Separation Logic for Higher-order Store

Bernhard Reus[1] and Jan Schwinghammer[2]

[1] Department of Informatics, University of Sussex, Brighton, UK
[2] Programming Systems Lab, Saarland University, Saarbrücken, Germany

Abstract. Separation Logic is a sub-structural logic that supports local reasoning for imperative programs. It is designed to elegantly describe sharing and aliasing properties of heap structures, thus facilitating the verification of programs with pointers. In past work, separation logic has been developed for heaps containing records of basic data types. Languages like C or ML, however, also permit the use of code pointers. The corresponding heap model is commonly referred to as "higher-order store" since heaps may contain commands which in turn are interpreted as partial functions between heaps.

In this paper we make Separation Logic and the benefits of local reasoning available to languages with higher-order store. In particular, we introduce an extension of the logic and prove it sound, including the Frame Rule that enables specifications of code to be extended by invariants on parts of the heap that are not accessed.

1 Introduction and Motivation

Since the beginning of program verification for high-level languages [8], pointers (and the aliasing they cause) have presented a major stumbling block for formal correctness proofs. Some of the pain of verifying pointer programs has been removed in recent years with the introduction of *Separation Logic*, developed by Reynolds, O'Hearn and others [25, 10, 14]. This is a variant of Hoare logic where assertions may contain the *separation conjunction*: The assertion $P * Q$ states that P and Q hold for disjoint parts of the heap store – in particular, there is no sharing between these regions. The separation connective allows for the elegant formulation of a *frame rule* which is key to local reasoning: In a triple $\{P\}\, c\, \{Q\}$, the assertions P and Q need to specify the code c only in terms of the heap cells that are actually used (the "footprint"). Clients can add invariants R for disjoint heap areas in a modular fashion, to obtain $\{P * R\}\, c\, \{Q * R\}$ without reproving c.

Some impressive results have been obtained within this formalism, including the verification of several algorithms operating on heap-based graph structures such as the Schorr-Waite graph marking algorithm [29, 4, 6]. Separation logic has been extended in several directions, covering shared-variable concurrent programs [13], modules [16] and higher-order procedures [5]. However, in all cases only values of *basic data types* can be stored. On the other hand, languages like C, ML, Scheme, and (implicitly) Java provide *code pointers*. In object-oriented programs, stored procedures are commonly used as callbacks. Moreover, code pointers “also appear in low-level programs that use techniques of higher-order or object-oriented programming” [25].

In this paper we address the problem of extending Separation Logic to languages that allow the storage of procedures. Reynolds emphasized the importance of code pointers in [25], speculating that the marriage of separation logic with continuation semantics could provide a way to reason about them. A step in this direction has been taken in [28], although mutual dependencies of stored procedures were initially excluded. Building on our results in [23, 22, 21] we suggest a much more direct extension of Separation Logic, by using a denotational semantics instead of an operational one. This allows us to model code pointers by means of a higher-order store, i.e., as a (mixed-variant) recursively defined domain where stores map locations to basic values *or to state transformers* (denoting partial maps from store to store).

The starting point for our work is [23] where a Hoare-style logic for a language with higher-order store is presented. This language assumes a global store and does not provide explicit means to allocate or dispose memory. The logic in [23] extends traditional Hoare logic by rules to reason about the *(mutual) recursion through the store* that becomes possible with command storage [11].

We extend the language of [23] with memory allocation constructs, and the logic with the rules of Separation Logic. The semantics of dynamically allocated memory raises a subtle point in connection with Separation Logic: soundness of the frame rule relies on the fact that the choice of a fresh location made by the allocation mechanism is irrelevant, as far as the logic is concerned. To the best of our knowledge, in all previous approaches this requirement has been enforced by making allocation *non-deterministic* so that valid predicates cannot possibly depend on assumptions about particular locations. However, in the presence of higher-order store where we have to solve recursive domain equations we found the use of (countable) non-determinism quite challenging (for instance, programs no longer denote ω-continuous functions, see also [7, 2]). Standard techniques [18] for proving the existence of recursively defined predicates over recursively defined domains are not immediately applicable.

Instead, our technical development takes place in a functor category so that the semantic domains are indexed by sets w of locations. Intuitively, w contains all the locations that are in use, and we can define a deterministic memory allocator. Non-determinism is not needed, due to the following observations:

- Every renaming $f : w \to w'$ between location sets gives rise to a corresponding transformation of programs. We can express, semantically, that two programs are equal up to a possible renaming of certain locations.
- We can identify a class of predicates (over stores) that are *invariant* under location renamings. This property captures the irrelevance of location names.

In contrast to previous uses of possible worlds models [24, 17, 15, 12] our semantics is not “tight” in the sense that stores may have allocated only a subset of the locations in w. Thus runtime errors are still possible by dereferencing dangling pointers. Memory faults are unavoidable because the language includes a *free*

Table 1. Syntax of expressions and commands

$x, y \in \textsc{Var}$		variables
$b, e \in \textsc{Exp} ::=$	$\mathsf{true} \mid \mathsf{false} \mid e_1{\le}e_2 \mid \neg b \mid b_1 \wedge b_2 \mid \ldots \mid$	boolean expressions
	$0 \mid -1 \mid 1 \mid \ldots \mid x \mid e_1 + e_2 \mid \ldots \mid$	integer expressions
	‘c’	quote (command as expression)
$c \in \textsc{Com} ::=$	$\mathsf{skip} \mid c_1;c_2 \mid \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi} \mid$	no op, composition, conditional
	$\mathsf{let}\ x = \mathsf{new}\ e\ \mathsf{in}\ c \mid \mathsf{free}\ x \mid$	memory allocation, disposal
	$[x]{:=}e \mid \mathsf{let}\ y{=}[x]\ \mathsf{in}\ c \mid$	assignment, lookup
	$\mathsf{eval}\ e$	unquote

operation that may create dangling pointers. Moreover, once stores are “taken apart” according to the separation conjunction, the concept of incomplete stores is convenient, even in the context of a statically typed language [20]. Nevertheless, with respect to the logic, *proved* programs do not yield memory faults.

In summary, we extend Separation Logic to higher-order store, thereby facilitating reasoning about code pointers. Technically, this is achieved by developing a functor category semantics that provides explicit location renamings, instead of using a non-determistic computation model. We believe this latter aspect is also of interest independently of the presence of higher-order store.

Structure of the paper. In Section 2 we present the syntax of programming language and logic, along with the proof rules. Section 3 develops the necessary background to interpret the language and logic, the semantics itself is given in Section 4. Section 5 concludes with an outlook on related and future work.

2 Programming Language and Logic

We present a variant of the language considered by Reus and Streicher [23], but extended with constructs for the dynamic allocation and disposal of memory cells. Two assumptions on the language simplify our presentation: First, we follow [5] in the slightly non-standard adoption of (ML-like) immutable identifiers. That is, all mutation takes place in the heap, whereas the stack variables are immutable. Second, expressions only depend on the stack but not on the heap.

2.1 Programming Language

The syntax of the language is given in Table 1. The set $\textsc{Exp}$ of expressions includes boolean and integer expressions. Additionally, a command c can be turned into an expression (delaying its execution), via the *quote* operation ‘c’.

The set $\textsc{Com}$ of commands consists of the usual no op, sequential composition, and conditional constructs. Because stack variables are not mutable, new memory is allocated by $\mathsf{let}\ x = \mathsf{new}\ e\ \mathsf{in}\ c$ that introduces an identifier with local scope c that is bound to (the location of) the new memory cell. We stress that the initial contents e may be a (quoted) command. This is also the case for an update, $[x]{:=}e$. The command $\mathsf{free}\ x$ disposes the memory cell that x denotes,

Table 2. Syntax of assertions

$A, B \in \textsc{pAssn} ::= \textbf{true} \mid e_1 \leq e_2 \mid$	pure basic predicates
$\neg A \mid A \wedge B \mid \forall x.\, A$	predicate logic connectives
$P, Q \in \textsc{Assn} ::= x \mapsto e \mid \textbf{emp} \mid P * Q \mid$	separation logic connectives
$A \mid P \wedge Q \mid P \vee Q \mid \forall x.\, P \mid \exists x.\, P$	predicate logic connectives

Table 3. Specific proof rules

Frame $\dfrac{\{P\}\, c\, \{Q\}}{\{P * R\}\, c\, \{Q * R\}}$	Free $\dfrac{}{\{x \mapsto \_\}\, \mathsf{free}\ x\, \{\textbf{emp}\}}$	New $\dfrac{\{P * x \mapsto e\}\, c\, \{Q\}}{\{P\}\, \mathsf{let}\ x = \mathsf{new}\ e\ \mathsf{in}\ c\, \{Q\}}\, x \notin fv(e, P, Q)$
Eval $\dfrac{\{P\}\, c\, \{Q\}}{\{P\}\, \mathsf{eval}\ \text{‘}c\text{’}\, \{Q\}}$	Assign $\dfrac{}{\{x \mapsto \_\}\, [x] := e\, \{x \mapsto e\}}$	Deref $\dfrac{\{P * x \mapsto e\}\, c[e/y]\, \{Q\}}{\{P * x \mapsto e\}\, \mathsf{let}\ y = [x]\ \mathsf{in}\ c\, \{Q\}}$
Rec $\dfrac{\bigwedge_{1 \leq i \leq n} \{P_1\}\, \mathsf{eval}\ x_1\, \{Q_1\} \ldots \{P_n\}\, \mathsf{eval}\ x_n\, \{Q_n\} \vdash \{P_i\}\, c_i\, \{Q_i\}}{\{P_j[\text{‘}\boldsymbol{c}\text{’}/\boldsymbol{x}]\}\, \mathsf{eval}\ \text{‘}c_j\text{’}\, \{Q_j[\text{‘}\boldsymbol{c}\text{’}/\boldsymbol{x}]\}}\, 1 \leq j \leq n$		

and let y=$[x]$ in c introduces a new stack variable y bound to the cell contents. Finally, eval e is the "unquote" command, i.e., if e denotes a quoted command c then c is executed. We give a formal semantics of this language in Section 4, after developing the necessary machinery in Section 3.

Note that Table 1 does not include any looping constructs – recursion can be expressed "through the store" [11]. Here is a simple example of a non-terminating command: $[x]$:=‘let y=$[x]$ in eval y’; let y=$[x]$ in eval y.

2.2 Assertions and Proof Rules

The assertions used in Hoare triples are built from the formulae of predicate logic and the additional separation logic assertions that describe the heap ($\_ \mapsto \_$, **emp**, and $P * Q$; cf. [25]). Note that in our language variables can also point to quoted code. The syntax of the assertions is given in Table 2. It is important to note that we distinguish between "pure" assertions pAssn, i.e., those that do not depend on the heap, and normal assertions Assn which do depend on the heap. Only the former allow negation. The reason for this will become clear when we give the semantics in Section 4.1. As usual, assertion $e_1 \leq e_2 \wedge e_2 \leq e_1$ is abbreviated $e_1 = e_2$ and assertion $\exists z.\, x \mapsto z$ is abbreviated to $x \mapsto \_$. For pure assertions pAssn the predicate **false** and connectives $\vee$, $\Rightarrow$, and $\exists x.A$ can be derived as usual using negation.

The inference rules of our program logic contain the standard Hoare rules (for skip, conditional, sequential composition, weakening and substitution), a standard axiomatization of predicate logic as well as an axiomatization of the Separation Logic connectives stating associativity and symmetry of $*$, neutrality of **emp** with respect to $*$, and some distributive laws (see e.g. [25]). The rules specific to our programming language are given in Table 3.

The frame rule extends triples by invariants for inaccessible parts of the heap. Rules (Free) and (Assign) specify the corresponding heap operations “tightly”. The inferences (New) and (Deref) combine heap allocation and dereferencing, resp., with local variable definitions (and hence are not tight). Unlike [5] our (New) permits (non-recursive) initializations (and self-reference can be introduced by assignment as in Section 2.3). Substitution on c is used in the premiss of (Deref) to avoid equations of the form $y = e$ that would be problematic when e is a stored procedure. Rule (Eval) is reminiscent of standard non-recursive procedure call rules; it expresses that evaluating a quoted command has the same effect as the command itself. Finally, (Rec) deals with recursion through the store. It is similar to the standard rule for Hoare-calculus with (mutually) recursive procedures, but since procedures are stored on the heap, they have to be accounted for in the assertions which leads to the substitution in the conclusion.

2.3 Example

The following command computes the sum $\Sigma n = \sum_{0\leq i\leq n} i$, where n is the value in the heap cell referred to by x, and stores the result at location y. Let c_P be

$$\mathsf{let}\;!y{=}\,[y]\;\mathsf{in}\;\mathsf{let}\;!x{=}\,[x]\;\mathsf{in}\;\mathsf{if}\;\;!x{\leq}0\;\;\mathsf{then}\;\;\mathsf{skip}\;\;\mathsf{else}\;\;[y]{:=}!y{+}!x;\,[x]{:=}!x{-}1;\,\mathsf{let}\,c{=}\,[f]\;\mathsf{in}\;\mathsf{eval}\;\;c\;\;\mathsf{fi}$$

and observe that $!x$ and $!y$ are (pointer) variable *names* representing the values in the cells denoted by x and y, respectively. The program is defined by recursion through the store: c_P calls the procedure stored in heap cell f, and if c_P is stored in cell f we have tied a “knot in the store.” We prove that this computes sums below omitting various applications of the weakening rule (which are easy to insert).

$$\textsc{New}\;\dfrac{;\;\dfrac{\begin{array}{c}\textsc{Frame}\;\dfrac{\textsc{Assign}\;\{f{\mapsto}\text{‘}\mathsf{skip}\text{’}\}\,[f]{:=}\text{‘}c_P\text{’}\,\{f{\mapsto}\text{‘}c_P\text{’}\}}{\{x{\mapsto}n * y{\mapsto}0 * f{\mapsto}\text{‘}\mathsf{skip}\text{’}\}\,[f]{:=}\text{‘}c_P\text{’}\,\{x{\mapsto}\_ * y{\mapsto}0 * f{\mapsto}\text{‘}c_P\text{’}\}} \\ \textsc{Deref}\;\dfrac{\textsc{Subst}\;\dfrac{\textsc{Rec}\;\dfrac{T}{\{x{\mapsto}n * y{\mapsto}m * f{\mapsto}\text{‘}c_P\text{’}\}\,\mathsf{eval}\;\text{‘}c_P\text{’}\,\{x{\mapsto}\_ * y{\mapsto}\Sigma n{+}m * f{\mapsto}\text{‘}c_P\text{’}\}}}{\{x{\mapsto}n * y{\mapsto}0 * f{\mapsto}\text{‘}c_P\text{’}\}\,\mathsf{eval}\;\text{‘}c_P\text{’}\,\{x{\mapsto}\_ * y{\mapsto}\Sigma n * f{\mapsto}\text{‘}c_P\text{’}\}}}{\{x{\mapsto}n * y{\mapsto}0 * f{\mapsto}\text{‘}c_P\text{’}\}\,\mathsf{let}\,c{=}\,[f]\;\mathsf{in}\;\mathsf{eval}\;\;c\,\{x{\mapsto}\_ * y{\mapsto}\Sigma n * f{\mapsto}\text{‘}c_P\text{’}\}}\end{array}}{\{x{\mapsto}n * y{\mapsto}0 * f{\mapsto}\text{‘}\mathsf{skip}\text{’}\}\,[f]{:=}\text{‘}c_P\text{’};\,\mathsf{let}\,c{=}\,[f]\;\mathsf{in}\;\mathsf{eval}\;\;c\,\{x{\mapsto}\_ * y{\mapsto}\Sigma n * f{\mapsto}\text{‘}c_P\text{’}\}}}{\{x{\mapsto}n * y{\mapsto}0\}\,\mathsf{let}\,f{=}\;\mathsf{new}\,\text{‘}\mathsf{skip}\text{’}\;\mathsf{in}\;[f]{:=}\text{‘}c_P\text{’};\,\mathsf{let}\,c{=}\,[f]\;\mathsf{in}\;\mathsf{eval}\;\;c\,\{x{\mapsto}\_ * y{\mapsto}\Sigma n * f{\mapsto}\text{‘}c_P\text{’}\}}$$

For the derivation tree T we assume

$$(\dagger)\qquad \{x{\mapsto}n * y{\mapsto}m * f{\mapsto}x_P\}\,\mathsf{eval}\;\;x_P\,\{x{\mapsto}n * y{\mapsto}\Sigma n{+}m * f{\mapsto}x_P\}$$

and prove $\{x{\mapsto}n * y{\mapsto}m * f{\mapsto}x_P\}\,c_P\,\{x{\mapsto}0 * y{\mapsto}\Sigma m{+}n * f{\mapsto}x_P\}$.

$$\textsc{Deref}^2\;\dfrac{\textsc{If}\;\dfrac{I_t \qquad I_f}{\{x{\mapsto}n * y{\mapsto}m * f{\mapsto}x_P\}\,\mathsf{if}\;n{\leq}0\;\mathsf{then\;skip\;else}\;\ldots\;\mathsf{fi}\,\{x{\mapsto}\_ * y{\mapsto}\Sigma n{+}m * f{\mapsto}x_P\}}}{\{x{\mapsto}n * y{\mapsto}m * f{\mapsto}x_P\}\,\underbrace{\mathsf{let}\;!y{=}\,[y]\;\mathsf{in}\;\ldots[y]{:=}!y{+}!x\ldots}_{c_P}\{x{\mapsto}\_ * y{\mapsto}\Sigma n{+}m * f{\mapsto}x_P\}}$$

where I_t and I_f, respectively, are:

$$\textsc{Weak}\ \dfrac{\textsc{Skip}\ \{x{\mapsto}n * y{\mapsto}m * f{\mapsto}x_P \wedge n \leq 0\}\,\mathsf{skip}\,\{x{\mapsto}n * y{\mapsto}m * f{\mapsto}x_P \wedge n \leq 0\}}{\{x{\mapsto}n * y{\mapsto}m * f{\mapsto}x_P \wedge n \leq 0\}\,\mathsf{skip}\,\{x{\mapsto}\_ * y{\mapsto}\Sigma n{+}m * f{\mapsto}x_P\}}$$

$$;^2\ \dfrac{\alpha \qquad \beta \qquad \gamma}{\{x{\mapsto}n * y{\mapsto}m * f{\mapsto}x_P \wedge n{>}0\}\ [y]{:=}\ldots; [x]{:=}\ldots; \mathsf{let}\ldots\{x{\mapsto}\_ * y{\mapsto}\Sigma n{+}m * f{\mapsto}x_P\}}$$

α, β and γ are as follows:

$$\textsc{Frame}\ \dfrac{\textsc{Assign}\ \{y{\mapsto}m\}\,[y]{:=}m{+}n\,\{y{\mapsto}m{+}n\}}{\{x{\mapsto}n * y{\mapsto}m * f{\mapsto}x_P\}\,[y]{:=}m{+}n\,\{x{\mapsto}n * y{\mapsto}m{+}n * f{\mapsto}x_P\}}$$

$$\textsc{Frame}\ \dfrac{\textsc{Assign}\ \{x{\mapsto}n\}\,[x]{:=}n{-}1\,\{x{\mapsto}n{-}1\}}{\{x{\mapsto}n * y{\mapsto}m{+}n * f{\mapsto}x_P\}\,[x]{:=}n{-}1\,\{x{\mapsto}n{-}1 * y{\mapsto}m{+}n * f{\mapsto}x_P\}}$$

$$\textsc{Deref}\ \dfrac{\textsc{Subst}\ \dfrac{(\dagger) \equiv \{x{\mapsto}n * y{\mapsto}m * f{\mapsto}x_P\}\,\mathsf{eval}\ x_P\,\{x{\mapsto}\_ * y{\mapsto}\Sigma n * f{\mapsto}x_P\}}{\{x{\mapsto}n{-}1 * y{\mapsto}m{+}n * f{\mapsto}x_P\}\,\mathsf{eval}\ \ x_P\,\{x{\mapsto}\_ * y{\mapsto}\Sigma n{+}m * f{\mapsto}x_P\}}}{\{x{\mapsto}n{-}1 * y{\mapsto}m{+}n * f{\mapsto}x_P\}\,\mathsf{let}\,c{=}\,[f]\,\mathsf{in\ eval}\ \ c\,\{x{\mapsto}\_ * y{\mapsto}\Sigma n{+}m * f{\mapsto}x_P\}}$$

Note how the Frame Rule is used to peel off those predicates of the assignment rule that do not relate to the memory cell affected.

3 A Model of Dynamic Higher-order Store

This section defines the semantic domains in which the language of Section 2 finds its interpretation. The semantic properties (*safety monotonicity* and *frame property*) that programs must satisfy to admit local reasoning [25] are re-formulated, using the renamings made available by the functor category machinery. Due to the higher-order character of stores, these predicates are recursive and their existence must be established. The framework of Pitts is used [18, 12].

3.1 Worlds

Fix a well-ordered, countably infinite set $\mathbb{L}$ of *locations* (e.g., the natural numbers). Let $\mathbb{W}$ be the category consisting of finite subsets $w \subseteq \mathbb{L}$ as objects and injections $f : w_1 \to w_2$ as morphisms. We call the objects w of $\mathbb{W}$ *worlds.* The intuition is that $w \in \mathbb{W}$ describes (a superset of) the locations currently in use; in particular, every location *not* in w will be fresh. The inclusion $w \subseteq w'$ is written $\iota_w^{w'}$, and the notation $f : w_1 \xrightarrow{\sim} w_2$ is used to indicate that f is a bijection.

The injections formalise a possible *renaming* of locations, as well as an *extension* of the set of available locations because of allocation.

3.2 Semantic Domains: Stores, Values and Commands

Let **pCpo** be the category of cpos (partial orders closed under taking least upper bounds of countable chains, but not necessarily containing a least element) and partial continuous functions. For a partial continuous function g we write $g(a)\downarrow$ if the application is defined, and $g(a)\uparrow$ otherwise. By $g;h$ we denote composition in diagrammatic order. Let **Cpo** be the subcategory of **pCpo** where the morphisms are *total* continuous functions. For cpos A and B we write $A \rightharpoonup B$ and $A \to B$ for the cpos of partial and total continuous functions from A to B, respectively, each with the pointwise ordering. For a family (A_i) of cpos, $\sum_i A_i$ denotes their disjoint union; we write its elements as $\langle i, a\rangle$ where $a \in A_i$.

For every $w \in \mathbb{W}$ we define a cpo of *w-stores* as records of values whose domain is a subset of w (viewed as discrete cpo). The fields of such a store contain values that may refer to locations in w:

$$St(w) = \mathsf{Rec}_w(Val(w)) = \sum_{w' \subseteq w} (w' \to Val(w)) \tag{1}$$

For $s = \langle w', s\rangle \in St(w)$ we set $\mathsf{dom}(s) = w'$ and may use the record notation $\{\!|l = v_l|\!\}_{l \in \mathsf{dom}(s)}$ where $s(l) = v_l$. The order on (1) is defined in the evident way, by $r \sqsubseteq s$ if and only if $\mathsf{dom}(r) = \mathsf{dom}(s)$ and $r(l) \sqsubseteq s(l)$ for all $l \in \mathsf{dom}(r)$.

A value (over locations $w \in \mathbb{W}$) is either a *basic value* in $BVal$, a location $l \in w$, or a *command*, i.e.,

$$Val(w) = BVal + w + Com(w) \tag{2}$$

We assume $BVal$ is a discretely ordered cpo that contains integers and booleans.

Commands $c \in Com(w)$ operate on the store; given an initial store the command may either diverge, terminate abnormally or terminate with a result store. Abnormal termination is caused by dereferencing dangling pointers which may refer to memory locations that either have not yet been allocated, or have already been disposed of. Thus, in contrast to [23] where store could not vary dynamically, we need to have a defined result error to flag undefined memory access. The possibility of *dynamic memory allocation* prompts a further refinement compared to [23]: a command should work for extended stores, too, and may also extend the store itself.

Formally, the collection of commands is given as a functor $Com : \mathbb{W} \longrightarrow \mathbf{Cpo}$, defined on objects by

$$Com(w) = \prod_{i:w \to w'} (St(w') \rightharpoonup (\mathsf{error} + \sum_{j:w' \to w''} St(w''))) \tag{3}$$

and on morphisms by the obvious restriction of the product,

$$Com(f : w_1 \to w_2)(c)_{i:w_2 \to w_3} = c_{(f;i)}$$

Viewing commands this way is directly inspired by Levy's model of an ML-like higher-order language with general references [12].

By considering $BVal$ as constant functor, and locations as the functor $\mathbb{W} \longrightarrow$ **Cpo** that acts on $f : w_1 \to w_2$ by sending $l \in w_1$ to $f(l) \in w_2$, Val can also be seen as a functor $\mathbb{W} \longrightarrow$ **Cpo**. Note that, by expanding the requirements (1), (2) and (3), Val is expressed in terms of a mixed-variant recursion. In Section 3.3 we address the issue of well-definedness of Val.

One might want to exclude run-time memory errors statically (which is possible assuming memory is never disposed, so that there is no way of introducing dangling pointers from within the language). An obvious solution to model this is by defining w-stores as $\prod_w Val(w)$, i.e., all locations occurring in values are guaranteed to exist in the current store. However, this approach means that there is no canonical way to extend stores, nor can values be restricted to smaller location sets. Consequently St is neither co- nor contravariantly functorial[1]. In contrast, our more permissive definition of stores (that may lead to access errors) *does* allow a functorial interpretation of St, as follows. For an injection $f : w_1 \to w_2$ we write $f^{-1} : \mathsf{im} f \to w_1$ for the right-inverse to f, and let

$$St(f) : \mathsf{Rec}_{w_1}(Val(w_1)) \to \mathsf{Rec}_{w_2}(Val(w_2))$$
$$St(f) = \lambda \langle w \subseteq w_1, s \rangle.\, \langle fw, f^{-1}; s; Val(f) \rangle$$

The case where f is a bijection then corresponds to a consistent renaming of the store and its contents. We will make some use of the functoriality of St in the following, to lift recursively defined predicates from values to stores.

For $s_1, s_2 \in St(w)$ we write $s_1 \perp s_2$ if their respective domains $w_1, w_2 \subseteq w$ are disjoint. In this case, their composition $s_1 * s_2 \in St(w)$ is defined by conjoining them in the obvious way, it is undefined otherwise. Observe that for $f : w \to w'$ we have $St(f)(s_1 * s_2) = St(f)(s_1) * St(f)(s_2)$; the right-hand side is defined because f is injective.

3.3 Domain Equations and Relational Structures on Bilimit-Compact Categories

This section briefly summarises the key results from [12, 27] about the solution of recursive domain equations in bilimit-compact categories. We will make use of the generalisation of Pitts' techniques [18] for establishing the well-definedness of (recursive) predicates, as outlined in [12].

Definition 1 (Bilimit-Compact Category [12]). A category $\mathbb{C}$ is *bilimit-compact* if

- $\mathbb{C}$ is **Cpo**-enriched and each hom-cpo $\mathbb{C}(A, B)$ has a least element $\perp_{A,B}$ such that $\perp \circ f = \perp = g \circ \perp$;
- $\mathbb{C}$ has an initial object; and
- in the category $\mathbb{C}^E$ of embedding-projection pairs of $\mathbb{C}$, every ω-chain $\Delta = D_0 \to D_1 \to \ldots$ has an O-colimit [27]. More precisely, there exists a cocone $(e_n, p_n)_{n<\omega} : \Delta \to D$ in $\mathbb{C}^E$ such that $\sqcup_{n<\omega}(p_n; e_n) = id_D$ in $\mathbb{C}(D, D)$.

[1] Levy [12] makes this observation for a similar, typed store model.

Table 4. Solving the domain equation: $F_{\mathit{Val}}, F_{\mathit{Com}} : \mathbb{C}^{op} \times \mathbb{C} \longrightarrow \mathbb{C}$ and $F_{\mathit{St}} : \mathbb{C} \longrightarrow \mathbb{C}$

On $\mathbb{C}$-objects A^-, A^+, B, worlds $w, w' \in \mathbb{W}$ and $f : w \to w'$,

$$\begin{aligned}
F_{\mathit{Val}}(A^-, A^+)(w) &= \mathit{BVal} + w + F_{\mathit{Com}}(A^-, A^+)(w) \\
F_{\mathit{Val}}(A^-, A^+)(f) &= \lambda v. \begin{cases} v & \text{if } v \in \mathit{BVal} \\ f(v) & \text{if } v \in w \\ F_{\mathit{Com}}(A^-, A^+)(f)(v) & \text{if } v \in F_{\mathit{Com}}(A^-, A^+)(w) \end{cases} \\
F_{\mathit{Com}}(A^-, A^+)(w) &= \textstyle\prod_{i:w\to w'} (F_{\mathit{St}}(A^-)(w') \rightharpoonup (\mathsf{error} + \sum_{j:w'\to w''} F_{\mathit{St}}(A^+)(w''))) \\
F_{\mathit{Com}}(A^-, A^+)(f) &= \lambda c \lambda i.\, c_{(f;i)}
\end{aligned}$$

$$\begin{aligned}
F_{\mathit{St}}(B)(w) &= \textstyle\sum_{w_1 \subseteq w} (w_1 \to B(w)) \\
F_{\mathit{St}}(B)(f) &= \lambda \langle w_1, s \rangle.\, \langle f w_1, f^{-1}; s; B(f) \rangle
\end{aligned}$$

On $\mathbb{C}$-morphisms $h = (h_w) : B^- \stackrel{.}{\rightharpoonup} A^-$ and $k = (k_w) : A^+ \stackrel{.}{\rightharpoonup} B^+$,

$$\begin{aligned}
F_{\mathit{Val}}(h, k)_w &= \lambda v. \begin{cases} v & \text{if } v \in \mathit{BVal} \text{ or } v \in w \\ F_{\mathit{Com}}(h, k)_w(v) & \text{if } v \in F_{\mathit{Com}}(A^-, A^+) \end{cases} \\
F_{\mathit{Com}}(h, k)_w &= \lambda c \lambda i{:}w \to w' \lambda s. \\
&\begin{cases} \text{undefined} & \text{if } F_{\mathit{St}}(h)_{w'}(s)\uparrow \\ & \text{or } F_{\mathit{St}}(h)_{w'}(s)\downarrow\ \wedge c_i(F_{\mathit{St}}(h)_{w'}(s))\uparrow \\ & \text{or } c_i(F_{\mathit{St}}(h)_{w'}(s)) = \langle j : w' \to w'', s' \rangle\ \wedge\ F_{\mathit{St}}(k)_{w''}(s')\uparrow \\ \mathsf{error} & \text{if } F_{\mathit{St}}(h)_{w'}(s)\downarrow\ \wedge c_i(F_{\mathit{St}}(h)_{w'}(s)) = \mathsf{error} \\ \langle j, F_{\mathit{St}}(k)_{w''}(s') \rangle & \text{if } c_i(F_{\mathit{St}}(h)_{w'}(s)) = \langle j : w' \to w'', s' \rangle\ \wedge\ F_{\mathit{St}}(k)_{w''}(s')\downarrow \end{cases} \\
F_{\mathit{St}}(k)_w &= \lambda \langle w_1, s \rangle. \begin{cases} \langle w_1, s; k_w \rangle & \text{if } \forall l \in w_1.\, k_w(s(l))\downarrow \\ \text{undefined} & \text{otherwise} \end{cases}
\end{aligned}$$

It follows that every locally continuous functor $F : \mathbb{C}^{op} \times \mathbb{C} \longrightarrow \mathbb{C}$ has a minimal invariant, i.e., an object D and isomorphism i in $\mathbb{C}$ such that $i : F(D, D) \cong D$ (unique up to unique isomorphism) and id_D is the least fixed point of the continuous endofunction $\delta : \mathbb{C}(D, D) \to \mathbb{C}(D, D)$ defined by $\delta(e) = i^{-1}; F(e, e); i$ [18]. To ease readability the isomorphism i is usually omitted below.

To solve the domain equation of the preceding subsection we shall be interested in the case where $\mathbb{C}$ denotes the category $[\mathbb{W}, \mathbf{Cpo}]$ of functors $\mathbb{W} \longrightarrow \mathbf{Cpo}$ and *partial* natural transformations, i.e., a morphism $e : A \stackrel{.}{\rightharpoonup} B$ in $\mathbb{C}$ is a family $e = (e_w)$ of partial continuous functions $e_w : A(w) \rightharpoonup B(w)$ such that $A(f); e_{w'} = e_w; B(f)$ for all $f : w \to w'$.

Lemma 2 (Bilimit-Compactness [12]). $\mathbb{C} = [\mathbb{W}, \mathbf{Cpo}]$ *is bilimit-compact.*

Thus, for well-definedness of *Val* it suffices to show that requirements (1), (2) and (3) induce a locally continuous functor $\mathbb{C}^{op} \times \mathbb{C} \longrightarrow \mathbb{C}$ for which *Val* is the minimal invariant. Table 4 defines such a functor F_{Val} in the standard way [18], by separating positive and negative occurrences of *St* in (3).

Lemma 3 (Minimal Invariant). $F_{\mathit{Val}} : \mathbb{C}^{op} \times \mathbb{C} \longrightarrow \mathbb{C}$ *is locally continuous. In particular, the minimal invariant* $\mathit{Val} = F_{\mathit{Val}}(\mathit{Val}, \mathit{Val})$ *exists.*

From this we can then define $\mathit{Com} = F_{\mathit{Com}}(\mathit{Val}, \mathit{Val})$ and $\mathit{St} = F_{\mathit{St}}(\mathit{Val})$ which satisfy (3) and (1). The minimal invariant in fact lives in the category of functors $\mathbb{W} \longrightarrow \mathbf{Cpo}$ and natural transformations that are *total*, i.e., those $e = (e_w) : A \stackrel{.}{\to} B$ where each e_w is a total continuous function $A(w) \to B(w)$. This is because F_{Val} restricts to this category, which is sub-bilimit-compact within $\mathbb{C}$ in the sense of [12]. A (normal) relational structure $\mathcal{R}$ on $\mathbb{C}$ in the sense of Pitts [18] is given as follows.

Definition 4 (Kripke Relational Structure). For each $A : \mathbb{W} \longrightarrow \mathbf{Cpo}$ let $\mathcal{R}(A)$ consist of the $\mathbb{W}$-indexed families $R = (R_w)$ of admissible predicates $R_w \subseteq A(w)$ such that for all $f : w \to w'$ and $a \in A(w)$,

$$a \in R_w \quad \Longrightarrow \quad A(f)(a) \in R_{w'} \qquad (\textsc{KripkeMon})$$

For each natural transformation $e = (e_w) : A \stackrel{.}{\rightharpoonup} B$ and all $R \in \mathcal{R}(A)$, $S \in \mathcal{R}(B)$,

$$e : R \subset S \quad :\Longleftrightarrow \quad \forall w \in \mathbb{W}\, \forall a \in A(w).\; a \in R_w \;\wedge\; e_w(a){\downarrow} \Longrightarrow\; e_w(a) \in S_w$$

Note that ($\textsc{KripkeMon}$) in particular covers the case where $f : w \to w$ is a bijection, i.e., the Kripke relations are invariant under permutation of locations.

For an object $A : \mathbb{W} \longrightarrow \mathbf{Cpo}$ and $\mathcal{R}$-relation $R = (R_w) \in \mathcal{R}(A)$ we let $\mathit{St}(R) \in \mathcal{R}(F_{\mathit{St}}(A))$ be the relation where $s \in \mathit{St}(R)_w$ if and only if $s(l) \in R_w$ for all $l \in \mathsf{dom}(s)$. It is easy to check admissibility and ($\textsc{KripkeMon}$). Two elementary properties are stated in the following lemma.

Lemma 5 (Relations over *St*). *Let $A, B : \mathbb{W} \longrightarrow \mathbf{Cpo}$, $R \in \mathcal{R}(A)$, $S \in \mathcal{R}(B)$. Let $e : A \stackrel{.}{\to} B$ and $w \in \mathbb{W}$.*

1. *If $e : R \subset S$ then $F_{\mathit{St}}(e) : \mathit{St}(R) \subset \mathit{St}(S)$.*
2. *If $s_1, s_2 \in F_{\mathit{St}}(A)(w)$ and $s_1 \perp s_2$ then $s_1 \in \mathit{St}(R)_w$ and $s_2 \in \mathit{St}(R)_w$ if and only if $s_1 * s_2 \in \mathit{St}(R)_w$.*

Theorem 6 (Invariant Relation [18]). *Let F_{Val} be the locally continuous functor for which Val is the minimal invariant. Suppose Φ maps $\mathcal{R}$-relations to $\mathcal{R}$-relations such that for all $R, R', S, S' \in \mathcal{R}(\mathit{Val})$ and $e \sqsubseteq \mathit{id}_{\mathit{Val}}$,*

$$e : R' \subset R \;\wedge\; e : S \subset S' \quad \Longrightarrow \quad F(e,e) : \Phi(R,S) \subset \Phi(R',S')$$

Then there exists a unique $\Delta \in \mathcal{R}(\mathit{Val})$ such that $\Phi(\Delta,\Delta) = \Delta$.

Proof. By [12], the proof of Pitts' existence theorem [18, Thm. 4.16] generalises from **Cppo** (pointed cpos and strict continuous maps) to arbitrary bilimit-compact categories. Since the $\mathcal{R}$-relations of Definition 4 are admissible in the sense of [18] and $\mathcal{R}$ has inverse images and intersections, the theorem follows. □

3.4 Safety Monotonicity and Frame Property

Safety monotonicity is the observation that if executing a command in heap h does not result in a memory fault, then this is also true when running the command in a heap that extends h. The second key semantic principle underlying separation logic is the observation that if execution of a command does not result in a memory fault (i.e., no dangling pointers are dereferenced), then running the command in an extended heap does not influence its observable behaviour — in particular, the additional heap region remains unaffected. The *frame property* [25] formalises this idea. Since the actual results of these executions may differ in the action of the memory allocator, the choice of locations is taken into account.

As the store may contain commands itself (which may be called in an execution), both safety monotonicity and frame property must already be *required* to hold of the data in the initial store. In order to give a sufficiently strong induction hypothesis later, we additionally require that the properties are *preserved* by the execution of commands. Unfortunately we cannot adopt separate definitions for safety monotonicity and the frame property (like [25]) but have to *combine* them. The reason is that safety monotonicity is not preserved by composition of commands, unless commands additionally satisfy the frame property.[2] Because of the higher-order store, both properties are expressed by mixed-variant recursive definitions, and existence of a predicate satisfying these definitions requires a proof. It is in this proof that both properties are needed simultaneously.

For this reason the following property LC (for "local commands") is proposed, subsuming both safety and frame property: For $R, S \in \mathcal{R}(\mathit{Val})$ let $\Phi(R,S)$ be the $\mathbb{W}$-indexed family of relations where

$$
\begin{array}{l}
c \in \Phi(R,S)_w \quad :\Longleftrightarrow \quad c \in \mathit{Com}(w) \implies \\
\quad \forall f{:}w\to w_2\,\forall i{:}w_2\overset{\sim}{\to}w_2'\,\forall g{:}w_2\to w_3\,\forall s_1,s_2\in \mathit{St}(R)_{w_2}\,\forall s'\in \mathit{St}(w_3).\ s_1\perp s_2 \implies \\
\qquad \left.\begin{array}{l}
c_f(s_1)\uparrow \implies c_{f;i}(\mathit{St}(i)(s_1 * s_2))\uparrow \\
\wedge\ c_f(s_1 * s_2) = \mathsf{error} \implies c_{f;i}(\mathit{St}(i)(s_1)) = \mathsf{error}
\end{array}\right\}\ \textit{safety mon.} \\
\qquad \left.\begin{array}{l}
\wedge\ c_f(s_1) \neq \mathsf{error} \wedge c_f(s_1 * s_2) = \langle g, s'\rangle \implies \\
\quad \exists g'{:}w_2'\to w_3'\,\exists j{:}w_3'\overset{\sim}{\to}w_3\,\exists s_1'\in \mathit{St}(w_3')\,\exists s_2'\in \mathit{St}(w_2). \\
\qquad c_{f;i}(\mathit{St}(i)(s_1)) = \langle g', s_1'\rangle\ \wedge\ s_2' \sqsubseteq s_2\ \wedge\ i;g';j = g \\
\qquad \wedge\ s' = \mathit{St}(j)(s_1' * \mathit{St}(i;g')(s_2'))\ \wedge\ s' \in \mathit{St}(S)_{w_3}
\end{array}\right\}\ \textit{frame property}
\end{array}
$$

and define the predicate $LC \in \mathcal{R}(\mathit{Val})$ on values as the fixpoint $LC = \Phi(LC, LC)$ of this functional.

This definition is complex so some remarks are in order. Besides combining safety and frame property, Φ strengthens the obvious requirements by allowing the use of a renaming i on the initial store as well. This provides a strong invariant that we need for the proof of Theorem 8 below in the case of sequential composition. To obtain the fixed point of Φ, Lemma 7 appeals to Theorem 6 which forced us to weaken the frame property to an inequality ($s_2' \sqsubseteq s_2$). This extends conservatively the usual notion of [25] to the case of higher-order stores.

Lemma 7 (Existence). *LC is well-defined, i.e., there exists a unique $LC \in \mathcal{R}(\mathit{Val})$ such that $LC = \Phi(LC, LC)$.*

[2] As pointed out to us by Hongseok Yang, this is neither a consequence of using a denotational semantics, nor of our particular formulation employing renamings rather than non-determinism; counter-examples can easily be constructed in a relational interpretation of commands.

Proof. One checks that Φ maps Kripke relations to Kripke relations, i.e. for all $R, S \in \mathcal{R}(\mathit{Val})$, $\Phi(R, S) \in \mathcal{R}(\mathit{Val})$. By Theorem 6 it remains to show for all $e \sqsubseteq \mathit{id}_{\mathit{Val}}$, if $e : R' \subset R$ and $e : S \subset S'$ then $F_{\mathit{Val}}(e, e) : \Phi(R, S) \subset \Phi(R', S')$. □

4 Semantics of Programs and Logic

Table 5 contains the interpretation of the language. Commands and expressions depend on environments because of free (stack) variables, so that $\mathcal{E}\,[\![e]\!] : \mathit{Env} \dot{\to} \mathit{Val}$ and $\mathcal{C}\,[\![c]\!] : \mathit{Env} \dot{\to} \mathit{Com}$ where the functor Env is $\mathit{Val}^{\textsc{Var}}$. The semantics of boolean and integer expressions is standard and omitted from Table 5; because of type mismatches (negation of integers, addition of booleans,. . .) expressions may denote error. The semantics of quote refers to the interpretation of commands and uses the injection of Com into Val. Sequential composition is interpreted by composition in the functor category but also propagates errors and non-termination. Conditional and skip are standard. The semantics of the memory commands is given in terms of auxiliary operations *extend* and *update*.

The following theorem shows the main result about the model: commands of the above language satisfy (and preserve) the locality predicate LC.

Theorem 8 (Locality). *Let $w \in \mathbb{W}$ and $\rho \in \mathit{Env}(w)$ such that $\rho(x) \in LC_w$ for all $x \in \textsc{Var}$. Let $c \in \textsc{Com}$. Then $[\![c]\!]_w\, \rho \in LC_w$.*

Proof. By induction on c. The case of sequential composition relies on LC taking safety monotonicity and frame property into account simultaneously. □

4.1 Interpretation of the Logic

The assertions of the logic are interpreted as predicates over St that are compatible with the possible-world structure. In contrast to the $\mathcal{R}$-relations of Section 3.3 they depend on environments, and downward-closure is required to prove the frame rule sound. This is made precise by the following relational structure $\mathcal{S}$.

Definition 9 (dclKripke Relational Structure). Let $\mathcal{S}$ consist of the $\mathbb{W}$-indexed families $p = (p_w)$ of predicates $p_w \subseteq \mathit{Env}(w) \times \mathit{St}(w)$ such that for all $f : w \to w'$, $\rho \in \mathit{Env}(w)$ and $s \in \mathit{St}(w)$,

Kripke Monotonicity. if $(\rho, s) \in p_w$ then $(\mathit{Env}(f)(\rho), \mathit{St}(f)(s)) \in p_{w'}$;
Downward Closure. $\{s \in \mathit{St}(w) \mid (\rho, s) \in p_w\}$ is downward-closed in $\mathit{St}(w)$.

For each natural transformation $e = (e_w) : \mathit{Val} \dot{\rightharpoonup} \mathit{Val}$ and $p, q \in \mathcal{S}$ we write $e : p \subset q$ if for all $w \in \mathbb{W}$, $\rho \in \mathit{Env}(w)$ and $s \in \mathit{St}(w)$,

$$(\rho, a) \in p_w \;\wedge\; (F_{\mathit{Env}}(e)_w(\rho)\!\downarrow \vee F_{\mathit{St}}(e)_w(s)\!\downarrow) \implies (F_{\mathit{Env}}(e)_w(\rho), F_{\mathit{St}}(e)_w(s)) \in q_w$$

where $F_{\mathit{Env}}(e) = {F_{\mathit{Val}}}^{\textsc{Var}}(e, e)$.

Table 5. Semantics of expressions and commands

$$
\begin{array}{ll}
\mathcal{E}\llbracket e \rrbracket : \mathit{Env} \dot{\to} \mathit{Val} + \mathsf{error} & \text{where } f : w \to w' \\
\mathcal{E}\llbracket \text{`}c\text{'} \rrbracket_w \rho & = \mathcal{C}\llbracket c \rrbracket_w \rho \\
\mathcal{C}\llbracket c \rrbracket : \mathit{Env} \dot{\to} \mathit{Com} & \text{where } f : w \to w' \text{ and } s \in \mathit{St}(w') \\
(\mathcal{C}\llbracket \mathsf{skip} \rrbracket_w \rho)_f(s) & = \langle id, s \rangle \\
(\mathcal{C}\llbracket c_1;c_2 \rrbracket_w \rho)_f(s) & = \begin{cases} \text{undefined} & \text{if } (\mathcal{C}\llbracket c_1 \rrbracket_w \rho)_f s \uparrow \\ \mathsf{error} & \text{if } (\mathcal{C}\llbracket c_1 \rrbracket_w \rho)_f s = \mathsf{error} \\ (\mathcal{C}\llbracket c_2 \rrbracket_w \rho)_{(f;g)} s' & \text{if } (\mathcal{C}\llbracket c_1 \rrbracket_w \rho)_f s = \langle g, s' \rangle \end{cases} \\
(\mathcal{C}\llbracket \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \rrbracket_w \rho)_f(s) & = \begin{cases} (\mathcal{C}\llbracket c_1 \rrbracket_w \rho)_f s & \text{if } \mathcal{E}\llbracket b \rrbracket_w \rho = \mathit{true} \\ (\mathcal{C}\llbracket c_2 \rrbracket_w \rho)_f s & \text{if } \mathcal{E}\llbracket b \rrbracket_w \rho = \mathit{false} \\ \mathsf{error} & \text{otherwise} \end{cases} \\
(\mathcal{C}\llbracket \mathsf{let}\ x{=}\mathsf{new}\ e\ \mathsf{in}\ c \rrbracket_w \rho)_f(s) & = \text{let } l = \min\{l \in \mathbb{L} \mid l \notin w\},\ s' = \mathit{extend}_{w',l}(\mathit{Val}(f)(\mathcal{E}\llbracket e \rrbracket_w \rho), s) \\
& \quad \text{in } (\mathcal{C}\llbracket c \rrbracket_{w' \cup \{l\}} (\mathit{Env}(f; \iota_{w'}^{w' \cup \{l\}})(\rho))[x := l])_{id} s'; \mathit{shift}_{(f;\iota_{w'}^{w' \cup \{l\}})} \\
(\mathcal{C}\llbracket \mathsf{free}\ x \rrbracket_w \rho)_f(s) & = \begin{cases} \langle id, \{\!|l' = s(l')|\!\}_{l' \in \mathsf{dom}(s),\, l' \neq l} \rangle & \\ \qquad \text{if } \exists l \in w.\, \mathcal{E}\llbracket x \rrbracket_w \rho = l \text{ and } f(l) \in \mathsf{dom}(s) & \\ \mathsf{error} \quad \text{otherwise} & \end{cases} \\
(\mathcal{C}\llbracket [x]{:=}e \rrbracket_w \rho)_f(s) & = \begin{cases} \langle id, \mathit{update}_{w'}(f(l),\ s,\ \mathit{Val}(f)(\mathcal{E}\llbracket e \rrbracket_w \rho)) \rangle & \\ \qquad \text{if } \exists l \in w.\, \mathcal{E}\llbracket x \rrbracket_\rho = l \text{ and } f(l) \in \mathsf{dom}(s) & \\ \qquad \text{and } \mathcal{E}\llbracket e \rrbracket_w \rho \in \mathit{Val}(w) & \\ \mathsf{error} \quad \text{otherwise} & \end{cases} \\
(\mathcal{C}\llbracket \mathsf{let}\ y = [x]\ \mathsf{in}\ c \rrbracket_w \rho)_f(s) & = \begin{cases} (\mathcal{C}\llbracket c \rrbracket_{w'} (\mathit{Env}(f)(\rho))[y := s(f(l))])_{id}(s); \mathit{shift}_f & \\ \qquad \text{if } \exists l \in w.\, \mathcal{E}\llbracket x \rrbracket_w \rho = l \text{ and } f(l) \in \mathsf{dom}(s) & \\ \mathsf{error} \quad \text{otherwise} & \end{cases} \\
(\mathcal{C}\llbracket \mathsf{eval}\ e \rrbracket_w \rho)_f(s) & = \begin{cases} (\mathcal{E}\llbracket e \rrbracket_w \rho)_f s & \text{if } \mathcal{E}\llbracket e \rrbracket_w \rho \in \mathit{Com}(w) \\ \mathsf{error} & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{l}
\mathit{shift}_f : (\mathsf{error} + \textstyle\sum_{g' : w' \to w''} \mathit{St}(w'')) \to (\mathsf{error} + \textstyle\sum_{g : w \to w''} \mathit{St}(w'')) \\
\mathit{shift}_f(v) \quad = \begin{cases} \mathsf{error} & \text{if } v = \mathsf{error} \\ \langle f; g', s' \rangle & \text{if } v = \langle g', s' \rangle \end{cases} \\
\mathit{extend}_{w,l} : \mathit{Val}(w) \times \mathit{St}(w) \to \mathit{St}(w \uplus \{l\}) \\
\mathit{extend}_{w,l}(v,s) \ = \mathit{St}(\iota_w^{w \cup \{l\}})(s) * \{\!|l = \mathit{Val}(\iota_w^{w \cup \{l\}})(v)|\!\} \\
\mathit{update}_w : w \times \mathit{St}(w) \times \mathit{Val}(w) \to \mathit{St}(w) \\
\mathit{update}_w(l,s,v) = \{\!|l{=}v|\!\} * \{\!|l'{=}s(l')|\!\}_{l' \in \mathsf{dom}(s), l' \neq l}
\end{array}
$$

Assertions $P \in \textsc{Assn}$ are interpreted by $\mathcal{S}$-relations $\mathcal{A}\llbracket P \rrbracket$. Some cases of the definition are given in Table 6. All assertions are indeed downward-closed in the store component, and that pure assertions denote either true or false since they do not depend on the heap. The interpretation shows that $\leq$ is not supposed to compare code (but yield false instead). Correspondingly, we assume the non-standard axiom $\neg(\text{`}c\text{'}_1 \leq e_2) \wedge \neg(e_1 \leq \text{`}c\text{'}_2)$ for the comparison operator.

We can now give the semantics of Hoare triples. Correctness is only ensured if the command in question is run on stores that contain local procedures only.

Table 6. Interpretation of assertions

$\mathcal{A}\llbracket P\rrbracket : \mathcal{S}$	
$(\rho, s) \in \mathcal{A}\llbracket \mathbf{true} \rrbracket_w$	$:\Leftrightarrow$ always
$(\rho, s) \in \mathcal{A}\llbracket e_1 \leq e_2 \rrbracket_w$	$:\Leftrightarrow \mathcal{E}\llbracket e_i \rrbracket_w \rho \notin \mathit{Com}(w) \ \wedge\ \mathcal{E}\llbracket e_1 \rrbracket_w \rho \ \leq\ \mathcal{E}\llbracket e_2 \rrbracket_w \rho$
$(\rho, s) \in \mathcal{A}\llbracket \neg A \rrbracket_w$	$:\Leftrightarrow (\rho, s) \notin \mathcal{A}\llbracket A \rrbracket_w$
$(\rho, s) \in \mathcal{A}\llbracket P \wedge Q \rrbracket_w$	$:\Leftrightarrow (\rho, s) \in \mathcal{A}\llbracket P \rrbracket_w \ \wedge\ (\rho, s) \in \mathcal{A}\llbracket Q \rrbracket_w$
$(\rho, s) \in \mathcal{A}\llbracket \forall x.\, P \rrbracket_w$	$:\Leftrightarrow \forall v \in \mathit{Val}(w).\ (\rho[x \mapsto v], s) \in \mathcal{A}\llbracket P \rrbracket_w$
$(\rho, s) \in \mathcal{A}\llbracket \mathbf{emp} \rrbracket_w$	$:\Leftrightarrow \mathsf{dom}(s) = \emptyset$
$(\rho, s) \in \mathcal{A}\llbracket P_1 * P_2 \rrbracket_w$	$:\Leftrightarrow \exists s_1, s_2 \in \mathit{St}(w).\, s = s_1 * s_2 \ \wedge\ (\rho, s_i) \in \mathcal{A}\llbracket P_i \rrbracket_w$
$(\rho, s) \in \mathcal{A}\llbracket x \mapsto e \rrbracket_w$	$:\Leftrightarrow \mathsf{dom}(s) = \{\rho(x)\} \ \wedge\ s(\rho(x)) \sqsubseteq \mathcal{E}\llbracket e \rrbracket_w \rho$

Definition 10 (Validity). Let $w \in \mathbb{W}$, $\rho \in \mathit{Env}(w)$, $s \in \mathit{St}(LC)_w$, $c \in \mathit{Com}(w) \cap LC_w$ and $p, q \in \mathcal{S}$. An auxiliary meaning of "semantical triples" with respect to a fixed world, written $(\rho, s) \models_w \{p\}\, c\, \{q\}$, holds if and only if for all $f : w \to w_1$,

$$\begin{aligned}&(\mathit{Env}(f)(\rho), \mathit{St}(f)(s)) \in p_{w_1} \ \wedge\ c_f(\mathit{St}(f)(s))\downarrow \quad \Longrightarrow \\ &\exists g : w_1 \to w_2\ \exists s' \in \mathit{St}(w_2).\ c_f(\mathit{St}(f)(s)) = \langle g, s' \rangle \ \wedge\ (\mathit{Env}(f;g)(\rho), s') \in q_{w_2}\end{aligned}$$

In particular, this implies the absence of errors. *Validity* of syntactic triples in context of assumptions (as used in the rules) is defined by

$$\begin{aligned}&\models \{P_1\}\, c_1\, \{Q_1\}, \ldots, \{P_n\}\, c_n\, \{Q_n\} \vdash \{P\}\, c\, \{Q\} \ :\Longleftrightarrow \\ &\forall w \in \mathbb{W}\, \forall \rho \in \mathit{Env}(w)\, \forall s \in \mathit{St}(LC)_w.\ \bigwedge_{1 \leq i \leq n} (\rho, s) \models_w \{\mathcal{A}\llbracket P_i \rrbracket\}\, \mathcal{C}\llbracket c_i \rrbracket_w \rho\, \{\mathcal{A}\llbracket Q_i \rrbracket\} \\ &\qquad\qquad \Longrightarrow\ (\rho, s) \models_w \{\mathcal{A}\llbracket P \rrbracket\}\, \mathcal{C}\llbracket c \rrbracket_w \rho\, \{\mathcal{A}\llbracket Q \rrbracket\}\end{aligned}$$

For an empty context we simply write $\models \{P\}\, c\, \{Q\}$ instead of $\models\ \vdash \{P\}\, c\, \{Q\}$.

Theorem 11 (Soundness). *The logic presented in Section 2.2 is sound with respect to our semantics.*

Proof. Lack of space permits only a sketch for the two most interesting rules.

Soundness of the frame rule (Frame). Except for exploiting the renaming of locations the proof uses the standard argument: Suppose $\{P\}\, c\, \{Q\}$ is valid, let $w_1 \in \mathbb{W}$, $\rho \in \mathit{Env}(w_1)$ and $s \in \mathit{St}(LC)_{w_1}$ such that $(\mathit{Env}(f)(\rho), \mathit{St}(f)(s)) \in \mathcal{A}\llbracket P * R \rrbracket_{w_2}$ and $c_f(\mathit{St}(f)(s))\downarrow$, where $f : w_1 \to w_2$. By definition $\mathit{St}(f)(s) = s_1 * s_2$ for some s_1, s_2 with $(\mathit{Env}(f)(\rho), s_1) \in \mathcal{A}\llbracket P \rrbracket_{w_2}$ and $(\mathit{Env}(f)(\rho), s_2) \in \mathcal{A}\llbracket R \rrbracket_{w_2}$. Now if $c_f(\mathit{St}(f)(s)) = \mathsf{error}$ then, by assumption $c \in LC_w$, also $c_f(s_1) = \mathsf{error}$ which contradicts validity of $\{P\}\, c\, \{Q\}$. Thus, $c_f(\mathit{St}(f)(s)) = \langle g, s' \rangle$ for some $g : w_2 \to w_3$ and $s' \in \mathit{St}(w_3)$. By $c \in LC_w$ there exist $g' : w_2 \to w_3'$, $j : w_3' \xrightarrow{\sim} w_3$, $s_1' \in \mathit{St}(w_3')$ and $s_2' \in \mathit{St}(w_2)$ such that $s_2' \sqsubseteq s_2$, $c_f(s_1) = \langle g', s_1' \rangle$ and

$$s' = \mathit{St}(j)(s_1') * \mathit{St}(g';j)(s_2') \tag{4}$$

Downward-closure of $\mathcal{A}\llbracket R \rrbracket$ entails $(Env(f)(\rho), s'_2) \in \mathcal{A}\llbracket R \rrbracket_{w_2}$, and therefore $(Env(f;g';j)(\rho), St(g';j)(s'_2)) \in \mathcal{A}\llbracket R \rrbracket_{w_3}$ by Kripke monotonicity. By validity of $\{P\}\,c\,\{Q\}$ we have $(Env(f;g')(\rho), s'_1) \in \mathcal{A}\llbracket Q \rrbracket_{w'_3}$. Kripke monotonicity of $\mathcal{A}\llbracket Q \rrbracket$ and (4) entail $(Env(f;g)(\rho), s') \in \mathcal{A}\llbracket Q * R \rrbracket_{w_3}$, proving $\models \{P * R\}\,c\,\{Q * R\}$.

Soundness of the recursion rule (Rec). This is proved along the lines of [23]: Pitts' technique (cf. Theorem 6) is used to establish existence of a suitable recursive $\mathcal{S}$-relation containing the commands defined by mutual recursion. As in [23] one shows for all assertions P that $\mathcal{A}\llbracket P \rrbracket \in \mathcal{S}$ satisfies the following properties: for all $w \in \mathbb{W}$, the set $\{s \mid (\rho, s) \in \mathcal{A}\llbracket P \rrbracket_w\}$ is downward closed, the set $\{\rho \mid (\rho, s) \in \mathcal{A}\llbracket P \rrbracket_w\}$ is upward closed, and $e : \mathcal{A}\llbracket P \rrbracket \subset \mathcal{A}\llbracket P \rrbracket$ for all $e \sqsubseteq id_{Val}$.

The first property is built into the definition of $\mathcal{S}$-relations, the latter two can be established by induction on assertions. Note that the way $x \mapsto e$ and $e_1 \leq e_2$ are defined in Table 6 is essential for this result. In particular, $e_1 \leq e_2$ had to be defined differently in [23] where the extra level of locations was absent. □

5 Conclusions and Further Work

We have presented a logic for higher-order store that admits a local reasoning principle in form of the (first-order) frame rule. Soundness relies on a denotational semantics employing powerful constructions known from domain theory.

The reasoning principle for recursion through the store (Rec) is based on explicitly keeping track of the code in assertions. Honda et al. [9] present a Hoare logic for a higher-order imperative language, including dynamically allocated memory and admitting function storage, where *nested triples* and self-reference in pre- and postconditions are used to express recursive specifications. Their semantics is based on operational techniques, local reasoning is not addressed. The next challenge is to design a logic that is similar to [9] by using triples in assertions to specify stored procedures, while admitting a frame rule proved sound with respect to the semantics developed in this paper.

Stored procedures are particularly important for object-oriented programming, and we are currently investigating how a separation logic for higher-order store can be extended to simple object-based languages like the object calculus to obtain a logic that combines the power of local reasoning with the principle ideas of Abadi and Leino's logic [1, 21]. To achieve that, our results need to be generalised from Hoare triples to more general *transition relations*. Separation conjunction in such a framework has been considered in [19].

There are several possibilities for further improvements. It would be interesting to see if the FM models of [26, 3], rather than a presheaf semantics, can simplify the semantics. It also needs to be investigated whether a higher-order frame rule can be proven sound in our setting analogous to [6, 5].

References

1. M. Abadi and K. R. M. Leino. A logic of object-oriented programs. In *Verification: Theory and Practice. Essays Dedicated to Zohar Manna on the Occasion of His 64th*

Birthday, pages 11–41. Springer, 2004.

2. K. R. Apt and G. D. Plotkin. Countable nondeterminism and random assignment. *Journal of the ACM*, 33(4):724–767, Oct. 1986.
3. N. Benton and B. Leperchey. Relational reasoning in a nominal semantics for storage. In Proc. TLCA'05, volume 3461 of *LNCS*, pages 86–101. Springer, 2005.
4. L. Birkedal, N. Torp-Smith, and J. C. Reynolds. Local reasoning about a copying garbage collector. In *Proc. 31st POPL*, pages 220–231. ACM Press, 2004.
5. L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules. In *Proc. 20th LICS*. IEEE Computer Society Press, 2005.
6. R. Bornat, C. Calcagno, and P. O'Hearn. Local reasoning, separation, and aliasing. In *Proc. 2nd workshop on Semantics, Program Analysis and Computing Environments for Memory Management*, 2004.
7. P. di Gianantonio, F. Honsell, and G. D. Plotkin. Uncountable limits and the lambda calculus. *Nordic Journal of Computing*, 2(2):126–145, 1995.
8. C. A. R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969.
9. K. Honda, N. Yoshida, and M. Berger. An observationally complete program logic for imperative higher-order functions. In *Proc. 20th LICS*, pages 270–279. IEEE Computer Society Press, 2005.
10. S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. *ACM SIGPLAN Notices*, 36(3):14–26, Mar. 2001.
11. P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, Jan. 1964.
12. P. B. Levy. *Call-By-Push-Value. A Functional/Imperative Synthesis*, volume 2 of *Semantic Structures in Computation*. Kluwer, 2004.
13. P. W. O'Hearn. Resources, concurrency and local reasoning. In *Proc. CONCUR'04*, volume 3170 of *LNCS*, pages 49–67. Springer, 2004.
14. 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. Springer, 2001.
15. P. W. O'Hearn and R. D. Tennent. Semantics of local variables. In *Applications of Categories in Computer Science*, volume 177 of *London Mathematical Society Lecture Note Series*, pages 217–238. Cambridge University Press, 1992.
16. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Proc. 31st POPL*, pages 268–280, 2004.
17. F. J. Oles. *A Category-theoretic approach to the semantics of programming languages.* PhD thesis, Syracuse University, 1982.
18. A. M. Pitts. Relational properties of domains. *Information and Computation*, 127:66–90, 1996.
19. A. Podelski and I. Schaefer. Local reasoning for termination. Workshop on Verification of Concurrent Systems with Dynamically Allocated Heaps, July 2005.
20. U. S. Reddy and H. Yang. Correctness of data representations involving heap data structures. *Science of Computer Programming*, 50(1–3):129–160, March 2004.
21. B. Reus and J. Schwinghammer. Denotational semantics for a program logic of objects. *Mathematical Structures in Computer Science*, 2006. To appear.
22. B. Reus and T. Streicher. Semantics and logic of object calculi. *Theoretical Computer Science*, 316:191–213, 2004.
23. B. Reus and T. Streicher. About Hoare logics for higher-order store. In *Proc. ICALP'05*, volume 3580 of *LNCS*, pages 1337–1348. Springer, 2005.

24. J. C. Reynolds. The essence of Algol. In J. W. deBakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. North-Holland, 1981.
25. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. 17th LICS*, pages 55–74. IEEE Computer Society, 2002.
26. M. R. Shinwell and A. M. Pitts. On a monadic semantics for freshness. *Theoretical Computer Science*, 342:28–55, 2005.
27. M. B. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal on Computing*, 11(4):761–783, Nov. 1982.
28. H. Thielecke. Frame rules from answer types for code pointers. In *Proc. 33rd POPL*, pages 309–319. ACM Press, 2006.
29. H. Yang. An example of local reasoning in BI pointer logic: the Schorr-Waite graph marking algorithm. In *Proc. 2nd workshop on Semantics, Program Analysis and Computing Environments for Memory Management*, Jan. 2001.