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, 15]. 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 [17, 2, 1, 12, 9, 3, 8, 7, 13]. 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 previous work of Reus and

Schwinghammer [15], 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 [15], and compare with our new model.

The first challenge is to find a model that validates the frame rule known from separation logic [17]. 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 [15]. 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 [16]. 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 [14]. In [15], these conditions were ensured

$$
\begin{array}{lll}
e \in \textsc{Exp} ::= \ldots \mid \text{`}C\text{'} & & \text{quote (command as expression)} \\
C \in \textsc{Com} ::= \mathsf{skip} \mid C_1;C_2 \mid \mathsf{if}\ (e_1{=}e_2)\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 & & \text{no op, sequencing, conditional} \\
\quad \mid \mathsf{let}\ x{=}\mathsf{new}\ (e_1,\ldots,e_n)\ \mathsf{in}\ C \mid \mathsf{free}\ e & & \text{allocation, disposal} \\
\quad \mid [e_1]{:=}e_2 \mid \mathsf{let}\ y{=}[e]\ \mathsf{in}\ C \mid \mathsf{eval}\ [e] & & \text{assignment, lookup, unquote}
\end{array}
$$

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, we use an admissible and downwards closure of the post-condition, similar to the use of $\perp\!\!\!\perp$-closure in [3] (see Section 4).

2 Programs, assertions and specifications

Programs The abstract syntax of the programming language is presented in Fig. 1. It is essentially as in [15], 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.

Assertions The assertions $P, Q, \ldots$ used in Hoare triples are built from the formulas of classical predicate logic and the additional separation logic assertions that describe the heap ($e \mapsto e'$, **emp**, $P * Q$ and $P \mathrel{-\!\!*} Q$; cf. [17]). Note that expressions in formulas can point to quoted code, as in $x \mapsto \text{`}C\text{'}$, so that they can be used to specify properties of stored procedures. 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 $x' \notin fv(e)$. We write $\Gamma \vdash P(\colon \mathsf{Assert})$ for some finite set of variables Γ, when the assertion P contains only free variables in Γ.

Specifications Specifications are formulas of first-order intuitionistic logic with equality. In addition, it includes Hoare triples as atomic formulas and invariant extensions $\varphi \otimes P$ (from [3]):

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

While assertions express properties of states, specifications describe properties of programs (sometimes using assertions inside Hoare triples). For a finite set Γ of variables, $\Gamma \vdash \varphi(\colon \mathsf{Spec})$ means that Γ includes all free variables of 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

PROOF RULES FOR STORED CODE

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

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

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

PROOF RULES FOR HOARE TRIPLES

$$(\forall x.\,\{P * x \mapsto e\}C\{Q\}) \ \Rightarrow\ \{P\}\mathsf{let}\ x = \mathsf{new}\ e\ \mathsf{in}\ C\{Q\} \quad (\text{where } x \notin fv(P, Q, e))$$

$$(\forall x.\,\{P * e \mapsto x\}C\{Q\}) \ \Rightarrow\ \{\exists x.\,P * e \mapsto x\}\mathsf{let}\ x = [e]\ \mathsf{in}\ C\{Q\} \quad (\text{where } x \notin fv(Q, e))$$

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

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

PROOF RULES FOR INVARIANT EXTENSION $- \otimes P$

$$\varphi \ \Rightarrow\ \varphi \otimes P \qquad \{P\}C\{P'\} \otimes Q \ \Leftrightarrow\ \{P * Q\}C\{P' * Q\}$$

$$(e_0 = e_1) \otimes Q \ \Leftrightarrow\ e_0 = e_1 \qquad (\varphi \otimes P) \otimes Q \ \Leftrightarrow\ \varphi \otimes (P * Q)$$

$$(\varphi \oplus \psi) \otimes P \ \Leftrightarrow\ (\varphi \otimes P) \oplus (\psi \otimes P) \quad (\text{where } \oplus \in \{\Rightarrow, \wedge, \vee\}) \qquad (\kappa x.\,\varphi) \otimes P \ \Leftrightarrow\ \kappa x.\,\varphi \otimes P \quad (\text{where } \kappa \in \{\forall, \exists\}, x \notin fv(P))$$

Fig. 2. Some proof rules

invariant extension $\varphi \otimes P$. Fig. 2 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 without the need to commit to a specific proof system for assertions.

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] These rules are similar to the rule for calling a parameterless, recursive procedure p declared as $p \Leftarrow C$, where C is the body of p that may contain a recursive call to p:

$$\frac{(\forall \vec{y}.\,\{P\}\mathsf{call}\ p\{Q\}) \vdash \forall \vec{y}.\,\{P\}C\{Q\}}{\forall \vec{y}.\,\{P\}\mathsf{call}\ p\{Q\}} \tag{1}$$

This rule is usually proved sound via fixpoint induction (note that p in the premiss semantically refers to any procedure with the required properties, whereas in the conclusion p refers to the declared procedure). For the language of Fig. 1,

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

the fact that stored procedures are in use means that the declaration of a procedure is now expressed by an assertion $e \mapsto \text{`}C\text{'}$, stating that e is a reference to the procedure with body C.

In Fig. 2 there are three rules for stored code that might call itself recursively, establishing partial correctness (and hence do not feature in the logic for *total* correctness of [6]). The first rule prohibits any access to the storing location e except through $\mathsf{eval}\ [e]$, whereas the second and third are more permissive. Note also that only the first two rules establish that the stored procedure called has not been altered. This is important in cases where the procedure gets updated. Updating code after its first call is a general pattern of usage of stored code, found e.g. in device drivers [5]: the first call is used for initialisation that further calls rely on.

It would be preferable to have just one rule for recursive procedures, e.g.

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

Alas we cannot easily prove such a rule sound. Our soundness proof for the recursion rules relies on pre- and post-condition satisfying properties (a) and (b), resp., as stated in the proof of Lemma 5 (see Section 4), and an arbitrary post-condition might violate property (b). We achieve soundness by restricting the shape of the post-condition to $Q * e \mapsto x$ and stipulating the side condition $x \notin fv(Q)$. As a consequence, we cannot instantiate the recursion rules with post-condition $e \mapsto x * e' \mapsto x$, as needed for a self-copying command $\mathsf{let}\ x{=}[e]\ \mathsf{in}\ [e']{:=}x$ stored at e. Yet, also $Q * e \mapsto x * e' \mapsto x$ satisfies property 2 mentioned above, and soundness of a corresponding recursion rule could be established analogously to Lemma 5. This means the logic is incomplete; our objective has been to find rules that are easy to apply on programs with "common" use of stored procedures.

Alternatively, we could have given a single rule for stored procedures, but with a *semantic* side-condition to rule out unsuitable post-conditions.

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

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

The command C implements the factorial function in an object-oriented style, 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 $\mathsf{eval}\ [o{+}2]$ of F_o.

The specification expresses that C computes 5! and stores it in cell $o{+}1$. The key step of the proof is the derivation

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

which shows the correctness of the stored procedure F_o. This step applies the first rule for stored procedures, and it illustrates the benefit of the rule. Here, 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; 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:

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

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

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:

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

The major step of the proof of C_1 is the use 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:

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

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

Since F_2 directly accesses cell $i{+}1$, which stores the procedure, and F_3 updates the storing cell, we have used the second rule for C_2 and the third for C_3. □

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 (i.e. a labelled smash product of arbitrary finite arity), which will be used to model *heaps*. Its underlying set is $Rec(A) = \big(\sum_{N \subseteq_{fin} Nats^+} (N \to A_\downarrow)\big)_\perp$, 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 $Rec(A)$ 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 $r \bullet r' \stackrel{def}{=}$ if $(r\#r')$ then $(r \cup r')$ else $\big($if $(r{=}\perp \vee r'{=}\perp)$ then $\perp$ 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 $id_{Com} = lfp(\lambda e{:}Com{\multimap}Com.\; i \circ F_{Com}(e, e) \circ i^{-1})$ [14]. The soundness proof of the rules for stored procedures exploits this fact.

Interpretation of the programming language Fig. 3 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. $[\![\text{`}C\text{'}]\!]^{\mathcal{E}}_\eta$ is defined to be $i([\![C]\!]_\eta)$ (which implicitly makes use of the embedding of Com into Val). In the defining equations in Fig. 3 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.

Interpretation of assertions Let $\mathcal{P}$ be the set of predicates $p \subseteq Heap$ that contain $\perp$. The separating conjunction for these predicates, known from separation logic [17], is defined by: $h \in p_1 * p_2 \stackrel{def}{\Leftrightarrow} \exists h_1, h_2.\; h = h_1 \bullet h_2 \;\wedge\; h_1 \in p_1 \;\wedge\; h_2 \in p_2$. 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 $p * q \subseteq p' * q'$.

[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{def}{=} h \\
[\![C_1;C_2]\!]_\eta\, h &\stackrel{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{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, ..., e_n\ \mathsf{in}\ C]\!]_\eta\, h &\stackrel{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{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{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{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{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. 3. Interpretation $[\![C]\!]_\eta \in \mathit{Heap} \multimap T_{err}(\mathit{Heap})$

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

Fig. 4. Interpretation $[\![P]\!]^{\mathcal{A}} : \mathit{Env} \to \mathcal{P}$ of assertions

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. Some cases of this interpretation of assertions are spelled out in Fig. 4.

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; it may be obtained as the intersection of all admissible, downward-closed subsets of Heap that include p).

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

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 $h \in Heap$, 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:

Lemma 1 (Frame rule). *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 $\{c \mid \{p\}c\{q\}$ 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 [15].

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 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.

Some cases of the definition of the satisfaction relation $\models$ are shown in Fig. 5. 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

$$
\begin{aligned}
\eta, p \models \varphi \wedge \psi &\stackrel{def}{\Leftrightarrow} \eta, p \models \varphi \text{ and } \eta, p \models \psi \\
\eta, p \models \varphi \Rightarrow \psi &\stackrel{def}{\Leftrightarrow} \text{for all } r \in \mathcal{P}, \text{ if } p \sqsubseteq r \text{ and } \eta, r \models \varphi, \text{ then } \eta, r \models \psi \\
\eta, p \models \varphi \otimes P &\stackrel{def}{\Leftrightarrow} \eta, p * [\![P]\!]^{\mathcal{A}}_{\eta} \models \varphi \\
\eta, p \models \{P\}C\{Q\} &\stackrel{def}{\Leftrightarrow} \models \{[\![P]\!]^{\mathcal{A}}_{\eta} * p\}\, [\![C]\!]_{\eta}\, \{[\![Q]\!]^{\mathcal{A}}_{\eta} * p\}
\end{aligned}
$$

Fig. 5. Interpretation $\eta, p \models \varphi$ of specifications

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.

Lemma 4 (Recursion). *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 $Com \times Com$ by

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

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.

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

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 $\forall c.\, \forall r' \sqsupseteq r.\, A_{\eta,r'}(c,c) \Rightarrow A_{\eta,r'}(c, [\![\text{`}C\text{'}]\!]_{\eta})$, we obtain $\forall c.\ S_{\eta,r}(c) \Rightarrow A_{\eta,r}(c, [\![\text{`}C\text{'}]\!]_{\eta})$ and therefore $S_{\eta,r}([\![\text{`}C\text{'}]\!]_{\eta})$ by (1). But then (2) shows $A_{\eta,r}([\![\text{`}C\text{'}]\!]_{\eta}, [\![\text{`}C\text{'}]\!]_{\eta})$, 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 [16], but many 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

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

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, one needs only 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 $j \stackrel{def}{=} Rec(\hat{l})$,

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

To see why it is suffices to prove (a) and (b), 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 $\vec{v} \in Val^n$

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

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

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

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 (a) 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 (b) 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 $\mathrm{Ad}([\![Q * e \mapsto x]\!]^{\mathcal{A}}_{\eta[x \mapsto d]} * r * r')$. $\square$

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 TOPLAS*, 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. *LMCS*, 2(5:1), 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*, 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. Bierman. Separation logic, abstraction and inheritance. In *Proc. 35th POPL*, 2008.
14. A. M. Pitts. Relational properties of domains. *Information and Computation*, 127:66–90, 1996.
15. B. Reus and J. Schwinghammer. Separation logic for higher-order store. In *Proc. CSL'06*, volume 4207 of *LNCS*, pages 575–590, 2006.
16. B. Reus and T. Streicher. About Hoare logics for higher-order store. In *Proc. ICALP'05*, LNCS, pages 1337–1348, 2005.
17. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. LICS'02*, pages 55–74, 2002.