A Simple Model of Separation Logic for Higher-order Store

Lars Birkedal[1], Bernhard Reus[2], Jan Schwinghammer[3], and Hongseok Yang[4]

[1] IT University of Copenhagen [2] University of Sussex, Brighton
[3] Saarland University, Saarbrücken [4] Queen Mary, University of London

Abstract. Separation logic is a Hoare-style logic for reasoning about pointer-manipulating programs. Its core ideas have recently been extended from low-level to richer, high-level languages. In this paper we develop a new semantics of the logic for a programming language where code can be stored (i.e., with higher-order store). The main improvement on previous work is the simplicity of the model. As a consequence, several restrictions imposed by the semantics are removed, leading to a considerably more natural assertion language with a powerful specification logic.

1 Introduction

Higher-order store is included in modern programming languages in the form of code pointers and storable objects. "Higher-order" here refers to the fact that one can keep not only data in the store but also procedures or commands that manipulate the store themselves. It is widely used in systems code, such as operating system kernels, device drivers and web servers. For instance, the Linux kernel keeps multiple linked lists whose nodes store code fragments, and calls those fragments in response to external events, such as a signal from a printer.

However, formal reasoning about higher-order store is still an open problem. Although several sound program logics for higher-order store have been proposed, they either are intended for machine code [4] or they fail to combine local reasoning with intuitive rules for stored code while maintaining the simplicity of Hoare logic for first-order store [6, 16]. The difficulty is that a logic for higher-order store should accommodate reasoning about "recursion through the store", a tricky implicit recursion implemented by stored procedures.

The goal of our research is to solve the problem of reasoning about higher-order store using separation logic. Separation logic is a program logic for reasoning modularly about programs with pointers. It has been demonstrated that the logic substantially simplifies formal program verification in low-level C-like programming languages as well as richer, higher-level languages [18, 2, 13, 1, 12, 9, 3, 8, 7, 14]. Our aim is to design program logics for higher-order store that keep all the benefits of separation logic, such as (higher-order) frame rules, while providing efficient, sound proof rules for recursion through the store.

In this paper, we investigate the semantic foundations for developing separation logic for higher-order store. We build on the earlier work of Reus and

Schwinghammer [16] which identified key semantic challenges for such a logic, and provided fairly sophisticated solutions based on functor categories. In this paper, we take different approaches to the various problems, and as a result obtain a more powerful logic and a substantially simpler semantic model.

We now give an overview of two key semantic challenges that are involved in developing separation logic for higher-order store. We outline how those challenges were addressed in earlier work [16], and compare this with our new model.

The first challenge is to find a model that validates the frame rule known from separation logic [18]. In traditional models of separation logic [10], the soundness of the frame rule relies on programs satisfying a frame property, which says that the meaning of each program phrase only relies on its "footprint". To ensure that all program phrases – in particular, memory allocation – satisfy the frame property, the models interpret commands as relations (i.e., functions from input states to *sets* of output states), and memory allocation denotes a function that nondeterministically picks new memory. Now, in a language with higher-order store, the semantics involves solving recursive domain equations. With nondeterministic memory allocation, one is naturally led to recursive domain equations using powerdomains. These are problematic not only because it is unclear whether they can be used to show the existence of recursive properties of the heap but also because programs would no longer denote ω-continuous functions, due to the *countable* nondeterminism arising from memory allocation. Instead, Reus and Schwinghammer considered a functor category, indexed over finite sets of locations, which made it possible to prove that programs obeyed a frame property without relying on a nondeterministic allocator. However, this involved two non-trivial aspects. First, recursive domain equations now had to be solved not in an ordinary category of domains, but in the functor category. Second, the frame property became a recursively defined property, whose existence required a separate non-trivial proof. While Reus and Schwinghammer succeeded in defining a model that validates the frame rule, the technical complications involved make it difficult to scale the ideas to richer languages and richer logics, e.g., with higher-order frame rules [3, 2, 11].

In this paper we validate the frame rule without relying on the frame property of programs. Instead, we "bake-in" the frame rule into the interpretation of Hoare triples, using an idea from [3]. (This is described in detail in Section 4.) In particular, this approach allows us to model memory allocation by a simple deterministic allocator, so that we can model the programming language using ordinary recursively defined domains, avoiding the complications in [16]. Furthermore, the approach also allows us to validate a whole range of higher-order frame rules and to include pointer arithmetic.

The second challenge is to validate proof rules for recursion through the store [17]. Such rules essentially amount to having recursively defined specifications, which denote recursive properties of the domain for commands. It is well-known that to establish the existence of such recursive properties of domains one needs additional conditions involving, in particular, admissibility and certain forms of downward closure [15]. In [16], these conditions were ensured

$e \in \text{Exp} ::= 0 \mid -1 \mid 1 \mid \ldots \mid e_1{+}e_2 \mid \ldots \mid x$		integer expressions, variable
$\mid$ 'C'		quote (command as expression)
$C \in \text{Com} ::= \mathsf{skip} \mid C_1;C_2 \mid \mathsf{if}\ (e_1{=}e_2)\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$		no op, sequencing, conditional
$\mid \mathsf{let}\ x{=}\mathsf{new}\ (e_1,\ldots,e_n)\ \mathsf{in}\ C \mid \mathsf{free}\ e$		allocation, disposal
$\mid [e_1]{:=}e_2 \mid \mathsf{let}\ y{=}[e]\ \mathsf{in}\ C \mid \mathsf{eval}\ [e]$		assignment, lookup, unquote

Fig. 1. Syntax of expressions and commands

by restricting the assertion language of the logic. In the present paper, we avoid such restrictions by changing the interpretation of triples and slightly modifying the recursion rules. In particular, the new interpretation uses an admissible and downwards closure of the post-condition, similar to the use of $\perp\!\!\perp$-closure in [3] (see Section 4 for more details).

2 Programs, assertions and specifications

Programs The abstract syntax of the programming language is presented in Fig. 1. It is essentially as in [16], with dynamic allocation (but here we assume a more realistic, deterministic memory allocator) and storable, parameterless procedures. The language is deliberately kept simple so that we can study higher-order store without distraction. We point out two features of the language which proved problematic for the semantics given in *loc. cit.* First, the language assumes that addresses are natural numbers, so that it is possible to apply arithmetic operations on addresses. Next, the language includes an allocator that deterministically picks n-consecutive cells. Specifically, it contains the command $\mathsf{let}\ x{=}\mathsf{new}\ (e_1,\ldots,e_n)\ \mathsf{in}\ C$ that works by allocating the first n free cells in the heap, initializing these cells with the values of $e_1,\ldots,e_n$, then binding x to the address of the first cell, and finally running the continuation C.

Assertions The assertions used in Hoare triples are built from the formulas of classical predicate logic and the additional separation logic assertions that describe the heap ($\_ \mapsto \_$, **emp**, $P * Q$ and $P \mathrel{-\!\!*} Q$; cf. [18]). The syntax of assertions is standard, and is given in Fig. 2. We remind the reader that expressions in formulas, such as a variable x, can point to quoted code, so that they can be used to specify properties of stored procedures. We write $\Gamma \vdash P(:\mathsf{Assert})$ for some finite set of variables Γ, when the assertion P contains only free variables in Γ. We use two abbreviations:

$$e \mapsto \_ \stackrel{def}{=} \exists x'.\, e \mapsto x' \qquad e \mapsto e_1,..,e_n \stackrel{def}{=} e \mapsto e_1 * e{+}1 \mapsto e_2 * .. * e{+}n{-}1 \mapsto e_n.$$

where $x' \notin fv(e)$. The first abbreviation expresses a heap cell e without specifying its content, and the second describes a block of n-consecutive cells starting from e. We also use the usual connectives **true**, $\neg P$, $P \vee Q$, $P \wedge Q$, and $\exists x.P$ of classical predicate logic that can be defined from the connectives in Fig. 2.

$$P, Q \in \textsc{Assn} ::= e_1{=}e_2 \mid e_1 \leq e_2 \mid \mathbf{false} \mid P \to Q \mid \forall x.\, P \quad \text{classical logic connectives}$$
$$\mid\ e_1 \mapsto e_2 \mid \mathbf{emp} \mid P * Q \mid P \mathrel{-\!\!*} Q \quad \text{separation logic connectives}$$

$$\varphi, \psi \in \textsc{Spec} ::= e_1{=}e_2 \mid \{P\}C\{Q\} \mid \varphi \otimes P \mid \mathbf{T} \mid \mathbf{F} \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \Rightarrow \psi \mid \exists x.\varphi \mid \forall x.\varphi$$

Fig. 2. Syntax of assertions and specifications

Proof Rules for Stored Code

$$\big((\forall \vec{y}.\, \{P\}\mathsf{eval}\ [e]\{Q\}) \ \Rightarrow\ \forall \vec{y}.\, \{P\}C\{Q\}\big) \ \Rightarrow\ \forall \vec{y}.\, \{P * e \mapsto \text{`}C\text{'}\}\mathsf{eval}\ [e]\{Q * e \mapsto \text{`}C\text{'}\} \quad (\text{where } \vec{y} \notin fv(e, C))$$

$$(\forall x.\ (\forall \vec{y}.\, \{P * e \mapsto x\}\mathsf{eval}\ [e]\{Q * e \mapsto x\}) \ \Rightarrow\ \forall \vec{y}.\, \{P * e \mapsto x\}C\{Q * e \mapsto x\}) \Rightarrow\ \forall \vec{y}.\, \{P * e \mapsto \text{`}C\text{'}\}\mathsf{eval}\ [e]\{Q * e \mapsto \text{`}C\text{'}\} \ (\text{where } x \notin fv(P, Q, \vec{y}, e, C), \vec{y} \notin fv(e, C))$$

$$(\forall x.\ (\forall \vec{y}.\, \{P * e \mapsto x\}\mathsf{eval}\ [e]\{Q\}) \ \Rightarrow\ \forall \vec{y}.\, \{P * e \mapsto x\}C\{Q\}) \Rightarrow\ \forall \vec{y}.\, \{P * e \mapsto \text{`}C\text{'}\}\mathsf{eval}\ [e]\{Q\} \quad (\text{where } x \notin fv(P, Q, \vec{y}, e, C), \vec{y} \notin fv(e, C))$$

Proof Rules for Hoare Triples

$$(\forall x.\, \{P * x \mapsto e\}C\{Q\}) \ \Rightarrow\ \{P\}\mathsf{let}\ x = \mathsf{new}\, e\ \mathsf{in}\ C\{Q\} \qquad (\text{where } x \notin fv(P, Q, e))$$
$$(\forall x.\, \{P * e \mapsto x\}C\{Q\}) \ \Rightarrow\ \{\exists x.\, P * e \mapsto x\}\mathsf{let}\ x = [e]\ \mathsf{in}\ C\{Q\} \quad (\text{where } x \notin fv(Q, e))$$
$$\{e \mapsto \_\}\mathsf{free}(e)\{\mathbf{emp}\} \qquad \{e \mapsto \_\}[e] := e'\{e \mapsto e'\}$$
$$\frac{[\![P]\!]^{\mathcal{A}}_{\eta} \subseteq [\![P']\!]^{\mathcal{A}}_{\eta} \text{ and } [\![Q']\!]^{\mathcal{A}}_{\eta} \subseteq [\![Q]\!]^{\mathcal{A}}_{\eta} \text{ for all } \eta \in [\![\Gamma]\!]}{\Gamma \ \vdash\ \{P'\}C\{Q'\} \Rightarrow \{P\}C\{Q\}}$$

Proof Rules for Invariant Extension $- \otimes P$

$$\varphi \ \Rightarrow\ \varphi \otimes P \qquad \{P\}C\{P'\} \otimes Q \ \Leftrightarrow\ \{P * Q\}C\{P' * Q\}$$
$$(e_0 = e_1) \otimes Q \ \Leftrightarrow\ e_0 = e_1 \qquad (\varphi \otimes P) \otimes Q \ \Leftrightarrow\ \varphi \otimes (P * Q)$$
$$(\varphi \oplus \psi) \otimes P \ \Leftrightarrow\ (\varphi \otimes P) \oplus (\psi \otimes P) \qquad (\kappa x.\, \varphi) \otimes P \ \Leftrightarrow\ \kappa x.\, \varphi \otimes P$$
$$(\text{where } \oplus \in \{\Rightarrow, \wedge, \vee\}) \qquad (\text{where } \kappa \in \{\forall, \exists\}, x \notin fv(P))$$

Fig. 3. Some proof rules

Specifications Specifications are formulas of first-order intuitionistic logic with equality. In addition, it includes Hoare triples as atomic formulas and invariant extensions (from [3]). While assertions express properties of states, specifications describe properties of programs (sometimes using assertions inside Hoare triples). The syntax of specifications is given in Fig. 2. We write $\Gamma \vdash \varphi(: \mathsf{Spec})$, for a finite set Γ of variables, to mean that Γ includes all the free variables in φ.

Proof rules Our specification logic includes all the usual proof rules of intuitionistic first-order logic with equality, and special rules for Hoare triples and invariant extension $\varphi \otimes P$. Fig. 3 lists some of those, where the context Γ for each specification is omitted. Note that the consequence rule uses semantically valid implications for *assertions*, some of which can be proved using the proof rules from classical logic and the logic of Bunched Implications. In this way, the consequence rule embeds reasoning about assertions into the specification logic.

Most of the rules in the figure are standard and known from separation logic. The only exceptions are the three proof rules for stored procedures.[1] The first of these three rules deals with the most conservative case where stored procedures do not access storing cells at all, except by calling eval $[e]$. The second rule considers stored procedures that do not change the storing cells, and the third rule allows stored procedures to modify storing cells.

Note that the assumption of the first rule for stored procedures hides the cell e completely, even when eval $[e]$ dereferences the cell. This hiding of e ensures that the code in the cell can access its storing cell e only by calling eval $[e]$ (thus allowing the procedure to call itself for evaluation), but not by any direct pointer-dereference operations. Therefore, the value of cell e does not change, as expressed by the postcondition of the concluding triple of the rule. The second rule allows direct accesses to cell e, but requires the code to be identical at procedure entry and exit. In-between, the procedure in the cell may be changed.

In contrast, the third rule for stored procedures does not even insist on the code x being restored in e at procedure exit, which allows the procedure to be changed by itself so that it has a different behaviour when called subsequently. Updating code after its first call is a general pattern of usage of stored code. This can be found e.g. in device drivers [5]. The first call is used for initialisation that further, repeated, calls rely on.

Example 1 (Factorial). Consider the following specification and implementation of the factorial function:

$$\begin{aligned} F_o &\stackrel{def}{=} \mathsf{let}\ x{=}[o]\ \mathsf{in}\ \mathsf{let}\ r{=}[o{+}1]\ \mathsf{in} \\ &\quad\ \mathsf{if}\ (x{=}0)\ \mathsf{then\ skip\ else}\ \big([o{+}1]{:=}r\cdot x;\ [o]{:=}x{-}1; \mathsf{eval}\ [o{+}2]\big) \\ C &\stackrel{def}{=} [o{+}2]{:=}`F_o\text{'}; \mathsf{eval}\ [o{+}2] \qquad\qquad o\ \vdash\ \{o \mapsto 5, 1, \_\}C\{o \mapsto 0, 5!, `F_o\text{'}\} \end{aligned}$$

The command C implements a factorial object using three consecutive cells $(o, o{+}1, o{+}2)$. The first two cells represent fields arg and res, and the third cell denotes a method that computes the factorial of arg (decrementing it as a side effect) and multiplies this onto res. Note that the procedure F_o stored in $o{+}2$ calls itself by recursion through the store; see the last instruction eval $[o{+}2]$ of F_o. The specification expresses that C computes 5! and stores it in cell $o{+}1$.

The main part of the proof is the below derivation

$$\cfrac{\cfrac{\cdots}{o \vdash (\forall ij.\{o \mapsto i, j\}\mathsf{eval}\,[o{+}2]\{o \mapsto 0, j\cdot i!\}) \Rightarrow (\forall ij.\{o \mapsto i, j\}F_o\{o \mapsto 0, j\cdot i!\})}}{o \vdash \forall ij.\ \{o \mapsto i, j, `F_o\text{'}\}\mathsf{eval}\ [o{+}2]\{o \mapsto 0, j\cdot i!, `F_o\text{'}\}}$$

[1] For simplicity, we do not consider mutually recursive stored procedures in the paper, but it is straightforward to extend our rules to handle them. Also, the first rule for stored procedures can be derived from the second and the higher-order frame rules, but it is included in order to point out the subtleties of reasoning about stored procedures.

which shows the correctness of the stored procedure F_o. The last step of the derivation applies the first rule for stored procedures, and it illustrates the benefit of the rule. In this case, the rule lets us hide the cell $o{+}2$ for code F_o in the premise, thereby giving a simple specification to discharge. The derivation of this specification itself is omitted, since it involves only routine applications of standard separation logic proof rules. □

Example 2. Next, we illustrate the typical use of the three rules for stored procedures with program C_n's below:

$$F_1 \stackrel{def}{=} \mathsf{let}\ j{=}[i]\ \mathsf{in}\ \big(\mathsf{if}\ j{=}0\ \mathsf{then\ skip\ else}\ ([i]{:=}j{-}1; \mathsf{eval}\ [i{+}1])\big)$$
$$F_2 \stackrel{def}{=} \mathsf{let}\ j{=}[i]\ \mathsf{in\ let}\ f{=}[i{+}1]\ \mathsf{in}\ \big([i]{:=}f; \mathsf{if}\ j{=}0\ \mathsf{then}\ [i]{:=}0\ \mathsf{else}\ ([i]{:=}j{-}1; \mathsf{eval}\ [i{+}1])\big)$$
$$F_3 \stackrel{def}{=} \mathsf{let}\ j{=}[i]\ \mathsf{in}\ \big(\mathsf{if}\ j{=}0\ \mathsf{then}\ ([i{+}1]{:=}\text{‘}\mathsf{skip}\text{’})\ \mathsf{else}\ ([i]{:=}j{-}1;\ \mathsf{eval}\ [i{+}1])\big)$$

$$C_n \stackrel{def}{=} [i{+}1]{:=}F_n; \mathsf{eval}\ [i{+}1]$$

All of the C_n's decrease the value of i to zero (rather inefficiently), using recursion through the store. Additionally, C_2 dereferences cell $i{+}1$ to get the stored procedure F_2 and copy it to cell i temporarily. C_3 replaces the stored procedure in $i{+}1$ by skip at the end of the execution. For these programs, we want to prove the following specifications:

$$i \vdash \{i{\mapsto}\_,\_\}C_1\{i{\mapsto}0, \text{‘}F_1\text{’}\}\ \ i \vdash \{i{\mapsto}\_,\_\}C_2\{i{\mapsto}0, \text{‘}F_2\text{’}\}\ \ i \vdash \{i{\mapsto}\_,\_\}C_3\{i{\mapsto}0, \text{‘}\mathsf{skip}\text{’}\}.$$

The major step of the proof of C_1 is the application of the first rule for stored procedures:

$$\frac{i \vdash \{i \mapsto \_\}\mathsf{eval}\ [i{+}1]\{i \mapsto 0\} \Rightarrow \{i \mapsto \_\}F_1\{i \mapsto 0\}}{i \vdash \{i \mapsto \_, \text{‘}F_1\text{’}\}\mathsf{eval}\ [i{+}1]\{i \mapsto 0, \text{‘}F_1\text{’}\}}$$

which shows a property of the stored procedure F_1. Note that the first rule successfully hides cell $i{+}1$ in the premise, giving us a simple subgoal to discharge. Similarly, the application of rules for stored procedures form the major steps of the proofs of the remaining triples for C_2 and C_3. Since F_2 directly accesses the cell $i{+}1$, which stores the procedure itself, and F_3 updates the storing cell, we use the second rule for C_2 and the third for C_3:

$$\frac{i \vdash \forall x.\ \{i \mapsto \_, x\}\mathsf{eval}\ [i{+}1]\{i \mapsto 0, x\} \Rightarrow \{i \mapsto \_, x\}F_2\{i \mapsto 0, x\}}{i \vdash \{i \mapsto \_, \text{‘}F_2\text{’}\}\mathsf{eval}\ [i{+}1]\{i \mapsto 0, \text{‘}F_2\text{’}\}}$$

$$\frac{i \vdash \forall x.\ \{i \mapsto \_, x\}\mathsf{eval}\ [i{+}1]\{i \mapsto 0, \text{‘}\mathsf{skip}\text{’}\} \Rightarrow \{i \mapsto \_, x\}C_3\{i \mapsto 0, \text{‘}\mathsf{skip}\text{’}\}}{i \vdash \{i \mapsto \_, \text{‘}F_3\text{’}\}C_3\{i \mapsto 0, \text{‘}\mathsf{skip}\text{’}\}}$$

□

3 Semantics of programs and assertions

Our interpretation of the programming language is based on a solution of a recursive domain equation, which is defined in the category **Cppo** of directed complete pointed partial orders (in short, cppos) and strict continuous functions.

Let $Nats^+$ be the set of positive natural numbers, ranged over by ℓ and n, and for $n \in Nats^+$, write $[n]$ for the set $\{1, \ldots, n\}$. For a cppo A, we consider a cppo of $Nats^+$*-labelled records* with entries from A (roughly, a labelled smash product of arbitrary finite arity), which will be used to model *heaps*. The underlying set of this cppo is

$$Rec(A) = \big(\textstyle\sum_{N \subseteq_{fin} Nats^+} (N \to A_\downarrow)\big)_\perp, \tag{1}$$

where $(N \to A_\downarrow)$ denotes the cpo of maps from the finite address set N to the cpo $A_\downarrow = A - \{\perp\}$ of non-bottom elements of A. For $\perp \neq \iota_N(r) \in Rec(A)$ we write $\mathsf{dom}(r) = N$ and use record notation $\{\!|\ell_1 = a_1, \ldots, \ell_n = a_n|\!\}$ if $N = \{\ell_1, \ldots, \ell_n\}$ and $r(\ell_i) = a_i$ for all $i \in [n]$. Note that field selection is actually application if the label is in the domain of the record (for our semantic definitions this restricted form of field selection will be sufficient). We shall also write $r[\ell \mapsto a]$ for the record that maps ℓ to a and all other $\ell' \in \mathsf{dom}(r)$ to $r(\ell')$ (assuming $\ell' \in \mathsf{dom}(r)$). In case that r is $\perp$, we define $r[\ell \mapsto a]$ to be $\perp$. The ordering on (1) is given by

$$r \sqsubseteq r' \quad \stackrel{def}{\Leftrightarrow} \quad r \neq \perp \;\Rightarrow\; \big(\mathsf{dom}(r) = \mathsf{dom}(r') \;\wedge\; \forall \ell \in \mathsf{dom}(r).\, r(\ell) \sqsubseteq r'(\ell)\big).$$

The *disjointness predicate* $r\#r'$ on records holds if $r, r' \neq \perp$ and $\mathsf{dom}(r) \cap \mathsf{dom}(r') = \emptyset$, and a continuous *(partial) combining operation* $r \bullet r'$ is defined by

$$r \bullet r' \;\stackrel{def}{=}\; \text{if } (r\#r') \text{ then } (r \cup r') \text{ else } \big(\text{if } (r{=}\perp \vee r'{=}\perp) \text{ then } \perp \text{ else undefined}\big).$$

The semantics of the programming language is given by a solution for the following domain equation:

$$Val = Integers_\perp \oplus Com_\perp \qquad Heap = Rec(Val) \qquad Com = Heap \multimap T_{err}(Heap)$$

where $T_{err}(D) = D \oplus \{error\}_\perp$ is the error monad. We usually omit the tags and (for $h \in Heap$) will simply write $h \in T_{err}(Heap)$ and $error \in T_{err}(Heap)$, resp. Recall that a solution $i : F_{Com}(Com, Com) \cong Com$ can be obtained by the usual inverse limit construction, where F_{Com} is the evident locally continuous functor obtained by separating negative and positive occurrences of Com in the right-hand sides of the three equations above.[2] Moreover, such a solution is a *minimal invariant*, in the sense that $id_{Com} = lfp(\lambda e{:}Com{\multimap}Com.\; i \circ F_{Com}(e, e) \circ i^{-1})$ [15]. The soundness proof of the rules for stored procedures exploits this fact.

Interpretation of the programming language Fig. 4 gives the interpretation $[\![C]\!]_\eta$ of commands in $Heap \multimap T_{err}(Heap)$ (which is isomorphic to Com), where $\eta \in Env \stackrel{def}{=} (Var \to Val_\downarrow)$ is an environment mapping identifiers to (non-bottom) values in Val. An interpretation function for expressions $[\![e]\!]^{\mathcal{E}}_\eta \in Val_\downarrow$ is assumed, where the only non-standard cases are quoted commands. $[\![`C']\!]^{\mathcal{E}}_\eta$ is defined to be $i([\![C]\!]_\eta)$ (which implicitly makes use of the embedding of Com into Val). In the defining equations in Fig. 4 we assume that $h \neq \perp$, and set $[\![C]\!]_\eta \perp = \perp$ for all C and η. Note that the conditional only permits restricted comparison of expressions, so that commands denote continuous functions.

[2] Formally, $F_{Com}(X, Y)$ is $Rec(Integers_\perp \oplus X_\perp) \multimap T_{err}(Rec(Integers_\perp \oplus Y_\perp))$.

$$
\begin{aligned}
[\![\mathsf{skip}]\!]_\eta\, h &\stackrel{\mathit{def}}{=} h\\
[\![C_1;C_2]\!]_\eta\, h &\stackrel{\mathit{def}}{=} \text{if } [\![C_1]\!]_\eta\, h \in \{\bot, \mathit{error}\} \text{ then } [\![C_1]\!]_\eta \text{ else } [\![C_2]\!]_\eta\,([\![C_1]\!]_\eta\, h)\\
[\![\mathsf{if}\ e{=}e'\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2]\!]_\eta\, h &\stackrel{\mathit{def}}{=} \text{if } \{[\![e_1]\!]^{\mathcal{E}}_\eta, [\![e_2]\!]^{\mathcal{E}}_\eta\} \subseteq \mathit{Com} \text{ then } \bot\\
&\qquad \text{else if } ([\![e]\!]^{\mathcal{E}}_\eta = [\![e']\!]^{\mathcal{E}}_\eta) \text{ then } [\![C_1]\!]_\eta\, h \text{ else } [\![C_2]\!]_\eta\, h\\
[\![\mathsf{let}\ x{=}\mathsf{new}\ e_1,\ldots,e_n\ \mathsf{in}\ C]\!]_\eta\, h &\stackrel{\mathit{def}}{=} \text{let } \ell = \min\{\ell \mid \forall \ell'.\,(\ell{\le}\ell'{<}\ell{+}n) \Rightarrow \ell' \notin \mathsf{dom}(h)\}\\
&\qquad \text{in } [\![C]\!]_{\eta[x\mapsto\ell]}\,(h \bullet \{\!|\ell{=}[\![e_1]\!]^{\mathcal{E}}_\eta, \ldots, \ell{+}n{-}1{=}[\![e_n]\!]^{\mathcal{E}}_\eta|\!\})\\
[\![\mathsf{free}\ e]\!]_\eta\, h &\stackrel{\mathit{def}}{=} \text{if } [\![e]\!]^{\mathcal{E}}_\eta \notin \mathsf{dom}(h) \text{ then } \mathit{error}\\
&\qquad \text{else (let } h' \text{ s.t. } h = h' \bullet \{\!|[\![e]\!]^{\mathcal{E}}_\eta{=}h([\![e]\!]^{\mathcal{E}}_\eta)|\!\} \text{ in } h')\\
[\![[e_1]{:=}e_2]\!]_\eta\, h &\stackrel{\mathit{def}}{=} \text{if } [\![e_1]\!]^{\mathcal{E}}_\eta \notin \mathsf{dom}(h) \text{ then } \mathit{error} \text{ else } (h[[\![e_1]\!]^{\mathcal{E}}_\eta \mapsto [\![e_2]\!]^{\mathcal{E}}_\eta])\\
[\![\mathsf{let}\ x{=}[e]\ \mathsf{in}\ C]\!]_\eta\, h &\stackrel{\mathit{def}}{=} \text{if } [\![e]\!]^{\mathcal{E}}_\eta \notin \mathsf{dom}(h) \text{ then } \mathit{error} \text{ else } [\![C]\!]_{\eta[x\mapsto h([\![e]\!]^{\mathcal{E}}_\eta)]}\, h\\
[\![\mathsf{eval}\ [e]]\!]_\eta\, h &\stackrel{\mathit{def}}{=} \text{if } ([\![e]\!]^{\mathcal{E}}_\eta \notin \mathsf{dom}(h) \vee h([\![e]\!]^{\mathcal{E}}_\eta) \notin \mathit{Com}) \text{ then } \mathit{error}\\
&\qquad \text{else } i^{-1}(h([\![e]\!]^{\mathcal{E}}_\eta))(h)
\end{aligned}
$$

Fig. 4. Interpretation of commands $[\![C]\!]_\eta \in \mathit{Heap} \multimap T_{err}(\mathit{Heap})$

$$
\begin{aligned}
&[\![e_1 \le e_2]\!]^{\mathcal{A}}_\eta \stackrel{\mathit{def}}{=} \{h \in \mathit{Heap} \mid h \ne \bot \Rightarrow [\![e_i]\!]^{\mathcal{E}}_\eta \in \mathit{Integers} \wedge [\![e_1]\!]^{\mathcal{E}}_\eta \le [\![e_2]\!]^{\mathcal{E}}_\eta\}\\
&[\![e_1 = e_2]\!]^{\mathcal{A}}_\eta \stackrel{\mathit{def}}{=} \{h \in \mathit{Heap} \mid h \ne \bot \Rightarrow [\![e_1]\!]^{\mathcal{E}}_\eta = [\![e_2]\!]^{\mathcal{E}}_\eta\}\\
&[\![\forall x.\, P]\!]^{\mathcal{A}}_\eta \stackrel{\mathit{def}}{=} \bigcap \{[\![P]\!]^{\mathcal{A}}_{\eta[x\mapsto v]} \mid v{\in}\mathit{Val}\} \qquad [\![P \to Q]\!]^{\mathcal{A}}_\eta \stackrel{\mathit{def}}{=} \{h \in \mathit{Heap} \mid h{\in}[\![P]\!]^{\mathcal{A}}_\eta \Rightarrow h{\in}[\![Q]\!]^{\mathcal{A}}_\eta\}\\
&[\![\mathbf{false}]\!]^{\mathcal{A}}_\eta \stackrel{\mathit{def}}{=} \{\bot\} \qquad [\![\mathbf{emp}]\!]^{\mathcal{A}}_\eta \stackrel{\mathit{def}}{=} \{\{\!||\!\}, \bot\} \qquad [\![P * Q]\!]^{\mathcal{A}}_\eta \stackrel{\mathit{def}}{=} [\![P]\!]^{\mathcal{A}}_\eta * [\![Q]\!]^{\mathcal{A}}_\eta\\
&[\![P \mathrel{-\!*} Q]\!]^{\mathcal{A}}_\eta \stackrel{\mathit{def}}{=} \{h \in \mathit{Heap} \mid \forall h' \in [\![P]\!]^{\mathcal{A}}_\eta.\ h \bullet h' \text{ is defined} \Rightarrow h \bullet h' \in [\![Q]\!]^{\mathcal{A}}_\eta\}\\
&[\![e \mapsto e']\!]^{\mathcal{A}}_\eta \stackrel{\mathit{def}}{=} \{h \in \mathit{Heap} \mid h \ne \bot \Rightarrow \mathsf{dom}(h) = \{[\![e]\!]^{\mathcal{E}}_\eta\} \wedge h([\![e]\!]^{\mathcal{E}}_\eta) = [\![e']\!]^{\mathcal{E}}_\eta\}
\end{aligned}
$$

Fig. 5. Interpretation $[\![P]\!]^{\mathcal{A}} : \mathit{Env} \to \mathcal{P}$ of assertions

Interpretation of assertions Let $\mathcal{P}$ be the set of predicates $p \subseteq \mathit{Heap}$ that contain $\bot$. The separating conjunction for these predicates, known from separation logic [18], is defined by: $h \in p_1 * p_2 \stackrel{\mathit{def}}{\Leftrightarrow} \exists h_1, h_2.\ h = h_1 \bullet h_2 \wedge h_1 \in p_1 \wedge h_2 \in p_2$. Note that $p * q \in \mathcal{P}$ whenever $p \in \mathcal{P}$ and $q \in \mathcal{P}$. Clearly '$*$' is associative and commutative, since '$\bullet$' is, and if $p \subseteq p'$ and $q \subseteq q'$ then $p * q \subseteq p' * q'$.

The poset $(\mathcal{P}, \subseteq)$ forms a complete boolean BI algebra.[3] Thus, we get a canonical BI hyperdoctrine $\mathbf{Set}(-, \mathcal{P})$, which soundly models classical (higher-order) predicate BI [1]. In particular this yields an interpretation for the quantifiers of our assertion language. This interpretation of assertions is spelled out explicitly in Fig. 5.

[3] The negation and false of this boolean algebra are slightly unusual, and they are defined by $\neg p \stackrel{\mathit{def}}{=} (\mathit{Heap} - p) \cup \{\bot\}$ and $\mathit{false} \stackrel{\mathit{def}}{=} \{\bot\}$. Conjunction, disjunction and true are defined as in the usual powerset boolean algebra.

4 Semantics of specifications

We now define the interpretation of specifications, and show how it addresses the two key challenges described in the introduction.

The most interesting components of our interpretation are semantic Hoare triples, which we will use to interpret (syntactic) Hoare triples. For each predicate $p \in \mathcal{P}$, let $\mathrm{Ad}(p)$ be the admissible, downward closure of p in $T_{err}(\mathit{Heap})$ (i.e., the smallest admissible, downward-closed subset of $T_{err}(\mathit{Heap})$ that includes p, which may be obtained as the intersection of all admissible, downward-closed subsets of *Heap* that include p).

Definition 1 (Semantic triple). *A* semantic Hoare triple *is a triple of predicates $p, q \in \mathcal{P}$ and function $c \in F_{Com}(Com, Com)$, written $\{p\}c\{q\}$. A semantic triple $\{p\}c\{q\}$ is* valid*, denoted $\models \{p\}c\{q\}$, if and only if, for all $r \in \mathcal{P}$ and all $h \in \mathit{Heap}$, we have that $h \in p * r \;\Rightarrow\; c(h) \in \mathrm{Ad}(q * r)$.*

Intuitively, a semantic triple $\{p\}c\{q\}$ specifies that c should transform an input state in p to an output state in q. Furthermore, the triple says that this transformation should modify only the portion of memory for p (because, otherwise, it would not preserve some invariant r when r was $*$-attached to the precondition p). Note that $\models \{p\}c\{q\}$ ensures the absence of memory errors for inputs in $p * r$ for all r, because $\mathrm{Ad}(q * r)$ cannot contain *error*.

We point out two important aspects of valid semantic Hoare triples and their relationships to the points raised in the introduction. First, the definition of validity includes a universal quantification over $*$-added invariants r. Since we will interpret (syntactic) Hoare triples using the validity of semantic triples, this universal quantification means that Hoare triples in our logic impose a stronger requirement on commands than the ones in standard separation logic. In particular, the requirement is strong enough to imply the frame rule, as indicated by the lemma below:

Lemma 1. *If $\models \{p\}c\{q\}$, then $\models \{p * r\}c\{q * r\}$ for all $r \in \mathcal{P}$.*

In this way, our model addresses the first challenge in the introduction regarding the soundness of the frame rule. Second, the definition of the validity takes the admissible, downward closure $\mathrm{Ad}(q * r)$ of post-conditions. As a result, whenever we define a subset of $F_{Com}(Com, Com)$ using a semantic Hoare triple, it is guaranteed that the resulting set is admissible and downward-closed:

Lemma 2. *For all $p, q \in \mathcal{P}$, the subset $\{c \mid \{p\}c\{q\} \text{ is valid}\}$ is an admissible, downward-closed subset of $F_{Com}(Com, Com)$.*

It is this property that lets us prove the soundness of the proof rules for stored procedures, without requiring any additional conditions, such as a syntactic restriction on assertions [16].

We interpret specifications following the usual Kripke semantics of intuitionstic logic. Our interpretation uses a particular Kripke structure that lets us validate all the higher-order frame rules, i.e., rules for invariant extension $\varphi \otimes P$. Concretely, the Kripke structure is the preorder $(\mathcal{P}, \sqsubseteq)$ where the relation $\sqsubseteq$ is

$\eta, p \models \mathbf{T}$	always	$\eta, p \models \varphi \wedge \psi$	$\overset{def}{\Leftrightarrow} \eta, p \models \varphi$ and $\eta, p \models \psi$
$\eta, p \models \mathbf{F}$	never	$\eta, p \models \varphi \vee \psi$	$\overset{def}{\Leftrightarrow} \eta, p \models \varphi$ or $\eta, p \models \psi$
$\eta, p \models e_1 = e_2$	$\overset{def}{\Leftrightarrow} [\![e_1]\!]^{\mathcal{E}}_\eta = [\![e_2]\!]^{\mathcal{E}}_\eta$	$\eta, p \models \varphi \otimes P$	$\overset{def}{\Leftrightarrow} \eta, p * [\![P]\!]^{\mathcal{A}}_\eta \models \varphi$

$\eta, p \models \{P\}C\{Q\} \overset{def}{\Leftrightarrow} \models \{[\![P]\!]^{\mathcal{A}}_\eta * p\}\, [\![C]\!]_\eta \, \{[\![Q]\!]^{\mathcal{A}}_\eta * p\}$
$\eta, p \models \varphi \Rightarrow \psi \quad \overset{def}{\Leftrightarrow}$ for all $r \in \mathcal{P}$, if $p \sqsubseteq r$ and $\eta, r \models \varphi$, then $\eta, r \models \psi$
$\eta, p \models \exists x.\, \varphi \quad \overset{def}{\Leftrightarrow}$ for some $v \in \mathit{Val}$, $(\eta[x \mapsto v], p \models \varphi)$
$\eta, p \models \forall x.\, \varphi \quad \overset{def}{\Leftrightarrow}$ for all $v \in \mathit{Val}$, $(\eta[x \mapsto v], p \models \varphi)$

Fig. 6. Interpretation $\eta, p \models \varphi$ of specifications

defined by: $p \sqsubseteq q \overset{def}{\Leftrightarrow} \exists r \in \mathcal{P}.\ p * r = q$. Each world p in this Kripke structure should be thought of as an invariant to be added by (higher-order) frame rules, and the preorder $p \sqsubseteq q$ denotes that q is obtained by extending p with some disjoint invariant r. This Kripke structure has been studied in [3], and we will use the results from that paper.

The semantics of specifications is given by the satisfaction relation $\models$ shown in Fig. 6. Note that Hoare triples are interpreted using the validity of semantic triples.

Soundness We recall one consequence of our semantics, which is discussed in more detail in [3]. It is the soundness of the generalized frame rule: $\varphi \Rightarrow \varphi \otimes P$. Since the interpretation follows the standard Kripke semantics, every formula φ satisfies the usual Kripke monotonicity: $\forall \eta, r, q.\, (\eta, r \models \varphi) \wedge (r \sqsubseteq q) \Rightarrow (\eta, q \models \varphi)$. Since $r \sqsubseteq q$ just means that $q = r * p$ for some p, the above monotonicity condition is equivalent to $\forall \eta, r, p.\, (\eta, r \models \varphi) \Rightarrow (\eta, r * p \models \varphi)$. This just means that adding an invariant p for each specification maintains the truth of a specification, and explains why the generalized frame rule is sound in our semantics.

Lemma 3 (Invariants, [3]). *All the axioms for invariant extensions are sound.*

Our semantics validates all the proof rules for specifications. In the following, we focus on the second rule for stored procedures. Proofs for the remaining rules can be found in the appendix.

Lemma 4 (Rec 2). *The second rule for stored procedures is sound.*

Proof. For each $\eta \in [\![\Gamma]\!]$ and $r \in \mathcal{P}$, define a predicate $A_{\eta,r}$ on $\mathit{Com} \times \mathit{Com}$ by

$$A_{\eta,r}(c,d) \overset{def}{\Leftrightarrow} \forall \vec{v} \in \mathit{Val}^n.\ \models \{[\![P * e \mapsto x]\!]^{\mathcal{A}}_{\eta_1} * r\} i^{-1}(d) \{[\![Q * e \mapsto x]\!]^{\mathcal{A}}_{\eta_1} * r\}$$

where $\eta_1 = \eta[\vec{y} \mapsto \vec{v}, x \mapsto c]$. Pick any $\eta \in [\![\Gamma]\!]$ and $r \in \mathcal{P}$. By the definition of $[\![\mathsf{eval}\ [e]]\!]$ and the usual substitution lemma (which holds for our interpretation), the soundness of the rule boils down to proving the following implication.

$$\big(\forall c \in \mathit{Com}. \forall r' \sqsupseteq r.\ A_{\eta,r'}(c,c) \Rightarrow A_{\eta,r'}(c, [\![`C']\!]_\eta)\big) \Rightarrow A_{\eta,r}([\![`C']\!]_\eta, [\![`C']\!]_\eta).$$

Suppose that there is a predicate $S_{\eta',r'}$ on *Com* parameterized by (η', r'), such that (1) $S_{\eta',r'}(c) \;\Leftrightarrow\; (\forall d{\in}Com.\; S_{\eta',r'}(d) \Rightarrow A_{\eta',r'}(d,c))$. Then, we have that (2) $\forall c.\, S_{\eta,r}(c) \Rightarrow A_{\eta,r}(c,c)$. Hence, assuming the precondition $\forall c. \forall r' \sqsupseteq r.\, A_{\eta,r'}(c,c){\Rightarrow}A_{\eta,r'}(c, [\![`C']\!]_\eta)$, we obtain $\forall c.\; S_{\eta,r}(c) \;\Rightarrow\; A_{\eta,r}(c, [\![`C']\!]_\eta)$ and therefore $S_{\eta,r}([\![`C']\!]_\eta)$ by (1). But then (2) shows $A_{\eta,r}([\![`C']\!]_\eta, [\![`C']\!]_\eta)$, as required. It remains to establish the existence of a predicate $S_{\eta',r'}$ satisfying (1). This is done in the following Lemma 5. ◻

Lemma 5 (Existence). *For all η, r, there exists $S_{\eta,r} \subseteq Com$ such that $S_{\eta,r}(c)$ holds iff $\forall d.\; S_{\eta,r}(d) \;\Rightarrow\; A_{\eta,r}(d,c)$, where $A_{\eta,r}$ is as in the proof of Lemma 4.*

Proof. The proof builds on the same technique as used in [17], but a lot of details have changed. Let $\mathcal{C}$ denote the set of admissible subsets of *Com*, which forms a complete lattice when ordered by $\subseteq$. Pick η and $r \in \mathcal{P}$. We define an operation $\Phi\colon \mathcal{C}^{op} \to \mathcal{C}$, by $S \;\mapsto\; \{c \in Com \mid \forall d.\, d \in S \Rightarrow A_{\eta,r}(d,c)\}$. That $\Phi(S)$ is admissible follows from the admissibility of $A_{\eta,r}(d,-)$, which itself comes from Lemma 2. The symmetrisation $\Phi^{\S}(S,T) \stackrel{def}{=} \langle \Phi(T), \Phi(S)\rangle$ of Φ is a monotonic map on the complete lattice $\mathcal{C}^{op} \times \mathcal{C}$ and thus has a least (pre-) fixed point (S^-, S^+), by Tarski's fixed point theorem. Then (S^+, S^-) is also a fixed point of $\Phi^{\S}$, so one obtains $S^+ \subseteq S^-$. A predicate $S_{\eta,r} \in \mathcal{C}$ with the required property $S_{\eta,r} = \Phi(S_{\eta,r})$ is obtained by proving the opposite inclusion.

To this end, for $l \sqsubseteq id_{Com}$ and $S_1, S_2 \in \mathcal{C}$, define $l : S_1 \subset S_2$ to mean that $\forall c \in S_1.\; l(c) \in S_2$. Note that from

$$l : S_1 \subset S_2 \Rightarrow (i \circ F_{Com}(l,l) \circ i^{-1}) : \Phi(S_2) \subset \Phi(S_1) \tag{1}$$

for all $l \sqsubseteq id_{Com}$, it follows by fixed point induction that $lfp(\lambda l.\, i \circ F_{Com}(l,l) \circ i^{-1}) : S^- \subset S^+$. This is equivalent to $id_{Com} : S^- \subset S^+$, i.e., $S^- \subseteq S^+$, because $lfp(...)$ is id_{Com} by the minimal invariant property of *Com*.

It remains to prove (1). For this, we will prove the following two properties. Let $\mathrm{Cl}^{\downarrow}(p)$ be the downward closure of a predicate p. For all environments η', heaps h and functions l with $l \sqsubseteq id_{Com}$, if $j \stackrel{def}{=} Rec(\hat{l})$,

1. $h \in [\![P * e \mapsto x]\!]^{\mathcal{A}}_{\eta'}$ implies $j(h) \in \mathrm{Cl}^{\downarrow}[\![P * e \mapsto x]\!]^{\mathcal{A}}_{\eta'[x \mapsto l(\eta'(x))]}$,
2. $h \in [\![Q * e \mapsto x]\!]^{\mathcal{A}}_{\eta'[x \mapsto l(\eta'(x))]}$ implies $j(h) \in \mathrm{Cl}^{\downarrow}[\![Q * e \mapsto x]\!]^{\mathcal{A}}_{\eta'}$.

To see why it is sufficient to prove those two properties, suppose $l \sqsubseteq id_{Com}$ satisfies $l : S_1 \subset S_2$. Pick $c \in \Phi(S_2)$. We have to show $(i \circ F_{Com}(l,l) \circ i^{-1})(c) \in \Phi(S_1)$. Thus, for all $d \in S_1$, we must show that $A_{\eta,r}(d, (i \circ F_{Com}(l,l) \circ i^{-1})(c))$ holds, i.e., for all $\vec{v} \in Val^n$

$$\models \{[\![P * e \mapsto x]\!]^{\mathcal{A}}_{\eta[\vec{y} \mapsto \vec{v}, x \mapsto d]} * r\} F_{Com}(l,l)(i^{-1}(c)) \{[\![Q * e \mapsto x]\!]^{\mathcal{A}}_{\eta[\vec{y} \mapsto \vec{v}, x \mapsto d]} * r\}. \tag{2}$$

For this, pick $d \in S_1$ and $\vec{v} \in Val^n$. Since $l : S_1 \subset S_2$, we have that $l(d) \in S_2$, and since $c \in \Phi(S_2)$, it must be the case that

$$\models \{[\![P * e \mapsto x]\!]^{\mathcal{A}}_{\eta[\vec{y} \mapsto \vec{v}, x \mapsto l(d)]} * r\} i^{-1}(c) \{[\![Q * e \mapsto x]\!]^{\mathcal{A}}_{\eta[\vec{y} \mapsto \vec{v}, x \mapsto l(d)]} * r\}. \tag{3}$$

We will now prove that (3) implies (2).

To simplify notation, we assume without loss of generality that η is such that $\eta(\vec{y}) = \vec{v}$. Pick $r' \in \mathcal{P}$ and $h \in [\![P * e \mapsto x]\!]^{\mathcal{A}}_{\eta[x \mapsto d]} * r * r'$. Let j be $Rec(\hat{l})$. Then, we have to show the set membership below:

$$F_{Com}(l,l)(i^{-1}(c))(h) \;=\; T_{err}(j)(i^{-1}(c)(j(h))) \;\in\; \mathrm{Ad}([\![Q * e \mapsto x]\!]^{\mathcal{A}}_{\eta[x \mapsto d]} * r * r').$$

By property 1 and definition of j, $j(h)$ is in $\mathrm{Cl}^{\downarrow}\big([\![P * e \mapsto x]\!]^{\mathcal{A}}_{\eta[x \mapsto l(d)]} * r * r'\big)$. So, we have (4) $i^{-1}(c)(j(h)) \in \mathrm{Ad}([\![Q * e \mapsto x]\!]^{\mathcal{A}}_{\eta[x \mapsto l(d)]} * r * r')$, because of (3) and the monotonicity of $i^{-1}(c)$. Note that by the property 2 and the definition of j, $T_{err}(j)$ should map heaps in $([\![Q * e \mapsto x]\!]^{\mathcal{A}}_{\eta[x \mapsto l(d)]} * r * r')$ to those in $\mathrm{Ad}([\![Q * e \mapsto x]\!]^{\mathcal{A}}_{\eta[x \mapsto d]} * r * r')$. Furthermore, for all continuous functions f on $T_{err}(Heap)$, if f maps every heap in a predicate p into $\mathrm{Ad}(q)$, it also maps all heaps in $\mathrm{Ad}(p)$ into $\mathrm{Ad}(q)$. Thus, since $T_{err}(j)$ is continuous, it maps heaps in $\mathrm{Ad}([\![Q * e \mapsto x]\!]^{\mathcal{A}}_{\eta[x \mapsto l(d)]} * r * r')$ into $\mathrm{Ad}([\![Q * e \mapsto x]\!]^{\mathcal{A}}_{\eta[x \mapsto d]} * r * r')$. By (4), this means that $T_{err}(j)(i^{-1}(c)(j(h)))$ belongs to $\mathrm{Ad}([\![Q * e \mapsto x]\!]^{\mathcal{A}}_{\eta[x \mapsto d]} * r * r')$, as desired.

Finally, we prove properties 1 and 2. To prove the first property, suppose that $h \in [\![P * e \mapsto x]\!]^{\mathcal{A}}_{\eta}$ and h is not $\bot$. (If h is $\bot$, it belongs to any predicate in $\mathcal{P}$, and gives the required conclusion.) Then, h can be split into h_0 and h_1 in $Heap$, such that $h_0 \in [\![P]\!]^{\mathcal{A}}_{\eta}$, $\mathsf{dom}(h_1) = \{[\![e]\!]^{\mathcal{E}}_{\eta}\}$, and $h_1([\![e]\!]^{\mathcal{E}}_{\eta}) = \eta(x)$. From this we obtain (5) $j(h_0) \in \mathrm{Cl}^{\downarrow}[\![P]\!]^{\mathcal{A}}_{\eta[x \mapsto l(\eta(x))]}$, because P does not contain x, and (6) $\mathsf{dom}(j(h_1)) = \{[\![e]\!]^{\mathcal{E}}_{\eta}\} \;\wedge\; j(h_1)([\![e]\!]^{\mathcal{E}}_{\eta}) = l(\eta(x))$, which means that $j(h_1) \in [\![e \mapsto x]\!]^{\mathcal{A}}_{\eta[x \mapsto l(\eta(x))]}$. (5) and (6) imply that $j(h)$ belongs to $\mathrm{Cl}^{\downarrow}[\![P * e \mapsto x]\!]^{\mathcal{A}}_{\eta[x \mapsto l(\eta(x))]}$.

For the second property, let $h \in [\![Q * e \mapsto x]\!]^{\mathcal{A}}_{\eta[x \mapsto l(\eta(x))]}$ such that h is not $\bot$. Then, h can be partitioned into heaps $h_0 \bullet h_1 = h$ such that $h_0 \in [\![Q]\!]^{\mathcal{A}}_{\eta[x \mapsto l(\eta(x))]}$ and $\mathsf{dom}(h_1) = \{[\![e]\!]^{\mathcal{E}}_{\eta}\} \wedge h_1([\![e]\!]^{\mathcal{E}}_{\eta}) = l(\eta(x))$. But $[\![Q]\!]^{\mathcal{A}}_{\eta[x \mapsto l(\eta(x))]} = [\![Q]\!]^{\mathcal{A}}$ because Q does not contain x. Furthermore, $h_1([\![e]\!]^{\mathcal{E}}_{\eta}) = l(\eta(x))$ and $j(h_1) \sqsubseteq h_1$ imply $j(h_1)([\![e]\!]^{\mathcal{E}}_{\eta}) \sqsubseteq \eta(x)$. Thus, $j(h_0) \in \mathrm{Cl}^{\downarrow}[\![Q]\!]^{\mathcal{A}}_{\eta}$ and $j(h_1) \in \mathrm{Cl}^{\downarrow}[\![e \mapsto x]\!]^{\mathcal{A}}_{\eta}$. This implies $j(h) \in \mathrm{Cl}^{\downarrow}[\![Q * e \mapsto x]\!]^{\mathcal{A}}_{\eta}$, as required. □

5 Conclusion and future work

We have developed a simple model of separation logic for a language with higher-order store. The model validates proof rules for recursion through the store and a wide range of higher-order frame rules. Future work includes extending the model to richer programming languages, in particular to languages with higher-order functions. In order to obtain modularity it is also necessary to develop a version of the logic where assertions do not contain code explicitly but rather abstract specifications of its behaviour. We are confident that the simplicity of the present model will make that possible. In future work we also plan to extend the relationally parametric model of separation logic in [3] to higher-order store.

References

1. B. Biering, L. Birkedal, and N. Torp-Smith. BI-hyperdoctrines, higher-order separation logic, and abstraction. *ACM Transactions on Programming Languages and Systems*, 29(5), 2007.
2. L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules for algol-like languages. *Logical Methods in Computer Science*, 2(5:1), Aug. 2006.
3. L. Birkedal and H. Yang. Relational parametricity and separation logic. In *Proc. FOSSACS'07*, volume 4423 of *LNCS*, 2007.
4. H. Cai, Z. Shao, and A. Vaynberg. Certified self-modifying code. In *Proc. PLDI'07*, pages 66–77, 2007.
5. J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers.* O'Reilly, 3rd edition, 2005.
6. K. Honda, N. Yoshida, and M. Berger. An observationally complete program logic for imperative higher-order functions. In *Proc. LICS'05*, pages 270–279, 2005.
7. N. Krishnaswami, J. Aldrich, and L. Birkedal. Modular verification of the subject-observer pattern via higher-order separation logic. In *FTfJP'07*, 2007.
8. A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract predicates and mutable ADTs in Hoare type theory. In *Proc. ESOP'07*, volume 4421 of *LNCS*, pages 189–204, 2007.
9. A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare type theory. In *Proc. ICFP'06*, pages 62–73, 2006.
10. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proc. CSL'01*, volume 2142 of *LNCS*, pages 1–18, 2001.
11. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Proc. of 31st POPL*, pages 268–280, 2004.
12. M. Parkinson. When separation logic met java. In *FTfJP'06*, 2006.
13. M. Parkinson and G. Biermann. Separation logic and abstraction. In *Proc. 32nd POPL*, 2005.
14. M. Parkinson and G. Biermann. Separation logic, abstraction and inheritance. In *Proc. 35th POPL*, 2008.
15. A. M. Pitts. Relational properties of domains. *Information and Computation*, 127:66–90, 1996.
16. B. Reus and J. Schwinghammer. Separation logic for higher-order store. In *Proc. CSL'06*, volume 4207 of *LNCS*, pages 575–590, 2006.
17. B. Reus and T. Streicher. About Hoare logics for higher-order store. In *Proc. ICALP'05*, LNCS, pages 1337–1348, 2005.
18. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. LICS'02*, pages 55–74, 2002.

Proof Rules for Stored Code

$$((\forall \vec{y}.\{P\}\mathsf{eval}\ [e]\{Q\}) \Rightarrow \forall \vec{y}.\{P\}C\{Q\}) \ \Rightarrow\ \forall \vec{y}.\{P * e \mapsto `C'\}\mathsf{eval}\ [e]\{Q * e \mapsto `C'\} \quad \text{(where } \vec{y} \notin fv(e, C))$$

$$(\forall x.\ (\forall \vec{y}.\ \{P * e \mapsto x\}\mathsf{eval}\ [e]\{Q * e \mapsto x\}) \ \Rightarrow\ \forall \vec{y}.\ \{P * e \mapsto x\}C\{Q * e \mapsto x\}) \ \Rightarrow\ \forall \vec{y}.\ \{P * e \mapsto `C'\}\mathsf{eval}\ [e]\{Q * e \mapsto `C'\} \quad \text{(where } x \notin fv(P, Q, \vec{y}, e, C), \vec{y} \notin fv(e, C))$$

$$(\forall x.\ (\forall \vec{y}.\ \{P * e \mapsto x\}\mathsf{eval}\ [e]\{Q\}) \ \Rightarrow\ \forall \vec{y}.\ \{P * e \mapsto x\}C\{Q\}) \ \Rightarrow\ \forall \vec{y}.\ \{P * e \mapsto `C'\}\mathsf{eval}\ [e]\{Q\} \quad \text{(where } x \notin fv(P, Q, \vec{y}, e, C), \vec{y} \notin fv(e, C))$$

Proof Rules for Hoare Triples

$$(\forall x.\{P\}C\{Q\}) \ \Rightarrow\ \{\exists x.\ P\}C\{\exists x.\ Q\} \ \text{(where } x \notin fv(C))$$
$$(\{P\}C\{Q\} \wedge \{P'\}C\{Q'\}) \ \Rightarrow\ \{P \vee P'\}C\{Q \vee Q'\}$$
$$(\{P\}C\{P_0\} \wedge \{P_0\}C'\{Q\}) \ \Rightarrow\ \{P\}C; C'\{Q\}$$
$$(\{P \wedge e_0{=}e_1\}C\{Q\} \wedge \{P \wedge e_0{\neq}e_1\}C'\{Q\}) \ \Rightarrow\ \{P\}\mathsf{if}\ (e_0{=}e_1)\ \mathsf{then}\ C\ \mathsf{else}\ C'\{Q\}$$
$$(\forall x.\ \{P * x \mapsto e\}C\{Q\}) \ \Rightarrow\ \{P\}\mathsf{let}\ x = \mathsf{new}\ e\ \mathsf{in}\ C\{Q\} \quad \text{(where } x \notin fv(P, Q, e))$$
$$(\forall x.\ \{P * e \mapsto x\}C\{Q\}) \ \Rightarrow\ \{\exists x.\ P * e \mapsto x\}\mathsf{let}\ x = [e]\ \mathsf{in}\ C\{Q\} \quad \text{(where } x \notin fv(Q, e))$$

$$\{\mathbf{emp}\}\mathsf{skip}\{\mathbf{emp}\} \qquad \{e \mapsto \_\}\mathsf{free}(e)\{\mathbf{emp}\} \qquad \{e \mapsto \_\}[e] := e'\{e \mapsto e'\}$$

$$\frac{[\![P]\!]^{\mathcal{A}}_{\eta} \subseteq [\![P']\!]^{\mathcal{A}}_{\eta} \text{ and } [\![Q']\!]^{\mathcal{A}}_{\eta} \subseteq [\![Q]\!]^{\mathcal{A}}_{\eta} \text{ for all } \eta \in [\![\Gamma]\!]}{\Gamma \ \vdash\ \{P'\}C\{Q'\} \Rightarrow \{P\}C\{Q\}}$$

Proof Rules for Invariant Extension $- \otimes P$

$$\varphi \ \Rightarrow\ \varphi \otimes P \qquad \{P\}C\{P'\} \otimes Q \ \Leftrightarrow\ \{P * Q\}C\{P' * Q\}$$
$$(e_0 = e_1) \otimes Q \ \Leftrightarrow\ e_0 = e_1 \qquad (\varphi \otimes P) \otimes Q \ \Leftrightarrow\ \varphi \otimes (P * Q)$$
$$(\varphi \oplus \psi) \otimes P \ \Leftrightarrow\ (\varphi \otimes P) \oplus (\psi \otimes P) \ \text{(where } \oplus \in \{\Rightarrow, \wedge, \vee\}) \qquad (\kappa x.\ \varphi) \otimes P \ \Leftrightarrow\ \kappa x.\ \varphi \otimes P \ \text{(where } \kappa \in \{\forall, \exists\}, x \notin fv(P))$$

Fig. 7. Proof rules

A Proofs

In this appendix, we provide soundness proofs for the proof rules of Fig. 7 (which include those in Fig. 3) and the key lemmas needed to establish these.

Lemma 6 (Admissible and downwards closure). $\mathrm{Ad}(\cdot)$ *is a closure operation: For all* $p, q \in \mathcal{P}$, $p \subseteq \mathrm{Ad}(p) = \mathrm{Ad}(\mathrm{Ad}(p))$ *(in particular,* $\bot \in \mathrm{Ad}(p)$*), and if* $p \subseteq q$ *then* $\mathrm{Ad}(p) \subseteq \mathrm{Ad}(q)$*. Moreover,* $\mathrm{Ad}(p * q) = \mathrm{Ad}(p) * \mathrm{Ad}(q)$.

Proof. That $\mathrm{Ad}(\cdot)$ is a closure operation is standard. That $\mathrm{Ad}(p * q) = \mathrm{Ad}(p) * \mathrm{Ad}(q)$ follows from the fact that if $h = h_1 \bullet h_2$ and $h' \sqsubseteq h$ then there exists a splitting $h' = h'_1 \bullet h'_2$ of h' such that $h'_1 \sqsubseteq h_1$ and $h'_2 \sqsubseteq h_2$. □

Lemma 7. *For all* $p, q \in \mathcal{P}$*,*

$$\models \{\mathrm{Ad}(p)\}c\{\mathrm{Ad}(q)\} \quad \Leftrightarrow \quad \models \{\mathrm{Ad}(p)\}c\{q\} \quad \Leftrightarrow \quad \models \{p\}c\{q\}$$

Proof. By the definition of semantic triples it is easily seen that validity of the first triple implies validity of the second, using

$$\mathrm{Ad}(q * r) = \mathrm{Ad}(q) * \mathrm{Ad}(r) = \mathrm{Ad}(\mathrm{Ad}(q) * r)$$

which holds by Lemma 6. That validity of the second triple implies validity of the third is an immediate consequence of $p \subseteq \mathrm{Ad}(p)$. To see that validity of the third triple implies that of the first, assume that $h \in \mathrm{Ad}(p) * r$. Then h splits into $h = h_1 \bullet h_2$ such that $h_1 \in \mathrm{Ad}(p)$ and $h_2 \in r$. Hence there exists a directed set H in p such that $h_1 \sqsubseteq \sqcup H$. Hence for all $h' \in H$, $c(h' \bullet h_2) \in \mathrm{Ad}(q * r)$, so that the continuity of c and the downward closure of $\mathrm{Ad}(q * r)$ yields $h \sqsubseteq \sqcup H \bullet h_2 \in \mathrm{Ad}(q * r)$. The result follows then by Lemma 6, which gives $\mathrm{Ad}(q * r) = \mathrm{Ad}(\mathrm{Ad}(q) * r)$. □

Lemma 8 (Auxiliary variable). *For all* $\eta \in \mathit{Env}$ *and* $r \in \mathcal{P}$*, if* $\eta, r \models \forall x.\{P\}C\{Q\}$ *where* x *is not free in* C*, then* $\eta, r \models \{\exists x.P\}C\{\exists x.Q\}$*.*

Proof. Assume that $\eta, r \models \forall x.\{P\}C\{Q\}$. Since $x \notin fv(C)$ this is equivalent to

(1) $$\models \{[\![P]\!]^{\mathcal{A}}_{\eta[x\mapsto v]} * r\}\, [\![C]\!]_\eta\, \{[\![Q]\!]^{\mathcal{A}}_{\eta[x\mapsto v]} * r\} \qquad \text{for all } v \in \mathit{Val}.$$

Now let $r' \in \mathcal{P}$ and $h \in [\![\exists x.P]\!]^{\mathcal{A}}_\eta * r * r'$. Thus there exists a splitting $h' \bullet h'' = h$ of h such that $h' \in [\![\exists x.P]\!]^{\mathcal{A}}_\eta$ and $h'' \in r * r'$. By definition, there exists $v \in \mathit{Val}$ such that $h' \in [\![P]\!]^{\mathcal{A}}_{\eta[x\mapsto v]}$. Hence $h \in [\![P]\!]^{\mathcal{A}}_{\eta[x\mapsto v]} * r * r'$ and $[\![C]\!]_\eta\, h \in \mathrm{Ad}([\![Q]\!]^{\mathcal{A}}_{\eta[x\mapsto v]} * r * r')$ by (1) above. Now $[\![Q]\!]^{\mathcal{A}}_{\eta[x\mapsto v]} \subseteq [\![\exists x.Q]\!]^{\mathcal{A}}_\eta$ implies $[\![Q]\!]^{\mathcal{A}}_{\eta[x\mapsto v]} * r * r' \subseteq [\![\exists x.Q]\!]^{\mathcal{A}}_\eta * r * r'$. Thus, $\mathrm{Ad}([\![Q]\!]^{\mathcal{A}}_{\eta[x\mapsto v]} * r * r') \subseteq \mathrm{Ad}([\![\exists x.Q]\!]^{\mathcal{A}}_\eta * r * r')$. The required $[\![C]\!]_\eta\, h \in \mathrm{Ad}([\![\exists x.Q]\!]^{\mathcal{A}}_\eta * r * r')$ follows from this. □

Lemma 9 (Disjunction). *For all* $c \in F_{\mathit{Com}}(\mathit{Com}, \mathit{Com})$*, and all* $r, p, p', q, q' \in \mathcal{P}$*, if the triples* $\{p * r\}c\{q * r\}$ *and* $\{p' * r\}c\{q' * r\}$ *are both valid then so is the triple* $\{(p \cup p') * r\}c\{(q \cup q') * r\}$*.*

Proof. Let $r' \in \mathcal{P}$ and $h \in (p \cup p') * r * r'$. We have to show that $c(h) \in \mathrm{Ad}((q \cup q') * r * r')$. By the assumptions we know that $c(h) \in \mathrm{Ad}(q * r * r')$ or $c(h) \in \mathrm{Ad}(q' * r * r)$. But $q, q' \subseteq q \cup q'$ implies $q * r * r', q' * r * r' \subseteq (q \cup q') * r * r'$, so that both $\mathrm{Ad}(q * r * r')$ and $\mathrm{Ad}(q' * r * r')$ are contained in $\mathrm{Ad}((q \cup q') * r * r')$, i.e., we have $\mathrm{Ad}(q * r * r') \cup \mathrm{Ad}(q' * r * r') \subseteq \mathrm{Ad}((q \cup q') * r * r')$. This gives $c(h) \in \mathrm{Ad}((q \cup q') * r * r)$, as desired. □

Remark 1 (Conjunction rule). As already pointed out in [3], the *conjunction rule*, $\{P\}C\{Q\} \wedge \{P'\}C\{Q'\} \Rightarrow \{P \wedge P'\}C\{Q \wedge Q'\}$, is not validated by the semantics due to the closure applied to the post-condition of a semantic triple. To prove the soundness of the rule, one would need

$$\mathrm{Ad}([\![Q]\!]^{\mathcal{A}}_\eta * r) \cap \mathrm{Ad}([\![Q']\!]^{\mathcal{A}}_\eta * r) \subseteq \mathrm{Ad}(([\![Q]\!]^{\mathcal{A}}_\eta \cap [\![Q']\!]^{\mathcal{A}}_\eta) * r) \qquad \text{for all } r, \eta.$$

However, this does not hold in general, as the following example shows [3, 11]: Consider

$$\eta = [x \mapsto \ell, y \mapsto 0], \quad Q = x \mapsto y, \quad Q' = \mathbf{emp}, \quad r = \mathit{Heap}, \quad h = (\{\!|\ell = 0|\!\}),$$

then $h \in \mathrm{Ad}([\![Q]\!]^{\mathcal{A}}_{\eta} * r) \cap \mathrm{Ad}([\![Q']\!]^{\mathcal{A}}_{\eta} * r)$ but $h \notin \mathrm{Ad}(([\![Q]\!]^{\mathcal{A}}_{\eta} \cap [\![Q']\!]^{\mathcal{A}}_{\eta}) * r)$.

Lemma 10 (Weakening). *For all $c \in F_{Com}(Com, Com)$ and all r, p, q, p', q' in $\mathcal{P}$, if $\models \{p' * r\}c\{q' * r\}$ and $p \subseteq p'$ and $q' \subseteq q$, then $\models \{p * r\}c\{q * r\}$.*

Proof. Let $r' \in \mathcal{P}$ and $h \in p * r * r'$. By $p \subseteq p'$ also $h \in p' * r * r'$. Hence $c(h) \in \mathrm{Ad}(q' * r * r')$, by $\models \{p'\}c\{q'\}$. Since $q' \subseteq q$ implies $\mathrm{Ad}(q' * r * r') \subseteq \mathrm{Ad}(q * r * r')$, we obtain $c(h) \in \mathrm{Ad}(q * r * r')$. We have just shown that $\models \{p * r\}c\{q * r\}$. □

Lemma 11 (Sequencing). *For all $c_1, c_2 \in F_{Com}(Com, Com)$ and all p, p_0, q, r in $\mathcal{P}$, the validity of semantic triples $\{p * r\}c_1\{p_0 * r\}$ and $\{p_0 * r\}c_2\{q * r\}$ implies that $\{p * r\}c\{q * r\}$ is valid, where c is defined by*

$$c(h) \stackrel{def}{=} \text{if } c_1(h) \in \{\bot, \mathit{error}\} \text{ then } c_1(h) \text{ else } c_2(c_1(h)).$$

Proof. Let $r' \in \mathcal{P}$ and $h \in \mathit{Heap}$ such that $h \in (p * r * r')$. We have to show that $c(h) \in \mathrm{Ad}(q * r * r')$. From the assumption we know (1) that $c_1(h) \in \mathrm{Ad}(p_0 * r * r')$ so that in particular $c_1(h) \neq \mathit{error}$. Moreover, if $c_1(h) = \bot$ then $c(h) = \bot \in \mathrm{Ad}(q * r * r')$ since $\bot \in q * r * r'$.

So let us now consider the case where (2) $c_1(h) \neq \bot$ and therefore $c(h) = c_2(c_1\, h)$. Then the assumption that $\{p_0 * r\}c_2\{q * r\}$ is a valid semantic triple entails that also $\{\mathrm{Ad}(p_0 * r)\}c_2\{q * r\}$ by Lemma 7. Thus, for all heaps $h' \in \mathrm{Ad}(p_0 * r * r') = \mathrm{Ad}(p_0 * r) * \mathrm{Ad}(r')$ we have $c_2(h') \in \mathrm{Ad}(q * r * \mathrm{Ad}(r'))$. Applied to (1) and (2), this yields the required $c(h) \in \mathrm{Ad}(q * r * r')$, by Lemma 6. □

Lemma 12 (Skip). *For all $p, r \in \mathcal{P}$, $\{p * r\}\lambda h.\, h\{p * r\}$ is valid.*

Proof. This follows from the fact that $q \subseteq \mathrm{Ad}(q)$ for all predicates $q \in \mathcal{P}$. □

Lemma 13 (Conditional). *For all η and r, if $\eta, r \models \{P \wedge e_0 = e_1\}C\{Q\}$ and $\eta, r \models \{P \wedge e_0 \neq e_1\}C\{Q\}$ then $\eta, r \models \{P\}\mathsf{if}\ (e_0 = e_1)\ \mathsf{then}\ C\ \mathsf{else}\ C'\{Q\}$.*

Proof. Let $r' \in \mathcal{P}$ and suppose $h \in [\![P]\!]^{\mathcal{A}}_{\eta} * r * r'$. Since $\bot \in \mathrm{Ad}([\![Q]\!]^{\mathcal{A}}_{\eta} * r * r')$ it suffices to consider the case where not both $[\![e_0]\!]^{\mathcal{E}}_{\eta}$ and $[\![e_1]\!]^{\mathcal{E}}_{\eta}$ are in Com. We have that either $[\![e_0]\!]^{\mathcal{E}}_{\eta} = [\![e_1]\!]^{\mathcal{E}}_{\eta}$ or $[\![e_0]\!]^{\mathcal{E}}_{\eta} \neq [\![e_1]\!]^{\mathcal{E}}_{\eta}$. We consider only the first case, the second is analogous.

Clearly if $[\![e_0]\!]^{\mathcal{E}}_{\eta} = [\![e_1]\!]^{\mathcal{E}}_{\eta}$ we have $h' \in [\![e_0 = e_1]\!]^{\mathcal{A}}_{\eta}$ for all $h' \in \mathit{Heap}$. Thus $h \in [\![P \wedge e_0 = e_1]\!]^{\mathcal{A}}_{\eta} * r * r'$, so that we obtain the required

$$[\![\mathsf{if}\ (e_0 = e_1)\ \mathsf{then}\ C\ \mathsf{else}\ C']\!]_{\eta}\, h \in \mathrm{Ad}([\![Q]\!]^{\mathcal{A}}_{\eta} * r * r')$$

from $[\![\mathsf{if}\ (e_0 = e_1)\ \mathsf{then}\ C\ \mathsf{else}\ C']\!]_{\eta}\, h = [\![C]\!]_{\eta}\, h$ and the assumption that $\eta, r \models \{P \wedge e_0 = e_1\}C\{Q\}$. □

Lemma 14 (Free). *For all η and all $r \in \mathcal{P}$ we have that*

$$\eta, r \models \{\llbracket \exists x'.e \mapsto x' \rrbracket_\eta^{\mathcal{A}}\} \llbracket \mathsf{free}\ e \rrbracket_\eta \{\llbracket \mathbf{emp} \rrbracket_\eta^{\mathcal{A}}\}.$$

Proof. Let $r' \in \mathcal{P}$ and $h \in (\llbracket \exists x'.e \mapsto x' \rrbracket_\eta^{\mathcal{A}} * r * r')$. Without loss of generality, we may assume $h \neq \bot$, for otherwise the result follows from the strictness of commands and $\bot \in \mathrm{Ad}(\llbracket \mathbf{emp} \rrbracket_\eta^{\mathcal{A}} * r * r')$. Thus there exist $h' \neq \bot \neq h''$ such that $h' \# h''$, and such that $h = h' \bullet h''$, $h' \in r * r'$ and $h'' \in \llbracket \exists x'.e \mapsto x' \rrbracket_\eta^{\mathcal{A}}$. By definition, this means $h'' = \{\!|\llbracket e \rrbracket_\eta^{\mathcal{E}} = h''(\llbracket e \rrbracket_\eta^{\mathcal{E}})|\!\}$, and in particular $\llbracket e \rrbracket_\eta^{\mathcal{E}} \in \mathsf{dom}(h'') \subseteq \mathsf{dom}(h)$. This yields

$$\llbracket \mathsf{free}\ e \rrbracket_\eta\, h \;=\; (\text{if } \llbracket e \rrbracket_\eta^{\mathcal{E}} \notin \mathsf{dom}(h) \text{ then } \mathit{error} \text{ else } h') \;=\; h'.$$

That $\eta, r \models \{\llbracket e \mapsto \_ \rrbracket_\eta^{\mathcal{A}}\} \llbracket \mathsf{free}\ e \rrbracket_\eta \{\llbracket \mathbf{emp} \rrbracket_\eta^{\mathcal{A}}\}$ holds is now a consequence of the fact that $h' \in r * r'$ implies $h' \in \mathrm{Ad}(r * r') = \mathrm{Ad}(\llbracket \mathbf{emp} \rrbracket_\eta^{\mathcal{A}} * r * r')$. □

Lemma 15 (New). *For all η and $r \in \mathcal{P}$, if $\eta, r \models \forall x.\{P * x \mapsto e\} C \{Q\}$ and x does not occur free in e, P or Q, then $\eta, r \models \{P\} \mathsf{let}\ x = \mathsf{new}\ e\ \mathsf{in}\ C \{Q\}$.*

Proof. Assume that for all $v \in \mathit{Val}$ the relation $\eta[x \mapsto v], r \models \{P * x \mapsto e\} C \{Q\}$ holds. Since x is not free in e, P or Q, this means that

$$(1) \qquad \models \{\llbracket P \rrbracket_\eta^{\mathcal{A}} * \llbracket x \mapsto e \rrbracket_{\eta[x \mapsto v]}^{\mathcal{A}} * r\} \llbracket C \rrbracket_{\eta[x \mapsto v]} \{\llbracket Q \rrbracket_\eta^{\mathcal{A}} * r\} \quad \text{for all } v.$$

We must prove validity of the triple $\{\llbracket P \rrbracket_\eta^{\mathcal{A}} * r\} \llbracket \mathsf{let}\ x = \mathsf{new}\ e\ \mathsf{in}\ C \rrbracket_\eta \{\llbracket Q \rrbracket_\eta^{\mathcal{A}} * r\}$.

Let $r' \in \mathcal{P}$ and $h \in \llbracket P \rrbracket_\eta^{\mathcal{A}} * r * r'$. Without loss of generality assume $h \neq \bot$, and let $\ell = \min(\mathit{Loc} - \mathsf{dom}(h))$. Thus $h \bullet \{\!|\ell = \llbracket e \rrbracket_\eta^{\mathcal{E}}|\!\} \in (\llbracket P \rrbracket_\eta^{\mathcal{A}} * \llbracket x \mapsto e \rrbracket_{\eta[x \mapsto \ell]}^{\mathcal{A}} * r * r')$. Since

$$\llbracket \mathsf{let}\ x = \mathsf{new}\ e\ \mathsf{in}\ C \rrbracket_\eta\, h = \llbracket C \rrbracket_{\eta[x \mapsto \ell]} (h \bullet \{\!|\ell = \llbracket e \rrbracket_\eta^{\mathcal{E}}|\!\}),$$

the lemma follows from (1). □

Lemma 16 (Deref). *The proof rule for dereference commands is sound. That is, if $x \notin fv(e, Q)$, then for all η and r,*

$$\eta, r \models \forall x.\{P * e \mapsto x\} C \{Q\}$$

implies

$$\eta, r \models \{\exists x.P * e \mapsto x\} \mathsf{let}\ x = [e]\ \mathsf{in}\ C \{Q\}.$$

Proof. Assume $\eta, r \models \forall x.\{P * e \mapsto x\} C \{Q\}$. Then, since $x \notin fv(Q)$, we have that

$$(1) \qquad \models \{\llbracket P * e \mapsto x \rrbracket_{\eta[x \mapsto v]}^{\mathcal{A}} * r\} \llbracket C \rrbracket_{\eta[x \mapsto v]} \{\llbracket Q \rrbracket_\eta^{\mathcal{A}} * r\} \qquad \text{for all } v \in \mathit{Val}.$$

Using this, we will prove that

$$\models \{\llbracket \exists x.P * e \mapsto x \rrbracket_\eta^{\mathcal{A}} * r\} \llbracket \mathsf{let}\ x = [e]\ \mathsf{in}\ C \rrbracket_\eta \{\llbracket Q \rrbracket_\eta^{\mathcal{A}} * r\}.$$

Let $r' \in \mathcal{P}$ and $h \in ([\![\exists x.P * e \mapsto x]\!]^{\mathcal{A}}_{\eta} * r * r')$. If $h = \bot$, $[\![\mathsf{let}\ x = [e]\ \mathsf{in}\ C]\!]_{\eta}\, h$ is also $\bot$, which belongs to $\mathrm{Ad}([\![Q]\!]^{\mathcal{A}}_{\eta} * r * r')$. The conclusion follows from this. Suppose that $h \neq \bot$. By definition, there exists a splitting $h' \bullet h'' = h$ of h such that $h' \in [\![\exists x.P * e \mapsto x]\!]^{\mathcal{A}}_{\eta}$ and $h'' \in r * r'$. In particular, $h' \in [\![P * e \mapsto x]\!]^{\mathcal{A}}_{\eta[x \mapsto v]}$ for some $v \in \mathit{Val}$. Since $x \notin \mathit{fv}(e)$ by assumption, this implies that

$$[\![e]\!]^{\mathcal{E}}_{\eta} = [\![e]\!]^{\mathcal{E}}_{\eta[x \mapsto v]} \in \mathsf{dom}(h') \subseteq \mathsf{dom}(h) \quad \wedge \quad h([\![e]\!]^{\mathcal{E}}_{\eta}) = v.$$

Thus we have $[\![\mathsf{let}\ x = [e]\ \mathsf{in}\ C]\!]_{\eta}\, h = [\![C]\!]_{\eta[x \mapsto v]}\, h$, and since $h = h' \bullet h'' \in [\![P * e \mapsto x]\!]^{\mathcal{A}}_{\eta[x \mapsto v]} * r * r'$ we obtain $[\![\mathsf{let}\ x = [e]\ \mathsf{in}\ C]\!]_{\eta}\, h \in \mathrm{Ad}([\![Q]\!]^{\mathcal{A}}_{\eta} * r * r')$ from (1). □

Lemma 17 (Assign). *For all η and all $r \in \mathcal{P}$ we have*

$$\eta, r \models \{\exists x'.e \mapsto x'\}[e] := e'\{e \mapsto e'\}\,.$$

Proof. Let $r' \in \mathcal{P}$ and $h \in [\![\exists x'.\, e \mapsto x']\!]^{\mathcal{A}}_{\eta} * r * r'$. Without loss of generality, assume $h \neq \bot$. By definition, there exist $h' \neq \bot \neq h''$ such that

$$(1) \qquad h' \# h'', \quad h = h' \bullet h'', \quad h' \in r * r', \quad \text{and} \quad h'' = \{\![\![e]\!]^{\mathcal{E}}_{\eta} = h([\![e]\!]^{\mathcal{E}}_{\eta})\}\!\}\,.$$

In particular, $[\![e]\!]^{\mathcal{E}}_{\eta} \in \mathsf{dom}(h'') \subseteq \mathsf{dom}(h)$. Thus,

$$[\![[e] := e']\!]_{\eta}\, h \;=\; h[[\![e]\!]^{\mathcal{E}}_{\eta} \mapsto [\![e']\!]^{\mathcal{E}}_{\eta}] \;=\; h' \bullet \{\![\![e]\!]^{\mathcal{E}}_{\eta} = [\![e']\!]^{\mathcal{E}}_{\eta}\}\!\}\,.$$

Since by definition $\{\![\![e]\!]^{\mathcal{E}}_{\eta} = [\![e']\!]^{\mathcal{E}}_{\eta}\}\!\} \in [\![e \mapsto e']\!]^{\mathcal{A}}_{\eta}$ and $h' \in r * r'$ by (1), the definition of $*$ gives $[\![[e] := e']\!]_{\eta}\, h \in \mathrm{Ad}([\![e \mapsto e']\!]^{\mathcal{A}}_{\eta} * r * r')$ as required. □

Lemma 18 (Rec 1). *The first recursion rule for stored procedures is sound.*

Proof. The first rule for stored procedures can be derived from the second and the higher-order frame rules. Concretely, using the rules for $-\otimes R$, one can derive the so called second-order frame rule in our logic:

$$\Big((\forall \vec{y}.\{P\}C\{Q\}) \Rightarrow \forall \vec{y}.\{P'\}C'\{Q'\}\Big) \Rightarrow \Big((\forall \vec{y}.\{P*R\}C\{Q*R\}) \Rightarrow \forall \vec{y}.\{P'*R\}C'\{Q'*R\}\Big)$$

This implication lets us show that the assumption of the second rule follows from that of the first rule:

$$\begin{aligned}
&\big((\forall \vec{y}.\{P\}\mathsf{eval}\ [e]\{Q\}) \Rightarrow \forall \vec{y}.\{P\}C\{Q\}\big)\\
&\Rightarrow\ \big((\forall \vec{y}.\{P * e \mapsto x\}\mathsf{eval}\ [e]\{Q * e \mapsto x\}) \Rightarrow \forall \vec{y}.\{P * e \mapsto x\}C\{Q * e \mapsto x\}\big)\\
&\Rightarrow\ \big(\forall x.\ \big((\forall \vec{y}.\{P * e \mapsto x\}\mathsf{eval}\ [e]\{Q * e \mapsto x\}) \Rightarrow \forall \vec{y}.\{P * e \mapsto x\}C\{Q * e \mapsto x\}\big)\big).
\end{aligned}$$

□

Lemma 19 (Rec 3). *The third recursion rule for stored procedures is sound.*

Proof. The proof is similar to that for the second recursion rule, given in Lemmas 4 and 5. The main difference is that now the property 2 for the postcondition from the proof of Lemma 5 is not needed, since the postcondition in the first recursion rule does not depend on x. □

B Example: object-based factorial

In this section we present a more detailed proof of the factorial function that uses higher-order store to implement recursive calls, using the proof rules of our logic.

Using pointers to blocks of consecutive cells, we can simulate objects in our language. Note that methods need to refer to the object itself (i.e., the implicit parameter *this* or *self*, whose role will be played by the variable o in the example below). An object is represented as a block of consecutive cells, each of which contains an integer (field) or a stored procedure (method). For example, the code C_{main} below constructs an object o that contains two fields arg and res, and a method fac that computes the factorial of arg (decrementing it as a side effect) and multiplies this onto res. The object o is implemented by a block of three consecutive cells, the first two of which represent argument and result fields, and the last of which stores the method *fac*. After this object is created, the factorial method is invoked. The recursive nature of the object o is implicitly expressed by the stored procedures that refer to the object o again. Note that by angle brackets we highlight that code depends on variables (denoted inside the brackets) declared earlier.

$$\begin{aligned}
C_{main} \stackrel{def}{=}\ & \mathsf{let}\ o{=}\mathsf{new}\ (5, 1, \text{`}\mathsf{skip}\text{'})\ \mathsf{in} \\
& \textit{// initialise the fac field of object o} \\
& [o{+}2]{:=}\text{`}C_{fac}\langle o\rangle\text{'} \\
& \textit{// call factorial method of o} \\
& \mathsf{eval}\ [o{+}2]
\end{aligned}$$

$$\begin{aligned}
C_{fac}\langle o\rangle \stackrel{def}{=}\ & \mathsf{let}\ xv{=}[o]\ \mathsf{in} \\
& \mathsf{let}\ rv{=}[o{+}1]\ \mathsf{in} \\
& \mathsf{if}\ (xv{=}0)\ \mathsf{then\ skip} \\
& \mathsf{else}\ [o{+}1]{:=}rv \cdot xv;\ [o]{:=}xv{-}1; \mathsf{eval}\ [o{+}2]\ \mathsf{fi}
\end{aligned}$$

Observe how the factorial is defined by recursion through the store (calling itself). Let $Obj_o\langle n, m\rangle$ be the specification defined as follows:

$$(o \mapsto n, m, \text{`}C_{fac}\langle o\rangle\text{'}).$$

According to the definition above, $Obj_o\langle n, m\rangle$ means that o is a pointer to a block of size 3, such that the content of the first cell of the block (i.e., the value of the argument field) is n, that of the second cell (i.e., the value of the result field) is m, and the last cell contains the factorial function.

Specifications for the methods above expressed in specification logic could be

$$o \vdash \forall n, m.\ \{Obj_o\langle n, m\rangle\}\mathsf{eval}\ [o{+}2]\{Obj_o\langle 0, m \cdot n!\rangle\}.$$

Note how *Obj* refers to the whole object stored in pointer o in the pre- and post-condition of the specifications. This is sensible as one expects to own the whole object if one calls a method. Let's see how we can derive the triple. Unrolling the definition of Obj, we obtain

$$o \vdash \forall n, m.\\ \{o \mapsto n, m, `C_{fac}\langle o\rangle'\}\mathsf{eval}\ [o{+}2]\{o \mapsto 0, m \cdot n!, `C_{fac}\langle o\rangle'\}$$

In what follows we prove the above specification using the third recursion rule. That is, we will show

$$o \vdash \varphi \Rightarrow \big(\forall n, m.\ \{o \mapsto n, m\}C_{fac}\langle o\rangle\{o \mapsto 0, m \cdot n!\}\big)$$

where the assumption φ is

$$\forall n, m.\ \{o \mapsto n, m\}\mathsf{eval}\ [o{+}2]\{o \mapsto 0, m \cdot n!\}.$$

Spelling out the definition of factorial procedure, applying the lookup rule twice and moving the universally quantified variables xv and rv into the context, we obtain as new goal

$$\begin{array}{l} o, n, m, xv, rv \vdash \varphi \Rightarrow \\ \{(o \mapsto xv, rv) \land xv{=}n \land rv{=}m\} \\ \quad \mathsf{if}\ (xv{=}0)\ \mathsf{then\ skip} \\ \quad \mathsf{else}\ [o{+}1]{:=}rv \cdot xv; [o]{:=}rv{-}1; \mathsf{eval}\ [o{+}2]\ \mathsf{fi} \\ \{o \mapsto 0, m \cdot n!\}. \end{array}$$

This gives us two new goals by the conditional axiom:

$$\begin{array}{l} o, n, m, xv, rv \vdash \varphi \Rightarrow \\ \{(o \mapsto xv, rv) \land xv{=}n \land rv{=}m \land xv{=}0\}\mathsf{skip}\{o \mapsto 0, m \cdot n!\} \end{array}$$

$$\begin{array}{l} o, n, m, xv, rv \vdash \varphi \Rightarrow \\ \{(o \mapsto xv, rv) \land xv{=}n \land rv{=}m \land xv{\neq}0\} \\ \quad [o{+}1]{:=}rv \cdot xv; [o]{:=}xv{-}1; \mathsf{eval}\ [o{+}2] \\ \{o \mapsto 0, m \cdot n!\} \end{array}$$

The first of them follows trivially from applying the consequence rule and the skip axiom. To prove the second one, we apply the axiom for composition and assignment and are left with

$$\begin{array}{l} \{(o \mapsto xv, rv) \land xv{=}n \land rv{=}m \land xv{\neq}0\} \\ \quad [o{+}1]{:=}rv \cdot xv; \\ \{(o \mapsto xv, rv \cdot xv) \land xv{=}n \land rv{=}m \land xv{\neq}0\} \\ \quad [o]{:=}xv{-}1; \\ \{(o \mapsto xv{-}1, rv \cdot xv) \land xv{=}n \land rv{=}m \land xv{\neq}0\} \\ \quad \mathsf{eval}\ [o{+}2] \\ \{o \mapsto 0, m \cdot n!\} \end{array}$$

where the first two steps follow from the assignment axiom. To show the last step after employing the consequence rule it suffices to show:[4]

$$\{o \mapsto n{-}1, m \cdot n\}\mathsf{eval}\ [vo.\mathrm{fac}]\{o \mapsto 0, m \cdot n \cdot (n{-}1)!\}$$

which we derive from φ by instantiating the quantified variables n, m of φ by $n{-}1$ and $m \cdot n$, resp.

Next, we prove

$$\vdash \{\mathbf{emp}\}C_{main}\{\mathit{Obj}_o\langle 0, 5!\rangle\}\,.$$

By the allocation rule and the introduction rule for $\forall$, it suffices to prove:

$$\begin{aligned} o \ \vdash \ & \{o \mapsto 5, 1, \text{`}\mathsf{skip}\text{'}\} \\ & \big([o{+}2]{:=}\text{`}C_{fac}\langle o\rangle\text{'}; \mathsf{eval}\ [o{+}2]\big) \\ & \{\mathit{Obj}_o\langle 0, 5!\rangle\} \end{aligned}$$

and thus by the composition rule we are left with two new goals:

$$o \vdash \{o \mapsto 5, 1, \text{`}\mathsf{skip}\text{'}\}[o{+}2]{:=}\text{`}C_{fac}\langle o\rangle\text{'}\{\mathit{Obj}_o\langle 5, 1\rangle\}$$

$$o \vdash \{\mathit{Obj}_o\langle 5, 1\rangle\}\mathsf{eval}\ [o{+}2]\{\mathit{Obj}_o\langle 0, 5!\rangle\}.$$

To show the first goal, we unroll the definition of *Obj* in the postcondition:

$$o \vdash \{o \mapsto 5, 1, \text{`}\mathsf{skip}\text{'}\}[o{+}2]{:=}\text{`}C_{fac}\langle o\rangle\text{'}\{o \mapsto 5, 1, \text{`}C_{fac}\text{'}\},$$

which follows from the assignment rule. The second goal follows from the specification of $\mathsf{eval}\ [o{+}2]$ that we proved earlier.

[4] Note that the $\cdot$ operator is defined not just for integers, but also for non-integer values. Our proof relies on only two properties of the operator: the operator is standard multiplication when restricted to integers, and it is associative.