Hoare logic for higher order store using simple semantics

Nathaniel Charlton

School of Informatics, University of Sussex
n.a.charlton@sussex.ac.uk

Abstract. We revisit the problem of providing a Hoare logic for higher order store programs, considered by Reus and Streicher (ICALP, 2005). In a higher order store program, the procedures/commands of the program are not fixed, but can be manipulated at runtime by the program itself; such programs provide a foundation to study language features such as reflection, dynamic loading and runtime code generation. By adapting the semantics of a proof system for a language with conventional (fixed) mutually recursive procedures, studied by von Oheimb (FSTTCS, 1999), we construct the same logic as Reus and Streicher, but using a much simpler model and avoiding unnecessary restrictions on the use of the proof rules. Furthermore our setup handles nondeterministic programs "for free". We also explain and demonstrate with an example that, contrary to what has been stated in the literature, such a proof system does support proofs which are (in a specific sense) modular.

Keywords: higher order store, Hoare logic, modular proof

1 Introduction

Higher order store is when a program's commands or procedures are not fixed, but are instead part of the mutable state which the program itself manipulates at runtime. Thus programming languages with higher order store allow programs to be self-modifying and self-configuring. Such languages can be used (as in e.g. [7]) to model phenomena such as runtime code generation and dynamic management of code, which is found in OS kernels, plugin systems and dynamic ("hot") software update systems.

Reus and Streicher [17] consider the problem of providing a Hoare logic for higher order store programs. They consider "arguably the simplest language that uses higher-order store" so that they can focus solely on how to account for higher order store. This language contains programs such as the following:

$$x := \text{`}\mathsf{run}\ x\text{'}\ ;\quad \mathsf{run}\ x \qquad (1) \qquad\qquad x := \text{`}n := 100\ ;\ x := \text{`}n := n - 1\text{''}\ ;\quad \mathsf{run}\ x\ ;\ \mathsf{run}\ x \qquad (2)$$

Program (1) constructs a non-terminating recursion: the command $\mathsf{run}\ x$ is written into variable x, and then invoked using the $\mathsf{run}\ x$ command. This leads to the code stored in x invoking itself endlessly. Program (1) should satisfy the Hoare

triple $\{true\} - \{false\}$. The fact that new recursions can be set up on-the-fly in this way was observed by Landin [9] and is sometimes called *recursion through the store* or *tying the knot*. In program (2) we store a self-modifying command in x: when first run, this command sets n to 100 and then replaces itself with the command $n := n - 1$. Program (2) should satisfy $\{true\} - \{n = 99\}$.

How should one give semantics to such a language? Reus and Streicher take the view that the commands stored in variables should be represented semantically as store transformers, that is, as (partial) functions $\mathsf{Store} \rightharpoonup \mathsf{Store}$. But since stores map variables to values, and values include commands, this makes the notions of command and store mutually recursive. Thus, [17] uses domain theory to solve the following system of recursive equations:

$$\mathsf{Store} = \mathsf{Var} \to \mathsf{Val} \qquad \mathsf{Val} = \mathsf{BVal} + \mathsf{Cmd} \qquad \mathsf{Cmd} = \mathsf{Store} \rightharpoonup \mathsf{Store}$$

where BVal is some basic values such as integers. The rest of the development of *loc. cit.*, which gives an assertion language and Hoare proof rules, then requires a rather intricate domain-theoretic argument. Specifically, results by Pitts [14] concerning the existence of invariant relations on recursively defined domains are needed. Even then, the domain theory leaks out into the logic: several of the rules are restricted in that they can only be used with assertions that are "downwards closed" (a semantic property which may not be intuitive to potential users of the logic, unless they are domain-theorists).

However, when one looks closely at the new proof rules in [17], one sees that they are not so new after all; rather, they are very similar to the rules used by von Oheimb [12] and later others (e.g. Nipkow [11] and Parkinson [13]) for reasoning about mutually recursive procedures in conventional procedural languages (where procedures cannot be updated at runtime). But, noticing this, we are left with an apparent disparity: von Oheimb's proof rules are justified using rather basic mathematics, namely proof by induction on the number of procedure calls made by an execution sequence. If the rules of [17] are so similar, why should the underlying semantics be much more complicated?

In Sections 2, 3 and 4 we resolve this question by showing that using induction on execution length we can reconstruct Reus and Streicher's logic. Not only does this give a simpler model of the logic, but we also eliminate restrictions which [17] places on the use of the proof rules. The key to making this work is to use a simpler, flat model of program states, where all values (including commands) are encoded as integers. In Section 5 we see that our setup handles nondeterministic programs "for free". Section 6 explains that, contrary to what has been stated in the literature, the logic studied here supports proofs which are modular (in a specific sense which we explain). An example of such a proof is provided. Section 7 discusses related and future work, and concludes.

The significance of our results is that they show that, for certain kinds of reasoning about higher order store, it is not necessary to use "sophisticated" domain-theoretic techniques, and in fact one fares better without them.

$$
\begin{array}{llll}
\text{expressions} & e & ::= & 0 \mid 1 \mid -1 \mid \ldots \mid e_1 + e_2 \mid e_1 = e_2 \mid \ldots \mid v \mid \text{`}C\text{'} \\
\text{commands} & C & ::= & \mathsf{nop} \mid x := e \mid C_1; C_2 \mid \mathsf{if}\ e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid \mathsf{run}\ x \\
\text{assertions} & P, Q & ::= & P_1 \wedge P_2 \mid \neg P \mid \forall v.P \mid e_1 \leq e_2 \\
\text{contexts} & \Gamma & ::= & \varepsilon \mid \{P\}\, e\, \{Q\}\, , \Gamma \\
\text{specifications} & S & ::= & \{P\}\, e\, \{Q\} \mid \Gamma \vdash \{P\}\, e\, \{Q\}
\end{array}
$$

Fig. 1. Syntax of expressions, commands, assertions, contexts and specifications.

$$
\frac{}{(x := e, s) \to (\mathsf{nop},\ \lambda v.\ \text{if } v = x \text{ then } [\![e]\!]^{\mathrm{ex}}_s \text{ else } s(v))}
\qquad
\frac{(C_1, s) \to (C_1', s')}{(C_1; C_2, s) \to (C_1'; C_2, s')}
$$

$$
\frac{}{(\mathsf{nop}; C, s) \to (C, s)}
\qquad
\frac{[\![e]\!]^{\mathrm{ex}}_s = 1}{(\mathsf{if}\ e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s) \to (C_1, s)}
$$

$$
\frac{[\![e]\!]^{\mathrm{ex}}_s \neq 1}{(\mathsf{if}\ e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s) \to (C_2, s)}
\qquad
\frac{C = \mathscr{G}^{-1}(s(x))}{(\mathsf{run}\ x, s) \to (C, s)}
$$

Fig. 2. Small-step operational semantics of programs.

2 Programs, assertions and specifications

We begin by giving the syntax and semantics of a programming language for higher order store programs, and accompanying assertion and specification languages. The syntax is essentially as in [17] except where indicated, but crucially the semantics takes a different approach.

Let $\mathsf{Store} \triangleq \mathsf{Var} \to \mathbb{Z}$ be the set of program stores and let $\mathsf{Env} \triangleq \mathsf{AuxVar} \to \mathbb{Z}$ be the set of environments for auxiliary variables (throughout we use $\triangleq$ to make definitions, to avoid confusion with the assignment symbol $:=$ which occurs in programs). We fix a bijection $\mathscr{G}$ mapping (syntactic) commands to integers; we will use this to encode commands. Thus all values, including commands, are encoded as integers. We call this a *flat* model because, unlike in [17], no complicated order structure on Store is required.

2.1 Expressions, programs and assertions

The syntax of expressions e, commands C and assertions P is given in Fig. 1. Variables v can be of two kinds: ordinary variables $x, y, \ldots \in \mathsf{Var}$, and auxiliary variables $\mathrm{p}, \mathrm{q}, \ldots \in \mathsf{AuxVar}$ which may not appear in programs. The quote expression 'C' turns the command C into a value so it can be stored in a variable, and run later. 'C' has no free variables. Using the available assertion language one can express *true*, $\vee$, $\exists$, $=$ and so on in the usual way. Because we use a flat

store model and encode commands as integers, we will not need the type check $\tau?e$ from [17], and the typed comparison $\leq_\tau$ becomes $\leq$ on integers.

Semantics of expressions: We write $[\![e]\!]^{\mathrm{ex}}_{s,\rho}$ for the value of expression e in store s and environment ρ. Where e contains no ordinary (resp. auxiliary) variables we omit s (resp. ρ). Expression evaluation is standard apart from the case of 'C', where we simply use the encoding $\mathscr{G}$, defining $[\![\text{`}C\text{'}]\!]^{\mathrm{ex}} \triangleq \mathscr{G}(C)$.

Semantics of programs: Fig. 2 gives a small-step operational semantics[1]. A *configuration* is a pair (C, s) of a command and a store, and is *terminal* if C is nop. One execution step is written $(C, s) \to (C', s')$ and if there is an execution sequence of zero or more steps from (C, s) to (C', s') we write $(C, s) \xrightarrow{*} (C', s')$. Note the semantics of the $\mathsf{run}\ x$ command : we just read an integer value from x, turn it back into a (syntactic) command with $\mathscr{G}^{-1}$, and run it.

Semantics of assertions: The semantics $[\![P]\!]^{\mathrm{as}}_{\rho} \subseteq \mathsf{Store}$ of assertion P in environment ρ is completely standard so we omit it. Entailment $P \Rightarrow Q$ means that for all ρ, $[\![P]\!]^{\mathrm{as}}_{\rho} \subseteq [\![Q]\!]^{\mathrm{as}}_{\rho}$.

2.2 Contexts and specifications

Next we introduce *contexts* Γ and *specifications* S, also shown in Fig. 1. A context is a collection of Hoare triples. A specification is either a Hoare triple $\{P\}e\{Q\}$, or a triple in context $\Gamma \vdash \{P\}\, e\, \{Q\}$. Intuitively the latter means that $\{P\}\, e\, \{Q\}$ holds in contexts where Γ holds. (In triple $\{P\}\, e\, \{Q\}$ the expression e must not contain free ordinary variables.) Before we can give semantics to contexts and specifications, we state our (partial correctness) interpretation of Hoare triples.

Definition 1. Semantics of Hoare triples. *We write* $\rho \vDash^n \{P\}\, e\, \{Q\}$ *to mean, putting* $C \triangleq \mathscr{G}^{-1}([\![e]\!]^{\mathrm{ex}}_{\rho})$*, that for all stores* $s \in [\![P]\!]^{\mathrm{as}}_{\rho}$*, if* $(C, s) \xrightarrow{*} (\mathsf{nop}, s')$ *in* n *steps or fewer then* $s' \in [\![Q]\!]^{\mathrm{as}}_{\rho}$*. We write* $\rho \vDash \{P\}\, e\, \{Q\}$ *to mean that* $\rho \vDash^n \{P\}\, e\, \{Q\}$ *for all* $n \in \mathbb{N}$*.*

The semantics $[\![S]\!]^{\mathrm{sp}} \subseteq \mathsf{Env}$ of specification S (and the semantics $[\![\Gamma]\!]^{\mathrm{co}} \subseteq \mathsf{Env} \times \mathbb{N}$ of context Γ) is then as in Fig. 3. For example, $\{A\}\, e\, \{B\} \vdash \{P\}\, e'\, \{Q\}$ means that in a context where command e satisfies triple $\{A\}\_\{B\}$ for executions of up to length $n-1$, the command e' satisfies triple $\{P\}\_\{Q\}$ for executions of up to length n. This semantics (which is not a conventional implication) will fit naturally with proof by induction on execution length. Note also that if $(\rho, n) \in [\![\Gamma]\!]^{\mathrm{co}}$ and $0 \leq m < n$ then $(\rho, m) \in [\![\Gamma]\!]^{\mathrm{co}}$.

We write $S_1, \ldots, S_k \vDash S$ to mean that (the conjunction of) specifications $S_1, \ldots, S_k$ entails the specification S, that is, if $[\![S_1]\!]^{\mathrm{sp}} = \ldots = [\![S_k]\!]^{\mathrm{sp}} = \mathsf{Env}$ then $[\![S]\!]^{\mathrm{sp}} = \mathsf{Env}$. (When $k = 0$ we write just $\vDash S$ meaning $[\![S]\!]^{\mathrm{sp}} = \mathsf{Env}$.)

[1] We could also carry out our development using denotational semantics, *as long as we kept a flat store model.* The flat store model is the important thing, and we are not trying to say that operational semantics is "better" than denotational semantics.

$$[\![\varepsilon]\!]^{\mathrm{co}} \triangleq \mathsf{Env} \times \mathbb{N} \qquad [\![\{P\}\, e\, \{Q\}, \Gamma]\!]^{\mathrm{co}} \triangleq \{(\rho, n) \mid \rho \vDash^n \{P\}\, e\, \{Q\}\} \cap [\![\Gamma]\!]^{\mathrm{co}}$$

$$[\![\{P\}\, e\, \{Q\}]\!]^{\mathrm{sp}} \triangleq \{\rho \in \mathsf{Env} \mid \rho \vDash \{P\}\, e\, \{Q\}\}$$

$$[\![\Gamma \vdash \{P\}\, e\, \{Q\}]\!]^{\mathrm{sp}} \triangleq \left\{ \rho \in \mathsf{Env} \,\middle|\, \begin{array}{l} \forall n \in \mathbb{N}, \text{ if } n = 0 \text{ or } (n > 0 \text{ and } (\rho, n-1) \in [\![\Gamma]\!]^{\mathrm{co}}) \\ \text{then } \rho \vDash^n \{P\}\, e\, \{Q\} \end{array} \right\}$$

Fig. 3. Semantics of contexts Γ and specifications S.

$$\mathrm{A}\ \frac{}{\{P[e/x]\}\,\text{`}x := e\text{'}\,\{P\}} \qquad \mathrm{S}\ \frac{\Gamma \vdash \{P\}\,\text{`}C_1\text{'}\,\{R\} \qquad \Gamma \vdash \{R\}\,\text{`}C_2\text{'}\,\{Q\}}{\Gamma \vdash \{P\}\,\text{`}C_1; C_2\text{'}\,\{Q\}}$$

$$\mathrm{I}\ \frac{\Gamma \vdash \{P \wedge e = 1\}\,\text{`}C_1\text{'}\,\{Q\} \qquad \Gamma \vdash \{P \wedge e \neq 1\}\,\text{`}C_2\text{'}\,\{Q\}}{\Gamma \vdash \{P\}\,\text{`}\mathsf{if}\ e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\text{'}\,\{Q\}}$$

$$\mathrm{W}\ \frac{\Gamma \vdash \{P'\}\, e\, \{Q'\}}{\Gamma \vdash \{P\}\, e\, \{Q\}}\ P \Rightarrow P', Q' \Rightarrow Q \qquad \epsilon\ \frac{}{\{P\}\,\text{`}\mathsf{nop}\text{'}\,\{P\}} \qquad \mathrm{C}\ \frac{\Gamma \vdash \{P\}\, e\, \{Q\}}{T, \Gamma \vdash \{P\}\, e\, \{Q\}}$$

Fig. 4. Standard Hoare logic rules, supported by the logic we study.

3 Proof rules

In this section we state the proof rules for the logic we work with. The logic features the standard Hoare logic rules for assignment, sequential composition, consequence etc. given in Fig. 4. These are presented as in [17] except that we are explicit about the presence of a context Γ. The interesting rules are those for running stored commands, given in Fig. 5. We can make these rules simpler than those of [17] in two respects, due to our flat store model. Firstly the side conditions about downwards closure of assertions have been eliminated. Secondly we can simply use equality on commands, rather than a partial order $\leq_{\mathrm{com}}$.

Rule (R) is used when we know the stored command C which will be invoked, and have already derived a triple for it. Rule (H) is for making use of contexts: in a context where $\{P \wedge x = \mathrm{p}\}\,\mathrm{p}\,\{Q\}$ holds for executions up to length $n-1$, $\{P \wedge x = \mathrm{p}\}\,\text{`}\mathsf{run}\ x\text{'}\,\{Q\}$ will hold for executions up to length n. Finally rule (μ) is used for reasoning about mutual recursion (through the store): intuitively the premise says that *if* all commands C_j involved satisfy their specifications $\{P_j\}\,\_\,\{Q_j\}$ for executions up to length $n-1$, *then* each command C_i also satisfies its specification for executions up to length n. Unsurprisingly when we later prove soundness of (μ) we will use induction on n.

We will see these rules in action in Section 6. As stated earlier, however, we emphasise that these rules can be seen as adaptations of von Oheimb's rules for

$$\text{R}\quad \frac{\Gamma \vdash \{P \wedge x = \text{`}C\text{'}\}\,\text{`}C\text{'}\,\{Q\}}{\Gamma \vdash \{P \wedge x = \text{`}C\text{'}\}\,\text{`}\mathsf{run}\ x\text{'}\,\{Q\}} \qquad \text{H}\quad \frac{}{\{P \wedge x = \mathrm{p}\}\,\mathrm{p}\,\{Q\} \ \vdash\ \{P \wedge x = \mathrm{p}\}\,\text{`}\mathsf{run}\ x\text{'}\,\{Q\}}$$

$$\mu\quad \frac{\bigwedge_{1\le i\le N} \Gamma, \{P_1\}\,\mathrm{p}_1\,\{Q_1\}, \ldots, \{P_N\}\,\mathrm{p}_N\,\{Q_N\} \ \vdash\ \{P_i\}\,\text{`}C_i\text{'}\,\{Q_i\}}{\bigwedge_{1\le i\le N} \Gamma \vdash \left\{P_i[\vec{C}/\vec{\mathrm{p}}]\right\}\,\text{`}C_i\text{'}\,\left\{Q_i[\vec{C}/\vec{\mathrm{p}}]\right\}} \quad \begin{array}{l}\mathrm{p}_1, \ldots, \mathrm{p}_N \text{ not free in } \Gamma\\ \vec{C} \text{ is } \text{`}C_1\text{'}, \ldots, \text{`}C_N\text{'}\\ \vec{p} \text{ is } p_1, \ldots, p_N\end{array}$$

Fig. 5. Hoare logic rules for the run statement which runs stored commands.

(conventional i.e. fixed) recursive procedures to a programming language with higher order store. Specifically, (R) and (μ) taken together strongly resemble the *Call* rule of [12], while (H) strongly resembles the *asm* rule. (For the reader's convenience, the *Call* and *asm* rules are reproduced in the Appendix, Sec. A.2.)

4 Soundness of the logic

Having stated the proof rules, we must of course show that they are sound. Soundness proofs for the standard rules (Fig. 4) are straightforward and omitted (except, as an example, the (S) rule is proved in the Appendix, Thm. 3). This leaves the rules for stored commands. We prove soundness of (H) and (μ); as the proofs for (R) and (H) are very similar, we defer the proof of (R) to the Appendix (Thm. 4). As the reader will see, no complicated theory is needed.

Theorem 1. *Rule (H) is sound.*

Proof. Let $\rho \in \mathsf{Env}$, $n \in \mathbb{N}$ be such that $n = 0$ or ($n > 0$ and $(\rho, n-1) \in [\![\{P \wedge x = \mathrm{p}\}\,\mathrm{p}\,\{Q\}]\!]^{\mathrm{co}}$). We must prove $\rho \vDash^n \{P \wedge x = \mathrm{p}\}\,\text{`}\mathsf{run}\ x\text{'}\,\{Q\}$. If $n = 0$ then this is trivially true, so let $n > 0$. Then from $(\rho, n-1) \in [\![\{P \wedge x = \mathrm{p}\}\,\mathrm{p}\,\{Q\}]\!]^{\mathrm{co}}$ we get (A.) $\rho \vDash^{n-1} \{P \wedge x = \mathrm{p}\}\,\mathrm{p}\,\{Q\}$.

Let $s \in [\![P \wedge x{=}\mathrm{p}]\!]^{\mathrm{as}}_\rho$ and s' be such that $(\mathsf{run}\ x, s) \xrightarrow{*} (\mathsf{nop}, s')$ in n steps or fewer; we are required to show $s' \in [\![Q]\!]^{\mathrm{as}}_\rho$. Due to the structure of the transition relation $\rightarrow$, we must have $(C, s) \xrightarrow{*} (\mathsf{nop}, s')$ in $n{-}1$ steps or fewer, where $C = \mathscr{G}^{-1}(s(x))$. From this, $s(x) = \rho(\mathrm{p})$ and (A.) we have $s' \in [\![Q]\!]^{\mathrm{as}}_\rho$ as required. □

Theorem 2. *Rule (μ) is sound.*

Proof. Let $\Phi(n)$ be the statement that for all $\rho \in \mathsf{Env}$ and all $i \in \{1, \ldots, N\}$,

$$n = 0 \text{ or } (n > 0 \text{ and } (\rho, n-1) \in [\![\Gamma]\!]^{\mathrm{co}}) \quad \text{implies} \quad \rho \vDash^n \{P_i[\vec{C}/\vec{\mathrm{p}}]\}\text{`}C_i\text{'}\{Q_i[\vec{C}/\vec{\mathrm{p}}]\}$$

It will suffice to prove $\Phi(n)$ for all $n \in \mathbb{N}$, which we shall do by induction.

Base case: Let $\rho \in \mathsf{Env}$ and let $i \in \{1, \ldots, N\}$. Let $\hat{\rho}$ be equal to ρ except at $\mathrm{p}_1, \ldots, \mathrm{p}_N$, which are mapped respectively to $[\![\text{`}C_1\text{'}]\!]^{\mathrm{ex}}, \ldots, [\![\text{`}C_N\text{'}]\!]^{\mathrm{ex}}$. From the premise of (μ), unpacking the definitions and instantiating n with 0 and ρ with $\hat{\rho}$, we obtain $\hat{\rho} \vDash^0 \{P_i\}\,\text{`}C_i\text{'}\,\{Q_i\}$. Using familar properties of substitution, this implies the thing we needed to prove, which is: $\rho \vDash^0 \{P_i[\vec{C}/\vec{\mathrm{p}}]\}\text{`}C_i\text{'}\{Q_i[\vec{C}/\vec{\mathrm{p}}]\}$.

Inductive case: Let $n > 0$ and let $\Phi(n-1)$ hold. Let $\rho \in \mathsf{Env}$ be such that $(\rho, n-1) \in [\![\Gamma]\!]^{\mathrm{co}}$ and let $i \in \{1, \ldots, N\}$. Define $\hat{\rho}$ as in the base case. We must prove $\rho \vDash^n \{P_i[\vec{C}/\vec{\mathrm{p}}]\}\text{`}C_i\text{'}\{Q_i[\vec{C}/\vec{\mathrm{p}}]\}$ which is equivalent to

$$\hat{\rho} \vDash^n \{P_i\}\text{`}C_i\text{'}\{Q_i\} \tag{3}$$

using familiar properties of substitution. Note that $(\hat{\rho}, n-1) \in [\![\Gamma]\!]^{\mathrm{co}}$ because $(\rho, n-1) \in [\![\Gamma]\!]^{\mathrm{co}}$ and $\mathrm{p}_1, \ldots, \mathrm{p}_N$ are not free in Γ. We next show

$$n - 1 = 0 \text{ or } (n - 1 > 0 \text{ and } (\hat{\rho}, n-2) \in [\![\Gamma]\!]^{\mathrm{co}}) \tag{4}$$

Suppose $n - 1 > 0$ i.e. $n > 1$. We know that $(\hat{\rho}, n-1) \in [\![\Gamma]\!]^{\mathrm{co}}$ and therefore $(\hat{\rho}, n-2) \in [\![\Gamma]\!]^{\mathrm{co}}$. So (4) holds. It now follows from (4) and the induction hypothesis $\Phi(n-1)$, instantiating ρ with $\hat{\rho}$, that

$$\text{for all } j \in \{1, \ldots, N\}, \quad \hat{\rho} \vDash^{n-1} \{P_j[\vec{C}/\vec{\mathrm{p}}]\}\text{`}C_j\text{'}\{Q_j[\vec{C}/\vec{\mathrm{p}}]\}$$

which in turn, using familiar properties of substitution, gives us

$$\text{for all } j \in \{1, \ldots, N\}, \quad \hat{\rho} \vDash^{n-1} \{P_j\}\mathrm{p}_j\{Q_j\} \tag{5}$$

From the premise of (μ), unpacking the definitions and instantiating n with n and ρ with $\hat{\rho}$, we find that (3) follows from (5) and $(\hat{\rho}, n-1) \in [\![\Gamma]\!]^{\mathrm{co}}$. □

5 Nondeterministic programs

Our setup handles nondeterminism for free, because nowhere in our proofs did we rely on determinism of the transition relation $\rightarrow$. For example, if we add a statement $\mathsf{choose}\ C_1\ C_2$ which makes transitions $(\mathsf{choose}\ C_1\ C_2, s) \rightarrow (C_1, s)$ and $(\mathsf{choose}\ C_1\ C_2, s) \rightarrow (C_2, s)$, it is easy to prove the appropriate Hoare rule:

$$\frac{\Gamma \vdash \{P\}\,\text{`}C_1\text{'}\,\{Q\} \qquad \Gamma \vdash \{P\}\,\text{`}C_2\text{'}\,\{Q\}}{\Gamma \vdash \{P\}\,\text{`}\mathsf{choose}\ C_1\ C_2\text{'}\,\{Q\}}$$

This is in contrast to the approach of [17] where nondeterminism causes problems. This is not an insignificant point: in [16], which extends the ideas of [17] to programs using pointers, a good deal of difficulty is caused by the nondeterminism of dynamic memory allocation. The explanation given is that in the presence of nondeterminism, "programs no longer denote ω-continuous functions".

6 Modular proofs

We now turn to the issue of modular proofs. In [17] the authors state that their logic "is not modular as all code must be known in advance and must be carried around in assertions"; they further state that it is "highly unlikely" that a modular logic exists at all, ascribing this to the lack of a "Bekic lemma" for their semantics. This belief is reiterated in later work, e.g. in [20]:

> However, the formulation ... 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 ... does not achieve this. (6)

(From here on the word "modular" is meant in the sense described in this quote.)

However, it is well known that proof rules such as von Oheimb's support modular proofs; this is the basis of program verifiers (e.g. [10, 4]) which check programs one procedure at a time. Therefore, noting their similarity to von Oheimb's rules, it stands to reason that the rules of Reus and Streicher which we work with should already support modular proofs. We now demonstrate using a simple program that this is indeed the case.

Consider the following program C_0:

$$f := \text{`}C_1\text{'} \;;\; g := \text{`}C_2\text{'} \;;\; \mathsf{run}\ f$$

where the commands C_1 and C_2 stored in f and g respectively are defined as follows:

$$\begin{array}{l} C_1 \triangleq \\ \quad \mathsf{if}\ (x \times x) + (y \times y) = (z \times z) \\ \quad \mathsf{then\ nop\ else\ run}\ g \end{array}$$

$$\begin{array}{l} C_2 \triangleq \\ \quad (\mathsf{if}\ x = n\ \mathsf{then}\ n := n + 1; x := 0 \\ \quad \mathsf{else\ if}\ y = n\ \mathsf{then}\ x := x + 1; y := 0 \\ \quad \mathsf{else\ if}\ z = n\ \mathsf{then}\ y := y + 1; z := 0 \\ \quad \mathsf{else}\ z := z + 1)\ ; \\ \quad \mathsf{run}\ f \end{array}$$

This program searches for a Pythagorean triple, that is, numbers x, y, z satisfying the predicate $R(x, y, z) \triangleq x^2 + y^2 = z^2$, stopping when one is found. Note that we establish a mutual recursion *through the store* between the C_1 code (stored in f) and the C_2 code (stored in g). The C_1 code tests whether the current values of variables x, y, z form a Pythagorean triple and terminates if so; otherwise C_1 runs the code in g to continue the search. The C_2 code updates x, y and z to the next triple to try, before invoking the code in f to perform the next test.

We would like to prove that this program works as intended, i.e. satifies

$$\{\mathit{true}\}\, C_0\, \{R(x, y, z)\} \tag{7}$$

But we would also like our proof to be modular, as described in (6), so that we do not have to completely redo our proof if we later change either C_1 (e.g. so that we search for values x, y, z with a different property) or C_2 (e.g. so that we exploit symmetry and only try triples where $x \leq y$). We shall now see how this can be accomplished. Let $T(e)$ be the triple

$$\{\, f = \mathrm{p} \wedge g = \mathrm{q} \,\} \;\; e \;\; \{\, R(x, y, z) \,\}$$

Then, we split our proof into three *independent* pieces.

For C_1: Prove $S_1 \triangleq$ $\quad \vDash T(\mathrm{p}), T(\mathrm{q}) \vdash T(\text{`}C_1\text{'})$

For C_2: Prove $S_2 \triangleq$ $\quad \vDash T(\mathrm{p}), T(\mathrm{q}) \vdash T(\text{`}C_2\text{'})$

For C_0: Prove $S_0 \triangleq$ $\quad S_1, S_2 \vDash \{\mathit{true}\}\, C_0\, \{R(x, y, z)\}$

Together, these three pieces trivially imply (7). We emphasise that in S_1 above the concrete code for C_2 does *not* appear, only a specification of its behaviour (on the left of $\vdash$), as described in (6). Similarly in S_2 the concrete code for C_1 does not appear, only a specification of its behaviour on the left of $\vdash$.

Proofs of S_0 and S_2 now follow (the proof of S_1 is deferred to the Appendix, Sec. A.1); these proofs demonstrate the use of the (R), (H), and (μ) rules. Note that only the proof for S_1 depends on the definition of predicate R.

Proof (for S_0 piece). In full, the proof obligation S_0 is

$$S_1, S_2 \vDash \{\mathit{true}\}\, f := \text{`}C_1\text{'} \;;\; g := \text{`}C_2\text{'} \;;\; \mathsf{run}\ f\, \{R(x, y, z)\}$$

Standard Hoare logic reasoning for assignments and sequential composition reduces this obligation to

$$S_1, S_2 \vDash \{f = \text{`}C_1\text{'} \wedge g = \text{`}C_2\text{'}\}\, \text{`}\mathsf{run}\ f\text{'}\, \{R(x, y, z)\}$$

By transitivity of $\vDash$ it is enough to show

$$S_1, S_2 \vDash \{f = \text{`}C_1\text{'} \wedge g = \text{`}C_2\text{'}\}\, \text{`}C_1\text{'}\, \{R(x, y, z)\} \tag{8}$$

and

$$\begin{array}{c} \{f = \text{`}C_1\text{'} \wedge g = \text{`}C_2\text{'}\}\, \text{`}C_1\text{'}\, \{R(x, y, z)\} \\ \vDash \quad \{f = \text{`}C_1\text{'} \wedge g = \text{`}C_2\text{'}\}\, \text{`}\mathsf{run}\ f\text{'}\, \{R(x, y, z)\} \end{array} \tag{9}$$

(9) is easily seen to be an instance of rule (R). To deduce (8) we start with the following instance of (μ)

$$\frac{\bigwedge_{i=1,2} \; T(p), T(q) \;\; \vdash \;\; T(\text{`}C_i\text{'})}{\bigwedge_{i=1,2} \{f = \text{`}C_1\text{'} \wedge g = \text{`}C_2\text{'}\}\, \text{`}C_i\text{'}\, \{R(x, y, z)\}}$$

If we use only the $i = 1$ part of the conclusion, we get

$$\frac{T(p), T(q) \vdash T(\text{`}C_1\text{'}) \qquad T(p), T(q) \vdash T(\text{`}C_2\text{'})}{\{f = \text{`}C_1\text{'} \wedge g = \text{`}C_2\text{'}\}\, \text{`}C_1\text{'}\, \{R(x, y, z)\}}$$

which is just (8) written in a different form.

□

Proof (for S_2 piece). By the (C) rule, S_2 will follow from $T(\mathrm{p}) \vdash T(\text{`}C_2\text{'})$. By the (S) rule this will follow from

$$T(\mathrm{p}) \vdash \left\{ f = \mathrm{p} \wedge g = \mathrm{q} \right\} \begin{array}{l} \text{`if } x = n \text{ then } n := n + 1; x := 0 \\ \text{else if } y = n \text{ then } x := x + 1; y := 0 \\ \text{else if } z = n \text{ then } y := y + 1; z := 0 \\ \text{else } z := z + 1\text{'} \end{array} \left\{ f = \mathrm{p} \wedge g = \mathrm{q} \right\}$$

and

$$T(\mathrm{p}) \vdash \{f = \mathrm{p} \wedge g = \mathrm{q}\}\, \text{`run } f\text{'}\, \{R(x, y, z)\}$$

The former is trivial; the latter is an instance of the (H) rule. □

Suppose we now replace our implementation C_2 with another implementation $\hat{C}_2$, which tries the triples (x, y, z) in a different order. We can reuse our existing proofs of S_1 and S_0; showing $\vDash T(\mathrm{p}), T(\mathrm{q}) \vdash T(\text{`}\hat{C}_2\text{'})$ is the only new work[2][3]. This proof is modular in the same way that proofs about programs with (fixed) recursive procedures can be made modular. If one uses the rules of [12] to show correctness of a program with mutually recursive procedures, and then changes the body of one procedure, then the verification conditions for all other procedures stay the same, and their existing proofs can be reused.

A remark on generalising the (μ) rule.

With the (μ) rule we can deal with recursion, even in cases where stored commands participating in a mutual recursion are updated during the recursion. However, as presented here (μ) only allows one to consider *finitely many* commands $C_1, \ldots, C_N$. If one extends the programming language, e.g. with features for runtime code generation, this may no longer be adequate: at runtime a program may have a countably infinite choice of commands it can generate.

We have also developed a generalised version of (μ) which covers these cases. The main idea is that in the generalised rule the variables $\mathrm{p}_1, \ldots, \mathrm{p}_N$ refer not to single commands, but to possibly infinite *sets* of commands. Assertions $x \in \mathrm{p}$

[2] Here we see why the lack of a "Bekic lemma" mentioned in [17] is not a problem. When we change the implementation of C_2 to $\hat{C}_2$, from the denotational viewpoint the application of the (μ) rule inside the proof of S_0 just "recomputes" the joint fixed point of C_1 and $\hat{C}_2$.

[3] Strictly, one might wish to explcitly put a universal quantification over R in proof obligations S_2 and S_0. This would be easy to add, but for our present purposes we simply note that the proofs of S_2 and S_0 do not rely on any properties of R, and thus will remain valid proofs whatever the choice of R.

are used instead of $x = \mathrm{p}$, and in the premise, each triple in the context now describes the behaviour of a whole set of commands.

7 Related work, future work and conclusions

Conclusions

We revisited the Hoare logic given in [17] for a higher order store programming language. We observed that the logic's proof rules strongly resemble those used by von Oheimb [12] for reasoning about programs with conventional (fixed) recursive procedures. This initially appeared puzzling, because whereas von Oheimb's rules are justified using a very simple semantic model, the proofs in [17] depend on the (significantly more complicated) domain-theoretic apparatus developed by Pitts.

We resolved this apparent disparity by giving a simple model of the same logic, using a flat store and induction on execution length to prove the recursion rule (μ). This also allowed us to drop restrictions on the use of the proof rules, and handle nondeterministic programs with no extra work. Finally we demonstrated that, contrary to what has been said in [17, 20], the logic does support modular proofs.

These results show that, for certain kinds of reasoning about higher order store, it is not necessary to use "sophisticated" domain-theoretic techniques, and in fact one fares better without them.

Related and future work

[16, 6] study the application of the ideas of [17] to a programming language with a heap, adding *separation logic* connectives [18] to the assertion language. Then, to overcome the perceived lack of modularity of these logics, [20, 21] add *nested Hoare triples*. However, these nested triples lead to even greater theoretical complications: [20, 21] use Kripke models based on recursively defined ultrametric spaces.

Hence, in the light of what we now know — that the logic of [17] can be given a very simple semantics, and already supports modular proofs — it will be interesting to revisit [20, 21] and see whether there is anything which can be accomplished using logics with nested triples, which cannot be supported using a more conventional logic of the kind considered in this paper. The *anti-frame rule* [15, 21] may be one such thing, but we cannot yet be sure. On the other hand, the kind of flat model used here can support syntactic operations such as testing syntactic equality of commands, and string-based runtime code generation (described for instance in [1, 19] respectively), which it appears the models of [20, 21] cannot.

We mention two other approaches to semantically justifying a logic with nested Hoare triples. In [8], total rather than partial correctness is used, and to reason about recursion the user of the logic must essentially perform an induction argument "on foot" in their proofs ([8] was the first paper to give a theory of

nested triples, there named *evaluation formulae*). In [5] the *step indexing* technique [2, 3] is used, where (unlike here) the interpretation of assertions is also indexed by the length of the execution sequence. Step indexing approaches are very similar in spirit to those based on ultrametric spaces.

Acknowledgements We thank the anonymous referees for their helpful comments. This research has been supported by EPSRC grant (EP/G003173/1) "*From Reasoning Principles for Function Pointers To Logics for Self-Configuring Programs*".

References

1. Appel, A.W.: Intensional equality ;=) for continuations. SIGPLAN Not. 31, 55–57 (February 1996)
2. Appel, A.W., McAllester, D.A.: An indexed model of recursive types for foundational proof-carrying code. ACM Trans. Program. Lang. Syst. 23(5), 657–683 (2001)
3. Benton, N., Hur, C.K.: Step-indexing: The good, the bad and the ugly. In: Modelling, Controlling and Reasoning About State. No. 10351 in Dagstuhl Seminar Proceedings, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany (2010)
4. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: FMCO. pp. 115–137 (2005)
5. Birkedal, L., Reus, B., Schwinghammer, J., Støvring, K., Thamsborg, J., Yang, H.: Step-indexed Kripke models over recursive worlds. In: POPL. pp. 119–132 (2011)
6. Birkedal, L., Reus, B., Schwinghammer, J., Yang, H.: A simple model of separation logic for higher-order store. In: ICALP (2). pp. 348–360 (2008)
7. Charlton, N., Horsfall, B., Reus, B.: Formal reasoning about runtime code update (2011), to appear in Proceedings of HotSWUp (Hot Topics in Software Upgrades) 2011
8. Honda, K., Yoshida, N., Berger, M.: An observationally complete program logic for imperative higher-order functions. In: LICS. pp. 270–279 (2005)
9. Landin, P.J.: The mechanical evaluation of expressions. Computer Journal 6(4), 308–320 (Jan 1964)
10. Møller, A., Schwartzbach, M.I.: The pointer assertion logic engine. In: PLDI '01 (Programming Language Design and Implementation). pp. 221–231. ACM Press, New York (2001)
11. Nipkow, T.: Hoare logics for recursive procedures and unbounded nondeterminism. In: Computer Science Logic (CSL 2002), volume 2471 of LNCS. pp. 103–119. Springer (2002)
12. von Oheimb, D.: Hoare logic for mutual recursion and local variables. In: FSTTCS. pp. 168–180 (1999)
13. Parkinson, M.J.: Local reasoning for Java. Ph.D. thesis, University of Cambridge, Computer Laboratory (Nov 2005)
14. Pitts, A.M.: Relational properties of domains. Inf. Comput. 127(2), 66–90 (1996)
15. Pottier, F.: Hiding local state in direct style: a higher-order anti-frame rule. In: LICS. pp. 331–340. Pittsburgh, Pennsylvania (Jun 2008)

16. Reus, B., Schwinghammer, J.: Separation logic for higher-order store. In: CSL. pp. 575–590 (2006)
17. Reus, B., Streicher, T.: About Hoare logics for higher-order store. In: ICALP. pp. 1337–1348 (2005)
18. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS. pp. 55–74 (2002)
19. Richards, G., Lebresne, S., Burg, B., Vitek, J.: An analysis of the dynamic behavior of JavaScript programs. In: PLDI. pp. 1–12 (2010)
20. Schwinghammer, J., Birkedal, L., Reus, B., Yang, H.: Nested Hoare triples and frame rules for higher-order store. In: CSL. pp. 440–454 (2009)
21. Schwinghammer, J., Yang, H., Birkedal, L., Pottier, F., Reus, B.: A semantic foundation for hidden state. In: FOSSACS. pp. 2–17 (2010)

A Appendix

Theorem 3. *Rule (S) is sound.*

$$\frac{\Gamma \vdash \{P\}\ `C_1'\ \{R\} \qquad \Gamma \vdash \{R\}\ `C_2'\ \{Q\}}{\Gamma \vdash \{P\}\ `C_1; C_2'\ \{Q\}}\ \mathrm{S}$$

Proof. For the sake of clarity, a sound proof rule

$$\frac{S_1 \quad \dots \quad S_k}{S}$$

means that $S_1, \dots, S_k \vDash S$.

Suppose that the premises of (S) hold. We must then show that the conclusion

$$\vDash \Gamma \vdash \{P\}\, `C_1; C_2'\, \{Q\}$$

holds. So let $\rho \in \mathsf{Env}$ and $n \in \mathbb{N}$ be such that either $n = 0$ or ($n > 0$ and $(\rho, n-1) \in [\![\Gamma]\!]^{\mathrm{co}}$). We must show

$$\rho \vDash^n \{P\}\, `C_1; C_2'\, \{Q\} \tag{10}$$

If $n = 0$ we are done, because $C_1; C_2$ cannot reach a terminal configuration in 0 steps. So assume $n > 0$ and $(\rho, n-1) \in [\![\Gamma]\!]^{\mathrm{co}}$. From this and the premises of (S) we deduce that

$$\rho \vDash^n \{P\}\, `C_1'\, \{R\} \tag{11}$$

and

$$\rho \vDash^n \{R\}\, `C_2'\, \{Q\} \tag{12}$$

To prove (10), let

$$(C_1; C_2, s^1) \to \cdots \to (\mathsf{nop}, s^K)$$

be an execution sequence such that $1 < K \leq n+1$ (so the execution consists of at most n steps) and $s^1 \in [\![P]\!]^{\mathrm{as}}_{\rho}$. This execution sequence must have the form

$$(C_1; C_2, s^1) \to \cdots \to (\mathsf{nop}; C_2, s^J) \to (C_2, s^{J+1}) \to \cdots \to (\mathsf{nop}, s^K)$$

where $1 \leq J < K$ and $s^J = s^{J+1}$, and there must exist another execution sequence, of $J-1$ steps, of the form

$$(C_1, s^1) \to \cdots \to (\mathsf{nop}, s^J)$$

By (11) we see that $s^J = s^{J+1} \in [\![R]\!]^{\mathrm{as}}_\rho$. Then applying (12) to the execution sequence

$$(C_2, s^{J+1}) \to \cdots \to (\mathsf{nop}, s^K)$$

we obtain $s^K \in [\![Q]\!]^{\mathrm{as}}_\rho$ as required. □

Theorem 4. *Rule (R) is sound.*

Proof. Suppose that the premise

$$\Gamma \vdash \{P \wedge x = \text{`}C\text{'}\}\ \text{`}C\text{'}\ \{Q\} \tag{13}$$

holds. We must prove the conclusion

$$\Gamma \vdash \{P \wedge x = \text{`}C\text{'}\}\ \text{`}\mathsf{run}\ x\text{'}\ \{Q\}$$

So let $\rho \in \mathsf{Env}$, $n \in \mathbb{N}$ be such that $n = 0$ or ($n > 0$ and $(\rho, n-1) \in [\![\Gamma]\!]^{\mathrm{co}}$). We must prove $\rho \vDash^n \{P \wedge x = \text{`}C\text{'}\}\ \text{`}\mathsf{run}\ x\text{'}\ \{Q\}$. If $n = 0$ then this is trivially true (since $\mathsf{run}\ x$ cannot reach a terminal configuration in 0 steps), so let $n > 0$. The premise (13) gives us

$$(\rho, n-1) \in [\![\Gamma]\!]^{\mathrm{co}} \quad \text{implies} \quad \rho \vDash^n \{P \wedge x = \text{`}C\text{'}\}\ \text{`}C\text{'}\ \{Q\}$$

But we already know $(\rho, n-1) \in [\![\Gamma]\!]^{\mathrm{co}}$ so we deduce

$$\rho \vDash^n \{P \wedge x = \text{`}C\text{'}\}\ \text{`}C\text{'}\ \{Q\} \tag{14}$$

Let $s \in [\![P \wedge x = \text{`}C\text{'}]\!]^{\mathrm{as}}_\rho$ and s' be such that $(\mathsf{run}\ x, s) \xrightarrow{*} (\mathsf{nop}, s')$ in n steps or fewer; we are required to show $s' \in [\![Q]\!]^{\mathrm{as}}_\rho$. Due to the structure of the transition relation $\to$, it must be the case that $(C, s) \xrightarrow{*} (\mathsf{nop}, s')$ in $n-1$ steps or fewer. From this and (14) we have $s' \in [\![Q]\!]^{\mathrm{as}}_\rho$ as required. □

A.1 Proof for S_1

Proof (for S_1 piece). By the (C) rule it will suffice to show $T(\mathrm{q}) \vdash T(\text{`}C_1\text{'})$ i.e.

$$T(\mathrm{q}) \vdash \left\{ f = \mathrm{p} \wedge g = \mathrm{q} \right\} \begin{array}{l} \text{`}\mathsf{if}\ (x \times x) + (y \times y) = (z \times z) \\ \mathsf{then\ nop} \\ \mathsf{else\ run}\ g\text{'} \end{array} \left\{ R(x, y, z) \right\}$$

Using the (I) rule it will be enough to show

$$T(\mathrm{q}) \vdash \left\{ \begin{array}{l} f = \mathrm{p} \\ \wedge\ g = \mathrm{q} \\ \wedge\ ((x \times x) + (y \times y) = (z \times z)) = 1 \end{array} \right\}\ \text{`}\mathsf{nop}\text{'}\ \left\{ R(x, y, z) \right\} \tag{15}$$

and

$$T(\mathrm{q}) \vdash \left\{ \begin{array}{l} f = \mathrm{p} \\ \wedge\ g = \mathrm{q} \\ \wedge\ ((x \times x) + (y \times y) = (z \times z)) \neq 1 \end{array} \right\} \text{`run } g\text{'} \ \{ R(x,y,z) \} \qquad (16)$$

(15) is easily proved using the definition of $R(x,y,z)$ as $x^2 + y^2 = z^2$ and the equivalence $(e_1{=}e_2) = 1 \Leftrightarrow e_1 = e_2$. We deduce (16) by the (W) rule from the following instance of (H):

$$T(\mathrm{q}) \vdash \{f = \mathrm{p} \wedge g = \mathrm{q}\} \text{`run } g\text{'} \{R(x,y,z)\}$$

□

A.2 Two of von Oheimb's proof rules

For the reader's convenience we reproduce here the two relevant rules from [12]:

$$\frac{\begin{array}{l}\textit{Call} \\ \Gamma \cup \{\{P_i\}\,\mathtt{Call}\ i\,\{Q_i\} \mid i \in ps\} \Vdash \{\{P_i\}\,\mathtt{body}\ i\,\{Q_i\} \mid i \in ps\} \qquad p \in ps\end{array}}{\Gamma \vdash \{P_p\}\,\mathtt{Call}\ p\,\{Q_p\}}$$

$$\frac{\begin{array}{l}\textit{asm} \\ t \in \Gamma\end{array}}{\Gamma \vdash t}$$

where $\Gamma \vdash t$ abbreviates $\Gamma \Vdash \{t\}$.

The idea of the *Call* rule is that to verify a family ps of mutually recursive procedures, one first gives a behavioural specification $\{P_i\}\_\{Q_i\}$ to each procedure $i \in ps$. One must then prove that the body of each procedure $i \in ps$ actually meets this specification, but when doing so, one is allowed to assume that calls to procedures in ps appearing in the body behave as specified. When doing such proofs, the *asm* rule allows one to make use of these assumptions.