Hoare logic for higher order store with simple foundations

Nathaniel Charlton

Department of Informatics, University of Sussex, Falmer, Brighton, United Kingdom

Abstract

We revisit the problem of providing a Hoare logic for a simple language 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. We present progress in three areas. Firstly, we present a new semantic model of the programming language, using flat states rather than domains. This model is much simpler and leads to a more powerful logic: unintuitive restrictions on proof rules are eliminated, non-deterministic programs are handled, programs which perform syntactic equality tests on commands can be reasoned about, and some convenient new proof rules are validated. Secondly we 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. Thirdly we extend the programming language with an operator for runtime specialisation of code, which is a simple form of runtime code generation. We provide new proof rules for reasoning about this operator, including a new recursion rule. We demonstrate these rules with an example.

Keywords: higher order store, Hoare logic, modular proof

1. Introduction

A programming language is said to feature *higher order store* 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. Higher order store has been suggested [1, 2, 3, 4, 5, 6] as a conceptual tool to help us understand various programming language features. These features include:

- *general reference cells* which can store functions, as found for example in ML

Email address: billiejoecharlton@gmail.com (Nathaniel Charlton)

- *dynamic loading and unloading of code*: even languages such as C allow programs to dynamically load and unload shared library code
- *dynamic software updating*, in which a software program is upgraded to a new version without shutting it down [7, 8]
- *runtime code generation*, in which routines optimised for specific inputs are generated at runtime [9]
- *method update* in object-based languages such as Abadi and Cardelli's imperative object calculus [10]

In all these situations, the set of code or commands making up the program changes at runtime.

Conventional Hoare logic [11, 12], however, does not account for higher order store, because it makes an implicit assumption that the program is fixed and immutable. To begin to address this shortcoming, Reus and Streicher [13] studied the problem of providing a Hoare logic for "arguably the simplest language that uses higher-order store". By working with a simple language the authors could focus solely on the issues created by higher order store. This language contained programs such as the following:

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

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 $\{\mathit{true}\} - \{\mathit{false}\}$. The fact that new recursions can be set up on-the-fly in this way was observed by Landin [14] 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 $\{\mathit{true}\} - \{n = 99\}$.

By developing a domain-theoretic model of the programming language, Reus and Streicher were able to give a suitable assertion language and associated Hoare logic rules. In particular, reasoning about recursion through the store is supported.

In this paper we revisit the original problem studied by Reus and Streicher [13], presenting new progress in three areas.

1. **We give a more powerful logic and a simpler semantic model.**

   The domain-theoretic model used by Reus and Streicher is rather complicated, and this has various undesireable consequences:

   - Proving soundness of the logic requires an intricate domain-theoretic argument; specifically, results by Pitts [15] concerning the existence of invariant relations on recursively defined domains are needed.

- The domain theory leaks out and pollutes the logic, so that some of the proof rules are restricted in that they can only be used with assertions that are "downwards closed". These restrictions are unintuitive because they come from the semantics and do not correspond to anything at the programming language level.
- Non-deterministic program statements cannot be added to the language.
- Reasoning about programs which perform syntactic equality testing on commands (which may be used as an optimisation) is not supported.

By using a simpler model based on flat states and operational semantics we address all these issues and additionally validate some convenient new proof rules.

2. **We demonstrate that modular proofs are in fact supported.**

   It is stated in [13] that the given logic does not support modular proofs. Contrary to this, we demonstrate with an example that one can indeed carry out proofs which are modular (in a specific sense which we shall make clear).

3. **We add features for code specialisation at runtime.**

   We extend the programming language with a *specialising quote* operator. This enables specialisation of code at runtime, which is a simple form of runtime code generation. We provide new proof rules for reasoning about runtime code specialisation, including a new recursion rule (μSETS), and demonstrate these rules with an example. The (μSETS) rule is needed because in the presence of runtime code specialisation there can be a countably infinite range of commands which could be generated during program execution.

The rest of this paper is structured as follows. Section 2 gives the syntax of the programs, assertions and specifications we work with and sets out our new semantic model for these. Section 3 presents proof rules extending those of [13], explaining the improvements we make, and shows their soundness. Section 4 explains and demonstrates with an example that our a proof system supports modular proofs. In Section 5 we introduce and justify further new proof rules for reasoning about runtime code specialisation, which we apply to an example program in Section 6. Section 7 puts our results into a broader context, discussing related and future work. Section 8 concludes.

2. Programs, assertions and specifications

In this section we begin by giving the syntax of a programming language for higher order store programs, and accompanying assertion and specification

$$
\begin{array}{llll}
\text{expressions} & e & ::= & 0 \mid 1 \mid \ldots \mid e_1 + e_2 \mid e_1 = e_2 \mid \ldots \mid x \mid \mathrm{x} \mid \text{`}C\text{'} \mid \text{`}C\text{'}_{\vec{v}=\vec{e}} \\
\text{commands} & C & ::= & \mathsf{nop} \mid x := e \mid C_1; C_2 \mid \mathsf{if}\ e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid \mathsf{run}\ e \mid \mathsf{choose}\ C_1\ C_2 \\
\text{assertions} & P, Q & ::= & P_1 \wedge P_2 \mid \neg P \mid \forall v.P \mid e_1 \leq e_2 \mid \mathrm{P}(e^+)
\end{array}
$$

$$
\begin{array}{llll}
\text{middle part of triple} & \mathcal{C} & ::= & e \mid \mathrm{P}(\cdot, e_1, \ldots, e_k) \\
 & & & (e, e_1, \ldots, e_k \text{ must not contain free program variables}) \\
\text{specifications} & S & ::= & \forall \vec{\mathrm{v}}.\, \{P\}\, \mathcal{C}\, \{Q\} \\
 & & & \mid P \quad \text{(lone } P \text{ must not contain free program variables)} \\
\text{specifications in context} & \Pi & ::= & S_1, \ldots, S_k \vdash S
\end{array}
$$

Figure 1: Syntax of expressions, commands, assertions and specifications.

languages. We briefly describe Reus and Streicher's semantic model for the languages, before introducing our new, simpler model. The reader will see that our model requires no complicated theory.

2.1. Syntax of programs, assertions and specifications

Fig. 1 gives the syntax of expressions e, commands C and assertions P. Variables 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. Predicate variables, in PVar, are typically named P, Q and so on.

The quote expression 'C' turns the command C into a value so it can be stored in a variable, and run later. The *specialising quote* 'C'$_{\vec{v}=\vec{e}}$ is similar, but when evaluated replaces the variables $\vec{v}$ in C with the current values of the expressions $\vec{e}$. This feature allows programs to perform code specialisation at runtime. The free variables of 'C'$_{\vec{v}=\vec{e}}$ are exactly those appearing in $\vec{e}$. (In fact 'C' is a special case of 'C'$_{\vec{v}=\vec{e}}$, where $\vec{v}$ is the empty list of variables; thus 'C' has no free variables.) The difference between the ordinary and specialising quotes is shown by the following two programs.

$$n := 10;\ x := \text{`}m := n\text{'};\ n := 0;\ \mathsf{run}\ x \tag{3}$$

$$n := 10;\ x := \text{`}m := a\text{'}_{a=n};\ n := 0;\ \mathsf{run}\ x \tag{4}$$

Program (3) terminates with $m = 0$ whereas program (4) terminates with $m = 10$.

We write $mod(C)$ for the set of program variables that are syntactically assigned to in the command C. (This need not coincide with the set of variables which C actually changes. The assigment $x := x$ changes no variables even though $mod(x := x) = \{x\}$; conversely $\mathsf{run}\ x$ may change any variable, depending on the code stored in x, yet $mod(\mathsf{run}\ x) = \emptyset$.) The specialising quote 'C'$_{\vec{v}=\vec{e}}$ only makes sense when $mod(C) \cap \vec{v} = \emptyset$. For example, '$x := y$'$_{x=3}$ does not

make sense because replacing x by 3 in $x := y$ leads to $3 := y$, which is not a syntactically well-formed command.

Using the available assertion language one can express the missing connectives *true*, $\vee$, $\exists$, $=$ and so on in the usual way.

Our setup features three extensions compared to that of [13]: the specialising quote $`C'_{\vec{v}=\vec{e}}$, predicates P in assertions and the non-deterministic choice statement choose. Aside from these extensions, the only difference is that because we will use a flat store model and encode commands as integers, we will not need a type check $\tau?e$ and the typed comparison operators $\leq_\tau$ become simply $\leq$.

Next we introduce *specifications* S and *specifications in context* Π, whose syntax is also shown in Fig. 1. A specification is either a Hoare triple describing the behaviour of commands, or an assertion which mentions no program variables. The latter form will be used to state properties of predicates P. The specification form $\{P\}\,\mathrm{P}(\cdot, e_1, \ldots, e_k)\,\{Q\}$, where a predicate P appears in the middle part of the triple, is used to say that all commands C satisfying $\mathrm{P}(C, e_1, \ldots, e_k)$ satisfy the triple $\{P\} \cdot \{Q\}$.

The specification in context $S_1, \ldots, S_k \vdash S$ says informally that in any context where specifications $S_1, \ldots, S_k$ hold, specification S must hold as well. Comma-separated sequences $S_1, \ldots, S_k$ of specifications will often be called *contexts* and named Γ.

2.2. The existing domain-theoretic semantics

Now we turn to the issue of giving semantics to the programming and assertion languages. Reus and Streicher took 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, [13] used 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.

2.3. Our new flat semantics

Instead of representing stored commands as functions mapping stores to stores, in this paper we represent them simply as syntax; this is the key idea of our model. Put another way, we use an intensional rather than an extensional representation. In fact for convenience all values, including commands, are encoded as integers. To this end we fix a bijection $\mathscr{G}$ mapping (syntactic) commands to integers; we will use this to encode commands. 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 call our stores and environments *flat* because, unlike in the domain-theoretic model [13], no complicated order structure on Store is required.

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

$$\frac{(C_1, s) \to (C_1', s')}{(C_1; C_2, s) \to (C_1'; C_2, s')} \qquad \frac{}{(\mathsf{nop}; C, s) \to (C, s)}$$

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

$$\frac{C = \mathscr{G}^{-1}([\![e]\!]^{\mathrm{ex}}_s)}{(\mathsf{run}\ e, s) \to (C, s)}$$

$$\frac{}{(\mathsf{choose}\ C_1\ C_2, s) \to (C_1, s)} \qquad \frac{}{(\mathsf{choose}\ C_1\ C_2, s) \to (C_2, s)}$$

Figure 2: Small-step operational semantics of programs.

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 $\text{`}C\text{'}_{\vec{v}=\vec{e}}$, where we use the encoding $\mathscr{G}$, defining

$$[\![\text{`}C\text{'}_{\vec{v}=\vec{e}}]\!]^{\mathrm{ex}}_{s,\rho} \quad \triangleq \quad \begin{cases} \mathscr{G}(C[\vec{v} \backslash [\![\vec{e}]\!]^{\mathrm{ex}}_{s,\rho}]) & \text{if } \mathit{mod}(C) \cap \vec{v} = \emptyset \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

(We cannot simply define $[\![\text{`}C\text{'}_{\vec{v}=\vec{e}}]\!]^{\mathrm{ex}}_{s,\rho} \triangleq \mathscr{G}(C[\vec{v} \backslash [\![\vec{e}]\!]^{\mathrm{ex}}_{s,\rho}])$ because in cases such as $\text{`}x := y\text{'}_{x=3}$ the substitution does not produce a well-formed command.) As a special case we have $[\![\text{`}C\text{'}]\!]^{\mathrm{ex}} \triangleq \mathscr{G}(C)$.

Semantics of programs. Fig. 2 gives a small-step operational semantics. 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 $\mathsf{run}\ e$ command: we just evaluate expression e to an integer value, turn it back into a (syntactic) command with $\mathscr{G}^{-1}$, and run it.

Semantics of assertions. Let $\mathsf{PEnv} \triangleq \mathsf{PVar} \to \mathbb{P}(\mathbb{Z}^+)$ be the set of environments for predicate variables. The semantics $[\![P]\!]^{\mathrm{as}}_{\rho,\chi} \subseteq \mathsf{Store}$ of assertion P in environment ρ and predicate environment χ is largely standard, and thus omitted. For predicate uses $\mathrm{P}(e_1, \ldots, e_k)$ we have

$$s \in [\![\mathrm{P}(e_1, \ldots, e_k)]\!]^{\mathrm{as}}_{\rho,\chi} \quad \triangleq \quad ([\![e_1]\!]^{\mathrm{ex}}_{s,\rho}, \ldots, [\![e_k]\!]^{\mathrm{ex}}_{s,\rho}) \in \chi(\mathrm{P})$$

For a formula P containing no predicate variables, we simply write $[\![P]\!]^{\mathrm{as}}_{\rho}$. Entailment $P \Rightarrow Q$ means that for all ρ and all χ, $[\![P]\!]^{\mathrm{as}}_{\rho,\chi} \subseteq [\![Q]\!]^{\mathrm{as}}_{\rho,\chi}$.

$$[\![\forall \vec{\mathrm{v}}.\, \{P\}\, \mathcal{C}\, \{Q\}]\!]^{\mathrm{sp}} \triangleq \left\{ (\rho, \chi, n) \;\middle|\; \begin{array}{l} \text{either } n = -1 \text{ or:} \\ \text{for all } \rho' \text{ agreeing with } \rho \\ \text{ except possibly at } \vec{\mathrm{v}}, \\ \rho', \chi \vDash^n \{P\}\, \mathcal{C}\, \{Q\} \end{array} \right\}$$

$$[\![P]\!]^{\mathrm{sp}} \triangleq \{(\rho, \chi) \mid [\![P]\!]^{\mathrm{as}}_{\rho,\chi} = \mathsf{Store}\} \times \mathbb{N}_{-1}$$

$$[\![S_1, \ldots, S_k \vdash S]\!]^{\mathrm{sic}} \triangleq \left\{ (\rho, \chi) \;\middle|\; \forall n \in \mathbb{N},\ \text{if } (\rho, \chi, n-1) \in \textstyle\bigcap_{i=1}^{k} [\![S_i]\!]^{\mathrm{sp}} \text{ then } (\rho, \chi, n) \in [\![S]\!]^{\mathrm{sp}} \right\}$$

Figure 3: Semantics of specifications and specifications in context.

Semantics of Hoare triples. The middle part $\mathcal{C}$ of a triple is interpreted to a set of commands as follows:

$$[\![e]\!]^{\mathrm{mid}}_{\rho,\chi} \quad \triangleq \quad \{\mathscr{G}^{-1}([\![e]\!]^{\mathrm{ex}}_{\rho})\}$$

$$[\![\mathrm{P}(\cdot, e_1, \ldots, e_k)]\!]^{\mathrm{mid}}_{\rho,\chi} \quad \triangleq \quad \{\mathscr{G}^{-1}(N) \mid (N, [\![e_1]\!]^{\mathrm{ex}}_{\rho}, \ldots, [\![e_k]\!]^{\mathrm{ex}}_{\rho})) \in \chi(\mathrm{P})\}$$

We use a partial correctness interpretation of Hoare triples as follows (where the index n is the number of execution steps for which the Hoare triple holds).

Definition 2.1. Meaning of Hoare triples. We write $\rho, \chi \vDash^n \{P\}\, \mathcal{C}\, \{Q\}$ (where $n \in \mathbb{N}$) to mean that for all $C \in [\![\mathcal{C}]\!]^{\mathrm{mid}}_{\rho,\chi}$ and all stores $s \in [\![P]\!]^{\mathrm{as}}_{\rho,\chi}$, if $(C, s) \xrightarrow{*} (\mathsf{nop}, s')$ in n steps or fewer then $s' \in [\![Q]\!]^{\mathrm{as}}_{\rho,\chi}$. □

Semantics of specifications. From now on we shall use $\mathbb{N}_{-1}$ as a shorthand for $\mathbb{N} \cup \{-1\}$. The semantics $[\![S]\!]^{\mathrm{sp}} \subseteq \mathsf{Env} \times \mathsf{PEnv} \times \mathbb{N}_{-1}$ of specification S, and the semantics $[\![\Pi]\!]^{\mathrm{sic}} \subseteq \mathsf{Env} \times \mathsf{PEnv}$ of specification in context Π, 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 $\{A\} \cdot \{B\}$ for executions of up to length $n-1$, the command e' satisfies triple $\{P\} \cdot \{Q\}$ 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, which we will employ in the next section. We lift the semantics $[\![-]\!]^{\mathrm{sp}}$ from specifications to contexts $\Gamma = S_1, \ldots, S_k$ using intersection.

We write $\Pi_1, \ldots, \Pi_k \vDash \Pi$ to mean that (the conjunction of) specifications in context $\Pi_1, \ldots, \Pi_k$ entails the specification in context Π, that is, if $[\![\Pi_1]\!]^{\mathrm{sic}} = \ldots = [\![\Pi_k]\!]^{\mathrm{sic}} = \mathsf{Env} \times \mathsf{PEnv}$ then $[\![\Pi]\!]^{\mathrm{sic}} = \mathsf{Env} \times \mathsf{PEnv}$. (When $k = 0$ we write just $\vDash S$ meaning $[\![\Pi]\!]^{\mathrm{sic}} = \mathsf{Env} \times \mathsf{PEnv}$.)

Remark 2.2. If $(\rho, \chi, n) \in [\![S]\!]^{\mathrm{sp}}$ and $-1 \leq m < n$ then $(\rho, \chi, m) \in [\![S]\!]^{\mathrm{sp}}$. □

3. Proof rules and their soundness

In this section we present a set of Hoare logic rules for reasoning about higher order store programs. These rules include all those of [13], with some

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

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

$$\text{W}\ \frac{\Gamma \vdash \{P'\}\,\mathcal{C}\,\{Q'\}}{\Gamma \vdash \{P\}\,\mathcal{C}\,\{Q\}}\ P \Rightarrow P', Q' \Rightarrow Q \qquad \epsilon\ \frac{}{\{P\}\,\text{`}\mathsf{nop}\text{'}\,\{P\}} \qquad \textsc{AddContext}\ \frac{\Gamma \vdash \{P\}\,\mathcal{C}\,\{Q\}}{\Gamma, S \vdash \{P\}\,\mathcal{C}\,\{Q\}}$$

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

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

useful generalisations and additions which we explain. We then provide simple proofs of the rules' soundness.

3.1. Meet the proof rules

We have split our proof rules into two groups: standard Hoare logic rules, and rules for reasoning about the run statement which runs stored commands. The standard rules for (A)ssignment, (S)equential composition, (W)eakening (i.e. consequence) etc. are given in Fig. 4. (If these rule names are a little cryptic, at least they are the same names as used in existing work [13, 16].) We are explicit about the presence of a context Γ whereas in [13] this is left implicit.

Dealing with non-determinism. The (Choose) rule for non-deterministic choice is our first new proof rule. The interest here is that, while our semantic model easily handles non-determinism, the domain-theoretic model from [13] cannot. The explanation given [3] is that in the presence of non-determinism, "programs no longer denote ω-continuous functions". Non-determinism is important because it arises naturally when one wishes to consider programs that use pointers and a dynamically allocated heap. In such programs the memory allocation operation in generally treated as non-deterministically choosing any unused heap space to allocate.

The second group of rules, those concerning the running of stored commands using run, is shown in Fig. 5. 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. Rule (μ) is used for reasoning about mutual recursion (through the

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

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

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

$$\forall\textsc{InstContext}\quad \frac{\Gamma, \forall\vec{\mathrm{v}}.\,\{P\}\,\mathcal{C}\,\{Q\}, (\{P\}\,\mathcal{C}\,\{Q\})[\vec{\mathrm{v}}\backslash\vec{e}] \;\vdash\; S}{\Gamma, \forall\vec{\mathrm{v}}.\,\{P\}\,\mathcal{C}\,\{Q\} \;\vdash\; S}\ e \text{ contains no prog. vars}$$

$$\mu\textsc{Direct}\quad \frac{\bigwedge_{1\le i\le N} \Gamma, \{P_1\}\,\text{`}C_1\text{'}\,\{Q_1\}, \ldots, \{P_N\}\,\text{`}C_N\text{'}\,\{Q_N\} \;\vdash\; \{P_i\}\,\text{`}C_i\text{'}\,\{Q_i\}}{\bigwedge_{1\le i\le N} \Gamma \vdash \{P_i\}\,\text{`}C_i\text{'}\,\{Q_i\}}$$

$$\textsc{HDirect}\quad \frac{}{\{P \wedge x = \text{`}C\text{'}\}\,\text{`}C\text{'}\,\{Q\} \;\vdash\; \{P \wedge x = \text{`}C\text{'}\}\,\text{`}\mathsf{run}\ x\text{'}\,\{Q\}}$$

Figure 5: Hoare logic rules associated with the run statement which runs stored commands.

store): intuitively the premise says that *if* all commands C_j involved satisfy their specifications $\{P_j\} \cdot \{Q_j\}$ 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.

The (R), (H) and (μ) rules come from [13], but due to our flat store model, we can make these rules simpler in two respects. Firstly, unintuitive side conditions about downwards closure of assertions have been eliminated. Secondly we simply use equality on commands, rather than a partial order $\leq_{\text{com}}$.

The ($\forall$InstContext) rule is for instantiating universal quantifiers over Hoare triples in the context. Such quantification is left implicit in [13]; we prefer to write the quantifiers and make explicit the rule for instantiating them.

The (μDirect) and (HDirect) rules, new in this paper, are "direct" versions of (μ) and (H). To explain the differences, as well as demonstrate all four

rules, we consider the program

$$x := \text{`}\mathsf{run}\ x\text{'};\ \mathsf{run}\ x$$

We noted in the introduction that this is the simplest possible program that uses recursion through the store. We shall prove that this program doesn't terminate, by showing

$$\{\mathit{true}\}\, x := \text{`}\mathsf{run}\ x\text{'};\ \mathsf{run}\ x\, \{\mathit{false}\}$$

This reduces (using (A) and (S)) to showing

$$\{x = \text{`}\mathsf{run}\ x\text{'}\}\, \text{`}\mathsf{run}\ x\text{'}\, \{\mathit{false}\} \tag{6}$$

One way to show this is it use the following instance of (μ):

$$\frac{\{x = \mathrm{p}\}\, \mathrm{p}\, \{\mathit{false}\} \ \vdash\ \{x = \mathrm{p}\}\, \text{`}\mathsf{run}\ x\text{'}\, \{\mathit{false}\}}{\{(x = \mathrm{p})[\mathrm{p}\backslash\text{`}\mathsf{run}\ x\text{'}]\}\, \text{`}\mathsf{run}\ x\text{'}\, \{\mathit{false}[\mathrm{p}\backslash\text{`}\mathsf{run}\ x\text{'}]\}}$$

The premise of this is trivially proved, being simply an instance of (H).

Note however that when using the (μ) rule we were forced into an "indirection", introducing an auxiliary variable p to name the code stored in x. If we instead use the (μDIRECT) and (HDIRECT) rules, this can be avoided: we can instead obtain (6) from the following instance of (μDIRECT).

$$\frac{\{x = \text{`}\mathsf{run}\ x\text{'}\}\, \text{`}\mathsf{run}\ x\text{'}\, \{\mathit{false}\} \ \vdash\ \{x = \text{`}\mathsf{run}\ x\text{'}\}\, \text{`}\mathsf{run}\ x\text{'}\, \{\mathit{false}\}}{\{x = \text{`}\mathsf{run}\ x\text{'}\}\, \text{`}\mathsf{run}\ x\text{'}\, \{\mathit{false}\}} \tag{7}$$

The premise of this is trivially proved, being an instance of (HDIRECT). In this case the benefit of using (μDIRECT) instead of (μ) is minimal, but in more complicated proofs it can be substantial.

The (μDIRECT) rule looks unusual; for instance, the premise of (7) has the form $A \vdash A$ which looks as though it should hold trivially. Indeed, in the domain-theoretic model of [13] this is the case and (μDIRECT) cannot be proved sound. But recall that in our model $\vdash$ is not a conventional implication, as the left and right sides talk about executions of different lengths. Thus the availability of (μDIRECT) and (HDIRECT) is another bonus of our simpler semantics.

Syntactic equality testing on commands. When programs can compare commands for syntactic equality (or, in functional languages, α-equivalence), new kinds of low-level programming and optimisation become available. Such syntactic tests in minimal programming languages have been described as "a proxy for the ability of machine code programs to compare code pointers for equality or to read executable instructions, Java programs to perform dynamic type tests or use reflection, or programs in popular dynamic languages to do all sorts of things" [17]. It is thus interesting to see that our flat model supports some reasoning about such syntactic, or intensional, operations.

In the domain-theoretic model of [13], stored commands are represented as semantic functions and information about their syntactic structure is thrown away; thus syntactic equality cannot be expressed. But in our flat model, commands are represented by their syntax (encoded as integers) so the standard equality operator naturally expresses syntactic equality.

As an example, let C_{iter} be the following higher order command.

$$\begin{array}{l}\mathsf{if}\ f = \text{`}\mathsf{nop}\text{'}\ \mathsf{then}\ \mathsf{nop}\ \mathsf{else}\\ \quad(\\ \quad\ y := \text{`}\mathsf{if}\ n = 0\ \mathsf{then}\ \mathsf{nop}\ \mathsf{else}\ (n := n{-}1\ ;\mathsf{run}\ f\ ;\ \mathsf{run}\ y)\text{'}\ ;\\ \quad\ \mathsf{run}\ y\\ \quad)\end{array}$$

C_{iter} executes the command stored in f repeatedly, n times. But as an optimisation, if the command in f is 'nop', it need not be run at all. With our proof rules we can prove various behaviours for C_{iter}, such as

$$\{f = \text{`}\mathsf{nop}\text{'} \wedge P\}\, C_{iter}\, \{P\}$$

for the $f = $ 'nop' case. For the $f \neq$ 'nop' case we can prove that if the command in f preserves invariant P and doesn't interfere with variables f, n, y, then C_{iter} also preserves invariant P:

$$\begin{array}{l}\forall \mathrm{f}_0, \mathrm{n}_0, \mathrm{y}_0.\ \left\{\begin{array}{c} P \\ \wedge\ f = \mathrm{f}_0 \\ \wedge\ n = \mathrm{n}_0 \\ \wedge\ y = \mathrm{y}_0 \end{array}\right\}\ \ \text{`}C\text{'}\ \ \left\{\begin{array}{c} P \\ \wedge\ f = \mathrm{f}_0 \\ \wedge\ n = \mathrm{n}_0 \\ \wedge\ y = \mathrm{y}_0 \end{array}\right\}\\ \models\\ \quad \{f = \text{`}C\text{'} \wedge f \neq \text{`}\mathsf{nop}\text{'} \wedge P\}\, C_{iter}\, \{P\}\end{array}$$

3.2. Soundness of the proof rules

Having stated the proof rules, we must of course show that they are sound. The standard rules (Fig. 4) are straightforward to prove so we omit the proofs. In any case, most of these rules are special cases of rules we introduce later (Section 5) to deal with runtime specialisation of code, so their soundness will be established as a by-product of later proofs.

This leaves the rules for running stored commands. We prove soundness of (H), (μ) and (μDIRECT); the proofs for (R) and (HDIRECT) are omitted as they are very similar to the proof for (H), but simpler. We emphasise that our proof of (μ) uses only the simple idea of induction on execution length, whereas the existing proof [13] depends on results by Pitts [15] concerning the existence of invariant relations on recursively defined domains.

Theorem 3.1. Rule (H) is sound.

Proof. Let $\rho \in \mathsf{Env}$, $\chi \in \mathsf{PEnv}$, $n \in \mathbb{N}$ be such that $(\rho, \chi, n{-}1) \in [\![\{P \wedge x = \mathrm{p}\}\, \mathrm{p}\, \{Q\}]\!]^{\mathrm{sp}}$). We must prove $\rho, \chi \models^n \{P \wedge x = \mathrm{p}\}\, \text{`}\mathsf{run}\ x\text{'}\, \{Q\}$. If $n = 0$ then this is trivially

true, so let $n > 0$. Then from $(\rho, \chi, n{-}1) \in [\![\{P \wedge x = \mathrm{p}\}\,\mathrm{p}\,\{Q\}]\!]^{\mathrm{sp}}$ we get (A.) $\rho, \chi \vDash^{n-1} \{P \wedge x = \mathrm{p}\}\,\mathrm{p}\,\{Q\}$.

Let $s \in [\![P \wedge x{=}\mathrm{p}]\!]^{\mathrm{as}}_{\rho,\chi}$ and s' be such that $(\mathsf{run}\ x, s) \xrightarrow{*} (\mathsf{nop}, s')$ in n steps or fewer; we are required to show $s' \in [\![Q]\!]^{\mathrm{as}}_{\rho,\chi}$. 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 $s' \in [\![Q]\!]^{\mathrm{as}}_{\rho,\chi}$ as required. □

Theorem 3.2. Rule (μ) is sound.

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

$$(\rho, \chi, n{-}1) \in [\![\Gamma]\!]^{\mathrm{sp}} \quad \text{implies} \quad \rho, \chi \vDash^{n} \{P_i[\vec{\mathrm{p}}\backslash\vec{C}]\}\text{`}C_i\text{'}\{Q_i[\vec{\mathrm{p}}\backslash\vec{C}]\}$$

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

Base case ($n = 0$): Let $\rho \in \mathsf{Env}$, $\chi \in \mathsf{PEnv}$ and let $i \in \{1, \ldots, N\}$ be such that $(\rho, \chi, -1) \in [\![\Gamma]\!]^{\mathrm{sp}}$. Let $\hat{\rho}$ be equal to ρ except at $\mathrm{p}_1, \ldots, \mathrm{p}_N$, which are mapped respectively to $[\![\text{`}C_1\text{'}]\!]^{\mathrm{ex}}, \ldots, [\![\text{`}C_N\text{'}]\!]^{\mathrm{ex}}$. Because $(\rho, \chi, -1) \in [\![\Gamma]\!]^{\mathrm{sp}}$ and $\mathrm{p}_1, \ldots, \mathrm{p}_N$ are not free in Γ, we have $(\hat{\rho}, \chi, -1) \in [\![\Gamma]\!]^{\mathrm{sp}}$. It follows from the definition of $[\![-]\!]^{\mathrm{sp}}$ that

$$(\rho, \chi, -1) \in [\![\{P_1\}\,\mathrm{p}_1\,\{Q_1\}, \ldots, \{P_N\}\,\mathrm{p}_N\,\{Q_N\}]\!]^{\mathrm{sp}}$$

Hence from the premise of (μ) we have $\hat{\rho}, \chi \vDash^{0} \{P_i\}\text{`}C_i\text{'}\{Q_i\}$. Using familar properties of substitution, this implies the thing we needed to prove, which is: $\rho, \chi \vDash^{0} \{P_i[\vec{\mathrm{p}}\backslash\vec{C}]\}\text{`}C_i\text{'}\{Q_i[\vec{\mathrm{p}}\backslash\vec{C}]\}$.

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

$$\hat{\rho}, \chi \vDash^{n} \{P_i\}\text{`}C_i\text{'}\{Q_i\} \tag{8}$$

using familiar properties of substitution. Note that $(\hat{\rho}, \chi, n{-}1) \in [\![\Gamma]\!]^{\mathrm{sp}}$ because $(\rho, \chi, n{-}1) \in [\![\Gamma]\!]^{\mathrm{sp}}$ and $\mathrm{p}_1, \ldots, \mathrm{p}_N$ are not free in Γ. We know that $-1 \leq n - 2$ so it follows by Remark 2.2 that $(\rho, \chi, n{-}2) \in [\![\Gamma]\!]^{\mathrm{sp}}$. It now follows from this and the induction hypothesis $\Phi(n-1)$, instantiating ρ with $\hat{\rho}$, that

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

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

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

From this it follows that:

$$(\hat{\rho}, \chi, n{-}1) \in [\![\{P_1\}\,\mathrm{p}_1\,\{Q_1\}, \ldots, \{P_N\}\,\mathrm{p}_N\,\{Q_N\}]\!]^{\mathrm{sp}} \tag{9}$$

From the premise of (μ), unpacking the definitions and instantiating n with n, ρ with $\hat{\rho}$ and χ with χ, we find that (8) follows from (9) and $(\hat{\rho}, n{-}1) \in [\![\Gamma]\!]^{\mathrm{sp}}$. □

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

where

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

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

Figure 6: A program for finding Pythagorean triples, using recursion through the store.

Theorem 3.3. Rule (μDIRECT) is sound.

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

$$(\rho, \chi, n{-}1) \in [\![\Gamma]\!]^{\mathrm{sp}} \quad \text{implies} \quad \rho, \chi \vDash^n \{P_i\}\text{`}C_i\text{'}\{Q_i\}$$

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

Base case ($n = 0$): Let $\rho \in \mathsf{Env}$, $\chi \in \mathsf{PEnv}$ and let $i \in \{1, \ldots, N\}$ be such that $(\rho, \chi, -1) \in [\![\Gamma]\!]^{\mathrm{sp}}$. It follows from the definition of $[\![-]\!]^{\mathrm{sp}}$ that

$$(\rho, \chi, -1) \in [\![\{P_1\}\,\text{`}C_1\text{'}\,\{Q_1\}, \ldots, \{P_N\}\,\text{`}C_N\text{'}\,\{Q_N\}]\!]^{\mathrm{sp}}$$

Then from the premise of (μDIRECT), unpacking the definitions and instantiating n with 0, we immediately obtain $\rho, \chi \vDash^0 \{P_i\}\,\text{`}C_i\text{'}\,\{Q_i\}$ as required.

Inductive case ($n > 0$): Let $\Phi(n-1)$ hold. Let $\rho \in \mathsf{Env}$ and $\chi \in \mathsf{PEnv}$ be such that $(\rho, \chi, n{-}1) \in [\![\Gamma]\!]^{\mathrm{sp}}$ and let $i \in \{1, \ldots, N\}$. We must prove $\rho, \chi \vDash^n \{P_i\}\text{`}C_i\text{'}\{Q_i\}$. We know that $-1 \leq n-2$ and $(\rho, \chi, n{-}1) \in [\![\Gamma]\!]^{\mathrm{sp}}$ so it follows by Remark 2.2 that $(\rho, \chi, n{-}2) \in [\![\Gamma]\!]^{\mathrm{sp}}$. It now follows from the induction hypothesis $\Phi(n-1)$ that

$$\text{for all } j \in \{1, \ldots, N\}, \ \ \rho, \chi \vDash^{n-1} \{P_j\}\text{`}C_j\text{'}\{Q_j\}$$

Hence we have

$$(\rho, \chi, n{-}1) \in [\![\Gamma, \{P_1\}\,\text{`}C_1\text{'}\,\{Q_1\}, \ldots, \{P_N\}\,\text{`}C_N\text{'}\,\{Q_N\}]\!]^{\mathrm{sp}}$$

Then by the premise of (μDIRECT) it follows that $(\rho, \chi, n) \in [\![\{P_i\}\,\text{`}C_i\text{'}\,\{Q_i\}]\!]^{\mathrm{sp}}$ whence $\rho, \chi \vDash^n \{P_i\}\text{`}C_i\text{'}\{Q_i\}$ as required. □

4. An example of a modular proof

We now turn to the issue of modular proofs. In [13] 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 [6]:

> 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.
>
> (10)

(From here on the word "modular" is meant in the sense described in this quote.) Nevertheless, we now demonstrate using a simple program that modular proofs are indeed possible.

Consider the example program in Fig. 6. 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. Let C_0 be the main program.

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

$$\{true\}\, C_0\, \{R(x, y, z)\} \tag{11}$$

But we would also like our proof to be modular, as described in (10), 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

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

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

For C_1: Prove $\Pi_1 \triangleq$ $\quad \vDash T(\mathrm{p}), T(\mathrm{q}) \vdash T(\text{`}C_1\text{'})$

For C_2: Prove $\Pi_2 \triangleq$ $\quad \vDash T(\mathrm{p}), T(\mathrm{q}) \vdash T(\text{`}C_2\text{'})$

For C_0: Prove $\Pi_0 \triangleq$ $\quad \Pi_1, \Pi_2 \vDash \{true\}\, C_0\, \{R(x, y, z)\}$

Together, these three pieces trivially imply (11). We emphasise that in Π_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 (10). Similarly in Π_2 the concrete code for C_1 does not appear, only a specification of its behaviour on the left of $\vdash$.

Proofs of Π_0 and Π_2 now follow (the proof of Π_1 is deferred to Appendix B); these proofs demonstrate the use of the (R), (H), and (μ) rules. Note that only the proof for Π_1 depends on the definition of predicate R.

Proof for Π_0 *piece.* In full, the proof obligation Π_0 is

$$\Pi_1, \Pi_2 \vDash \{true\}\, f := \text{`}C_1\text{'}\ ;\ g := \text{`}C_2\text{'}\ ;\ \mathsf{run}\ f\, \{R(x,y,z)\}$$

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

$$\Pi_1, \Pi_2 \vDash \{f = \text{`}C_1\text{'} \wedge g = \text{`}C_2\text{'}\}\, \text{`}\mathsf{run}\ f\text{'}\, \{R(x,y,z)\}$$

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

$$\Pi_1, \Pi_2 \vDash \{f = \text{`}C_1\text{'} \wedge g = \text{`}C_2\text{'}\}\, \text{`}C_1\text{'}\, \{R(x,y,z)\} \tag{12}$$

and

$$\begin{array}{ll} & \{f = \text{`}C_1\text{'} \wedge g = \text{`}C_2\text{'}\}\, \text{`}C_1\text{'}\, \{R(x,y,z)\} \\ \vDash & \{f = \text{`}C_1\text{'} \wedge g = \text{`}C_2\text{'}\}\, \text{`}\mathsf{run}\ f\text{'}\, \{R(x,y,z)\} \end{array} \tag{13}$$

(13) is easily seen to be an instance of rule (R). To deduce (12) we start with the following instance of (μ)

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

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 (12) written in a different form. □

Proof for Π_2 *piece.* By the (AddContext) rule, Π_2 will follow from $T(\mathrm{p}) \vdash T(\text{`}C_2\text{'})$. By the (S) rule this will follow from

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

and

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

The former is trivial since f and g are not updated; 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 Π_1 and Π_0; showing $\vDash T(\mathrm{p}), T(\mathrm{q}) \vdash T(\text{`}\hat{C}_2\text{'})$ is the only new work[12]. This proof is modular in the same way that proofs about programs with (fixed) recursive procedures can be made modular. Suppose, for instance, one uses the rules of [18] to show correctness of a program with mutually recursive procedures. If one then changes the body of one procedure, the verification conditions for all other procedures stay the same, and their existing proofs can be reused.

5. Proof rules for runtime code specialisation

In this section we discuss how to reason about programs which perform runtime code generation. We focus on a simple form of runtime code generation, namely the *specialisation* of code at runtime, which our language provides via the specialising quote $\text{`}C\text{'}_{\vec{v}=\vec{e}}$. In particular, we will see that the recursion rule (μ) becomes inadequate in the presence of code specialisation at runtime, and we will propose a more powerful recursion rule (μSETS).

Recall that the specialising quote $\text{`}C\text{'}_{\vec{v}=\vec{e}}$ turns command C into a value, but first replacing the variables $\vec{v}$ in C with the current values of the expressions $\vec{e}$.

To see how the specialising quote might be used, let us consider an optimisation[3] to our program for finding Pythagorean triples from Section 4. When trying a sequence of triples such as

$$(3,4,1),\ (3,4,2),\ (3,4,3)\ ,\ldots,\ (3,4,m),\ \ldots$$

our current program calculates the squares of 3 and 4 and sums them anew for each triple in the sequence, which is wasteful. Our optimised program, shown in Fig. 7, avoids this. Every time one of the variables x or y is updated, the value of $x^2 + y^2$ is calculated and a piece of code specialised with this value is stored in f.

This program is trickier to reason about than the non-optimised version. In the next section we motivate and introduce further proof rules to support convenient reasoning about such programs.

5.1. Meet some further proof rules

The first set of new rules we introduce appears in Fig. 8. These are for reasoning about the execution of commands produced using the specialising

[1]Here we see why the lack of a "Bekic lemma" mentioned in [13] 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 Π_0 just "recomputes" the joint fixed point of C_1 and $\hat{C}_2$.

[2]Strictly, one might wish to explcitly put a universal quantification over R in proof obligations Π_2 and Π_0. This would be easy to add, but for our present purposes we simply note that the proofs of Π_2 and Π_0 do not rely on any properties of R, and thus will remain valid proofs whatever the choice of R.

[3]This optimisation is for the purposes of illustration; we are not seriously trying to get an efficient algorithm for finding Pythagorean triples.

$f :=$ ‘if $m = (z \times z)$ then nop else run g’$_{m=(x\times x)+(y\times y)}$;
$g :=$ ‘C_2’ ;
run f

where

$C_2 \triangleq$
(if $x = n$ then
$n := n + 1; x := 0$;
$f :=$ ‘if $m = (z \times z)$ then nop else run g’$_{m=(x\times x)+(y\times y)}$
else if $y = n$ then
$x := x + 1; y := 0$;
$f :=$ ‘if $m = (z \times z)$ then nop else run g’$_{m=(x\times x)+(y\times y)}$
else if $z = n$ then
$y := y + 1; z := 0$;
$f :=$ ‘if $m = (z \times z)$ then nop else run g’$_{m=(x\times x)+(y\times y)}$
else $z := z + 1$) ;
run f

Figure 7: The program for finding Pythagorean triples, optimised using runtime code generation.

quote. Since the ordinary quote ‘C’ is a special case of the specialising quote ‘C’$_{\vec{v}=\vec{e}}$, these rules (with the exception of (TrivialSpec)) are generalisations of the standard Hoare rules in Fig. 4. The side conditions involving $mod(-)$ are present to ensure that the specialisations can take place properly (recall that in Definition (5) we had to deal with cases such as ‘$x := y$’$_{x=3}$ which, if allowed, would not produce well-formed commands).

In addition to these rules, we must introduce a generalisation of the recursion rule (μ). The reason is that the (μ) rule only allows one to consider a recursion between *finitely many* commands $C_1, \ldots, C_N$. Yet in our optimised example program, there is a *countable infinity* of commands which could end up stored in f, namely the set of all commands of the form

$$\text{if } \mathfrak{m} = (z \times z) \text{ then nop else run } g$$

where $\mathfrak{m}$ is a sum of two squares.

Fig. 9 gives the (μSets) rule which we introduce to deal with such cases, and several other associated rules. The main idea in (μSets) is that instead of using variables $\mathrm{p}_1, \ldots, \mathrm{p}_N$ to refer to single commands, we now use predicates $\mathrm{P}_1, \ldots, \mathrm{P}_N$ to refer to possibly infinite *sets* of commands. Each triple in the context now describes the behaviour of a whole set of commands.

Shortly we will demonstrate these rules by applying them to the optimised program of Fig. 7, but first we will prove the soundness of the new rules.

$$\textsc{ASpec}\quad \frac{}{\{P[x\backslash(e[\vec{v}\backslash\vec{e}])]\}\,`x := e'_{\vec{v}=\vec{e}}\,\{P\}}\;\; x \notin \vec{v}$$

$$\textsc{SSpec}\quad \frac{\Gamma \vdash \{P\}\,`C_1'_{\vec{v}=\vec{e}}\,\{R\} \qquad \Gamma \vdash \{R\}\,`C_2'_{\vec{v}=\vec{e}}\,\{Q\}}{\Gamma \vdash \{P\}\,`C_1; C_2'_{\vec{v}=\vec{e}}\,\{Q\}}\;\; mod(C_1, C_2) \cap \vec{v} = \emptyset$$

$$\textsc{ISpec}\quad \frac{\Gamma \vdash \{P \wedge (e[\vec{v}\backslash\vec{e}]) = 1\}\,`C_1'_{\vec{v}=\vec{e}}\,\{Q\} \qquad \Gamma \vdash \{P \wedge (e[\vec{v}\backslash\vec{e}]) \neq 1\}\,`C_2'_{\vec{v}=\vec{e}}\,\{Q\}}{\Gamma \vdash \{P\}\,`\mathsf{if}\ e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2'_{\vec{v}=\vec{e}}\,\{Q\}}\;\; mod(C_1, C_2) \cap \vec{v} = \emptyset$$

$$\textsc{ChooseSpec}\quad \frac{\Gamma \vdash \{P\}\,`C_1'_{\vec{v}=\vec{e}}\,\{Q\} \qquad \Gamma \vdash \{P\}\,`C_2'_{\vec{v}=\vec{e}}\,\{Q\}}{\Gamma \vdash \{P\}\,`\mathsf{choose}\ C_1\ C_2'_{\vec{v}=\vec{e}}\,\{Q\}}\;\; mod(C_1, C_2) \cap \vec{v} = \emptyset$$

$$\textsc{TrivialSpec}\quad \frac{\Gamma \vdash \{P\}\,`C'\,\{Q\}}{\Gamma \vdash \{P\}\,`C'_{\vec{v}=\vec{e}}\,\{Q\}}\;\; \text{none of } \vec{v} \text{ appear in } C$$

Figure 8: Hoare logic rules for reasoning about specialised commands.

5.2. Soundness of the proof rules

We first address the rules for reasoning about specialised commands (Fig. 8). Here we give proofs for the (ASpec) and (SSpec) rules. The proof for (ISpec) appears in Appendix A; proofs for the other two rules are easier and omitted.

Theorem 5.1. Rule (ASpec) is sound.

Proof. Suppose $x \notin \vec{v}$; we must prove $\vDash \{P[x\backslash(e[\vec{v}\backslash\vec{e}])]\}\,`x := e'_{\vec{v}=\vec{e}}\,\{P\}$. So let $\rho \in \mathsf{Env}$, $\chi \in \mathsf{PEnv}$ and $n \in \mathbb{N}$. We must prove

$$(\rho, \chi, n) \in [\![\{P[x\backslash(e[\vec{v}\backslash\vec{e}])]\}\,`x := e'_{\vec{v}=\vec{e}}\,\{P\}]\!]^{\mathrm{sp}}$$

which (since n cannot be -1) means we must show

$$\rho, \chi \vDash^n \{P[x\backslash(e[\vec{v}\backslash\vec{e}])]\}\,`x := e'_{\vec{v}=\vec{e}}\,\{P\}$$

Because $x \notin \vec{v}$ we have

$$\begin{aligned} [\![`x := e'_{\vec{v}=\vec{e}}]\!]^{\mathrm{ex}}_{\rho} &= \mathscr{G}((x := e)[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}]) \\ &= \mathscr{G}(x := (e[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}])) \end{aligned}$$

and hence

$$\begin{aligned} [\![`x := e'_{\vec{v}=\vec{e}}]\!]^{\mathrm{mid}}_{\rho,\chi} &= \{\mathscr{G}^{-1}(\mathscr{G}(x := (e[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}])))\} \\ &= \{x := (e[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}])\} \end{aligned}$$

μSETS

$$\frac{\bigwedge_{1\le i\le N} \Gamma, \forall \vec{\mathrm{v}}_1\, \{P_1\}\, \mathrm{P}_1(\cdot,\vec{e}_1)\, \{Q_1\}, \ldots, \forall \vec{\mathrm{v}}_N\, \{P_N\}\, \mathrm{P}_N(\cdot,\vec{e}_N)\, \{Q_N\} \;\vdash\; \{P_i\}\, \mathrm{P}_i(\cdot,\vec{e}_i)\, \{Q_i\}}{\bigwedge_{1\le i\le N} \Gamma \vdash \{P_i\}\, \mathrm{P}_i(\cdot,\vec{e}_i)\, \{Q_i\}}$$

$\vec{\mathrm{v}}_i$ is all auxiliary variables free in $P_i, Q_i, \vec{e}_i$
No variable in $\vec{\mathrm{v}}_1, \ldots, \vec{\mathrm{v}}_N$ is free in Γ

HSETS

$$\frac{P \;\Rightarrow\; \mathrm{P}(e, e_1, \ldots, e_k)}{\forall \mathrm{x}_1, \ldots, \mathrm{x}_k.\, \{P \wedge e_1{=}\mathrm{x}_1 \wedge \cdots \wedge e_k{=}\mathrm{x}_k\}\, \mathrm{P}(\cdot, \mathrm{x}_1, \ldots, \mathrm{x}_k)\, \{Q\} \vdash \{P\}\, \text{`}\mathsf{run}\ e\text{'}\, \{Q\}}\ \mathrm{x}_1, \ldots, \mathrm{x}_k \text{ fresh}$$

RSETS

$$\frac{P \;\Rightarrow\; \mathrm{P}(e, e_1, \ldots, e_k) \qquad \Gamma \vdash \{P \wedge e_1{=}\mathrm{x}_1 \wedge \cdots \wedge e_k{=}\mathrm{x}_k\}\, \mathrm{P}(\cdot, \mathrm{x}_1, \ldots, \mathrm{x}_k)\, \{Q\}}{\Gamma \vdash \{P\}\, \text{`}\mathsf{run}\ e\text{'}\, \{Q\}}\ \mathrm{x}_1, \ldots, \mathrm{x}_k \text{ fresh}$$

WWITHCONTEXT

$$\frac{\Gamma, R \vdash \{P'\}\, e\, \{Q'\}}{\Gamma, R \vdash \{P\}\, e\, \{Q\}}\ P \wedge R \Rightarrow P', Q' \wedge R \Rightarrow Q$$

INTROPREDDEF

$$\frac{R \vdash \{P\}\, e\, \{Q\}}{\{P\}\, e\, \{Q\}}$$

assertion R is satisfiable and has no free variables
P, Q contain no predicate symbols

USEPREDDEF

$$\frac{\Gamma \vdash \{P\}\, \text{`}C\text{'}_{\vec{v}=\vec{e}}\, \{Q\}}{\Gamma,\ \forall \mathrm{c} \forall \vec{\mathrm{x}}.\mathrm{P}(\mathrm{c}, \vec{\mathrm{x}}) \Leftrightarrow \mathrm{c} = \text{`}C\text{'}_{\vec{\mathrm{v}}=\vec{\mathrm{x}}} \vdash \{P\}\, \mathrm{P}(\cdot, \vec{e})\, \{Q\}}\ \mathit{mod}(C) \cap \vec{\mathrm{v}} = \emptyset$$

Figure 9: Hoare logic rules for reasoning about possibly infinite sets of commands.

So let $s \in [\![P[x\backslash(e[\vec{v}\backslash\vec{e}])]]\!]^{\mathrm{as}}_{\rho,\chi}$ be such that $(x := (e[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}]), s) \xrightarrow{*} (\mathsf{nop}, s')$ in n steps or fewer; it suffices to show $s' \in [\![P]\!]^{\mathrm{as}}_{\rho,\chi}$. Define $c \triangleq [\![e[\vec{v}\backslash\vec{e}]]\!]^{\mathrm{ex}}_{\rho}$. By familiar properties of substitution we have $s \in [\![P[x\backslash c]]\!]^{\mathrm{as}}_{\rho,\chi}$. By the structure of the transition relation, we know that s' must be reached from s in one step, and

$$s' \quad = \quad \lambda v.\ \text{if } v = x \text{ then } [\![e[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}]]\!]^{\mathrm{ex}}_{\rho} \text{ else } s(v)$$

which (because $[\![e[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}]]\!]^{\mathrm{ex}}_{\rho} = [\![e[\vec{v}\backslash\vec{e}]]\!]^{\mathrm{ex}}_{\rho} = c$) is equal to

$$\lambda v.\ \text{if } v = x \text{ then } c \text{ else } s(v)$$

From this and $s \in [\![P[x\backslash c]]\!]^{\mathrm{as}}_{\rho,\chi}$ it follows by familiar properties of substitution that $s' \in [\![P]\!]^{\mathrm{as}}_{\rho,\chi}$ as required. □

Theorem 5.2. Rule (SSpec) is sound.

Proof. Suppose that the premises and side condition of (SSpec) hold. We must then show that the conclusion

$$\vDash \Gamma \vdash \{P\}\, \text{`}C_1; C_2\text{'}_{\vec{v}=\vec{e}}\, \{Q\}$$

holds. So let $\rho \in \mathsf{Env}$, $\chi \in \mathsf{PEnv}$ and $n \in \mathbb{N}$ be such that $(\rho, \chi, n{-}1) \in [\![\Gamma]\!]^{\mathrm{sp}}$. We must show

$$\rho, \chi \vDash^n \{P\}\, \text{`}C_1; C_2\text{'}_{\vec{v}=\vec{e}}\, \{Q\} \tag{14}$$

From the side condition it follows that we are in the first case of the definition (5) and thus

$$[\![\text{`}C_1; C_2\text{'}_{\vec{v}=\vec{e}}]\!]^{\mathrm{ex}}_{\rho} \;=\; \mathscr{G}((C_1; C_2)[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}]) \;=\; \mathscr{G}((C_1[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}] \;;\; C_2[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}]))$$

Hence

$$\begin{aligned}[\![\text{`}C_1; C_2\text{'}_{\vec{v}=\vec{e}}]\!]^{\mathrm{mid}}_{\rho,\chi} \;&=\; \{\mathscr{G}^{-1}(\mathscr{G}((C_1[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}] \;;\; C_2[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}])))\} \\ &=\; \{C_1[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}] \;;\; C_2[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}]\}\end{aligned}$$

From $(\rho, \chi, n{-}1) \in [\![\Gamma]\!]^{\mathrm{sp}}$ and the premises of (SSpec) we deduce that

$$\rho, \chi \vDash^n \{P\}\, \text{`}C_1\text{'}_{\vec{v}=\vec{e}}\, \{R\} \tag{15}$$

and

$$\rho, \chi \vDash^n \{R\}\, \text{`}C_2\text{'}_{\vec{v}=\vec{e}}\, \{Q\} \tag{16}$$

To prove (14), let

$$(C_1[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}] \;;\; C_2[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}], s^1) \to \cdots \to (\mathsf{nop}, s^K)$$

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

$$\begin{aligned}&(C_1[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}] \;;\; C_2[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}], s^1) \\ \to\;& \cdots \\ \to\;& (\mathsf{nop}; C_2[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}], s^J) \\ \to\;& (C_2[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}], s^{J+1}) \\ \to\;& \cdots \\ \to\;& (\mathsf{nop}, s^K)\end{aligned}$$

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

$$(C_1[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}], s^1) \to \cdots \to (\mathsf{nop}, s^J)$$

By (15) and $[\![\text{`}C_1\text{'}_{\vec{v}=\vec{e}}]\!]^{\mathrm{mid}}_{\rho,\chi} = \{C_1[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}]\}$ we see that $s^J = s^{J+1} \in [\![R]\!]^{\mathrm{as}}_{\rho,\chi}$. Then similarly applying (16) to the execution sequence

$$(C_2[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}], s^{J+1}) \to \cdots \to (\mathsf{nop}, s^K)$$

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

Now we address the rules for reasoning about possibly infinite sets of commands (Fig. 9). Here we focus on the proofs for the (μSets) and (HSets) rules. The rules (UsePredDef), (IntroPredDef) and (RSets) are proved sound in Appendix A; the proof for (WWithContext) is easier and omitted. The proof for (μSets) is by induction on execution length, as was the proof for the (μ) rule.

Theorem 5.3. Rule (μSets) is sound.

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

$$(\rho, \chi, n{-}1) \in [\![\Gamma]\!]^{\mathrm{sp}} \quad \text{implies} \quad \rho, \chi \vDash^n \{P_i\}\mathrm{P}_i(\cdot, \vec{e}_i)\{Q_i\}$$

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

Base case ($n = 0$): Let $\rho \in \mathsf{Env}$, $\chi \in \mathsf{PEnv}$ and let $i \in \{1, \ldots, N\}$ be such that $(\rho, \chi, -1) \in [\![\Gamma]\!]^{\mathrm{sp}}$. It follows from the definition of $[\![-]\!]^{\mathrm{sp}}$ that

$$(\rho, \chi, -1) \in [\![\forall \vec{\mathrm{v}}_1\, \{P_1\}\, \mathrm{P}_1(\cdot, \vec{e}_1)\, \{Q_1\}, \ldots, \forall \vec{\mathrm{v}}_N\, \{P_N\}\, \mathrm{P}_N(\cdot, \vec{e}_N)\, \{Q_N\}]\!]^{\mathrm{sp}}$$

Then from the premise of (μDirect), unpacking the definitions and instantiating n with 0, we immediately obtain $\rho, \chi \vDash^0 \{P_i\}\, \mathrm{P}_i(\cdot, \vec{e}_i)\, \{Q_i\}$ as required.

Inductive case ($n > 0$): Let $\Phi(n-1)$ hold. Let $\rho \in \mathsf{Env}$ and $\chi \in \mathsf{PEnv}$ be such that $(\rho, \chi, n{-}1) \in [\![\Gamma]\!]^{\mathrm{sp}}$ and let $i \in \{1, \ldots, N\}$. We must prove $\rho, \chi \vDash^n \{P_i\}\mathrm{P}_i(\cdot, \vec{e}_i)\{Q_i\}$. We know that $-1 \leq n{-}2$ and $(\rho, \chi, n{-}1) \in [\![\Gamma]\!]^{\mathrm{sp}}$ so it follows by Remark 2.2 that $(\rho, \chi, n{-}2) \in [\![\Gamma]\!]^{\mathrm{sp}}$.
The next part of our argument is to show

$$(\rho, \chi, n{-}1) \in [\![\forall \vec{\mathrm{v}}_1\, \{P_1\}\, \mathrm{P}_1(\cdot, \vec{e}_1)\, \{Q_1\}, \ldots, \forall \vec{\mathrm{v}}_N\, \{P_N\}\, \mathrm{P}_N(\cdot, \vec{e}_N)\, \{Q_N\}]\!]^{\mathrm{sp}} \quad (17)$$

So let $j \in \{1, \ldots, N\}$ and we need to show $(\rho, \chi, n{-}1) \in [\![\forall \vec{\mathrm{v}}_j\, \{P_j\}\, \mathrm{P}_j(\cdot, \vec{e}_j)\, \{Q_j\}]\!]^{\mathrm{sp}}$. Let $\hat{\rho}$ agree with ρ except possibly at $\vec{\mathrm{v}}_j$; then it will suffice to prove $\hat{\rho}, \chi \vDash^{n-1} \{P_j\}\, \mathrm{P}_j(\cdot, \vec{e}_j)\, \{Q_j\}$. Since variables $\vec{\mathrm{v}}_j$ do not appear free in Γ, we have $(\hat{\rho}, \chi, n{-}2) \in [\![\Gamma]\!]^{\mathrm{sp}}$. It then follows from the induction hypothesis $\Phi(n-1)$ that $\hat{\rho}, \chi \vDash^{n-1} \{P_j\}\mathrm{P}_j(\cdot, \vec{e}_j)\{Q_j\}$ as required. Hence we have established (17).

Then by the premise of (μSets) it follows that $(\rho, \chi, n) \in [\![\{P_i\}\, \mathrm{P}_i(\cdot, \vec{e}_i)\, \{Q_i\}]\!]^{\mathrm{sp}}$ whence $\rho, \chi \vDash^n \{P_i\}\mathrm{P}_i(\cdot, \vec{e}_i)\{Q_i\}$ as required.

□

Theorem 5.4. Rule (HSets) is sound.

Proof. Let $\rho \in \mathsf{Env}$, $\chi \in \mathsf{PEnv}$, $n \in \mathbb{N}$ be such that

$$(\rho, \chi, n{-}1) \in [\![\forall \mathrm{x}_1, \ldots, \mathrm{x}_k.\, \{P \wedge e_1{=}\mathrm{x}_1 \wedge \cdots \wedge e_k{=}\mathrm{x}_k\}\, \mathrm{P}(\cdot, \mathrm{x}_1, \ldots, \mathrm{x}_k)\, \{Q\}]\!]^{\mathrm{sp}}) \quad (18)$$

We must prove $\rho, \chi \vDash^n \{P\}\, \text{`}\mathsf{run}\ e\text{'}\, \{Q\}$. If $n = 0$ then this is trivially true, so assume $n > 0$. Let $s \in [\![P]\!]^{\mathrm{as}}_{\rho,\chi}$ and s' be such that $(\mathsf{run}\ e, s) \xrightarrow{*} (\mathsf{nop}, s')$ in n steps or fewer; we are required to show $s' \in [\![Q]\!]^{\mathrm{as}}_{\rho,\chi}$.

Define environment ρ' as follows:

$$\rho'(\mathrm{v}) \quad \triangleq \quad \begin{cases} [\![e_i]\!]^{\mathrm{ex}}_{s,\rho} & \text{if v is } \mathrm{x}_i \\ \rho(\mathrm{v}) & \text{otherwise} \end{cases}$$

From (18), $n > 0$ and the definition of ρ' it follows that

$$\rho', \chi \models^{n-1} \{P \wedge e_1{=}\mathrm{x}_1 \wedge \cdots \wedge e_k{=}\mathrm{x}_k\}\, \mathrm{P}(\cdot, \mathrm{x}_1, \ldots, \mathrm{x}_k)\, \{Q\} \tag{19}$$

From $s \in [\![P]\!]^{\mathrm{as}}_{\rho,\chi}$, the freshness of $\mathrm{x}_1, \ldots, \mathrm{x}_n$ and the definition of ρ' we see that

$$s \in [\![P \wedge e_1{=}\mathrm{x}_1 \wedge \cdots \wedge e_k{=}\mathrm{x}_k]\!]^{\mathrm{as}}_{\rho',\chi} \tag{20}$$

Define $C \triangleq \mathscr{G}^{-1}([\![e]\!]^{\mathrm{ex}}_s)$. From (20) and the premise $P \Rightarrow \mathrm{P}(e, e_1, \ldots, e_k)$ we have $s \in [\![\mathrm{P}(e, e_1, \ldots, e_k)]\!]^{\mathrm{as}}_{\rho',\chi}$, which means that

$$([\![e]\!]^{\mathrm{ex}}_s, [\![\mathrm{x}_1]\!]^{\mathrm{ex}}_{\rho'}, \ldots, [\![\mathrm{x}_k]\!]^{\mathrm{ex}}_{\rho'}) \in \chi(\mathrm{P})$$

From this and $C = \mathscr{G}^{-1}([\![e]\!]^{\mathrm{ex}}_s)$ it follows that

$$C \in [\![\mathrm{P}(\cdot, \mathrm{x}_1, \ldots, \mathrm{x}_k)]\!]^{\mathrm{mid}}_{\rho',\chi} \tag{21}$$

Due to the structure of the transition relation $\rightarrow$, we must have $(C, s) \xrightarrow{*} (\mathsf{nop}, s')$ in $n{-}1$ steps or fewer. Combining this with (19), (20) and (21) we find that $s' \in [\![Q]\!]^{\mathrm{as}}_{\rho',\chi}$. Because $\mathrm{x}_1, \ldots, \mathrm{x}_k$ do not appear in Q, we have $s' \in [\![Q]\!]^{\mathrm{as}}_{\rho,\chi}$ as required. □

6. Proof of our program which uses runtime code specialisation

We now have all the rules required to show correctness of our optimised program for finding Pythagorean triples (Fig. 7), which uses runtime specialisation of code. In this section we sketch the correctness proof.

Let C_0 be the main program. We shall prove:

$$\{\mathit{true}\}\, C_0\, \{R(x, y, z)\} \tag{22}$$

We will use two predicates in our proof, Check$(-,-)$ and Next$(-)$. These correspond to the two kinds of stored commands used by our program: commands to *check* whether x, y, z is a Pythagorean triple, and commands to generate the *next* triple to try. Let *DefCheck* and *DefNext* respectively be the following formulae, which will function as our definitions of the Check and Next predicates.

$$\forall \mathrm{c}, \mathrm{x}.\ \mathrm{Check}(\mathrm{c}, \mathrm{x}) \quad \Leftrightarrow \quad \mathrm{c} = \text{`}\mathsf{if}\ m = (z \times z)\ \mathsf{then\ nop\ else\ run}\ g\text{'}_{m=\mathrm{x}}$$

$$\forall \mathrm{c}.\ \mathrm{Next}(\mathrm{c}) \quad \Leftrightarrow \quad \mathrm{c} = \text{`}C_2\text{'}$$

These formulae are trivially satisfiable and have no free variables so we can use the IntroPredDef rule to reduce the proof obligation (22) to

$$\mathit{DefCheck}, \mathit{DefNext} \ \vdash\ \{\mathit{true}\}\, C_0\, \{R(x, y, z)\}$$

This breaks down by sequential composition to showing three things:

$$\begin{array}{l} \mathit{DefCheck}, \mathit{DefNext} \\ \vdash \\ \{\mathit{true}\}\ f := \text{`}\mathsf{if}\ m = (z \times z)\ \mathsf{then\ nop\ else\ run}\ g\text{'}_{m=(x\times x)+(y\times y)}\ \big\{\mathrm{Check}(f, x^2+y^2)\big\} \end{array} \tag{23}$$

$$\begin{array}{l} \mathit{DefCheck}, \mathit{DefNext} \\ \vdash \\ \big\{\mathrm{Check}(f, x^2+y^2)\big\}\ g := \text{`}C_2\text{'}\ \big\{\mathrm{Check}(f, x^2+y^2) \wedge \mathrm{Next}(g)\big\} \end{array} \tag{24}$$

$$\begin{array}{l} \mathit{DefCheck}, \mathit{DefNext} \\ \vdash \\ \big\{\mathrm{Check}(f, x^2+y^2) \wedge \mathrm{Next}(g)\big\}\ \mathsf{run}\ f\ \{R(x,y,z)\} \end{array} \tag{25}$$

Of these, (23) and (24) show that f and g are initialised with appropriate pieces of (checking and generating) code. We prove them using a combination of the assignment rule (A) and the (WWithContext) rule, which allows us to make use of the predicate definitions *DefCheck*, *DefNext* from the context while doing consequence rule style reasoning. This leaves (25) which by (RSets) and (W) will follow from:

$$\begin{array}{l} \mathit{DefCheck}, \mathit{DefNext} \\ \vdash \\ \big\{\mathrm{m} = x^2+y^2 \wedge \mathrm{Check}(f, \mathrm{m}) \wedge \mathrm{Next}(g)\big\}\ \mathrm{Check}(\cdot, \mathrm{m})\ \{R(x,y,z)\} \end{array} \tag{26}$$

In our earlier proof in Section 4, we used $T(-)$ as a shorthand when writing our behavioural specifications. In this proof we will use shorthand in a similar way: this time we take T_{Check} and T_{Next} to be the following triples.

$$T_{\mathrm{Check}} \quad \triangleq \quad \left\{ \begin{array}{l} \mathrm{Check}(f, x^2+y^2) \\ \wedge\ \mathrm{Next}(g) \\ \wedge\ \mathrm{m} = x^2+y^2 \end{array} \right\}\ \mathrm{Check}(\cdot, \mathrm{m}) \quad \{\ R(x,y,z)\ \}$$

$$T_{\mathrm{Next}} \quad \triangleq \quad \left\{ \begin{array}{l} \mathrm{Check}(f, x^2+y^2) \\ \wedge\ \mathrm{Next}(g) \end{array} \right\}\ \mathrm{Next}(\cdot) \quad \{\ R(x,y,z)\ \}$$

We now use the following instance of the (μSets) rule (where we afford ourselves the liberty of writing multiple conclusions):

$$\frac{\mathit{DefCheck}, \mathit{DefNext}, \forall \mathrm{m}.T_{\mathrm{Check}}, T_{\mathrm{Next}} \ \vdash\ T_{\mathrm{Check}}, T_{\mathrm{Next}}}{\mathit{DefCheck}, \mathit{DefNext} \vdash T_{\mathrm{Check}}, T_{\mathrm{Next}}}$$

Thus to get (26) it is enough to prove

$$\mathit{DefCheck}, T_{\mathrm{Next}} \vdash T_{\mathrm{Check}} \tag{27}$$

and

$$\mathit{DefCheck}, \mathit{DefNext}, \forall \mathrm{m}.T_{\mathrm{Check}} \vdash T_{\mathrm{Next}} \tag{28}$$

First we shall prove (27) which, in full, is:

$$\begin{array}{l} \forall \mathrm{c}, \mathrm{x}.\ \mathrm{Check}(\mathrm{c}, \mathrm{x}) \ \Leftrightarrow\ \mathrm{c} = \text{`}\mathsf{if}\ m = (z \times z)\ \mathsf{then\ nop\ else\ run}\ g\text{'}_{m=\mathrm{x}}, \\ \big\{\mathrm{Check}(f, x^2+y^2) \wedge \mathrm{Next}(g)\big\}\,\mathrm{Next}(\cdot)\,\{R(x,y,z)\} \\ \vdash\ \big\{\mathrm{Check}(f, x^2+y^2) \wedge \mathrm{Next}(g) \wedge \mathrm{m} = x^2+y^2\big\}\,\mathrm{Check}(\cdot, \mathrm{m})\,\{R(x,y,z)\} \end{array}$$

Since $m \notin \mathit{mod}(\mathsf{if}\ m = (z \times z)\ \mathsf{then\ nop\ else\ run}\ g)$, the (UsePredDef) rule reduces this to:

$$\begin{array}{l} \big\{\mathrm{Check}(f, x^2+y^2) \wedge \mathrm{Next}(g)\big\}\,\mathrm{Next}(\cdot)\,\{R(x,y,z)\} \\ \vdash \left\{\begin{array}{l} \mathrm{Check}(f, x^2+y^2) \\ \wedge\ \mathrm{Next}(g) \\ \wedge\ \mathrm{m} = x^2+y^2 \end{array}\right\}\ \text{`}\mathsf{if}\ m = (z \times z)\ \mathsf{then\ nop\ else\ run}\ g\text{'}_{m=\mathrm{m}}\ \ \{\ R(x,y,z)\ \} \end{array}$$

We next apply the (ISpec) rule, which we can do because $m \notin \mathit{mod}(\mathsf{nop}, \mathsf{run}\ g)$, leaving us to prove

$$\begin{array}{l} \big\{\mathrm{Check}(f, x^2+y^2) \wedge \mathrm{Next}(g)\big\}\,\mathrm{Next}(\cdot)\,\{R(x,y,z)\} \\ \vdash \left\{\begin{array}{l} \mathrm{m} = z \times z \\ \wedge\ \mathrm{Check}(f, x^2+y^2) \\ \wedge\ \mathrm{Next}(g) \\ \wedge\ \mathrm{m} = x^2+y^2 \end{array}\right\}\ \text{`}\mathsf{nop}\text{'}_{m=\mathrm{m}}\ \ \{\ R(x,y,z)\ \} \end{array}$$

and

$$\begin{array}{l} \big\{\mathrm{Check}(f, x^2+y^2) \wedge \mathrm{Next}(g)\big\}\,\mathrm{Next}(\cdot)\,\{R(x,y,z)\} \\ \vdash \left\{\begin{array}{l} \mathrm{m} \neq z \times z \\ \wedge\ \mathrm{Check}(f, x^2+y^2) \\ \wedge\ \mathrm{Next}(g) \\ \wedge\ \mathrm{m} = x^2+y^2 \end{array}\right\}\ \text{`}\mathsf{run}\ g\text{'}_{m=\mathrm{m}}\ \ \{\ R(x,y,z)\ \} \end{array}$$

Using the (TrivialSpec) rule, along with (AddContext) and (W) to discard some unneeded assumptions, these two obligations reduce to:

$$\left\{\begin{array}{l} \mathrm{m} = z \times z \\ \wedge\ \mathrm{m} = x^2+y^2 \end{array}\right\}\ \ \mathsf{nop}\ \ \{\ R(x,y,z)\ \}$$

and

$$\begin{array}{l} \big\{\mathrm{Check}(f, x^2+y^2) \wedge \mathrm{Next}(g)\big\}\,\mathrm{Next}(\cdot)\,\{R(x,y,z)\} \\ \vdash \left\{\begin{array}{l} \mathrm{Check}(f, x^2+y^2) \\ \wedge\ \mathrm{Next}(g) \end{array}\right\}\ \ \mathsf{run}\ g\ \ \{\ R(x,y,z)\ \} \end{array}$$

The first of these is trivial; the second is an instance of the (HSets) rule because $\mathrm{Check}(f, x^2+y^2) \wedge \mathrm{Next}(g)\ \Rightarrow\ \mathrm{Next}(g)$.

To finish we prove (28) which, in full, is:

$$\begin{array}{l}\forall \mathrm{c}, \mathrm{x}.\ \mathrm{Check}(\mathrm{c}, \mathrm{x}) \ \Leftrightarrow\ \mathrm{c} = \text{`}\mathsf{if}\ m = (z \times z)\ \mathsf{then\ nop\ else\ run}\ g\text{'}_{m=\mathrm{x}},\\ \forall \mathrm{c}.\ \mathrm{Next}(\mathrm{c}) \ \Leftrightarrow\ \mathrm{c} = \text{`}C_2\text{'},\\ \forall \mathrm{m}.\ \{\mathrm{Check}(f, x^2+y^2) \wedge \mathrm{Next}(g) \wedge \mathrm{m} = x^2+y^2\}\ \mathrm{Check}(\cdot, \mathrm{m})\ \{R(x,y,z)\}\\ \vdash\ \{\mathrm{Check}(f, x^2+y^2) \wedge \mathrm{Next}(g)\}\ \mathrm{Next}(\cdot)\ \{R(x,y,z)\}\end{array}$$

The (UsePredDef) rule reduces this to

$$\begin{array}{l}\forall \mathrm{c}, \mathrm{x}.\ \mathrm{Check}(\mathrm{c}, \mathrm{x}) \ \Leftrightarrow\ \mathrm{c} = \text{`}\mathsf{if}\ m = (z \times z)\ \mathsf{then\ nop\ else\ run}\ g\text{'}_{m=\mathrm{x}},\\ \forall \mathrm{x}.\ \{\mathrm{Check}(f, x^2+y^2) \wedge \mathrm{Next}(g) \wedge \mathrm{x} = x^2+y^2\}\ \mathrm{Check}(\cdot, \mathrm{x})\ \{R(x,y,z)\}\\ \vdash\ \{\mathrm{Check}(f, x^2+y^2) \wedge \mathrm{Next}(g)\}\ \text{`}C_2\text{'}\ \{R(x,y,z)\}\end{array}$$

By (S) and (AddContext) it suffices to prove

$$\begin{array}{l}\forall \mathrm{c}, \mathrm{x}.\ \mathrm{Check}(\mathrm{c}, \mathrm{x}) \ \Leftrightarrow\ \mathrm{c} = \text{`}\mathsf{if}\ m = (z \times z)\ \mathsf{then\ nop\ else\ run}\ g\text{'}_{m=\mathrm{x}}\\ \vdash\ \{\mathrm{Check}(f, x^2+y^2) \wedge \mathrm{Next}(g)\}\ \hat{C}\ \{\mathrm{Check}(f, x^2+y^2) \wedge \mathrm{Next}(g)\}\end{array} \tag{29}$$

where $\hat{C}$ is the if-then-else cascade which makes up most of C_2 in Fig. 7, and

$$\begin{array}{l}\forall \mathrm{x}.\ \{\mathrm{Check}(f, x^2+y^2) \wedge \mathrm{Next}(g) \wedge \mathrm{x} = x^2+y^2\}\ \mathrm{Check}(\cdot, \mathrm{x})\ \{R(x,y,z)\}\\ \vdash\ \{\mathrm{Check}(f, x^2+y^2) \wedge \mathrm{Next}(g)\}\ \mathsf{run}\ f\ \{R(x,y,z)\}\end{array} \tag{30}$$

The proof of (29) is easy; as in the proof of the first initialisation statement (23), the (WWithContext) rule allows us to use the defining formula for Check from the context.

To prove (30) we first use the (∀InstContext) and (AddContext) rules; by these (30) will follow from

$$\begin{array}{l}\{\mathrm{Check}(f, x^2+y^2) \wedge \mathrm{Next}(g) \wedge \mathrm{m} = x^2+y^2\}\ \mathrm{Check}(\cdot, \mathrm{m})\ \{R(x,y,z)\}\\ \vdash\ \{\mathrm{Check}(f, x^2+y^2) \wedge \mathrm{Next}(g)\}\ \mathsf{run}\ f\ \{R(x,y,z)\}\end{array}$$

and this is an instance of (RSets).

7. Related and future work

Our semantic model was partly inspired by a paper by von Oheimb [18], which reports soundness proofs of rules for reasoning about mutually recursive procedures. That paper uses operational semantics, and uses induction to prove the recursion rules as we do. A minor difference is that our inductions are on execution length whereas in [18] the number of calls made in an execution is used. Similar induction arguments are also used for example by Parkinson [19] to give a logic for Java. The fact that logics such as von Oheimb's and Parkinson's support modular proofs is what led us to suspect that the logic we work with should also support modular proofs (which we confirmed in Section 4).

The original paper by Reus and Streicher [13] spawned a line of follow-up work [3, 5, 6, 20, 21, 22] addressing various aspects of reasoning about higher order store programs using domain-theoretic semantics. Initially, [3, 5] studied the application of the ideas of [13] to a programming language with a heap, adding *separation logic* connectives [23] to the assertion language. As we remarked earlier, difficulties concerning the non-determinism of dynamic memory allocation had to be overcome. Then, to address the perceived lack of modularity of these logics, *nested Hoare triples* [6, 20] were added. However, these nested triples lead to even greater theoretical complications: [6, 20] use Kripke models based on recursively defined ultrametric spaces.

Hence, in the light of what we now know — that the logic of [13] can be given a simple semantics, and already supports modular proofs — it will be interesting to revisit [6, 20] 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* [24, 20] may be one such thing, but we cannot yet be sure.

We mention two other approaches to semantically justifying a logic with nested Hoare triples. In [25], 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 ([25] was the first paper to give a theory of nested triples, there named *evaluation formulae*). In [26] the *step indexing* technique [27, 17] 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, with some small differences in the resulting logic [28, §6].

The research on higher order store mentioned above is theoretical in character. Research by Jacobs, Smans and Piessens [29] on the verification of dynamically loaded and unloaded kernel modules has a more applied flavour, as does work by Cai, Shao and Vaynberg [30] on verifying self-modifying assembly code and work by Charlton, Horsfall and Reus [31] on verifying dynamic software updates. Overall, however, there has been little exploration of how the ideas of higher order store can be put to work in the verification of real software programs. By its nature, of course, the question of practical applicability cannot be resolved by further theoretical work, so research focused on applications is needed.

Other future work is to adapt the ideas of Section 5 to other forms of runtime code generation. For example, programs in JavaScript and Perl can build up commands represented as source code strings at runtime, and then have these parsed and run with a special *eval* operation.

7.1. An even simpler approach to higher order store?

In this paper we have made progress on the problem of providing a Hoare logic with features designed to deal with higher order store. Yet this is not the only possible approach to solving higher order store verification problems. We conclude our discussion of related and future work by examining an alternative method.

Suppose we are given a verification problem consisting of a higher order store program P and a specification Φ of the behaviour P should have; our goal is to prove $P \vDash \Phi$ for some appropriate satisfaction relation $\vDash$. The alternative idea is to develop a *translation* $P \vDash \Phi \mapsto P' \vDash \Phi'$ of verification problems such that:

1. the translated program P' uses no higher order store features, and
2. the translation is *sound*, i.e. $P' \vDash \Phi'$ implies $P \vDash \Phi$.

Once this is done, we can attack any higher order store verification problem $P \vDash \Phi$ by translating it to $P' \vDash \Phi'$ and then attempting to solve this translated problem. Since P' does not use higher order store, we can attack $P' \vDash \Phi'$ with the many "off the shelf" tools for automated software verification.

This approach is used successfully by Hayden et al. [32] for verifying dynamic software updates. There, verification problems in which a program applies updates to itself as it runs – a use of higher order store – are translated into problems concerning a conventional procedural program, which uses ordinary variables (integers and Booleans) to track which updates have been applied. The translation used by Hayden et al. is easy to understand, preserves the structure of programs and in fact gives $P' \vDash \Phi'$ iff $P \vDash \Phi$.

We believe it will be fruitful to apply this approach to other verification problems of practical interest which (in one formulation) involve higher order store.

8. Conclusions

We revisited the problem of providing a Hoare logic for a simple language for higher order store programs. Firstly we presented a simpler semantic model of the programming language, using flat states rather than domains. This model leads to straightforward soundness proofs, yet gives rise to a more powerful logic. We eliminated unintuitive restrictions on proof rules, and added some convenient new rules. Additionally our model handles non-determinism and testing of syntactic equality on commands. Secondly we explained and demonstrated with an example that, contrary to what has been stated in the literature, the proof system we work with does support modular proofs. Thirdly we extended the programming language with an operator for runtime specialisation of code, and gave rules for reasoning about this operator, including a new recursion rule. We demonstrated these rules with an example.

Overall, our 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.

Acknowledgements

The author acknowledges the support of EPSRC grant (EP/G003173/1) "*From Reasoning Principles for Function Pointers To Logics for Self-Configuring Programs*".

References

[1] A. J. Ahmed, A. W. Appel, R. Virga, A stratified semantics of general references, in: [33].

[2] J. Schwinghammer, A typed semantics of higher-order store and subtyping, in: M. Coppo, E. Lodi, G. M. Pinna (Eds.), ICTCS, volume 3701 of *Lecture Notes in Computer Science*, Springer, 2005, pp. 390–405.

[3] B. Reus, J. Schwinghammer, Separation logic for higher-order store, in: Z. Ésik (Ed.), CSL, volume 4207 of *Lecture Notes in Computer Science*, Springer, 2006, pp. 575–590.

[4] B. Reus, From reasoning principles for function pointers to logics for self-configuring programs: case for support, 2008. Department of Informatics, University of Sussex.

[5] L. Birkedal, B. Reus, J. Schwinghammer, H. Yang, A simple model of separation logic for higher-order store, in: L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, I. Walukiewicz (Eds.), ICALP (2), volume 5126 of *Lecture Notes in Computer Science*, Springer, 2008, pp. 348–360.

[6] J. Schwinghammer, L. Birkedal, B. Reus, H. Yang, Nested Hoare triples and frame rules for higher-order store, in: E. Grädel, R. Kahle (Eds.), CSL, volume 5771 of *Lecture Notes in Computer Science*, Springer, 2009, pp. 440–454.

[7] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, I. Neamtiu, Mutatis mutandis: Safe and predictable dynamic software updating, ACM Trans. Program. Lang. Syst. 29 (2007).

[8] G. Bierman, M. Hicks, P. Sewell, G. Stoyle, Formalizing dynamic software updating, in: 2nd International Workshop on Unanticipated Software Evolution (USE 2003), pp. 13–23.

[9] D. Keppel, S. J. Eggers, R. R. Henry, A Case for Runtime Code Generation, Technical Report UWCSE 91-11-04, University of Washington Department of Computer Science and Engineering, 1991.

[10] M. Abadi, L. Cardelli, An imperative object calculus, in: P. D. Mosses, M. Nielsen, M. I. Schwartzbach (Eds.), TAPSOFT, volume 915 of *Lecture Notes in Computer Science*, Springer, 1995, pp. 471–485.

[11] C. A. R. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (1969) 576–580.

[12] K. R. Apt, Ten years of Hoare's logic: A survey - part I, ACM Trans. Program. Lang. Syst. 3 (1981) 431–483.

[13] B. Reus, T. Streicher, About Hoare logics for higher-order store, in: L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, M. Yung (Eds.), ICALP, volume 3580 of *Lecture Notes in Computer Science*, Springer, 2005, pp. 1337–1348.

[14] P. J. Landin, The mechanical evaluation of expressions, Computer Journal 6 (1964) 308–320.

[15] A. M. Pitts, Relational properties of domains, Inf. Comput. 127 (1996) 66–90.

[16] N. Charlton, Hoare logic for higher order store using simple semantics, in: L. D. Beklemishev, R. de Queiroz (Eds.), WoLLIC, volume 6642 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 52–66.

[17] N. Benton, C.-K. Hur, Step-indexing: The good, the bad and the ugly, in: A. Ahmed, N. Benton, L. Birkedal, M. Hofmann (Eds.), Modelling, Controlling and Reasoning About State, number 10351 in Dagstuhl Seminar Proceedings, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, Dagstuhl, Germany, 2010.

[18] D. von Oheimb, Hoare logic for mutual recursion and local variables, in: C. P. Rangan, V. Raman, R. Ramanujam (Eds.), FSTTCS, volume 1738 of *Lecture Notes in Computer Science*, Springer, 1999, pp. 168–180.

[19] M. J. Parkinson, Local reasoning for Java, Ph.D. thesis, University of Cambridge, Computer Laboratory, 2005.

[20] J. Schwinghammer, H. Yang, L. Birkedal, F. Pottier, B. Reus, A semantic foundation for hidden state, in: C.-H. L. Ong (Ed.), FOSSACS, volume 6014 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 2–17.

[21] N. Charlton, B. Reus, Specification patterns and proofs for recursion through the store, in: O. Owe, M. Steffen, J. A. Telle (Eds.), FCT, volume 6914 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 310–321.

[22] N. Charlton, B. Horsfall, B. Reus, Crowfoot: A verifier for higher-order store programs, in: V. Kuncak, A. Rybalchenko (Eds.), VMCAI, volume 7148 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 136–151.

[23] J. C. Reynolds, Separation logic: A logic for shared mutable data structures, in: [33], pp. 55–74.

[24] F. Pottier, Hiding local state in direct style: A higher-order anti-frame rule, in: LICS, IEEE Computer Society, 2008, pp. 331–340.

[25] K. Honda, N. Yoshida, M. Berger, An observationally complete program logic for imperative higher-order functions, in: LICS, IEEE Computer Society, 2005, pp. 270–279.

[26] L. Birkedal, B. Reus, J. Schwinghammer, K. Støvring, J. Thamsborg, H. Yang, Step-indexed kripke models over recursive worlds, in: T. Ball, M. Sagiv (Eds.), POPL, ACM, 2011, pp. 119–132.

[27] A. W. Appel, D. A. McAllester, An indexed model of recursive types for foundational proof-carrying code, ACM Trans. Program. Lang. Syst. 23 (2001) 657–683.

[28] J. Schwinghammer, L. Birkedal, B. Reus, H. Yang, Nested Hoare triples and frame rules for higher-order store, Logical Methods in Computer Science 7 (2011).

[29] B. Jacobs, J. Smans, F. Piessens, Verification of unloadable modules, in: M. Butler, W. Schulte (Eds.), FM, volume 6664 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 402–416.

[30] H. Cai, Z. Shao, A. Vaynberg, Certified self-modifying code, in: J. Ferrante, K. S. McKinley (Eds.), PLDI, ACM, 2007, pp. 66–77.

[31] N. Charlton, B. Horsfall, B. Reus, Formal reasoning about runtime code update, in: S. Abiteboul, K. Böhm, C. Koch, K.-L. Tan (Eds.), ICDE Workshops, IEEE, 2011, pp. 134–138.

[32] C. M. Hayden, S. Magill, M. Hicks, N. Foster, J. S. Foster, Specifying and verifying the correctness of dynamic software updates, in: R. Joshi, P. Müller, A. Podelski (Eds.), VSTTE, volume 7152 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 278–293.

[33] 17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings, IEEE Computer Society, 2002.

Appendix A. Further soundness proofs

Theorem Appendix A.1. Rule (ISPEC) is sound.

Proof. Suppose that the premises and side condition of (ISPEC) hold. We must then show that the conclusion

$$\Gamma \vdash \{P\}\,\text{`}\mathsf{if}\ e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\text{'}_{\vec{v}=\vec{e}}\,\{Q\}$$

holds. So let $\rho \in \mathsf{Env}$, $\chi \in \mathsf{PEnv}$ and $n \in \mathbb{N}$ be such that $(\rho, \chi, n-1) \in [\![\Gamma]\!]^{\mathrm{sp}}$. We must show

$$\rho, \chi \vDash^n \{P\}\,\text{`}\mathsf{if}\ e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\text{'}_{\vec{v}=\vec{e}}\,\{Q\} \tag{A.1}$$

From the side condition it follows that $mod(\mathsf{if}\ e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2) \cap \vec{v} = \emptyset$, so we are in the first case of Definition (5) and thus

$$\begin{aligned} [\![\text{`}\mathsf{if}\ e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\text{'}_{\vec{v}=\vec{e}}]\!]^{\mathrm{ex}}_{\rho} &= \mathscr{G}((\mathsf{if}\ e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2)[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}]) \\ &= \mathscr{G}(\mathsf{if}\ e[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}]\ \mathsf{then}\ (C_1[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}])\ \mathsf{else}\ (C_2[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}])) \end{aligned}$$

It then follows that

$$\begin{aligned}\llbracket \text{`if } e \text{ then } C_1 \text{ else } C_2\text{'}_{\vec{v}=\vec{e}} \rrbracket^{\mathrm{mid}}_{\rho,\chi} &= \{\mathscr{G}^{-1}(\mathscr{G}(\text{if } e[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}] \text{ then } (C_1[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}]) \text{ else } (C_2[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}])))\} \\ &= \{\text{if } e[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}] \text{ then } (C_1[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}]) \text{ else } (C_2[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}])\}\end{aligned}$$

So let $s \in [\![P]\!]^{\mathrm{as}}_{\rho,\chi}$ be such that

$$(\text{if } e[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}] \text{ then } (C_1[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}]) \text{ else } (C_2[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}]), s) \xrightarrow{*} (\mathsf{nop}, s')$$

in n steps or fewer; to prove (A.1) it suffices to show that $s' \in [\![Q]\!]^{\mathrm{as}}_{\rho,\chi}$. By the structure of the transition relation, there are two possibilities:

1. $[\![e[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}]]\!]^{\mathrm{ex}}_{s} = 1$ and $(C_1[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}], s) \xrightarrow{*} (\mathsf{nop}, s')$ in $n-1$ steps or fewer.
2. $[\![e[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}]]\!]^{\mathrm{ex}}_{s} \neq 1$ and $(C_2[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}], s) \xrightarrow{*} (\mathsf{nop}, s')$ in $n-1$ steps or fewer.

We will only do the proof for case 1 because that for case 2 is very similar. So suppose $[\![e[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}]]\!]^{\mathrm{ex}}_{s} = 1$. It follows from this by familiar properties of substitution that $[\![e[\vec{v}\backslash\vec{e}]]\!]^{\mathrm{ex}}_{s,\rho} = 1$. From this and $s \in [\![P]\!]^{\mathrm{as}}_{\rho,\chi}$ it follows that $s \in [\![P \wedge (e[\vec{v}\backslash\vec{e}]) = 1]\!]^{\mathrm{as}}_{\rho,\chi}$.

Our plan is now to use the first premise of (Is). To do this, we note that

$$\llbracket \text{`}C_1\text{'}_{\vec{v}=\vec{e}} \rrbracket^{\mathrm{mid}}_{\rho,\chi} = \{\mathscr{G}^{-1}(\llbracket \text{`}C_1\text{'}_{\vec{v}=\vec{e}} \rrbracket^{\mathrm{ex}}_{\rho})\} = \{\mathscr{G}^{-1}(\mathscr{G}(C_1[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}]))\} = \{C_1[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}]\}$$

Combining this with conditions we have already checked, namely:

- $(\rho, \chi, n-1) \in [\![\Gamma]\!]^{\mathrm{sp}}$
- $(C_1[\vec{v}\backslash[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}], s) \xrightarrow{*} (\mathsf{nop}, s')$ in $n-1$ steps or fewer
- $s \in [\![P \wedge (e[\vec{v}\backslash\vec{e}]) = 1]\!]^{\mathrm{as}}_{\rho,\chi}$

we find that $s' \in [\![Q]\!]^{\mathrm{as}}_{\rho,\chi}$ as required. □

Theorem Appendix A.2. Rule (IntroPredDef) is sound.

Proof. Suppose the premise $R \vdash \{P\}\, e\, \{Q\}$ and side-conditions hold; we must prove $\vDash \{P\}\, e\, \{Q\}$. So let $\rho \in \mathsf{Env}$, $\chi \in \mathsf{PEnv}$ and $n \in \mathbb{N}$. We must prove $(\rho, \chi, n) \in [\![\{P\}\, e\, \{Q\}]\!]^{\mathrm{sp}}$. The assertion R is satisfiable so there exist $\hat{\rho} \in \mathsf{Env}$, $\hat{\chi} \in \mathsf{PEnv}$ and $s \in \mathsf{Store}$ such that $s \in [\![R]\!]^{\mathrm{as}}_{\rho,\chi}$. But R cannot contain free program variables, so in fact $[\![R]\!]^{\mathrm{as}}_{\hat{\rho},\hat{\chi}} = \mathsf{Store}$. Because R contains no free auxiliary variables either, it follows that $[\![R]\!]^{\mathrm{as}}_{\rho,\hat{\chi}} = \mathsf{Store}$. This means that $(\rho, \hat{\chi}, n-1) \in [\![R]\!]^{\mathrm{sp}}$. From this and the rule's premise it follows that $(\rho, \hat{\chi}, n) \in [\![\{P\}\, e\, \{Q\}]\!]^{\mathrm{sp}}$. From this it follows that $(\rho, \chi, n) \in [\![\{P\}\, e\, \{Q\}]\!]^{\mathrm{sp}}$ as required, because P, Q contain no predicate symbols so it does not matter if we use χ instead of $\hat{\chi}$. □

Theorem Appendix A.3. Rule (UsePredDef) is sound.

Proof. Suppose the premise

$$\Gamma \vdash \{P\}\, \text{`}C\text{'}_{\vec{v}=\vec{e}}\, \{Q\}$$

and side-condition hold; we must prove

$$\Gamma,\ \forall \mathrm{c} \forall \vec{\mathrm{x}}.\mathrm{P}(\mathrm{c}, \vec{\mathrm{x}}) \Leftrightarrow \mathrm{c} = \text{`}C\text{'}_{\vec{\mathrm{v}}=\vec{\mathrm{x}}} \vdash \{P\}\, \mathrm{P}(\cdot, \vec{e})\, \{Q\}$$

So let $\rho \in \mathsf{Env}$, $\chi \in \mathsf{PEnv}$, $n \in \mathbb{N}$ be such that

$$(\rho, \chi, n-1) \in [\![\Gamma,\ \forall \mathrm{c} \forall \vec{\mathrm{x}}.\mathrm{P}(\mathrm{c}, \vec{\mathrm{x}}) \Leftrightarrow \mathrm{c} = \text{`}C\text{'}_{\vec{\mathrm{v}}=\vec{\mathrm{x}}}]\!]^{\mathrm{sp}} \tag{A.2}$$

We must prove

$$(\rho, \chi, n) \in [\![\{P\}\, \mathrm{P}(\cdot, \vec{e})\, \{Q\}]\!]^{\mathrm{sp}}$$

So let $C' \in [\![\mathrm{P}(\cdot, \vec{e})]\!]^{\mathrm{mid}}_{\rho,\chi}$, and let $s \in [\![P]\!]^{\mathrm{as}}_{\rho,\chi}$ be such that $(C', s) \xrightarrow{*} (\mathsf{nop}, s')$ in n steps or fewer. We must prove $s' \in [\![Q]\!]^{\mathrm{as}}_{\rho,\chi}$.

It follows from $C' \in [\![\mathrm{P}(\cdot, \vec{e})]\!]^{\mathrm{mid}}_{\rho,\chi}$ that there exists $N \in \mathbb{N}$ such that $C' = \mathscr{G}^{-1}(N)$ and $(N, [\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}) \in \chi(\mathrm{P})$ (where we lifted $[\![-]\!]^{\mathrm{ex}}_{-}$ to sequences of expressions). From (A.2) it follows that

$$[\![\forall \mathrm{c} \forall \vec{\mathrm{x}}.\mathrm{P}(\mathrm{c}, \vec{\mathrm{x}}) \Leftrightarrow \mathrm{c} = \text{`}C\text{'}_{\vec{\mathrm{v}}=\vec{\mathrm{x}}}]\!]^{\mathrm{as}}_{\rho,\chi} = \mathsf{Store} \tag{A.3}$$

We define $\hat{\rho}$ to be equal to ρ except that c is mapped to N and the variables $\vec{\mathrm{x}}$ are mapped to values $[\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}$. From (A.3) and the definition of $\hat{\rho}$ we have

$$[\![\mathrm{P}(\mathrm{c}, \vec{\mathrm{x}}) \Leftrightarrow \mathrm{c} = \text{`}C\text{'}_{\vec{\mathrm{v}}=\vec{\mathrm{x}}}]\!]^{\mathrm{as}}_{\hat{\rho},\chi} = \mathsf{Store}$$

But $\hat{\rho}$ is also constructed to ensure

$$[\![\mathrm{P}(\mathrm{c}, \vec{\mathrm{x}})]\!]^{\mathrm{as}}_{\hat{\rho},\chi} = \mathsf{Store}$$

so $[\![\mathrm{c}]\!]^{\mathrm{ex}}_{\hat{\rho}} = [\![\text{`}C\text{'}_{\vec{\mathrm{v}}=\vec{\mathrm{x}}}]\!]^{\mathrm{ex}}_{\hat{\rho}}$. But $[\![\mathrm{c}]\!]^{\mathrm{ex}}_{\hat{\rho}} = N$, so $[\![\text{`}C\text{'}_{\vec{\mathrm{v}}=\vec{\mathrm{x}}}]\!]^{\mathrm{ex}}_{\hat{\rho}} = N$. Thus (since $mod(C) \cap \vec{\mathrm{v}} = \emptyset$) we have $\mathscr{G}(C[\vec{v} \backslash [\![\vec{\mathrm{x}}]\!]^{\mathrm{ex}}_{\hat{\rho}}]) = N$. But $C' = \mathscr{G}^{-1}(N)$ so in fact $C' = C[\vec{v} \backslash [\![\vec{\mathrm{x}}]\!]^{\mathrm{ex}}_{\hat{\rho}}]$. By the definition of $\hat{\rho}$ we have $[\![\vec{\mathrm{x}}]\!]^{\mathrm{ex}}_{\hat{\rho}} = [\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}$, and hence $C' = C[\vec{v} \backslash [\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}]$.

Now we use the premise of (UsePredDef). We know $(\rho, \chi, n-1) \in [\![\Gamma]\!]^{\mathrm{sp}}$ so it follows that $(\rho, \chi, n) \in [\![\{P\}\, \text{`}C\text{'}_{\vec{v}=\vec{e}}\, \{Q\}]\!]^{\mathrm{sp}}$ from which we have

$$\rho, \chi \vDash^{n} \{P\}\, \text{`}C\text{'}_{\vec{v}=\vec{e}}\, \{Q\} \tag{A.4}$$

Now

$$[\![\text{`}C\text{'}_{\vec{v}=\vec{e}}]\!]^{\mathrm{mid}}_{\rho,\chi} = \{\mathscr{G}^{-1}([\![\text{`}C\text{'}_{\vec{v}=\vec{e}}]\!]^{\mathrm{ex}}_{\rho})\} = \{C[\vec{v} \backslash [\![\vec{e}]\!]^{\mathrm{ex}}_{\rho}]\} = C'$$

Combining this with (A.4) we can easily show that $s' \in [\![Q]\!]^{\mathrm{as}}_{\rho,\chi}$ as required.

□

Theorem Appendix A.4. Rule (RSets) is sound.

Proof. Let $\rho \in \mathsf{Env}$, $\chi \in \mathsf{PEnv}$, $n \in \mathbb{N}$ be such that $(\rho, \chi, n{-}1) \in [\![\Gamma]\!]^{\mathrm{sp}}$. We must prove $\rho, \chi \vDash^n \{P\}\,\text{`}\mathsf{run}\ e\text{'}\,\{Q\}$. If $n = 0$ then this is trivially true, so let $n > 0$. Define ρ' as follows:

$$\rho'(\mathrm{v}) \quad \triangleq \quad \begin{cases} [\![e_i]\!]^{\mathrm{ex}}_{s,\rho} & \text{if v is } \mathrm{x}_i \\ \rho(\mathrm{v}) & \text{otherwise} \end{cases}$$

By this, the freshness of $\mathrm{x}_1, \ldots, \mathrm{x}_k$ and $(\rho, \chi, n{-}1) \in [\![\Gamma]\!]^{\mathrm{sp}}$ we have $(\rho', \chi, n{-}1) \in [\![\Gamma]\!]^{\mathrm{sp}}$. From this, the rule's second premise and Remark (2.2) we have

$$\rho', \chi \vDash^{n-1} \{P \wedge e_1{=}\mathrm{x}_1 \wedge \cdots \wedge e_k{=}\mathrm{x}_k\}\,\mathrm{P}(\cdot, \mathrm{x}_1, \ldots, \mathrm{x}_k)\,\{Q\} \tag{A.5}$$

Let $s \in [\![P]\!]^{\mathrm{as}}_{\rho,\chi}$ and s' be such that $(\mathsf{run}\ e, s) \xrightarrow{*} (\mathsf{nop}, s')$ in n steps or fewer; we are required to show $s' \in [\![Q]\!]^{\mathrm{as}}_{\rho,\chi}$. From $s \in [\![P]\!]^{\mathrm{as}}_{\rho,\chi}$ and the definition of ρ' it follows that

$$s \in [\![P \wedge e_1{=}\mathrm{x}_1 \wedge \cdots \wedge e_k{=}\mathrm{x}_k]\!]^{\mathrm{as}}_{\rho',\chi} \tag{A.6}$$

Define $C \triangleq \mathscr{G}^{-1}([\![e]\!]^{\mathrm{ex}}_s)$. From $s \in [\![P]\!]^{\mathrm{as}}_{\rho,\chi}$ and the premise $P \Rightarrow \mathrm{P}(e, e_1, \ldots, e_k)$ we have $s \in [\![\mathrm{P}(e, e_1, \ldots, e_k)]\!]^{\mathrm{as}}_{\rho,\chi}$, which means that

$$([\![e]\!]^{\mathrm{ex}}_s, [\![e_1]\!]^{\mathrm{ex}}_{s,\rho}, \ldots, [\![e_1]\!]^{\mathrm{ex}}_{s,\rho})) \in \chi(\mathrm{P})$$

By this and the definition of ρ' we have

$$([\![e]\!]^{\mathrm{ex}}_s, [\![\mathrm{x}_1]\!]^{\mathrm{ex}}_{\rho'}, \ldots, [\![\mathrm{x}_1]\!]^{\mathrm{ex}}_{\rho'})) \in \chi(\mathrm{P})$$

From this and $C = \mathscr{G}^{-1}([\![e]\!]^{\mathrm{ex}}_s)$ it follows that

$$C \in [\![\mathrm{P}(\cdot, \mathrm{x}_1, \ldots, \mathrm{x}_k)]\!]^{\mathrm{mid}}_{\rho',\chi} \tag{A.7}$$

Due to the structure of the transition relation $\rightarrow$, we must have $(C, s) \xrightarrow{*} (\mathsf{nop}, s')$ in $n{-}1$ steps or fewer. Combining this with (A.5), (A.6) and (A.7) we find that $s' \in [\![Q]\!]^{\mathrm{as}}_{\rho',\chi}$. Because $\mathrm{x}_1, \ldots, \mathrm{x}_k$ do not appear in Q, we have $s' \in [\![Q]\!]^{\mathrm{as}}_{\rho,\chi}$ as required. □

Appendix B. Proof for Π_1

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

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

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

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

and

$$T(\mathrm{q}) \vdash \left\{ \begin{array}{l} f = \mathrm{p} \\ \wedge\, g = \mathrm{q} \\ \wedge\, ((x \times x) + (y \times y) = (z \times z)) \neq 1 \end{array} \right\} \quad \text{`run } g\text{'} \quad \{\ R(x,y,z)\ \} \tag{B.2}$$

(B.1) 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 (B.2) by the (W) rule from the following instance of (H):

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

□