(E) \cup \{x\}$.

3.2 Semantics of formulae

The semantics of the logic is given in Fig. 3.

We use the following notation in formulating the semantics:

- $h \sharp h'$ indicates that the domains of heaps h and h' are disjoint;
- $h \cdot h'$ denotes the union of disjoint heaps (i.e., the union of functions with disjoint domains).

We express the semantics of the formulae in an environment ρ mapping formula variables to set of states: $\rho : Var_v \rightharpoonup \mathcal{P}(S \times H)$. The semantics of a formula in an environment ρ is the set of states which satisfy it, and is expressed by: $[\![\cdot]\!]_\rho : BI^{\mu\nu} \rightharpoonup \mathcal{P}(S \times H)$

We call $[\![P]\!]$ the semantics of a formula P in an empty environment $[\![P]\!] = [\![P]\!]_\emptyset$. We also define a forcing relation of the form:

$$s, h \models P \text{ if and only if } s, h \in [\![P]\!]$$

and an equivalence:
$P \equiv Q$ if and only if $\forall \rho.([\![P]\!]_\rho = [\![Q]\!]_\rho)$ or ($[\![P]\!]_\rho$ and $[\![Q]\!]_\rho$ both do not exist).

$$\begin{array}{ll}
[\![E = E']\!]_\rho & = \{s, h \mid [\![E]\!]^s = [\![E']\!]^s\} \\
[\![E \mapsto E_1, E_2]\!]_\rho & = \{s, h \mid dom(h) = \{[\![E]\!]^s\} \\
 & \quad \text{and } h([\![E]\!]^s) = \langle [\![E_1]\!]^s, [\![E_2]\!]^s \rangle\} \\
[\![\texttt{false}]\!]_\rho & = \emptyset \\
[\![P \Rightarrow Q]\!]_\rho & = ((S \times H) \setminus [\![P]\!]_\rho) \cup [\![Q]\!]_\rho \\
[\![\exists x.P]\!]_\rho & = \{s, h \mid \exists v \in Val.[s|x \to v], h \in [\![P]\!]_\rho\} \\
[\![\texttt{emp}]\!]_\rho & = \{s, h \mid h = []\} \\
[\![P * Q]\!]_\rho & = \{s, h \mid \exists h_0, h_1.\ h_0 \sharp h_1,\ h = h_0 \cdot h_1 \\
 & \quad s, h_0 \in [\![P]\!]_\rho \text{ and } s, h_1 \in [\![Q]\!]_\rho\} \\
[\![P -\!\!* Q]\!]_\rho & = \{s, h \mid \forall h'. \text{ if } h \sharp h' \text{ and } s, h' \in [\![P]\!]_\rho \text{ then} \\
 & \quad s, h \cdot h' \in [\![Q]\!]_\rho\} \\
[\![X_v]\!]_\rho & = \rho(X_v) \text{ , if } X_v \in dom(\rho) \\
[\![\mu X_v\ .\ P]\!]_\rho & = \mathrm{lfp}^{\subseteq}_\emptyset \lambda X.\ [\![P]\!]_{[\rho|X_v \to X]} \\
[\![\nu X_v\ .\ P]\!]_\rho & = \mathrm{gfp}^{\subseteq}_\emptyset \lambda X.\ [\![P]\!]_{[\rho|X_v \to X]} \\
[\![P\ \wr E/x \wr]\!]_\rho & = \{s, h \mid [s \mid x \to [\![E]\!]^s], h \in [\![P]\!]_\rho\}
\end{array}$$

Fig. 3. Semantics of $BI^{\mu\nu}$

In both cases μ and ν, the X in λX is a fresh variable over sets of elements in $S \times H$ which does not already occur in ρ.

Notice that $[\![\cdot]\!]_\rho$ is only a partial function. In definitions above, $\mathrm{lfp}^{\subseteq}_\emptyset \phi$ *(*$\mathrm{gfp}^{\subseteq}_\emptyset \phi$*)* is the least fixpoint *(greatest fixpoint)* of ϕ on the poset $\langle \mathcal{P}(S \times H), \subseteq \rangle$, if it exists. Otherwise, $[\![\mu X_v.P]\!]_\rho$ *(*$[\![\nu X_v.P]\!]_\rho$*)* is not defined. For example, this is the case for $\mu X_v\ .\ (X_v \Rightarrow \texttt{false})$.

The syntactical criterions for formulae with defined semantics (like parity of negation under a fixpoint, etc.) are the usual ones knowing that in terms of monotonicity, $-\!\!*$ acts like $\Rightarrow$, $*$ acts like $\wedge$, and $\wr$ / $\wr$ does not interfere. The fixpoint theory gives us criteria (using Tarski's fixpoint theorem) for the existence of $[\![P]\!]_\rho$, but no criteria for nonexistence. Nonetheless, we have these facts:

- if P is $E = E$ or $E \mapsto E_1, E_2$ or `false` or `emp`, then $[\![P]\!]_\rho$ always exists; $\lambda X.[\![P]\!]_{[\rho|X_v \to X]}$ is monotonic and antitonic.
- $[\![X_v]\!]_\rho$ exists if and only if $X_v \in dom(\rho)$; $\lambda X.[\![X_v]\!]_{[\rho|Y_v \to X]}$ is monotonic and not antitonic.
- If P is $Q \Rightarrow R$ or $Q -\!\!* R$, then $[\![P]\!]_\rho$ exists if and only if $[\![Q]\!]_\rho$ and $[\![R]\!]_\rho$ exist; $\lambda X.[\![P]\!]_{[\rho|X_v \to X]}$ is monotonic if and only if $\lambda X.[\![R]\!]_{[\rho|X_v \to X]}$ is monotonic and $\lambda X.[\![Q]\!]_{[\rho|X_v \to X]}$ is antitonic; $\lambda X.[\![P]\!]_{[\rho|X_v \to X]}$ is antitonic if and only if $\lambda X.[\![R]\!]_{[\rho|X_v \to X]}$ is antitonic and $\lambda X.[\![Q]\!]_{[\rho|X_v \to X]}$ is monotonic.

– $[\![Q * R]\!]_\rho$ exists if and only if $[\![Q]\!]_\rho$ and $[\![R]\!]_\rho$ exist; $\lambda X.[\![Q * R]\!]_{[\rho|X_v \to X]}$ is monotonic/antitonic if and only if $\lambda X.[\![R]\!]_{[\rho|X_v \to X]}$ and $\lambda X.[\![Q]\!]_{[\rho|X_v \to X]}$ are monotonic/antitonic.
– If P is $\exists x.\ Q$ or $Q\ \wr E/x \wr$, then $[\![P]\!]_\rho$ exists if and only if $[\![Q]\!]_\rho$ exists; $\lambda X.[\![P]\!]_{[\rho|X_v \to X]}$ is monotonic/antitonic if and only if $\lambda X.[\![Q]\!]_{[\rho|X_v \to X]}$ is monotonic/antitonic.
– If $\mu\nu \in \{\mu, \nu\}$ and if $\lambda X.[\![P]\!]_{[\rho|X_v \to X]}$ exists and is monotonic, then $[\![\mu\nu X_v.\ P]\!]_\rho$ exists and $\lambda X.[\![\mu\nu X_v.\ P]\!]_{[\rho|X_v \to X]}$ is monotonic and antitonic.
– If $\mu\nu \in \{\mu, \nu\}$ and if $\lambda X.[\![P]\!]_{[\rho|X_v \to X|Y_v \to Y]}$ is monotonic/antitonic, and $\lambda Y.[\![P]\!]_{[\rho|X_v \to X|Y_v \to Y]}$ exists and is monotonic, then $\lambda X.[\![\mu\nu Y_v.\ P]\!]_{[\rho|X_v \to X]}$ is monotonic/antitonic.

3.3 Interpretation of Triples

Hoare triples are of the form $\{P\}C\{Q\}$, where P and Q are assertions in $BI^{\mu\nu}$ and C is a command. The interpretation ensures that well-specified commands do not get stuck. (In this, it differs from the usual interpretation of Hoare triples [4].)

Definition 3. *$\{P\}C\{Q\}$ is a* true triple *if and only if $\forall s, h$, if $s, h \models P$ and $FV(Q) \subseteq dom(s)$, then*
– *C, s, h is safe*
– *if $C, s, h \rightsquigarrow^* s', h'$, then $s', h' \models Q$.*

This is a partial correctness interpretation; with looping, it does not guarantee termination. This is the reason for expressing "weakest liberal preconditions" for our backward analysis and not "weakest preconditions". However, the safety requirement rules out certain runtime errors and, as a result, we do not have that $\{\texttt{true}\}C\{\texttt{true}\}$ holds for all commands. For example, $\{\texttt{true}\}x := \texttt{nil}; x.1 := 3\{\texttt{true}\}$ is not a true triple.

3.4 Fixpoints and postponed substitution

In this section, we discuss our motivations for adding fixpoints and postponed substitution to separation logic. We show that the postponed substitution connective, $\wr\ /\ \wr$, is not classical substitution, $\{\ /\ \}$, and that the usual variable renaming theorem does not hold for $\{\ /\ \}$. We develop the needed concepts in a series of vignettes:

First motivation Our initial motivation for adding fixpoint operators to separation logic came from the habit of the separation logic community of informally defining recursive formulae and using them in proofs of correctness.

Since we have added fixpoint operators to the logic, we can formally and correctly express, for example, that x is a non-cyclic finite linear list as

$$\texttt{nclist}(x) = \mu X_v.\ (x = \texttt{nil}) \vee \exists x_1, x_2.(\texttt{isval}(x_1) \wedge (x \mapsto x_1, x_2\ *\ X_v\ \wr x_2/x \wr))$$

and that x is non-cyclic finite or infinite list

$$\texttt{nclist}(x) = \nu X_v.\ (x = \texttt{nil}) \vee \exists x_1, x_2.(\texttt{isval}(x_1) \wedge (x \mapsto x_1, x_2\ *\ X_v\ \wr x_2/x\wr))$$

where $\texttt{isval}(x) = (x = \texttt{true}) \vee (x = \texttt{false}) \vee (\exists n.n = x + 1)$

In earlier papers [18], Reynolds and O'Hearn use the definition,

$$\texttt{nclist}(x) = (x = \texttt{nil}) \vee \exists x_1, x_2.(\texttt{isval}(x_1) \wedge (x \mapsto x_1, x_2\ *\ \texttt{nclist}(x_2)))$$

which is not within the syntax of separation logic.

Second motivation The second motivation was the formulations of the *wlp* ($\{\ ?\ \}C\{P\}$) and *sp* ($\{P\}C\{\ ?\ \}$) in the case of `while` commands, which was not possible earlier. This problem is nontrivial: For separation logic without fixpoints, we might express *sp* as

$$sp(P, \texttt{while}\ \ E\ \ \texttt{do}\ \ C) = (\text{lfp}^{\models}_{\texttt{false}}\ \ \lambda X.sp(X \wedge E = \texttt{true}, C) \vee P) \wedge\ (E = \texttt{false})$$

with $\text{lfp}^{\models}_{\texttt{false}}\ \lambda X.F(X)$ defined, if it exists, as a formula P which satisfies:

- $P \equiv F(P)$
- for any formula Q, ($Q \equiv F(Q)$ implies $P \models Q$)

where

- $Q \models P$ if and only if $[\![Q]\!] \subseteq [\![P]\!]$ or $[\![Q]\!]$ and $[\![P]\!]$ are both not defined;
- $P \equiv Q$ if and only if $P \models Q$ and $Q \models P$.

This implies that during the computation of the *sp*, each time a `while` loop occurs, we must find a formula in existing separation logic that was provably the fixpoint, so that we could continue the computation of the *sp*. In another sense, this "work" could be seen as the "work" of finding the strongest loop invariant in the application of the usual rule for `while` loop.

Our addition of fixpoints (and the related postponed substitution) allows us to express the *sp* directly within the logic:

$$sp(P, \texttt{while}\ E\ \texttt{do}\ C) = (\mu X_v.sp(X_v \wedge E = \texttt{true}, C) \vee P) \wedge\ (E = \texttt{false}).$$

Although the definitions of the *wlp* and *sp* for the `while` loop are simple and elegant, the "work" of finding loop invariants is not skipped, however it is now postponed for when we have a specific proof to undertake. For example, we are working on translations of formulae into some other domains, and we have to find an approximation of the translation of fixpoints which is precise and not too expensive to compute. The advantage here is that this work of building the translation is done once and for all, then the analysis can be fully automated while the methodology of a proof system and finding loop invariant implies hand work.

$\wr$ / $\int$ is not { / } In this paper, we use the notation $P\{E/x\}$ for capture-avoiding syntactical substitution (that is, the usual substitution of variables). Recall that $\wr$ / $\int$ is a connective of the logic (called *postponed substitution*) and is not equivalent to { / }. It might be helpful for the reader to understand $\wr$ / $\int$ to look at the formula $P\wr E/x\int$ as (Moggi's [9]) call-by-value, $\texttt{let}\ x = E\ \texttt{in}\ P$.

The distinction between $\wr$ / $\int$ and { / } can be viewed in this example, where the command will be stuck in any state that has no value in its stack for y:

$$\{\texttt{true}\}x := y\{\texttt{true}\} \text{ is false}$$

This implies that the classical axiom for assignment, $\{P\{y/x\}\}x := y\{P\}$, is *unsound*.

In other versions of separation logic [10], $\{P\{y/x\}\}x := y\{P\}$ was sound, since the definition of a true triple required $FV(C,Q) \subseteq dom(s)$ and not merely $FV(Q) \subseteq dom(s)$, as here, *and also because there was no recursion.*

We believe that our definition (and our choice to allow stacks to be partial functions) is better since it does not require variables of the program to have a default value in the stack and it checks whether a variable has been assigned before we try to access its value. In any case, the addition of fixpoints does not require stacks to be partial functions. (Indeed, if stacks were total functions, then more laws would hold for $\wr$ / $\int$, but the latter's definition would remain different from { / }'s.)

Unfolding As usual, we have $\mu X_v.P \equiv P\{\mu X_v.P/X_v\}$
and $\nu X_v.P \equiv P\{\nu X_v.P/X_v\}$

{/}: No variable renaming Surprisingly, we have $\exists y.P \not\equiv \exists z.P\{z/y\}$ with $z \notin Var(P)$ *(when $y \neq z$)*. Here are two counterexamples, which expose the difficulties:

Counterexample 1:

$$[\![\nu X_v.y = 3 \wedge \exists y.(X_v \wedge y = 5)]\!] \not\equiv [\![\nu X_v.y = 3 \wedge \exists z.(X_v \wedge z = 5)]\!]$$

The left-hand side denotes the empty set, while the right-hand side denotes $[\![y = 3]\!]$. Here are the detailed calculations:

$$\begin{aligned}
&[\![\nu X_v.y = 3 \wedge \exists \boldsymbol{y}.(X_v \wedge \boldsymbol{y} = 5)]\!]_\emptyset \\
&= \mathrm{gfp}^{\subseteq}_{\emptyset} \lambda Y.\ [\![y = 3 \wedge \exists \boldsymbol{y}.(X_v \wedge \boldsymbol{y} = 5)]\!]_{[X_v \to Y]} \\
&= \mathrm{gfp}^{\subseteq}_{\emptyset} \lambda Y.\ [\![y = 3]\!]_{[X_v \to Y]} \cap [\![\exists \boldsymbol{y}.(X_v \wedge \boldsymbol{y} = 5)]\!]_{[X_v \to Y]} \\
&= \mathrm{gfp}^{\subseteq}_{\emptyset} \lambda Y.\ \{s,h \mid s(y) = 3\} \cap \{s,h \mid \exists v.[s \mid \boldsymbol{y} \to v], h \in [\![X_v \wedge \boldsymbol{y} = 5]\!]_{[X_v \to Y]}\} \\
&= \mathrm{gfp}^{\subseteq}_{\emptyset} \lambda Y.\ \{s,h \mid s(y) = 3\} \cap \{s,h \mid \exists v.[s \mid \boldsymbol{y} \to v], h \in Y \wedge [s \mid \boldsymbol{y} \to v](\boldsymbol{y}) = 5\} \\
&= \mathrm{gfp}^{\subseteq}_{\emptyset} \lambda Y.\ \{s,h \mid s(y) = 3\} \cap \{s,h \mid [s \mid \boldsymbol{y} \to 5], h \in Y\} \\
&= \boldsymbol{\emptyset}
\end{aligned}$$

$$
\begin{aligned}
&[\![\nu X_v.y = 3 \wedge \exists \boldsymbol{z}.(X_v \wedge \boldsymbol{z} = 5)]\!]_\emptyset \\
&= \mathrm{gfp}^{\subseteq}_{\emptyset} \lambda Y.\ [\![y = 3 \wedge \exists \boldsymbol{z}.(X_v \wedge \boldsymbol{z} = 5)]\!]_{[X_v \to Y]} \\
&= \mathrm{gfp}^{\subseteq}_{\emptyset} \lambda Y.\ [\![y = 3]\!]_{[X_v \to Y]} \cap [\![\exists \boldsymbol{z}.(X_v \wedge \boldsymbol{z} = 5)]\!]_{[X_v \to Y]} \\
&= \mathrm{gfp}^{\subseteq}_{\emptyset} \lambda Y.\ \{s, h \mid s(y) = 3\} \cap \{s, h \mid \exists v.[s \mid \boldsymbol{z} \to v], h \in [\![X_v \wedge \boldsymbol{z} = 5]\!]_{[X_v \to Y]}\} \\
&= \mathrm{gfp}^{\subseteq}_{\emptyset} \lambda Y.\ \{s, h \mid s(y) = 3\} \cap \{s, h \mid \exists v.[s \mid \boldsymbol{z} \to v], h \in Y \wedge [s \mid \boldsymbol{z} \to v](\boldsymbol{z}) = 5\} \\
&= \mathrm{gfp}^{\subseteq}_{\emptyset} \lambda Y.\ \{s, h \mid s(y) = 3\} \cap \{s, h \mid [s \mid \boldsymbol{z} \to 5], h \in Y\} \\
&= \boldsymbol{\{s, h \mid s(y) = 3\}}
\end{aligned}
$$

Here is some intuition: For the left-hand side, $y = 3$ says that all the states defined by the assertion must bind y to 3, and "$\exists y.X_v \wedge y = 5$" says that for all those states defined by the assertion, we can bind y such that it satisfies $y = 5$, *even as it satisfies* $y = 3$, *due to the recursion*, which is impossible, so we have $\emptyset$ as the denotation.

For the right-hand side, $y = 3$ asserts again that y binds to 3, and $\exists z.X_v \wedge z = 5$ says that for all states in the assertion's denotation, we bind 5 to z, which is indeed possible, so we have $[\![y = 3]\!]$ as the denotation of the assertion.

Counterexample 1 shows that variable renaming has a special behavior when applied to a formula which is not v-closed.

Counterexample 2:

$$[\![\exists y.\nu X_v.y = 3 \wedge \exists y.(X_v \wedge y = 5)]\!] \not\equiv [\![\exists z.\nu X_v.z = 3 \wedge \exists y.(X_v \wedge y = 5)]\!]$$

The left-hand side denotes the empty set, while the right-hand side denotes $S \times H$.

To see this, note that the left-hand side's semantics is essentially the same as its counterpart in the first counterexample. As for the right-hand side, if we apply the semantics of the right-hand side of the first counterexample, we see that $[\![\nu X_v.z = 3 \wedge \exists y.(X_v \wedge y = 5)]\!] = [\![z = 3]\!]$, signifying that all the states are such that we bind 5 to z. So, we have $S \times H$ as the denotation of the right-hand side.

Counterexample 2 shows that variables occurring free in the bodies of fixpoint formulae are subject to *dynamic binding* with respect to unrolling the recursive formulae via postponed substitution.

Full substitution The previous counterexample 2 leads to the definition of a new substitution:

Definition 4. *Let* $\{[\ /\ ]\}$ *be a full syntactical variable substitution:* $P\{[z/y]\}$ *is* P *in which all* y *are replaced by* z *wherever they occur, for example:*
$(\exists y.P)\{[z/y]\} \triangleq \exists z.(P\{[z/y]\})$, $(P\,\wr E/x\wr)\{[z/y]\} \triangleq (P\{[z/y]\})[E\{z/y\}/x\{z/y\}]$

The variable renaming theorem for $BI^{\mu\nu}$ (Th. 1) We define $class(z, s, h)$ as the set of states containing the state, s, h, and all other states identical to s, h except for z:

Definition 5. $class(z, s, h) = \{s', h \mid$ *for all* $v \in Val, [s' \mid z \to v] = [s \mid z \to v]\}$

Alternatively, we can say that $class(z, s, h)$ is that set which satisfies:

- $s_{\restriction_{dom(s)\setminus\{z\}}}, h \in class(z, s, h)$
- $\forall v.[s \mid z \rightarrow v], h \in class(z, s, h)$

Definition 6. *For $z \in Var$, $X \in \mathcal{P}(S \times H)$, define*

$$nodep(z, X) \triangleq True \text{ iff } \forall s, h \in X.class(z, s, h) \subseteq X$$

We extend this definition to environments as well:

$$nodep(z, \rho) \triangleq True \text{ iff } (\forall X_v \in dom(\rho).nodep(z, \rho(X_v))$$

Proposition 1. *If $nodep(z, \rho)$, $FV_v(P) \subseteq dom(\rho)$, $z \notin FV(P)$ and $[\![P]\!]_\rho$ exists, then $nodep(z, [\![P]\!]_\rho)$*

The idea is, if P is v-closed and z does not occur free in P, then $\forall v.$ $(s, h \in [\![P]\!]$ iff $[s \mid z \rightarrow v], h \in [\![P]\!])$. Yet another phrasing goes, if z does not occur free in a v-closed formula, then the set of states satisfying the formula does not have any particular values for z.

Now, let

- $s^{\bullet}_{y,z} \triangleq \begin{cases} [s \mid y \rightarrow s(z)] & \text{if } z \in dom(s) \\ s & \text{if } z \notin dom(s) \end{cases}$
- $\rho^{\bullet}_{y,z} \triangleq [\forall X_v \in dom(\rho).\ X_v \rightarrow \{s, h \mid s^{\bullet}_{y,z}, h \in \rho(X_v)\}]$

Proposition 2. *If $nodep(z, \rho)$ and $z \notin Var(P)$, then $[\![P\{[z/y]\}]\!]_{\rho^{\bullet}_{y,z}} = \{s, h \mid s^{\bullet}_{y,z}, h \in [\![P]\!]_\rho\}$*

Theorem 1. *If P is v-closed, $z \notin Var(P)$ and $y \notin FV(P)$, then $P \equiv P\{[z/y]\}$.*
In particular, $\exists y.P \equiv \exists z.(P\{[z/y]\})$.

Proof (Th. 1). The proof follows from Propositions 1 and 2, assuming that all fixpoints are defined from monotonic functionals.
□

Equivalences on $\wr$ / $\int$ We define $is(E) \triangleq E = E$, which is just a formula ensuring that E has a value in the current state. If we had chosen that stacks were only total functions, $is(E)$ would always be equivalent to `true` and there would be more simplifications. We have these facts:

- If P does not contain any v-variable or fixpoint or postponed substitution, then $P \wr E/x \int \equiv P\{E/x\} \wedge is(E)$.
- If $x \notin Var(E)$ then $P \wr E/x \int \equiv \exists x.x = E \wedge P$
- If P is v-closed and if $x_1 \notin Var(E)$ and $x_1 \neq x_2$, then:
  $(\exists x_1.P) \wr E/x_2 \int \equiv \exists x_1.(P \wr E/x_2 \int)$.
- $(\exists x.P) \wr E/x \int \equiv (\exists x.P) \wedge is(E)$.
- $(A \vee C) \wr E/x \int \equiv (A \wr E/x \int) \vee (C \wr E/x \int)$.

– If $y \notin \mathit{Var}(P)$, then
$(\mu X_v.P)\ \wr y/x \int \equiv (\mu X_v.P\{\ \wr y/x \int\}) \wedge is(y)$
$(\nu X_v.P)\ \wr y/x \int \equiv (\nu X_v.P\{\ \wr y/x \int\}) \wedge is(y)$.

Concerning the last item, one would want a similar equivalence for E instead of y, but this is not possible since $(P\ \wr E'/x \int)\{[E/x]\}$ is not defined because $P\ \wr E'/E \int$ is not defined. (The last argument must be a variable.) This explains why $\wr\ /\ \int$ must be a connective.
To understand the last equivalence, we must return to the programming point of view, seeing fixpoints as `while` loops and $\wr\ /\ \int$ as assignments, so that the precondition for $x := w$; `while` $x = y$ `do` $x := x + 1$ is the same as the one for `while` $w = y$ `do` $w := w + 1$. (In Sect. 4, we will learn that this will be $(\nu X_v.(x \neq y) \vee ((x = y) \wedge X_v\ \wr x+1/x \int))\ \wr w/x \int \equiv is(w) \wedge (\nu X_v.(w \neq y) \vee ((w = y) \wedge X_v\ \wr w+1/w \int))$.)

Example of unfolding Let $\texttt{nclist42}(x) \triangleq \mu X_v.(x = \texttt{nil}) \vee \exists x_2.((x \mapsto 42, x_2) * X_v\ \wr x_2/x \int)$ with $x_2 \neq x$. Let's prove that $X_v\ \wr x_2/x \int$ is equivalent to $\texttt{nclist42}(x_2)$.

$$
\begin{array}{lcl}
\texttt{nclist42}(x) & \triangleq & \mu X_v.(x = \texttt{nil}) \vee \exists x_2.((x \mapsto 42, x_2) * X_v\ \wr x_2/x \int) \\
(unfolding) & = & (x = \texttt{nil}) \vee \exists x_2.((x \mapsto 42, x_2) * \\
 & & ((\mu X_v.(x = \texttt{nil}) \vee \exists x_2.((x \mapsto 42, x_2) * X_v\ \wr x_2/x \int))\ \wr x_2/x \int)) \\
\text{(Th. 1)} & = & (x = \texttt{nil}) \vee \exists x_2.((x \mapsto 42, x_2) * \\
 & & ((\mu X_v.(x = \texttt{nil}) \vee \exists x_3.((x \mapsto 42, x_3) * X_v\ \wr x_3/x \int))\ \wr x_2/x \int)) \\
(\text{simplify } \wr\ /\ \int \text{ case } \mu) & = & (x = \texttt{nil}) \vee \exists x_2.((x \mapsto 42, x_2) * \\
 & & (\mu X_v.(x_2 = \texttt{nil}) \vee \exists x_3.((x_2 \mapsto 42, x_3) * X_v\ \wr x_3/x_2 \int))) \\
 & \triangleq & (x = \texttt{nil}) \vee \exists x_2.((x \mapsto 42, x_2) * \texttt{nclist42}(x_2))
\end{array}
$$

So we have $\texttt{nclist42}(x_2) \equiv (\texttt{nclist42}(x))\ \wr x_2/x \int$, as expected.

Why is $BI + \mu + \nu \neq BI^{\mu\nu}$? Or, why do we need to add $\wr\ /\ \int$ to the syntax? Informally stated, one can view the fixpoint as a while loop and $\wr\ /\ \int$ as an assignment, then if we have a while loop followed by an assignment, we cannot include the assignment within the loop. So, if an analysis postponed the computation of while loop (fixpoint), then it also has to postpone the computation of assignment ($\wr\ /\ \int$).

The need for $\wr\ /\ \int$ is not surprising. In [4], de Bakker proved that for his simple logic with fixpoints, there was no *sp* for the while loop statements.

Indeed, for P without any μ, ν, X_v in it, we have $P\ \wr E/x \int \equiv P\{E/x\} \wedge is(E)$. But for $BI^{\mu\nu}$ without the connective $\wr\ /\ \int$, there is no formula in the logic equivalent to $P\ \wr E/x \int$, which means that $\wr\ /\ \int$ has to be in the logic syntax. For example, $(\exists y.P)\ \wr E/x \int \not\equiv \exists y.(P\ \wr E/x \int)$ when $y \neq x$ but $y \in \mathit{Var}(E)$ but the renaming theorem: $\exists y.P \equiv \exists z.P\{z/y\}$ with $z \notin \mathit{Var}(P)$ does not hold, so the attempt to find an equivalent formula for $(\exists y.P)\ \wr E/x \int$ will fail.

4 Backward analysis

We now define the *weakest liberal precondition* (*wlp*) semantics of the while-loop language with pointers; see Fig. 4. Most of the clauses are from Ishtiaq and O'Hearn [7], but our definition for `while` E `do` C is new and crucial. We add to *wlp* a parameter $V \in \mathcal{P}(Var)$, such that when choosing fresh variables, they are not in V.

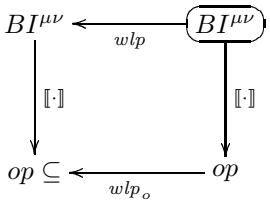
If we can establish $\{P\}C\{\texttt{true}\}$, then we will know that execution of C is safe in any state satisfying P. So for our backward analysis, in Fig. 4 we express *wlp* such that

Theorem 2. $[\![wlp(P,C)]\!] = wlp_o([\![P]\!], C)$.

Corollary 1. $\{wlp(P,C)\}C\{P\}$ *is true.*

Proof. To prove that our definition indeed defines *wlp*, we formally relate it to the inverse state-transition function wlp_o: The definition of a true triple implies that:

$$\{wlp(P,C)\}C\{P\} \text{ true} \quad \text{if and only if} \quad \left(\begin{array}{c}[\![wlp(P,C)]\!] \\ \cap\{s,h \mid FV(P) \subseteq dom(s)\}\end{array}\right) \subseteq wlp_o([\![P]\!], C)$$



To prove that our analysis is correct, we express wlp_o for each command, and prove by induction on the syntax of C that for each C and P, we have $[\![wlp(P,C)]\!] \subseteq wlp_o([\![P]\!], C)$. To prove that those preconditions are the weakest we establish that $[\![wlp(P,C)]\!] = wlp_o([\![P]\!], C)$
□

Example: $wlp(\texttt{true}, \texttt{while}\ i > 0\ \texttt{do}\ (x := x \cdot 2; i := i - 1)) \triangleq \nu X_v.((x \leq 0 \wedge \texttt{true}) \vee (i > 0 \wedge \exists x_1, x_2.\ (X_v\ \wr i - 1/i \wr\ \wr x_2/x \wr) \wedge (x \mapsto x_1, x_2)))$, which simplifies to $\nu X_v.i \leq 0 \vee (i > 0 \wedge \exists x_1, x_2.\ X_v\ \wr i - 1/i \wr\ \wr x_2/x \wr \wedge x \mapsto x_1, x_2)$.

5 Forward analysis

In the previous section, we defined *wlp* for C and P such that $\{wlp(P,C)\}C\{P\}$ is true. Unfortunately, the strongest postcondition semantics $sp(P,C)$ is not always defined — we can find C and P such that there exists no Q that makes $\{P\}C\{Q\}$ true. This is due to the fact that a true triple requires C to be executable from all states satisfying P *and also such that* $FV(Q) \subseteq dom(s)$ which is obviously *not* the case for some C and P. (For example, $\{\texttt{true}\}x := nil; y := x.1\{?\}$ has no solution, since all states satisfy P but the command can never be executed — `nil`.1 is not defined).

We therefore split the analysis into two steps. The first step checks whether C is executable from all states satisfying P or not. The second step gives $sp(P,C)$

$$
\begin{aligned}
wlp(P, C) &= wlp_{\emptyset}(P, C) \\
wlp_V(P, x := E) &= P\,\wr E/x\wr \\
wlp_V(P, x := E.i) &= \exists x_1 \exists x_2.(P\,\wr x_i/x\wr \wedge (E \hookrightarrow x_1, x_2)) \\
&\quad \text{with } x_i \notin V \cup FV(E, P) \\
wlp_V(P, E.1 := E') &= \exists x_1 \exists x_2.(E \mapsto x_1, x_2) * ((E \mapsto E', x_2) \mathrel{-\!\!*} P) \\
&\quad \text{with } x_i \notin V \cup FV(E, E', P) \\
wlp_V(P, E.2 := E') &= \exists x_1 \exists x_2.(E \mapsto x_1, x_2) * ((E \mapsto x_1, E') \mathrel{-\!\!*} P) \\
&\quad \text{with } x_i \notin V \cup FV(E, E', P) \\
wlp_V(P, x := \texttt{cons}(E_1, E_2)) &= \forall x'.(x' \mapsto E_1, E_2) \mathrel{-\!\!*} P\,\wr x'/x\wr \\
&\quad \text{with } x' \notin V \cup FV(E_1, E_2, P) \\
wlp_V(P, \texttt{dispose}(E)) &= P * (\exists a \exists b.(E \mapsto a, b)) \\
&\quad \text{with } a, b \notin V \cup FV(E) \\
wlp_V(P, C_1; C_2) &= wlp_V(wlp_V(P, C_2), C_1) \\
wlp_V(P, \texttt{if } E \texttt{ then } C_1 \texttt{ else } C_2) &= (E = \texttt{true} \wedge wlp_V(P, C_1)) \\
&\quad \vee (E = \texttt{false} \wedge wlp_V(P, C_2)) \\
wlp_V(P, \texttt{skip}) &= P \\
wlp_V(P, \texttt{while } E \texttt{ do } C_1) &= \nu X_v.((E = \texttt{true} \wedge wlp_{V \cup Var(E,P)}(X_v, C_1)) \\
&\quad \vee (E = \texttt{false} \wedge P)) \\
&\quad \text{with } X_v \text{ not in } P
\end{aligned}
$$

Fig. 4. Weakest liberal preconditions

that makes the triple $\{P\}C\{sp(P, C)\}$ true if C is executable from all states satisfying P. **Step 1 :** $wlp(\texttt{true}, C)$**:**

$$(\forall s, h \in [\![P]\!].\ C, s, h \text{ is safe}) \text{ if and only if } (P \models wlp(\texttt{true}, C))$$

The first step expresses the $wlp(\texttt{true}, C)$ formulae, which are the formulae given in Fig. 4, instantiated for $P = \texttt{true}$.
Step 2: $sp(P, C)$**:** This is given in Fig. 5.
This gives us

Theorem 3. *$sp_o([\![P]\!], C) = [\![sp(P, C)]\!]$.*

Corollary 2. *$\{P \wedge wlp(true, C)\}C\{sp(P \wedge wlp(true, C), C)\}$ is always true. In case $P \models wlp(\texttt{true}, C)$ this is equivalent to $\{P\}C\{sp(P, C)\}$ is true.*

Corollary 3. *If $P \not\models wlp(\texttt{true}, C)$, then C cannot be executable from all states satisfying P. But for those states from which C is executable, the final states satisfy $sp(P \wedge wlp(\texttt{true}, C), C)$.*

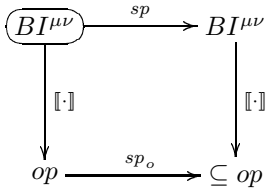
Our $sp(P, C)$ makes the triple $\{P\}C\{sp(P, C)\}$ always true in the usual definition of Hoare triples (which is $\{P\}C\{Q\}$ true iff $sp_o([\![P]\!], C) \subseteq [\![Q]\!]$).

Proof. To prove that our definition indeed defines sp, we formal relate it to the inverse state-transition function sp_o. The definition of a true triple implies that

$$\{P\}C\{Q\} \text{ true if and only if } P \models wlp(\texttt{true}, C) \wedge sp_o([\![P]\!], C) \subseteq [\![Q]\!]$$

To prove that our analysis is correct, we expressed sp_o for each command, and proved by induction on the syntax of C that for each C, and P, we have

If $P \models wlp(\texttt{true}, C)$
then $sp_o([\![P]\!], C) \subseteq [\![sp(P, C)]\!]$



But since sp_o is defined such that it only collects the final states of successful computations, we must only prove that for each C and P: $sp_o([\![P]\!], C) \subseteq [\![sp(P, C)]\!]$.

Finally, to prove that those postconditions are the strongest we have established that $sp_o([\![P]\!], C) = [\![sp(P, C)]\!]$ □

$$
\begin{aligned}
sp(P, C) &= sp_{\emptyset}(P, C) \\
sp_V(P, x := E) &= \exists x'.\ P\,\wr x'/x\wr \wedge x = E\{x'/x\} \\
&\quad \text{with } x' \notin V \cup FV(E, P) \\
sp_V(P, x := E.i) &= \exists x'.\ P\,\wr x'/x\wr \wedge x = (E\{x'/x\}).i \\
&\quad \text{with } x' \notin V \cup FV(E, P) \\
sp_V(P, E.1 := E') &= \exists x_1 \exists x_2.(E \mapsto E', x_2) * ((E \mapsto x_1, x_2) \mathrel{-\!\!*} P) \\
&\quad \text{with } x_i \notin V \cup FV(E, E', P) \\
sp_V(P, E.2 := E') &= \exists x_1 \exists x_2.(E \mapsto x_1, E') * ((E \mapsto x_1, x_2) \mathrel{-\!\!*} P) \\
&\quad \text{with } x_i \notin V \cup FV(E, E', P) \\
sp_V(P, x := \texttt{cons}(E_1, E_2)) &= \exists x'.(P\,\wr x'/x\wr * (x \mapsto E_1\{x'/x\}, E_2\{x'/x\})) \\
&\quad \text{with } x' \notin V \cup FV(E_1, E_2, P) \\
sp_V(P, \texttt{dispose}(E)) &= \exists x_1, x_2.\ ((E \mapsto x_1, x_2) \mathrel{-\!\!*} P) \\
&\quad \text{with } x_1, x_2 \notin V \cup FV(E, P) \\
sp_V(P, C_1; C_2) &= sp_V(sp_V(P, C_1), C_2) \\
sp_V(P, \texttt{if } E \texttt{ then } C_1 \texttt{ else } C_2) &= sp_V(P \wedge E = \texttt{true}, C_1) \\
&\quad \vee sp_V(P \wedge E = \texttt{false}, C_2) \\
sp_V(P, \texttt{skip}) &= P \\
sp_V(P, \texttt{while } E \texttt{ do } C_1) &= (\mu X_v.sp_{V \cup Var(E,P)}(X_v \wedge E = \texttt{true}, C_1) \vee P) \\
&\quad \wedge (E = \texttt{false}) \\
&\quad \text{with } X_v \text{ not in } P
\end{aligned}
$$

Fig. 5. Strongest postconditions

Example: $sp(\texttt{true}, i := 0; x := \texttt{nil}; \texttt{while}\ i \neq 5\ \texttt{do}\ x := \texttt{cons}(i, x); i := i + 1) \triangleq i = 5 \wedge \mu X_v.((\exists x'.(\exists i'.\texttt{true}\,\wr i'/i\wr \wedge i = 0)\,\wr x'/x\wr \wedge x = \texttt{nil}) \vee \exists i'.(\exists x'.(X_v \wedge i \neq 5)\,\wr x'/x\wr * (x \mapsto i, x'))\,\wr i'/i\wr \wedge i = i' + 1)$, which is after simplifications, $i = 5 \wedge \mu X_v.((i = 0 \wedge x = \texttt{nil}) \vee \exists x'.i'.i = i' + 1 \wedge i' \neq 5 \wedge (X_v[x'/x] * (x \mapsto i', x')))$

6 Variations of $BI^{\mu\nu}$

Our version of $BI^{\mu\nu}$ is not unique. One variant would preserve the usual renaming theorem but at the price of additional complexity in defining fixpoint formulas: the v-variables become functions whose parameters are the free variables of the formula. Instead of having postponed substitution, one would have an application connective. The syntax reads

$$\begin{array}{l}\texttt{nonCircularList}(x) = \\ \quad \mu X_v(x).\ (x = \texttt{nil}) \vee \exists x_1, x_2.(\texttt{isval}(x_1) \wedge (x \mapsto x_1, x_2\ *\ X_v(x_2)))\end{array}$$

When considering our example to the renaming theorem, one states

$$\exists y.\nu X_v(y).y = 3 \wedge \exists y.(X_v(y) \wedge y = 5)$$

which becomes equivalent to

$$\exists z.\nu X_v(z).z = 3 \wedge \exists y.(X_v(y) \wedge y = 5)$$

Those formulas are not precisely stated; let us try to formalize. The changes are that $\rho : Var_v \rightharpoonup (\mathcal{P}(Var) \times \mathcal{P}(S \times H))$ and that $[\![\cdot]\!]_\rho \ : \ BI^{\mu\nu\Lambda} \rightharpoonup (\mathcal{P}(S \times H) \uplus (\mathcal{P}(Var) \times \mathcal{P}(S \times H)))$.

The semantics for fixpoints and postponed substitution would be:

$$\begin{array}{ll}[\![\mu X_v(x_1, ..., x_n)\ .\ P]\!]_\rho & = \mathrm{lfp}^{\subseteq}_{\emptyset} \lambda X.\ [\![P]\!]_{[\rho | X_v \to ((x_1,...,x_n),X)]} \\ [\![\nu X_v(x_1, ..., x_n)\ .\ P]\!]_\rho & = \mathrm{gfp}^{\subseteq}_{\emptyset} \lambda X.\ [\![P]\!]_{[\rho | X_v \to ((x_1,...,x_n),X)]} \\ [\![P(E_1, ..., E_n)]\!]_\rho & = \left\{ s, h \mid \begin{array}{l} \exists x_1, ..., x_n, X.\ [\![P]\!]_\rho = ((x_1, ..., x_n), X) \\ \wedge [s \mid x_1 \to [\![E_1]\!]^s \mid ... \mid x_n \to [\![E_n]\!]^s], h \in X \end{array} \right\}\end{array}$$

But this implies that to write $\mu X_v(x_1, ..., x_n).\ P$, we must consider the free variables in P. (Maybe this would help the users of the logic!)

Another important point is that $[\![\nu X_v.X_v]\!] = S \times H$, while $[\![\nu X_v(x).X_v(x)]\!] = \{s, h \mid x \in dom(s)\}$. That is, if one allows partial functions for stacks (as we do), the meaning changes.

To update our definition, $wlp(P, x := E) = P\,\wr E/x \wr$, we require a function connective, and we write $wlp(P, x := E) = (\Lambda x.P)(E)$. (we use Λ instead of λ to avoid confusion between a real function and the new function connective.) And instead of writing $\mu X_v(x_1, ..., x_n).P$, we would write $\mu(\Lambda(x_1, ..., x_n).X_v).P$. The non-circular linear list example reads as follows:

$$\begin{array}{l}\texttt{nonCircularList}(x) = \\ \quad \mu X_v.\ (x = \texttt{nil}) \vee \exists x_1, x_2.(\texttt{isval}(x_1) \wedge (x \mapsto x_1, x_2\ *\ (\Lambda x.X_v)(x_2)))\end{array}$$

This implies a new semantics: First, we define a new recursive type, $res = \mathcal{P}(S \times H) \uplus (Var \times res)$. Next, we define

$$\begin{array}{ll} apply & : \ (Exp \times (Var \times res)) \to res \\ apply(E, (x, S)) & = \{s, h \mid [s \mid x \to [\![E]\!]^s], h \in S\} \text{ if } S \in \mathcal{P}(S \times H) \\ apply(E, (x, (y, S))) & = (y, apply(E, (x, S)))\end{array}$$

$$
\begin{array}{lll}
[\![.]\!]_\rho & : & BI^{\mu\nu\Lambda} \rightharpoonup res \\
 & & ... \\
[\![\Lambda x.P]\!]_\rho & = & ((x), [\![P]\!]_\rho) \\
[\![\mu X_v \ . \ P]\!]_\rho & = & \mathrm{lfp}^{\subseteq}_{\emptyset} \lambda X. \ [\![P]\!]_{[\rho | X_v \rightarrow X]} \\
[\![\nu X_v \ . \ P]\!]_\rho & = & \mathrm{gfp}^{\subseteq}_{\emptyset} \lambda X. \ [\![P]\!]_{[\rho | X_v \rightarrow X]} \\
[\![P(E)]\!]_\rho & = & apply(E, [\![P]\!]_\rho) \ \text{ if } \exists x, X. \ [\![P]\!]_\rho = (x, X)
\end{array}
$$

With this semantics, in *wlp* or *sp*, the only change is that wherever $P\{E/x\}$ appears, it should be replaced by $(\Lambda x.P)E$.

As for our counterexample of the renaming theorem,

$$\exists y.\nu X_v.y = 3 \wedge \exists y.(((\Lambda y.X_v)y) \wedge y = 5)$$

becomes equivalent to

$$\exists z.\nu X_v.z = 3 \wedge \exists y.(((\Lambda z.X_v)y) \wedge y = 5)$$

But again, this is not the usual renaming theorem.

What we propose now is a mix of the last two semantics, where the renaming theorem will not always hold, but if one wants it to hold, then one must verify, wherever there is a fixpoint, that the fixpoint should be written in the format, $\mu X_v(FV(P)).P$. The user must be aware that $(\Lambda x.P)(x)$ is not always equivalent to P (because $(\Lambda x.P)(x)$ implies that x can be evaluated in the actual context (i.e. $x \in dom(s)$)).

Now $\rho \ : \ Var_v \rightharpoonup res$, and the example becomes

$$\exists y.\nu X_v(y).y = 3 \wedge \exists y.(X_v(y) \wedge y = 5)$$

which is equivalent to

$$\exists z.\nu X_v(z).z = 3 \wedge \exists y.(X_v(y) \wedge y = 5)$$

The semantics goes as follows; omitted clauses are the same as those in Sect. 3.2.

$$
\begin{array}{lll}
[\![.]\!]_\rho & : & BI^{\mu\nu\Lambda} \rightharpoonup res \\
 & & ... \\
[\![\Lambda x.P]\!]_\rho & = & ((x), [\![P]\!]_\rho) \\
[\![\mu X_v \ . \ P]\!]_\rho & = & \mathrm{lfp}^{\subseteq}_{\emptyset} \lambda X. \ [\![P]\!]_{[\rho | X_v \rightarrow X]} \\
[\![\mu X_v(x_1, ..., x_n) \ . \ P]\!]_\rho & = & \mathrm{lfp}^{\subseteq}_{\emptyset} \lambda X. \ [\![P]\!]_{[\rho | X_v \rightarrow (x_1, (..., (x_n, X)))]} \\
[\![\nu X_v \ . \ P]\!]_\rho & = & \mathrm{gfp}^{\subseteq}_{\emptyset} \lambda X. \ [\![P]\!]_{[\rho | X_v \rightarrow X]} \\
[\![\nu X_v(x_1, ..., x_n) \ . \ P]\!]_\rho & = & \mathrm{gfp}^{\subseteq}_{\emptyset} \lambda X. \ [\![P]\!]_{[\rho | X_v \rightarrow (x_1, (..., (x_n, X)))]} \\
[\![P(E)]\!]_\rho & = & apply(E, [\![P]\!]_\rho) \ \text{ if } \exists x, X. \ [\![P]\!]_\rho = (x, X)
\end{array}
$$

(We gave the semantics for $\mu X_v.P$ and $\nu X_v.P$ separately but they are the nullary case of the other considering that $(x_1, (..., (x_0, X))) = X$).

This semantics preserves our *wlp* and *sp* formulas (switching $P\{E/x\}$ with $(\Lambda x.P)(E)$) and allows the user to have functions for v-variable and a restricted renaming theorem.

Remark Since the first draft of this paper, Yang et al. [16] present another way to add fixpoints to separation logic. They add the connective $\texttt{let rec}\ X_v(x_1,...,x_n) = P\ \texttt{in}\ Q$ which is a mix of `let` and recursion. Their logic is not equivalent to ours, even just for the `let` part. To be equivalent, "let" should only be a substitution i.e $[\![\texttt{let}\ X_v = P\ \texttt{in}\ Q]\!]_\rho \triangleq [\![Q]\!]_{[\rho|X_v \to [\![P]\!]_\rho]}$ but they define it as $[\![\ \texttt{let}\ X_v = P\ \texttt{in}\ Q]\!]^y_\rho \triangleq \left\{ s,h \mid s,h \in [\![Q]\!]^y_{[\rho|X_v \to S\times\{h|s,h\in[\![P]\!]^y_\rho\}]} \right\}$. This could be seen roughly as a "postponed substitution" on v-variables. It behaves not as usual, for example $[\![\texttt{let}\ X_v = (x=5)\ \texttt{in}\ \exists x.X_v]\!] = [\![x=5]\!]$. One deficiency of their work is that its correctness proof relies on the correctness of separation-logic triples, which is not proved for their version of separation logic with fixpoints. They cannot directly use our triples because of the inequivalence between our logic and theirs. Still, we believe that we can express in our logic $\texttt{let rec}\ X_v(x_1,...,x_n) = P\ \texttt{in}\ Q$ when $FV(P) \subseteq \{x_1,...,x_n\}$, but this will not be simple since their recursion represents fixpoints on $\mathcal{P}(H)$ while ours represents fixpoints on $\mathcal{P}(S \times H)$.

7 Conclusion

We have proposed an extension of separation logic, with fixpoint connectives and postponed substitution. This lets us express formulae of recursive definitions within the logic and solve problems that cannot be handled by lightweight checking tools. Second, we can now express the *sp* and *wlp* in the case of `while` statements. (To the best of our knowledge, there is no forward analysis using separation logic in the literature). We expressed the *sp* and *wlp* operators for all commands without any syntactical restrictions on the formulae provided as pre- or post-conditions. This leads to automatic analysis which take a different approach from the usual set-of-rules analysis that require significant human intervention.

We are applying our extended separation logic to develop verification tools that go beyond lightweight checking: we use separation logic as an interface language for abstract interpretation of programs that manipulate heap storage. Recursively defined assertions summarize the *shape properties* [11, 8] of objects that live in the heap, permitting abstract interpretation to compute the shape of a program's dynamic data structures. The wp- and sp-semantic formulations given in this paper express the denotations of program components, allowing compositional abstract interpretation [3]. (A typical abstract interpretation is a whole-program analysis using a weakly expressive property language; a compositional analysis requires a richer property language, which is often defined in an ad-hoc fashion). Research in these directions is underway, both in the author's own work and in other recent efforts [16].

Acknowledgments I am grateful to Patrick Cousot, Radhia Cousot, and David Schmidt for their guidance and helpful discussions. I would like to thank Josh Berdine for discussions and comments that helped improve this version of the article and inspired Sect. 6.

References

1. P. Cousot and R. Cousot. Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints In *POPL'77*, pages 238-252, Los Angeles, CA, 1977. ACM Press.
2. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL'79*, pages 269–282, San Antonio, Texas, 1979. ACM Press.
3. P. Cousot and R. Cousot. Modular Static Program Analysis. In *LNCS 2304*, page 159-178, 2002.
4. J. W. de Bakker. *Mathematical Theory of Program Correctness.* Prentice Hall, Englewood Cliffs, NJ, 1980.
5. E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, NJ, 1976.
6. C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12:576–580, 1969.
7. S. Ishtiaq and P. O'Hearn. BI as an assertion language for mutable data structures. In *POPL'01*, pages 14–26, 2001.
8. P. Lam, V. Kuncak and M. Rinard Generalized Typestate Checking for Data Structure Consistency In *VMCAI'05*, 2005.
9. E. Moggi. Notions of Computation and Monads. In *Information and Computation*, Vol. 93, pages 55–92, 1991.
10. J. C. Reynolds. Separation logic : A logic for shared mutable data structures. In *LICS'02*, pages 55–74, Denmark, 2002. IEEE Computer Society.
11. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems, 2002.
12. É.-J. Sims. Extending Separation Logic with Fixpoints and Postponed Substitution. In *AMAST'04*, LNCS 3116, pages 475–490, 2004. Springer.
13. É.-J. Sims. Pointers static analysis and BI-logic. *Mémoire de DEA*, École Polytechnique, France, DEA Programmation, 2002.
14. P. R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637, Saint-Malo, France, 1992. Springer-Verlag.
15. H. Yang An example of local reasoning in BI pointer logic:the Schorr-Waite graph marking algorithm In *SPACE'01*, London, 2001.
16. H. Yang O. Lee and K. Yi Automatic Verification of Pointer Programs Using Grammar-Based Shape Analysis In *ESOP'05*, Edinburgh, 2005.
17. H. Yang and P. O'Hearn. A semantic basis for local reasoning. In *FoSSaCS'02*, Lecture Notes in Computer Science, pages 402–416. Springer, 2002.
18. H. Yang P. O'Hearn and J. Reynolds. Syntactic control of interference. In *POPL'04*, Italy, 2004. ACM Press, New York, NY.