# Nested Hoare Triples and Frame Rules for Higher-order Store

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

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

**Abstract.** Separation logic is a Hoare-style logic for reasoning about programs with heap-allocated mutable data structures. As a step toward extending separation logic to high-level languages with ML-style general (higher-order) storage, we investigate the compatibility of nested Hoare triples with several variations of higher-order frame rules.
The interaction of nested triples and frame rules can be subtle, and the inclusion of certain frame rules is in fact unsound. A particular combination of rules can be shown consistent by means of a Kripke model where worlds live in a recursively defined ultrametric space. The resulting logic allows us to elegantly prove programs involving stored code. In particular, it leads to natural specifications and proofs of invariants required for dealing with recursion through the store.

## 1 Introduction

Many programming languages permit not only the storage of first-order data, but also forms of higher-order store. Examples are code pointers in C, and ML-like general references. It is therefore important to have modular reasoning principles for these language features. Separation logic is an effective formalism for modular reasoning about pointer programs, in low-level C-like programming languages and, more recently, also in higher-level languages [6, 7, 9, 13]. However, its assertions are usually limited to talk about first-order data.

In previous work, we have begun the study of separation logic for languages with higher-order store [2, 12]. A challenge in this research is the combination of proof rules from separation logic for modular reasoning, and proof rules for code stored on the heap. Ideally, a program logic for higher-order store provides sufficiently expressive proof rules that, e.g., can deal with recursion through the store, and at the same time interact well with (higher-order) frame rules, which enable modular program verification.

Our earlier work shows that separation logic is consistent with higher-order store. However, the formulation of [2, 12] has a shortcoming: code is treated like any other data in that assertions can only mention concrete commands. For modular reasoning, it is clearly desirable to abstract from particular code and instead (partially) specify its behaviour. For example, when verifying mutually

recursive procedures on the heap, one would like to consider each procedure in isolation, relying on properties but not the implementations of the others. The recursion rule in [2, 12] does not achieve this. A second, and less obvious consequence of lacking behavioural specifications for code in assertions is that one cannot take full advantage of the frame rules of separation logic. For instance, the language in [2] can simulate higher-order procedures by passing arguments through the heap, but the available (higher-order) frame rules are not useful here because an appropriate specification for this encoding is missing.

In this article, we address these shortcomings by investigating a program logic in which stored code can be specified using Hoare triples, i.e., an assertion language with *nested triples*. This is an obvious idea, but the combination of nested triples and frame rules turns out to be tricky: the most natural combination turns out to be unsound.

The main technical contributions of this paper are therefore: (1) the observation that certain "deep" frame rules can be unsound, (2) the suggestion of a "good" combination of nested Hoare triples and frame rules, and (3) the verification of those by means of an elegant Kripke model, where the worlds are themselves world-dependent sets of heaps. The worlds form a complete metric space and (the denotation of) the tensor $\otimes$, needed to generically express higher-order frame rules, is contractive; as a consequence, our logic permits recursively defined assertions.

After introducing the syntax of language and assertions in Section 2 we discuss some unsound combinations of rules in Section 3, which also contains the suggested set of rules for our logic. The soundness of the logic is then shown in Section 4.

## 2 Syntax of Programs and Assertions

We consider a simple imperative programming language extended with operations for stored code and heap manipulation. The syntax of the language is shown in Fig. 1. The expressions in the language are integer expressions, variables, and the quote expression '$C$' for representing an unevaluated command $C$. The integer or code value denoted by expression $e_1$ can be stored in a heap cell $e_0$ using $[e_0]{:=}e_1$, and this stored value can later be looked up and bound to the (immutable) variable $y$ by $\mathsf{let}\ y{=}[e_0]\ \mathsf{in}\ D$. In case the value stored in cell $e_0$ is code '$C$', we can run (or "evaluate") this code by executing $\mathsf{eval}\,[e_0]$. Our language also provides constructs for allocating and disposing heap cells such as $e_0$ above. We point out that, as in ML, all variables $x, y, z$ in our language are *immutable*, so that once they are bound to a value, their values do not change. This property of the language lets us avoid side conditions on variables when studying frame rules. Finally, we do not include while loops in our language, since they can be expressed by stored code (using Landin's knot).

Our assertion language is standard first-order intuitionistic logic, extended with separating connectives $emp, *$, the points-to predicate $\mapsto$ [13], and with recursively defined assertions. The syntax of assertions appears in Fig. 1. Each

| | |
|---|---|
| $d \in Exp ::= \mathsf{0} \mid -\mathsf{1} \mid \mathsf{1} \mid \ldots \mid d_1{+}d_2 \mid \ldots \mid x$ | integer expressions, variable |
| $\mid$ '$C$' | quote (command as expression) |
| $C \in Com ::= [e_1]{:=}e_2 \mid \mathsf{let}\ y{=}[e]\ \mathsf{in}\ C \mid \mathsf{eval}\,[e]$ | assignment, lookup, unquote |
| $\mid$ $\mathsf{let}\ x{=}\mathsf{new}\ (e_1,\ldots,e_n)\ \mathsf{in}\ C \mid \mathsf{free}\ e$ | allocation, disposal |
| $\mid$ $\mathsf{skip} \mid C_1;C_2 \mid \mathsf{if}\ (e_1{=}e_2)\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$ | no op, sequencing, conditional |
| $P,Q \in Assn ::= \mathsf{false} \mid \mathsf{true} \mid P \vee Q \mid P \wedge Q \mid P \Rightarrow Q$ | intuitionistic-logic connectives |
| $\mid$ $\forall x.P \mid \exists x.P \mid \mathsf{int}(e) \mid e_1{=}e_2 \mid e_1{\leq}e_2$ | quantifiers, atomic formulas |
| $\mid$ $e_1 \mapsto e_2 \mid \mathit{emp} \mid P * Q$ | separating connectives |
| $\mid$ $\{P\}e\{Q\} \mid P \otimes Q \mid \ldots$ | Hoare triple, invariant extension |

**Fig. 1.** Syntax of expressions, commands and assertions

$$P \circ R \stackrel{def}{=} (P \otimes R) * R$$

$$\{P\}e\{Q\} \otimes R \Leftrightarrow \{P \circ R\}e\{Q \circ R\} \quad (\kappa x.P) \otimes R \Leftrightarrow \kappa x.(P \otimes R) \qquad (\kappa \in \{\forall, \exists\}, x \notin fv(R))$$

$$(P \otimes R) \otimes R' \Leftrightarrow P \otimes (R \circ R') \qquad (P \oplus Q) \otimes R \Leftrightarrow (P \otimes R) \oplus (Q \otimes R) \quad (\oplus \in \{\Rightarrow, \wedge, \vee, *\})$$

$$P \otimes R \Leftrightarrow P \qquad (P \text{ is one of } \mathsf{true}, \mathsf{false}, \mathit{emp}, e{=}e', e{\mapsto}e' \text{ and } \mathsf{int}(e))$$

**Fig. 2.** Axioms for distributing $- \otimes R$

assertion describes a property of states, which consist of an immutable stack and a mutable heap. Formula *emp* means that the heap component of the state is empty, and $P * Q$ means that the heap component can be split into two, one satisfying $P$ and the other satisfying $Q$, both evaluated with respect to the same stack. The points-to predicate $e_0 \mapsto e_1$ states that the heap component consists of only one cell $e_0$ whose contents is (some approximation of) $e_1$.

One interesting aspect of our assertion language is that it includes Hoare triples $\{P\}e\{Q\}$ and invariant extensions $P \otimes Q$; previous work [4, 2] does not treat them as assertions, but as so-called *specifications*, which form a different syntactic category. Intuitively, $\{P\}e\{Q\}$ means that $e$ denotes code satisfying $\{P\}\_\{Q\}$, and $P \otimes Q$ denotes a modification of $P$ where all the pre- and post-conditions of triples inside $P$ are $*$-extended with $Q$. For instance, the assertion $(\exists k.\,(1 \mapsto k) \wedge \{\mathit{emp}\}k\{\mathit{emp}\}) \otimes (2{\mapsto}0)$ is equivalent to $(\exists k.\,(1 \mapsto k) \wedge \{2{\mapsto}0\}k\{2{\mapsto}0\})$. This assertion says that cell 1 is the only cell in the heap and it stores code $k$ that satisfies the triple $\{2{\mapsto}0\}\_\{2{\mapsto}0\}$. This intuition of the $\otimes$ operator can also be seen in the set of axioms in Fig. 2, which let us distribute $\otimes$ through all the constructs of the assertion language.

Note that since triples are assertions, they can appear in pre- and post-conditions of triples. This *nested* use of triples is useful in reasoning, because it allows one to specify stored code behaviourally, in terms of properties that it satisfies. Another important consequence of having these new constructs as assertions is that they allow us to study proof rules for exploiting locality of stored code systematically, as we will describe shortly.

The last case ... in Fig. 1 represents pre-defined assertions, including recursively defined ones. In particular, it contains all recursively defined assertions

of the form $R = P \otimes R$, where $R$ does not appear in $P$. These assertions are always well-defined (because $\otimes$ is "contractive" in its second argument, as shown in Lemma 4), and they let us reason about self-applying stored code, without using specialized rules [2]. We will say more about the use of recursively defined predicates and their existence in Sections 3 and 4.[5]

We shall make use of two abbreviations. The first is $P \circ R$, which stands for $(P \otimes R) * R$ (already used in Fig. 2). This abbreviation is often used to add an invariant $R$ to a Hoare triple $\{P\}e\{Q\}$, so as to obtain $\{P \circ R\}e\{Q \circ R\}$. We use $\circ$ instead of $*$ here to extend not only $P$ by $R$ but also ensure, via $\otimes$, that all Hoare triples nested inside $P$ preserve $R$ as an invariant. The $\circ$ operator has been introduced in [11], where it is credited to Paul-André Melliès and Nicolas Tabareau. The second abbreviation is for the "$\mapsto$" operator: $e_1 \mapsto P[e_2] \stackrel{def}{=} e_1 \mapsto e_2 \wedge P[e_2]$ and $e_1 \mapsto P[\_] \stackrel{def}{=} \exists x.\, e_1 \mapsto P[x]$. Here $x$ is a fresh (logic) variable and $P[\cdot]$ is an assertion with an expression hole, such as $\{Q\} \cdot \{R\}$, $\mathsf{int}(\cdot)$, $\cdot = e$ or $\cdot \leq e$.

## 3 Proof Rules for Higher-order Store

In our formal setting, reasoning about programs is done by deriving the judgement $\Gamma \vdash P$, where $P$ is an assertion expressing properties of programs and $\Gamma$ is a list of variables containing all the free variables in $P$. For instance, to prove that command $C$ stores at cell 1 the code that initializes cell 10 to 0,[6] we need to derive $\Gamma \vdash \{1 \mapsto \_\}\text{`}C\text{'}\{1 \mapsto \{10 \mapsto \_\}\_\{10 \mapsto 0\}\}$. In this section, we describe inference rules and axioms for assertions that let one efficiently reason about programs. We focus on those related to higher-order store.

**Standard proof rules** The proof rules include the standard proof rules for intuitionistic logic and the logic of bunched implications [8] (not repeated here). Moreover, the proof rules include variations of standard separation logic proof rules, see Fig. 3.[7] The figure neither includes the rule for executing stored code with $\mathsf{eval}\,[e]$ nor the frame rule for adding invariants to triples; the reason for this omission is that these two rules raise nontrivial issues in the presence of higher-order store and nested triples, as we will now discuss.

**Frame rule for higher-order store** The frame rule is the most important rule in separation logic, and it formalizes the intuition of local reasoning, where

[5] More generally, we may need to solve mutually recursive assertions $\langle R_1, \ldots, R_n \rangle = \langle P_1 \otimes (R_1 * \ldots * R_n), \ldots, P_n \otimes (R_1 * \ldots * R_n) \rangle$ in order to deal with mutually recursive stored procedures. For brevity we omit formal syntax for such; see Theorem 12 for the semantic existence proof.

[6] One concrete example of such a command $C$ is [1]:='[10]:=0'.

[7] The Update, Free and Skip rules in the figure are not the usual small axioms in separation logic, since they contain assertion $P$ describing the unchanged part. Since we have the standard frame rule for $*$, we could have used small axioms instead here. But we chose not to do this, because the current non-small axioms make it easier to follow our discussions on frame rules and higher-order store in the next subsection.

$$\textsc{Deref}\quad \frac{\Gamma, x \vdash \{P * e \mapsto x\}\text{`}C\text{'}\{Q\}}{\Gamma \vdash \{\exists x.P * e \mapsto x\}\text{`}\mathsf{let}\ x{=}[e]\ \mathsf{in}\ C\text{'}\{Q\}}(x{\notin}\mathsf{fv}(e,Q)) \qquad \textsc{Update}\quad \frac{}{\Gamma \vdash \{e \mapsto \_ * P\}\text{`}[e] := e_0\text{'}\{e \mapsto e_0 * P\}}$$

$$\textsc{New}\quad \frac{\Gamma, x \vdash \{P * x \mapsto e\}\text{`}C\text{'}\{Q\}}{\Gamma \vdash \{P\}\text{`}\mathsf{let}\ x{=}\mathsf{new}\ e\ \mathsf{in}\ C\text{'}\{Q\}}(x{\notin}\mathsf{fv}(P,e,Q)) \qquad \textsc{Free}\quad \frac{}{\Gamma \vdash \{e \mapsto \_ * P\}\text{`}\mathsf{free}(e)\text{'}\{P\}}$$

$$\textsc{Skip}\quad \frac{}{\Gamma \vdash \{P\}\text{`}\mathsf{skip}\text{'}\{P\}} \qquad \textsc{Seq}\quad \frac{\Gamma \vdash \{P\}\text{`}C\text{'}\{R\} \quad \Gamma \vdash \{R\}\text{`}D\text{'}\{Q\}}{\Gamma \vdash \{P\}\text{`}C; D\text{'}\{Q\}}$$

$$\textsc{If}\quad \frac{\Gamma \vdash \{P \wedge e_0{=}e_1\}\text{`}C\text{'}\{Q\} \quad \Gamma \vdash \{P \wedge e_0{\neq}e_1\}\text{`}D\text{'}\{Q\}}{\Gamma \vdash \{P\}\text{`}\mathsf{if}\ (e_0{=}e_1)\ \mathsf{then}\ C\ \mathsf{else}\ D\text{'}\{Q\}} \qquad \textsc{Conseq}\quad \frac{\Gamma \vdash P' \Rightarrow P \quad \Gamma \vdash Q \Rightarrow Q'}{\Gamma \vdash \{P\}e\{Q\} \Rightarrow \{P'\}e\{Q'\}}$$

**Fig. 3.** Proof rules from separation logic

proofs focus on the footprints of the programs we verify. Developing a similar rule in our setting is challenging, because nested triples allow for several choices regarding the shape of the rule. Moreover, the recursive nature of the higher-order store muddies the water and it is difficult to see which choices actually make sense (i.e., do not lead to inconsistency).

To see this problem more clearly, consider the rules below:

$$\frac{\Gamma \vdash \{P\}e\{Q\}}{\Gamma \vdash \{P \ \square\ R\}e\{Q \ \square\ R\}} \ \text{and} \ \frac{}{\Gamma \vdash \{P\}e\{Q\} \Rightarrow \{P \ \square\ R\}e\{Q \ \square\ R\}} \ \text{for } \square \in \{*, \circ\}.$$

Note that we have four choices, depending on whether we use $\square = *$ or $\square = \circ$ and on whether we have an inference rule or an axiom. If we choose $*$ for $\square$, we obtain *shallow* frame rules that add $R$ to the outermost triple $\{P\}e\{Q\}$ only; they do not add $R$ in nested triples appearing in pre-condition $P$ and post-condition $Q$. On the other hand, if we choose $\circ$ for $\square$, since $(A \circ R) = (A \otimes R * R)$, we obtain *deep* frame rules that add the invariant $R$ not just to the outermost triple but also to all the nested triples in $P$ and $Q$.

The distinction between inference rule and axiom has some bearing on where the frame rule can be applied. With the axiom version, we can apply the frame rule not just to valid triples, but also to nested triples appearing in pre- or post-conditions. With the inference rule version, however, we cannot add invariants to (or remove invariants from) nested triples.

Ideally, we would like to have the axiom versions of the frame rules for both $*$ and $\circ$. Unfortunately, this is not possible for $\circ$. Adding the axiom version for $\circ$ makes our logic unsound. The source of the problem is that with the axiom version for $\circ$, one can add invariants selectively to some, but not necessarily all, nested triples. This flexibility can be abused to derive incorrect conclusions.

Concretely, with the axiom version for $\circ$ we can make the following derivation:

$$\dfrac{\dfrac{\Gamma \vdash \{P \circ S\} e \{Q \circ S\}}{\Gamma \vdash \{P\} e \{Q\} \otimes S}\otimes\text{-Dist} \quad \dfrac{\dfrac{}{\Gamma \vdash \{P\} e \{Q\} \Rightarrow \{P \circ R\} e \{Q \circ R\}}\textsc{Frame}}{\Gamma \vdash \{P\} e \{Q\} \otimes S \Rightarrow \{P \circ R\} e \{Q \circ R\} \otimes S}\otimes\text{-Mono}}{\dfrac{\Gamma \vdash \{P \circ R\} e \{Q \circ R\} \otimes S}{\Gamma \vdash \{(P \circ R) \circ S\} e \{(Q \circ R) \circ S\}}\otimes\text{-Dist}}\ \textsc{ModusPon}$$

Here we use the distribution axioms for $\otimes$ in Fig. 2 and the monotonicity of $- \otimes R$. This derivation means that when adding $R$ to nested triples, we can skip the triples in the $S$ part of the pre- and post-conditions of $\{P \circ S\} e \{Q \circ S\}$. This flexibility leads to the unsoundness:

**Proposition 1.** *Adding the axiom version of the frame rule for $\circ$ renders our logic unsound.*

*Proof.* Let $R$ be the predicate defined by $R = (3 \mapsto \{1 \mapsto \_\}\_\{1 \mapsto \_\}) \otimes R$. Then, we can derive the triple:

$$(\dagger)\quad \dfrac{\dfrac{k \vdash \{2 \mapsto \{1 \mapsto \_\}\_\{1 \mapsto \_\} \circ R\} k \{2 \mapsto \_ \circ R\}}{k \vdash \{(2 \mapsto \{1 \mapsto \_\}\_\{1 \mapsto \_\} \circ 1 \mapsto \_) \circ R\} k \{(2 \mapsto \_ \circ 1 \mapsto \_) \circ R\}}}{k \vdash \{2 \mapsto \text{`}\mathsf{free}(-1)\text{'} * 1 \mapsto \_ * R\} k \{2 \mapsto \_ * 1 \mapsto \_ * 3 \mapsto \{1 \mapsto \_ * R\}\_\{1 \mapsto \_ * R\}\}}$$

Here the first step uses the derivation above for adding invariants selectively, and the last step uses the Consequence axiom with the below two implications:

$$\begin{aligned} 2 \mapsto \{1 \mapsto \_\}\_\{1 \mapsto \_\} \circ 1 \mapsto \_ \circ R &\iff 2 \mapsto \{1 \mapsto \_ * 1 \mapsto \_ * R\}\_\{1 \mapsto \_ * 1 \mapsto \_ * R\} * 1 \mapsto \_ * R \\ &\iff 2 \mapsto \{\mathsf{false}\}\_\{\mathsf{false}\} * 1 \mapsto \_ * R \\ &\impliedby 2 \mapsto \text{`}\mathsf{free}(-1)\text{'} * 1 \mapsto \_ * R \end{aligned}$$

$$\begin{aligned} 2 \mapsto \_ \circ 1 \mapsto \_ \circ R &\iff 2 \mapsto \_ * 1 \mapsto \_ * R \iff 2 \mapsto \_ * 1 \mapsto \_ * ((3 \mapsto \{1 \mapsto \_\}\_\{1 \mapsto \_\}) \otimes R) \\ &\iff 2 \mapsto \_ * 1 \mapsto \_ * 3 \mapsto \{1 \mapsto \_ * R\}\_\{1 \mapsto \_ * R\}. \end{aligned}$$

Consider $C \equiv \mathsf{let}\ x{=}[2]\ \mathsf{in}\ [3]{:=}x$. When $P[y] \equiv \{1 \mapsto \_\} y \{1 \mapsto \_\} \otimes R$,

$$\dfrac{\dfrac{}{\vdash \{2 \mapsto P[\_] * 3 \mapsto P[\_]\}\text{`}C\text{'}\{2 \mapsto P[\_] * 3 \mapsto P[\_]\}}}{\vdash \{2 \mapsto \{1 \mapsto \_\}\_\{1 \mapsto \_\} \circ R\}\text{`}C\text{'}\{2 \mapsto \_ \circ R\}}$$

Now we instantiate $k$ in $(\dagger)$ with $C$, discharge the premise of the resulting derivation with the above derivation for $C$, and obtain

$$\dfrac{\dfrac{}{\vdash \{2 \mapsto \text{`}\mathsf{free}(-1)\text{'} * 1 \mapsto \_ * R\}\text{`}C\text{'}\{2 \mapsto \_ * 1 \mapsto \_ * 3 \mapsto \{1 \mapsto \_ * R\}\_\{1 \mapsto \_ * R\}\}}}{\vdash \{2 \mapsto \text{`}\mathsf{free}(-1)\text{'} * 1 \mapsto \_ * R\}\text{`}C\text{;}\,\mathsf{free}(2)\text{'}\{1 \mapsto \_ * 3 \mapsto \{1 \mapsto \_ * R\}\_\{1 \mapsto \_ * R\}\}}$$

Here the second step uses the rules Free and Seq in Fig. 3. But the post-condition of the conclusion here is equivalent to $1 \mapsto \_ * R$ by the definition of $R$ and the distribution axioms for $\otimes$. Thus, as our rule for $\mathsf{eval}$ will show later, we should be able to conclude that

$$\vdash \{2 \mapsto \text{`}\mathsf{free}(-1)\text{'} * 1 \mapsto \_ * R\}\text{`}C\text{;}\,\mathsf{free}(2)\text{;}\,\mathsf{eval}\,[3]\text{'}\{1 \mapsto \_ * 3 \mapsto \{1 \mapsto \_ * R\}\_\{1 \mapsto \_ * R\}\}$$

However, since $-1$ is not even an address, the program $(C; \mathsf{free}(2); \mathsf{eval}\,[3])$ always faults, contradicting the requirement of separation logic that proved programs run without faulting. □

Notice that in the derivation above it is essential that $R$ is a recursively defined assertion, otherwise we would not obtain that 2 and 3 point to code satisfying the same $P$.

Fortunately, the second best choice leads to a consistent proof system:

**Proposition 2.** *Both the inference rule version of the frame rule for $\circ$ and the axiom version for $*$ are sound in our semantics, which will be given in Section 4. In fact, the semantics also validates the following more general version of the rule for $\circ$:*

$$\frac{\Gamma \vdash P}{\Gamma \vdash P \otimes R}$$

**Rule for executing stored code** An important and challenging part of the design of a program logic for higher-order store is the design of a proof rule for $\mathsf{eval}\,[e]$, the command that executes code stored at $e$. Indeed, the rule should overcome two challenges directly related to the recursive nature of higher-order store: (1) implicit recursion through the store (i.e., Landin's knot), and (2) extensional specifications of stored code.

These two challenges are addressed, using the expressiveness of our assertion language, by the following rule for $\mathsf{eval}\,[e]$:

$$\textsc{Eval}\quad \frac{\Gamma, k \vdash R[k] \Rightarrow \{P * e \mapsto R[\_]\}k\{Q\}}{\Gamma \vdash \{P * e \mapsto R[\_]\}\text{`}\mathsf{eval}\,[e]\text{'}\{Q\}}$$

This rule states that in order to prove $\{P * e \mapsto R[\_]\}\text{`}\mathsf{eval}\,[e]\text{'}\{Q\}$ for executing stored code in $[e]$ under the assumption that $e$ points to arbitrary code $k$ (expressed by the $\_$ which is an abbreviation for $\exists k.e \mapsto R[k]$), it suffices to show that the specification $R[k]$ implies that $k$ itself fulfils triple $\{P * e \mapsto R[\_]\}k\{Q\}$.

In the above rule we do not make any assumptions about what code $e$ actually points to, as long as it fulfils the specification $R$. It may even be updated between recursive calls. However, for recursion through the store, $R$ must be recursively defined as it needs to maintain itself as an invariant of the code in $e$.

The Eval rule crucially relies on the expressiveness of our assertion language, especially the presence of nested triples and recursive assertions. In our previous work, we did not consider nested triples. As a result, we had to reason explicitly with stored code, rather than properties of the code, as illustrated by one of our old rules for $\mathsf{eval}$ [2]:

$$\textsc{OldEval}\quad \frac{\Gamma \vdash \{P\}\text{`}\mathsf{eval}\,[e]\text{'}\{Q\} \Rightarrow \{P\}\text{`}C\text{'}\{Q\}}{\Gamma \vdash \{P * e \mapsto \text{`}C\text{'}\}\text{`}\mathsf{eval}\,[e]\text{'}\{Q * e \mapsto \text{`}C\text{'}\}}$$

$$\frac{}{\Gamma \vdash \{P * e \mapsto \forall \boldsymbol{y}.\, \{P\}\_\{Q\}\}\text{`}\mathsf{eval}\,[e]\text{'}\{Q * e \mapsto \forall \boldsymbol{y}.\, \{P\}\_\{Q\}\}}\ \textsc{EvalNonRec1}$$

$$\frac{}{\Gamma \vdash \{P * e \mapsto \forall \boldsymbol{y}.\, \{P * e \mapsto \_\}\_\{Q\}\}\text{`}\mathsf{eval}\,[e]\text{'}\{Q\}}\ \textsc{EvalNonRecUpd}$$

$$\frac{}{\Gamma \vdash \{P \circ R\}\text{`}\mathsf{eval}\,[e]\text{'}\{Q \circ R\}}\ \textsc{EvalRec}\ (\text{where } R = (e \mapsto \forall \boldsymbol{y}.\, \{P\}\_\{Q\} * P_0) \otimes R)$$

**Fig. 4.** Derived rules from Eval

Here the actual code $C$ is specified explicitly in the pre- and post-conditions of the triple. In both rules the intuition is that the premise states that the body of the recursive procedure fulfils the triple, under the assumption that the recursive call does so as well. In the Eval rule this is done without direct reference to the code itself, but rather via a $k$ satisfying $R$. The soundness proof of OldEval proceeded along the lines of Pitts' method for establishing relational properties of domains [10]. On the other hand, Eval relies on the availability of recursive assertions, the existence of which is guaranteed by Banach's fixpoint theorem.

From the Eval rule one can easily derive the axioms of Fig. 4. The first two axioms are for non-recursive calls. This can be seen from the fact that in the pre-condition of the nested triples $e$ does not appear at all or does not have a specification, respectively. Only the third axiom EvalRec allows for recursive calls. The idea of this axiom is that one assumes that the code in $[e]$ fulfils the required triple provided the code that $e$ points to at call-time fulfils the triple as well. Let us look at the actual derivation of EvalRec to make this evident. We write $S[k] \equiv \forall \boldsymbol{y}.\, \{P \circ R\}k\{Q \circ R\}$ such that for the original $R$ of rule EvalRec we have $R \Leftrightarrow (e \mapsto S[\_] * (P_0 \otimes R))$. Note that $\Gamma$ contains the variables $\boldsymbol{y}$ which may appear freely in $P$ and $Q$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, k \vdash (\forall \boldsymbol{y}.\, \{P \circ R\}k\{Q \circ R\}) \Rightarrow \{P \circ R\}k\{Q \circ R\}}\ \text{FOL}}{\Gamma, k \vdash S[k] \Rightarrow \{(P \otimes R) * (P_0 \otimes R) * e \mapsto S[\_]\}k\{Q \circ R\}}\ \textsc{Conseq}}{\Gamma \vdash \{(P \otimes R) * (P_0 \otimes R) * e \mapsto S[\_]\}\text{`}\mathsf{eval}\,[e]\text{'}\{Q \circ R\}}\ \textsc{Eval}}{\Gamma \vdash \{P \circ R\}\text{`}\mathsf{eval}\,[e]\text{'}\{Q \circ R\}}\ \textsc{Conseq}$$

In the derivation tree above, the axiom used at the top is simply a first-order axiom for $\forall$ elimination. The quantified variables $\boldsymbol{y}$ are substituted by the variables with the same name from the context. After an application of the EvalRec rule those variables $\boldsymbol{y}$ can then be substituted further.

The use of recursive specification $R = (e \mapsto \forall \boldsymbol{y}.\, \{P\}\_\{Q\} * P_0) \otimes R$ is essential here as it allows us to unroll the definition so that the Eval rule can be applied. Note that in the logic of [5], which also uses nested triples but features neither a specification logic nor expresses any frame rules or axioms, such recursive specifications are avoided. This is possible under the assumption that code does not change during recursion. One can then express the recursive $R$ above as

$$\frac{\Gamma \vdash P}{\Gamma \vdash P \otimes R}\ \otimes\text{-Frame} \qquad \frac{}{\Gamma \vdash \{P\}e\{Q\} \Rightarrow \{P*R\}e\{Q*R\}}\ *\text{-Frame} \qquad \frac{\Gamma, k \vdash R[k] \Rightarrow \{P * e \mapsto R[\_]\}k\{Q\}}{\Gamma \vdash \{P * e \mapsto R[\_]\}\text{`}\mathsf{eval}\,[e]\text{'}\{Q\}}\ \textsc{Eval}$$

**Fig. 5.** Proof rules specific to higher-order store

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

**Fig. 6.** Interpretation of commands $[\![C]\!]_\eta \in \mathit{Heap} \multimap T_{err}(\mathit{Heap})$

follows (we can omit the $P_0$ now, as this is only needed for mutually recursively defined triples):

$$e \mapsto \{e \mapsto k * P\}k\{e \mapsto k * Q\}\,.$$

The question however remains how the assertion can be proved for some concrete '$C$' in $[e]$. In *loc.cit.* this is done by an induction on some appropriate argument, as total correctness is considered only. Note that our old OldEval can be viewed as a fixpoint induction rule for proving such specifications, if one quantifies away the concrete appearances of '$C$'. In any case, our new Eval is obviously elegant to use, and it does not only allow for recursion through the store but also disentangles the reasoning from the concrete code stored in the heap.

To conclude this section, Fig. 5 summarizes a particular choice of proof-rule set from the current and previous subsections. Soundness is proved in Section 4.

## 4 Semantics of Nested Triples

This section develops a model for the programming language and logic we have presented. The semantics of programs, given in the next subsection using an untyped domain-theoretic model, is standard. The following semantics of the logic is, however, unusual; it is a possible world semantics where the worlds live in a recursively defined *metric* space. Finally, we discuss the existence of recursively defined assertions, which have been used in the previous sections.

**Semantics of expressions and commands** The interpretation of the programming language is given in the category $\mathbf{Cppo}_\perp$ of pointed cpos and strict continuous functions, and is the same as in our previous work [2]. That is, commands denote strict continuous functions $[\![C]\!]_\eta \in \mathit{Heap} \multimap T_{err}(\mathit{Heap})$ where

$$\mathit{Heap} = \mathit{Rec}(\mathit{Val}) \quad \mathit{Val} = \mathit{Integers}_\perp \oplus \mathit{Com}_\perp \quad \mathit{Com} = \mathit{Heap} \multimap T_{err}(\mathit{Heap}) \quad (1)$$

In these equations, $T_{err}(D) = D \oplus \{\mathit{error}\}_\perp$ denotes the error monad, and $\mathit{Rec}(D)$ denotes records with entries from $D$ and labelled by positive natural numbers. Formally, $\mathit{Rec}(D) = \left(\sum_{N \subseteq_{fin} Nats^+} (N \to D_\downarrow)\right)_\perp$ where $(N \to D_\downarrow)$ is the cpo of maps from the finite address set $N$ to the cpo $D_\downarrow = D - \{\perp\}$ of non-bottom elements of $D$. We use some evident record notations, such as $\{\!|\ell_1 = d_1, \ldots, \ell_n = d_n|\!\}$ for the record mapping label $\ell_i$ to $d_i$, and $\mathsf{dom}(r)$ for the set of labels of a record $r$. The *disjointness predicate* $r \# r'$ on records holds if $r$ and $r'$ are not $\perp$ and have disjoint domains, and a partial *combining operation* $r \cdot r'$ is defined by

$$r \cdot r' \stackrel{def}{=} \text{if } r \# r' \text{ then } r \cup r' \text{ else } \perp \ .$$

The interpretation of commands is repeated in Fig. 6 (assuming $h \neq \perp$). The interpretation of the quote operation, '$C$', uses the injection of *Com* into *Val*. The interpretation of the remaining expressions is entirely standard and omitted.

A solution to equation (1) for *Heap* can be obtained by the usual inverse limit construction [14] in the category $\mathbf{Cppo}_\perp$. This solution is an SFP domain (e.g., [15]), and thus comes equipped with an increasing chain $\pi_n : \mathit{Heap} \to \mathit{Heap}$ of continuous projection maps, satisfying $\pi_0 = \perp$, $\bigsqcup_{n \in \omega} \pi_n = id_{Heap}$, and $\pi_n \circ \pi_m = \pi_{\min\{n,m\}}$ . The image of each $\pi_n$ is finite, hence each $\pi_n(h)$ is a compact element of *Heap*. Moreover, the projections are compatible with composition of heaps: we have $\pi_n(h \cdot h') = \pi_n(h) \cdot \pi_n(h')$ for all $h, h'$.

**Semantic domain for assertions** A subset $p \subseteq \mathit{Heap}$ is *admissible* if $\perp \in p$ and $p$ is closed under taking least upper bounds of $\omega$-chains. It is *uniform* [3] if it is closed under the projections, i.e., if $p$ satisfies that $h \in p \Rightarrow \pi_n(h) \in p$ for all $n$. We write *UAdm* for the set of all uniform admissible subsets of *Heap*. For $p \in \mathit{UAdm}$, $p_{[n]}$ denotes the image of $p$ under $\pi_n$. Note that also $p_{[n]} \in \mathit{UAdm}$.

The uniform admissible subsets will form the basic building block when interpreting the assertions of our logic. Since assertions in general depend on invariants for stored code, the space of semantic predicates Pred will consist of functions $W \to \mathit{UAdm}$ from a set of "worlds," describing the invariants, to the collection of uniform admissible subsets of heaps. But, the invariants for stored code are themselves semantic predicates, and the interaction between Pred and $W$ is governed by (the semantics of) $\otimes$. Hence we seek a space of worlds $W$ that is "the same" as $W \to \mathit{UAdm}$. We obtain such a $W$ using metric spaces.

Recall that a 1-bounded ultrametric space $(X, d)$ is a metric space where the distance function $d : X \times X \to \mathbb{R}$ takes values in the closed interval $[0, 1]$ and satisfies the strong triangle inequality $d(x, y) \leq \max\{d(x, z), d(z, y)\}$, for all $x, y, z \in X$. An (ultra-)metric space is complete if every Cauchy sequence

has a limit. A function $f : X_1 \to X_2$ between metric spaces $(X_1, d_1), (X_2, d_2)$ is *non-expansive* if for all $x, y \in X_1$, $d_2(f(x), f(y)) \leq d_1(x, y)$. It is *contractive* if for some $\delta < 1$, $d_2(f(x), f(y)) \leq \delta \cdot d_1(x, y)$ for all $x, y \in X_1$.

The complete, 1-bounded ultrametric spaces and non-expansive functions between them form a Cartesian closed category *CBUlt*. Products in *CBUlt* are given by the set-theoretic product where the distance is the maximum of the componentwise distances, and exponentials are given by the non-expansive functions equipped with the sup-metric. A functor $F : \mathit{CBUlt}^{op} \times \mathit{CBUlt} \longrightarrow \mathit{CBUlt}$ is *locally non-expansive* if $d(F(f, g), F(f', g')) \leq \max\{d(f, f'), d(g, g')\}$ for all non-expansive $f, f', g, g'$, and it is *locally contractive* if $d(F(f, g), F(f', g')) \leq \delta \cdot \max\{d(f, f'), d(g, g')\}$ for some $\delta < 1$. By multiplication of the distances of $(X, d)$ with a shrinking factor $\delta < 1$ one obtains a new ultrametric space, $\delta \cdot (X, d) = (X, d')$ where $d'(x, y) = \delta \cdot d(x, y)$. By shrinking, a locally non-expansive functor $F$ yields a locally contractive functor $(\delta \cdot F)(X_1, X_2) = \delta \cdot (F(X_1, X_2))$.

The set *UAdm* of uniform admissible subsets of *Heap* becomes a complete, 1-bounded ultrametric space when equipped with the following distance function: $d(p, q) \stackrel{def}{=} \mathit{if}\ (p \neq q)\ \mathit{then}\ (2^{-\max\{i \in \omega \mid p_{[i]} = q_{[i]}\}})\ \mathit{else}\ 0$. Note that $d$ is well-defined: first, because $\pi_0 = \bot$ and $\bot \in p$ for all $p \in \mathit{UAdm}$ the set $\{i \in \omega \mid p_{[i]} = q_{[i]}\}$ is non-empty; second, this set is finite, because $p \neq q$ implies $p_{[i]} \neq q_{[i]}$ for all sufficiently large $i$ by the uniformity of $p, q$ and using $\bigsqcup_{n \in \omega} \pi_n = id_{Heap}$.

**Theorem 3.** *There exists an ultrametric space $W$ and an isomorphism $\iota$ from $\frac{1}{2} \cdot (W \to \mathit{UAdm})$ to $W$ in CBUlt.*

*Proof.* By an application of America & Rutten's existence theorem for fixed points of locally contractive functors on complete ultrametric spaces [1], applied to $F(X, Y) = \frac{1}{2} \cdot (X \to \mathit{UAdm})$. See [3] for details of a similar application. □

We write Pred for $\frac{1}{2} \cdot (W \to \mathit{UAdm})$ and $\iota^{-1} : W \cong \text{Pred}$ for the inverse to $\iota$.

For an ultrametric space $(X, d)$ and $n \in \omega$ we use the notation $x \stackrel{n}{=} y$ to mean that $d(x, y) \leq 2^{-n}$. By the ultrametric inequality, each $\stackrel{n}{=}$ is an equivalence relation on $X$ [3]. Since all non-zero distances in *UAdm* are of the form $2^{-n}$ for some $n \in \omega$, this is also the case for the distance function on $W$. Therefore, to show that a map is non-expansive it suffices to show that $f(x) \stackrel{n}{=} f(y)$ whenever $x \stackrel{n}{=} y$. The definition of Pred has the following consequence: for $p, q \in \text{Pred}$, $p \stackrel{n}{=} q$ iff $p(w) \stackrel{n-1}{=} q(w)$ for all $w \in W$. This fact is used repeatedly in our proofs.

For $p, q \in \mathit{UAdm}$, the separating conjunction $p * q$ is defined as usual, by $h \in p * q \stackrel{def}{\Leftrightarrow} \exists h_1, h_2.\ h = h_1 \cdot h_2 \wedge h_1 \in p \wedge h_2 \in q$. This operation is lifted to non-expansive functions $p_1, p_2 \in \text{Pred}$ pointwise, by $(p_1 * p_2)(w) = p_1(w) * p_2(w)$. This is well-defined, and moreover determines a non-expansive operation on the space Pred. The corresponding unit for the lifted $*$ is the non-expansive function $\mathsf{emp} = \lambda w.\{\{\|\}, \bot\}$ (i.e., $p * \mathsf{emp} = \mathsf{emp} * p = p$, for all $p$). We let $emp = \iota(\mathsf{emp})$.

**Lemma 4.** *There exists a non-expansive map $\circ : W \times W \to W$ and a map $\otimes : \text{Pred} \times W \to \text{Pred}$ that is non-expansive in its first and contractive in its second argument, satisfying $q \circ r = \iota(\iota^{-1}(q) \otimes r * \iota^{-1}(r))$ and $p \otimes r = \lambda w.p(r \circ w)$ for all $p \in \text{Pred}$ and $q, r \in W$.*

*Proof.* The defining equations of both operations give rise to contractive maps, which have (unique) fixed points by Banach's fixed point theorem. □

**Lemma 5.** *$(W, \circ, emp)$ is a monoid in CBUlt. Moreover, $\otimes$ is an action of this monoid on* Pred*.*

*Proof.* First, *emp* is a left-unit for $\circ$, $emp \circ q = \iota((\lambda w.\iota^{-1}(emp)(q \circ w)) * \iota^{-1}(q)) = \iota(\iota^{-1}(q)) = q$. Using this, one shows that it is also a right-unit for $\circ$. Next, one shows by induction that for all $n \in \omega$, $\circ$ is associative up to distance $2^{-n}$, from which associativity follows. By the 1-boundedness of $W$ the base case is clear. For the inductive step $n > 0$, by definition of the distance function on Pred it suffices to show that for all $w \in W$, $\iota^{-1}((p \circ q) \circ r)(w) \overset{n-1}{=} \iota^{-1}(p \circ (q \circ r))(w)$. This equation follows from the definition of $\circ$ and the inductive hypothesis.

That $\otimes$ forms an action of $W$ on Pred follows from these properties of $\circ$. First, $p \otimes emp = \lambda w.p(emp \circ w) = p$ since *emp* is a unit for $\circ$. Next, $(p \otimes q) \otimes r = \lambda w.p(q \circ (r \circ w)) = \lambda w.p((q \circ r) \circ w) = p \otimes (q \circ r)$ by the associativity of $\circ$. □

**Semantics of triples and assertions** Since assertions appear in the pre- and post-conditions of Hoare triples, and triples can be nested inside assertions, the interpretation of assertions and triples must be defined simultaneously. To achieve this, we first define a notion of semantic triple.

**Definition 6 (Semantic triple).** *A* semantic Hoare triple *consists of predicates $p, q \in$* Pred *and a strict continuous function $c \in Heap \multimap T_{err}(Heap)$, written $\{p\}c\{q\}$. For $w \in W$, a semantic triple $\{p\}c\{q\}$ is* forced by *$w$, denoted $w \models \{p\}c\{q\}$, if for all $r \in UAdm$ and all $h \in Heap$:*

$$h \in p(w) * \iota^{-1}(w)(emp) * r \;\Rightarrow\; c(h) \in \mathrm{Ad}(q(w) * \iota^{-1}(w)(emp) * r),$$

*where* $\mathrm{Ad}(r)$ *denotes the least downward closed and admissible set of heaps containing $r$. A semantic triple is* valid*, written $\models \{p\}c\{q\}$, if $w \models \{p\}c\{q\}$ for all $w \in W$. We extend semantic triples from $Com = Heap \multimap T_{err}(Heap)$ to all $d \in Val$, by $w \models \{p\}d\{q\}$ iff $d = c$ for some command $c \in Com$ and $w \models \{p\}c\{q\}$. A triple holds* approximately up to level *$k$, $w \models_k \{p\}d\{q\}$, if $w \models \{p\}\pi_k; d; \pi_k\{q\}$.*

Thus, semantic triples bake in the first-order frame property (by conjoining $r$), and "close" the "open" recursion (by applying the world $w$, on which the triple implicitly depends, to *emp*). The admissible downward closure that is applied to the entire post-condition is in line with a partial correctness interpretation of triples. In particular, it entails that the sets $\{c \in Com \mid w \models_k \{p\}c\{q\}\}$ and $\{c \in Com \mid w \models \{p\}c\{q\}\}$ are admissible and downward closed subsets of *Com*. Finally, semantic triples are non-expansive, in the sense that if $w \overset{n}{=} w'$ for $n > 0$ and $w \models \{p\}c\{q\}$, then $w' \models_{n-1} \{p\}c\{q\}$. This observation plays a key role in the following definition of the semantics of nested triples. Another useful observation is that $w \models \{p\}c\{q\}$ is equivalent to $\forall k \in \omega.\, w \models_k \{p\}c\{q\}$.

Assertions are interpreted as elements $[\![P]\!]_\eta \in$ Pred. Note that $(UAdm, \subseteq)$ is a complete Heyting BI algebra. Using the pointwise extension of the operations of

$$
\begin{array}{rl}
[\![\mathsf{true}]\!]_\eta\, w = \mathit{Heap} & [\![P \wedge Q]\!]_\eta\, w = [\![P]\!]_\eta\, w \cap [\![Q]\!]_\eta\, w \\
[\![\mathsf{false}]\!]_\eta\, w = \{\bot\} & [\![P \vee Q]\!]_\eta\, w = [\![P]\!]_\eta\, w \cup [\![Q]\!]_\eta\, w \\
[\![\mathit{emp}]\!]_\eta\, w = \{\{\!|\}\!\}, \bot\} & [\![P * Q]\!]_\eta\, w = [\![P]\!]_\eta\, w * [\![Q]\!]_\eta\, w \\
[\![e_1 \mapsto e_2]\!]_\eta\, w = \{h \mid h \sqsubseteq \{\!|[\![e_1]\!]_\eta = [\![e_2]\!]_\eta|\!\}\} & [\![P \otimes Q]\!]_\eta\, w = ([\![P]\!]_\eta \otimes \iota([\![Q]\!]_\eta))w \\
[\![e_1 = e_2]\!]_\eta\, w = \{h \mid h \neq \bot \Rightarrow [\![e_1]\!]_\eta = [\![e_2]\!]_\eta\} & [\![\forall x.P]\!]_\eta\, w = \bigcap_{d \in \mathit{Val}} [\![P]\!]_{\eta[x:=d]}\, w \\
[\![\exists x.P]\!]_\eta\, w = \{h \mid \forall n \in \omega.\ \pi_n(h) \in \bigcup_{d \in \mathit{Val}} [\![P]\!]_{\eta[x:=d]}\, w\} & \\
[\![P \Rightarrow Q]\!]_\eta\, w = \{h \mid \forall n \in \omega.\ \pi_n(h) \in [\![P]\!]_\eta\, w \text{ implies } \pi_n(h) \in [\![Q]\!]_\eta\, w\} & \\
[\![\{P\}e\{Q\}]\!]_\eta\, w = \mathrm{Ad}\{h \in \mathit{Heap} \mid \mathit{rnk}(h) > 0 \Rightarrow w \models_{\mathit{rnk}(h)-1} \{[\![P]\!]_\eta\}\, [\![e]\!]_\eta\, \{[\![Q]\!]_\eta\}\} &
\end{array}
$$

**Fig. 7.** Semantics of assertions

this algebra to the set of non-expansive functions $W \to \mathit{UAdm}$, we also obtain a complete Heyting BI algebra on $\mathrm{Pred} = \frac{1}{2} \cdot (W \to \mathit{UAdm})$ which soundly models the intuitionistic predicate BI part of the assertion logic. Moreover, the monoid action of $W$ on Pred serves to model the invariant extension of the assertion logic. In order to define a non-expansive interpretation of nested triples we will use the following definition:

**Definition 7 (Rank of a heap).** *If $h$ is a compact element of Heap, then the least $r$ for which $\pi_r(h) = h$ is the* rank *of $h$, abbreviated $\mathit{rnk}(h)$, otherwise the rank is undefined.*

The interpretation of assertions is spelled out in detail in Fig. 7. The interpretation of a nested triple $\{P\}e\{Q\}$ is *not* independent of the heap, unlike the (more traditional) semantics of "top-level" triples, i.e. $\models\{p\}c\{q\}$. More precisely, the definition in Fig. 7 means that triples as assertions depend on the rank of the current heap. This is necessary to provide a *non-expansive* function from $W$ to $\mathit{UAdm}$ that provides enough "approximation information." Simpler definitions like $\{h \in \mathit{Heap} \mid w \models \{[\![P]\!]_\eta\}\, [\![e]\!]_\eta\, \{[\![Q]\!]_\eta\}\}$ are heap independent but not non-expansive. A similar approach has been taken in [3] to force non-expansiveness for a reference type constructor for ML-style references.

As a consequence of this interpretation, the axiom $\{\{A\}e\{B\} \wedge A\}e\{B\}$ does not hold; the inner triple is only approximately valid up to the level of the rank of the argument heap. Analogously, the following rule does not appear to be sound in our semantics:

$$\frac{\{A\}e\{B\} \Rightarrow \{P\}e'\{Q\}}{\{\{A\}e\{B\} \wedge P\}e'\{Q\}}$$

The opposite direction does actually hold. Axioms and rules like these are used, e.g., in [5] in proofs for recursion through the store; instead we use (EVAL).

**Soundness of the proof rules** We prove soundness of the proof rules listed in Sections 2 and 3. We first consider the distribution axioms for $- \otimes R$ in Fig. 2.

**Lemma 8 ($\otimes$-Dist, 1).** *The axiom $(P \otimes Q) \otimes R \Leftrightarrow P \otimes (Q \circ R)$ is valid.*

*Proof.* An instance of the fact that $\otimes$ is a monoid action (Lemma 5). □

**Lemma 9 ($\otimes$-Dist, 2).** *The axiom* $\{P\}e\{Q\} \otimes R \Leftrightarrow \{P \circ R\}e\{Q \circ R\}$ *is valid.*

*Proof.* The statement follows from the following claim: for all $p, q, r \in \mathrm{Pred}$, strict continuous $c : \mathit{Heap} \multimap T_{err}(\mathit{Heap})$ and all $w \in W$, $\iota(r) \circ w \models \{p\}c\{q\}$ iff $w \models \{p \otimes \iota(r) * r\}c\{q \otimes \iota(r) * r\}$. The proof of this claim uses the equation

$$(p \otimes \iota(r) * r)(w) * \iota^{-1}(w)(\mathit{emp}) = p(\iota(r) \circ w) * \iota^{-1}(\iota(r) \circ w)(\mathit{emp}),$$

which is a consequence of the definitions of $\otimes$ and $\circ$. □

The proofs of the remaining distribution axioms are easy since the logical connectives are interpreted pointwise, and *emp* and $(e_1 \mapsto e_2)$ are constant.

Next, we consider the rules for higher-order store given in Fig. 5.

**Lemma 10 ($\otimes$-Frame).** *The* $\otimes$-Frame *rule is sound: if* $h \in p(w)$ *for all* $h \in \mathit{Heap}$ *and* $w \in W$*, then* $h \in (p \otimes \iota(r))(w)$ *for all* $h \in \mathit{Heap}$*,* $w \in W$ *and* $r \in \mathrm{Pred}$.

*Proof.* Assume $\forall h.\forall w.h \in p(w)$. Let $r \in \mathrm{Pred}$, $w \in W$ and $h \in \mathit{Heap}$. We show that $h \in (p \otimes \iota(r))(w)$. Note that $(p \otimes \iota(r))(w) = p(\iota(r) \circ w)$ by the definition of $\otimes$. So, for $w' \stackrel{def}{=} \iota(r) \circ w$, the assumption shows that $h \in p(w') = (p \otimes \iota(r))(w)$. □

**Lemma 11 ($*$-Frame).** $\{P\}e\{Q\} \Rightarrow \{P * R\}e\{Q * R\}$ *is valid for all* $P, Q, R, e$.

*Proof.* We show that for all $w \in W$, $p, q, r \in \mathrm{Pred}$ and $c \in \mathit{Com}$, if $w \models \{p\}c\{q\}$, then $w \models \{p * r\}c\{q * r\}$. This implies the lemma as follows. If $k$ is the rank of $\pi_n(h)$ and $\pi_n(h) \in [\![\{P\}e\{Q\}]\!]\, w$, then $w \models_{k-1} \{[\![P]\!]_\eta\}\, [\![e]\!]_\eta\, \{[\![Q]\!]_\eta\}$. This lets us conclude $w \models_{k-1} \{[\![P * R]\!]_\eta\}\, [\![e]\!]_\eta\, \{[\![Q * R]\!]_\eta\}$, which in turn implies that $\pi_n(h)$ belongs to $[\![\{P * R\}e\{Q * R\}]\!]\, w$.

Assume $w \models \{p\}c\{q\}$. We must show that $w \models \{p * r\}c\{q * r\}$. Let $r' \in \mathit{UAdm}$ and assume $h \in (p * r)(w) * \iota^{-1}(w)(\mathit{emp}) * r' = p(w) * \iota^{-1}(w)(\mathit{emp}) * (r(w) * r')$. Since $w \models \{p\}c\{q\}$, it follows that $c(h) \in \mathrm{Ad}(q(w) * \iota^{-1}(w)(\mathit{emp}) * (r(w) * r')) = \mathrm{Ad}((q * r)(w) * \iota^{-1}(w)(\mathit{emp}) * r')$, which establishes $w \models \{p * r\}c\{q * r\}$. □

The proofs of the remaining rules (i.e., Eval and those in Fig. 3) are similar but omitted due to lack of space.

**Semantics of recursive assertions** The following general fixed point theorem is a consequence of Banach's fixed point theorem, and it allows us to introduce recursively defined assertions in the logic, as used in previous sections.

**Theorem 12 (Mutually recursive predicates).** *Let* $I$ *be a set and suppose that, for each* $i \in I$, $F_i : \mathrm{Pred}^I \to \mathrm{Pred}$ *is a contractive function. Then there exists a unique* $\boldsymbol{p} = (p_i)_{i \in I} \in \mathrm{Pred}^I$ *such that* $F_i(\boldsymbol{p}) = p_i$*, for all* $i \in I$.

Note that this theorem is sufficiently general to permit the mutual recursive definition of even infinite families of predicates.

As established in Lemma 4, $\otimes$ is contractive in its right-hand argument. Thus, for fixed $P$ and $\eta$, the map $F(r) = [\![P]\!]_\eta \otimes \iota(r)$ on Pred is contractive

and has a unique fixed point, $r$. Given an equation $R = P \otimes R$ we take $r$ as the interpretation of $R$, and note that indeed $[\![R]\!]_\eta = r = [\![P]\!]_\eta \otimes \iota(r) = [\![P \otimes R]\!]_\eta$. Along the same lines, we can interpret mutually recursive assertions: $R_1 = P_1 \otimes (R_1 * \ldots * R_n), \quad \ldots, \quad R_n = P_n \otimes (R_1 * \ldots * R_n)$. Using the non-expansiveness of $*$ as an operation on Pred, these equations give rise to contractive functions $F_i(r_1, \ldots, r_n) = [\![P_i]\!]_\eta \otimes \iota(r_1 * \ldots * r_n)$.

*Acknowledgments* We would like to thank François Pottier, Kristian Støvring and Jacob Thamsborg for helpful discussions. Kristian suggested that $\otimes$ is a monoid action. Partial support has been provided by FNU project 272-07-0305 "Modular reasoning about software", EPSRC projects EP/G003173/1 "From reasoning principles for function pointers to logics for self-configuring programs" and EP/E053041/1 "Scalable program analysis for software verification".

## References

1. P. America and J. J. M. M. Rutten. Solving reflexive domain equations in a category of complete metric spaces. *J. Comput. Syst. Sci.*, 39(3):343–375, 1989.
2. L. Birkedal, B. Reus, J. Schwinghammer, and H. Yang. A simple model of separation logic for higher-order store. In *ICALP*, volume 5126 of *LNCS*, pages 348–360. Springer, 2008.
3. L. Birkedal, K. Støvring, and J. Thamsborg. Realizability semantics of parametric polymorphism, general references, and recursive types. In *FOSSACS*, volume 5504 of *LNCS*, pages 456–470. Springer, 2009.
4. L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules for Algol-like languages. *Log. Methods Comput. Sci.*, 2(5:1), 2006.
5. K. Honda, N. Yoshida, and M. Berger. An observationally complete program logic for imperative higher-order functions. In *LICS*, pages 270–279. IEEE Computer Society, 2005.
6. N. Krishnaswami, L. Birkedal, J. Aldrich, and J. Reynolds. Idealized ML and Its Separation Logic. Available at http://www.cs.cmu.edu/~neelk/, 2007.
7. A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: dependent types for imperative programs. In *ICFP*, pages 229–240. ACM, 2008.
8. P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *B. Symb. Log.*, 5(2):215–244, 1999.
9. M. Parkinson and G. Biermann. Separation logic, abstraction and inheritance. In *POPL*, pages 75–86. ACM, 2008.
10. A. M. Pitts. Relational properties of domains. *Inf. Comput.*, 127:66–90, 1996.
11. F. Pottier. Hiding local state in direct style: a higher-order anti-frame rule. In *LICS*, pages 331–340. IEEE Computer Society, 2008.
12. B. Reus and J. Schwinghammer. Separation logic for higher-order store. In *CSL*, volume 4207 of *LNCS*, pages 575–590. Springer, 2006.
13. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.
14. M. B. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM J. Comput.*, 11(4):761–783, 1982.
15. T. Streicher. *Domain-theoretic Foundations of Functional Programming.* World Scientific, 2006.