About Hoare Logics for Higher-order Store*

Bernhard Reus[1]** and Thomas Streicher[2]

[1] University of Sussex, Brighton BN1 9QH, UK
[2] TU Darmstadt, 64298 Darmstadt, Germany

Abstract. We present a Hoare logic for a simple imperative while-language with stored commands, ie. stored parameterless procedures. Stores that may contain procedures are called higher-order. Soundness of our logic is established by using denotational rather than operational semantics. The former is employed to elegantly account for an inherent difficulty of higher-order store, namely that assertions necessarily describe recursive predicates on a recursive domain. In order to obtain proof rules for mutually recursive procedures, assertions have to explicitly refer to the code of the procedures.

1 Introduction and Motivation

Hoare logic for imperative languages has been invented in the late 60es [7] and since then extended in many directions (for a survey see e.g. [4]). Procedures are a typical example. For a simple while language with parameterless recursive procedures it is common to apply the following rule (see [4]) for a procedure p declared with body C:

$$(proc)\quad \frac{\{P\}\,p\,\{Q\} \vdash \{P\}\,C\,\{Q\}}{\{P\}\,p\,\{Q\}}$$

In order to verify the effect of a procedure call, one has to show that the procedure body satisfies the very same effect under the assumption that the call already does so. Semantically, this corresponds to a form of fixpoint induction where admissibility of the semantical predicates is guaranteed automatically as store is modeled by a flat domain. Thus, in rule (*proc*) the $\{P\}\,p\,\{Q\}$ in the conclusion refers to the fixpoint of the definition rec $p \Leftarrow C[p]$ whereas on the left hand side in the premise it refers to an arbitrary implementation of p.

The situation changes dramatically if one allows stored procedures, ie. if procedures are kept in the store – in the same way as basic data like numbers – and called by their (variable) name. For example, $\mathsf{run}\,x$ invokes the procedure stored in variable x. The semantics of programs, being state transformers, now

* Both authors have been partially supported by APPSEM II (Applied Semantics), a thematic network funded by the IST programme of the European Union, IST-2001-38957.

** The first author has been partially supported by the EPSRC under grant GR/R65190/01, "Programming Logics for Denotations of Recursive Objects".

becomes implicitly higher-order, as they depend on the (code in the) store which contains such transformers itself. For this reason such stores are sometimes called "higher-order" or even recursive.

Landin had already observed[1] that in such situations one is able to tie "knots through the store". Put differently, recursion through the store becomes available such that additional fixpoint operators are obsolete. Consider e.g. the example $x := C; \mathsf{run}\ x$ which first stores a command C in x and then runs this command. But C itself may contain a command $\mathsf{run}\, x$, in which case we obtain a recursive procedure. In traditional semantics (see [9, 14, 6]) the semantics of a procedure is a fixpoint. This is fine as long as newly added code can only call the old procedures and not vice-versa. In object-oriented languages, however, new subclasses can change the semantics of the old classes (that is the whole point of object-orientation) and the traditional semantics cannot cope with that. For languages with higher-order store this is no problem, as recursion is through the store and not by fixpoint.

To the best of our knowledge, there is no Hoare-calculus for partial correctness of (even simple) imperative languages with higher-order store in the literature. However, in [8] a calculus for *total* correctness of programs with higher-order store has been presented recently where soundness is based on induction on a termination measure. The semantics does not make use of domain theory and does not seem to be easily extendible to partial correctness.

Several (fully-abstract) models using games (or abstract versions of games) have been developed but they focus on observational equivalence, e.g. [3, 10, 11].

A Hoare-like calculus for an object-based language, Abadi and Cardelli's imperative object calculus [1], has been suggested in [2]. In that language, simple field values and method closures are kept together in the same store. Hence, the store is higher-order. In [2], the program logic does not use Hoare-triples but specifications that refer to the state before and after method execution. Consequently, in [2], method specifications can only use static information about other methods and thus cannot cope with callbacks or dynamic loading where specifications may change at runtime. Method update had to be disallowed. Note that in this paper our stored procedures can be updated.

In [16, 17], we have presented a denotational technique to understand and model such object logics. This has been extended to a complete analysis of the entire Abadi-Leino calculus in [15]. Separately, in [5], Calcagno and O'Hearn set out to put ideas of separation logic [12] into a program logics for objects in a traditional Hoare-triple style but had problems with the object introduction rule.

In this paper we present a simple imperative language (Sect. 2) with higher-order store but without objects, and an assertion language (Sect. 3). We present some new proof rules (Sect. 4), give examples (Sect. 5) and prove soundness (Sect. 6). We finish with an outlook where the results of this paper may lead us and how related work could be helpful.

[1] as Peter O'Hearn pointed out to the first author.

Table 1. BNF syntax of $\mathcal{L}$

$x \in \mathsf{Var}$			variable
$e \in \mathsf{Exp}$	$::=$	x	variable expression
	$\mid$	k	numbers and other constants
	$\mid$	$e \circ_2 e$	binary operators
	$\mid$	$\circ_1 e$	unary operators
	$\mid$	$`s'$	quote (command as expression)
$s \in \mathsf{Com}$	$::=$	nop	no op
	$\mid$	$x := e$	assignment
	$\mid$	$s; s$	sequential composition
	$\mid$	$\mathsf{if}\, e\, \mathsf{then}\, s\, \mathsf{else}\, s$	conditional
	$\mid$	$\mathsf{run}\ x$	unquote (run the command in x)

The language in use is arguably the simplest language that uses higher-order store. It is thus an ideal candidate to investigate the problems caused by higher-order store in isolation. Using denotational semantics we will discover where exactly the difficulties of higher-order store are rooted.

2 The Programming Language

Syntax First we define the programming language syntax of our language, called $\mathcal{L}$. Let Var be the set of (countable) program variables, Exp the side effect free expressions, and Com the statements (commands). A BNF grammar for $\mathcal{L}$ is presented in Table 1.

The simplest non-terminating loop can be written as $x := `\mathsf{run}\, x'; \mathsf{run}\, x$.

Semantics The semantics is developed in a category of cpo-s and partial continuous maps (predomains). For any (pre-)domain there is, as usual, a partial order $\sqsubseteq$, and for a partial continuous function $f \in A \rightharpoonup B$ and $a \in A$, we write $f(a)\downarrow$ to state that the application is defined.

Let BVal be the set of basic first-order values like numbers or booleans ordered discretely. Values and stores are defined by the following system of (pre-)domain equations

$$\mathsf{Val} = \mathsf{BVal} + [\mathsf{St} \rightharpoonup \mathsf{St}] \qquad \mathsf{St} = \mathsf{Val}^{\mathsf{Var}}$$

Stores in St map variables into values in Val. The fact that state transformers can be values reflects the fact that the store is higher-order. Note that for a store σ, a variable x, and a value a we write $\sigma[x \mapsto a]$ for the map σ' defined as

$$\sigma'(y) = \begin{cases} a & \text{if } y \equiv x \\ \sigma(y) & \text{otherwise} \end{cases}$$

The equations for higher-order store can thus be rewritten in one equation as follows:

$$\mathsf{St} = (\mathsf{BVal} + [\mathsf{St} \rightharpoonup \mathsf{St}])^{\mathsf{Var}}$$

Table 2. Semantics of $\mathcal{L}$

$[\![x]\!]^e\ \sigma$	$= \sigma(x)$
$[\![`s']\!]^e\ \sigma$	$= [\![s]\!]$
$[\![\mathsf{k}]\!]^e\ \sigma$	$= k$
$[\![e_1 \circ_2 e_2]\!]^e\ \sigma$	$= \circ_2([\![e_1]\!]^e\ \sigma, [\![e_2]\!]^e\ \sigma)$
$[\![\circ_1\ e_1]\!]^e\ \sigma$	$= \circ_1([\![e_1]\!]^e\ \sigma)$
$[\![\mathsf{nop}]\!]\ \sigma$	$= \sigma$
$[\![x := e]\!]\ \sigma$	$= \sigma[x \mapsto [\![e]\!]^e\ \sigma]$
$[\![s_1; s_2]\!]\ \sigma$	$= [\![s_2]\!]([\![s_1]\!]\ \sigma)$
$[\![\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2]\!]\ \sigma$	$= \begin{cases} [\![s_1]\!]\sigma & \text{if } [\![e]\!]^e\sigma = \mathit{true} \\ [\![s_2]\!]\sigma & \text{if } [\![e]\!]^e\sigma = \mathit{false} \\ \text{undefined otherwise} \end{cases}$
$[\![\mathsf{run}\ x]\!]\ \sigma$	$= \sigma(x)(\sigma)$

or equivalently, by setting $\mathsf{Cl} = [\mathsf{St} \rightharpoonup \mathsf{St}]$,

$$\mathsf{Cl} = [(\mathsf{BVal}+\mathsf{Cl})^{\mathsf{Var}} \rightharpoonup (\mathsf{BVal}+\mathsf{Cl})^{\mathsf{Var}}]$$

The mixed-variant functor for which Cl is the solution is given by its object and morphism part below:

$$F(X,Y) = [(\mathsf{BVal}+X)^{\mathsf{Var}} \rightharpoonup (\mathsf{BVal}+Y)^{\mathsf{Var}}]$$

If $e : X \rightharpoonup Y$ then let $\hat{e} = (\mathsf{BVal} + e)^{\mathsf{Var}} : (\mathsf{BVal} + X)^{\mathsf{Var}} \rightharpoonup (\mathsf{BVal} + Y)^{\mathsf{Var}}$. More precisely, for a store $\sigma \in (\mathsf{BVal} + X)^{\mathsf{Var}}$, $\hat{e}(\sigma)$ is defined as follows:

$$\hat{e}(\sigma)(x) = \begin{cases} \sigma(x) & \text{if } \sigma(x) \in \mathsf{BVal} \\ e(\sigma(x)) & \text{if } \sigma(x) \in X \end{cases}$$

Now we can define the morphism part:

$$F(e,f) = \lambda h : F(X,Y).\ \hat{f} \circ h \circ \hat{e}.$$

For $e \in [\mathsf{Cl} \rightharpoonup \mathsf{Cl}]$ let $e^A \in [\mathsf{Cl}^A \rightharpoonup \mathsf{Cl}^A]$ be defined by $e^A(h)(a) = e(h(a))$. We can interpret $\mathcal{L}$ using an interpretation function for expressions $[\![\_]\!]^e : \mathsf{Exp} \to [\mathsf{St} \rightharpoonup \mathsf{Val}]$ and commands $[\![\_]\!] : \mathsf{Com} \to [\mathsf{St} \rightharpoonup \mathsf{St}]$ as presented in Table 2.

The last equation $[\![\mathtt{run}\, x]\!]\sigma = \sigma(x)(\sigma)$ is reminiscent of the self-application semantics of method call in OO-languages.

3 The Assertion Language

The assertion language is based on the assertions of the classic Hoare-calculus with the difference, though, that expressions can also refer to stored procedures.

Table 3. Syntax of Assertions

$n, p \in \mathsf{AuxVar}$		
$\tau \in \mathsf{Type}$	$::= \mathsf{bool} \mid \mathsf{int} \mid \mathsf{com}$	types
$e \in \mathsf{Exp}$	$::= n \mid p$	auxiliary variables in BVal and Cl
	$\mid x \mid \mathsf{k} \mid e \circ_2 e \mid \circ_1 e \mid$'s'	
$P \in \mathsf{Asrt}$	$::= \mathit{false}$	falsity
	$\mid P \wedge P$	conjunction
	$\mid \neg P$	negation
	$\mid \forall n.\, P$	universal quantification
	$\mid \tau ?\, e$	type check
	$\mid e \leq_\tau e$	comparison

Table 4. Semantics of Assertions

$$
\begin{array}{ll}
(\!|\mathit{false}|\!)\,\eta & = \emptyset \\
(\!|P \wedge Q|\!)\,\eta & = (\!|P|\!)\,\eta \cap (\!|Q|\!)\,\eta \\
(\!|\forall n.\, P|\!)\,\eta & = \cap_{v \in \mathsf{Val}} (\!|P|\!)\,\eta[n \mapsto v] \\
(\!|\tau ?\, e|\!)\,\eta & = \{\, \sigma \mid [\![e]\!]^e\,\eta\,\sigma \in [\![\tau]\!] \,\} \text{ where } [\![\mathsf{com}]\!] = \mathsf{Cl},\ [\![\mathsf{int}]\!] = \mathbb{Z},\ [\![\mathsf{bool}]\!] = \mathbb{B} \\
(\!|\neg P|\!)\ \ \eta & = \{\, \sigma \mid \sigma \notin (\!|P|\!)\,\eta \,\} \\
(\!|e_1 \leq_\tau e_2|\!)\,\eta & = \{\, \sigma \mid\ [\![e_1]\!]^e\,\eta\,\sigma \in [\![\tau]\!] \wedge [\![e_2]\!]^e\,\eta\,\sigma \in [\![\tau]\!]\ \wedge\ [\![e_1]\!]^e\,\eta\,\sigma \sqsubseteq [\![e_2]\!]^e\,\eta\,\sigma \,\}
\end{array}
$$

Syntax The syntax of assertions is presented in Table 3. They may contain expressions of $\mathcal{L}$ which have no side effects. Note that $e = e$ and $\phi \vee \phi$ can be expressed using $\leq$, and $\wedge$ and $\neg$, respectively.

As already known from classic Hoare-calculus one needs "ghost variables" (also called *auxiliary variables*) to be able to refer to values in the pre-execution state. For example, in the Hoare-triple $\{x = n\}\, fac\, \{x = n!\}$ we have a program variable x, and an auxiliary variable n. The countable set of auxiliary ghost variables is called AuxVar. Throughout the paper we use x, y, z to denote instances of program variables in Var and n, p, q to denote instances of auxiliary variables in AuxVar where n is usually used for auxiliary variables of basic type and p and q for auxiliary variables for commands (procedures). It is important not to confuse those different types of variables.

Semantics The denotational semantics of assertions is standard but we have to take care of auxiliary variables. For those variables an additional environment is in use of type $\mathsf{Env} = \mathsf{Val}^{\mathsf{AuxVar}}$. Correspondingly, an interpretation function for assertions must have type $(\!|\_|\!) : \mathsf{Asrt} \to \mathsf{Env} \to \mathcal{P}(\mathsf{St})$ and its equations can be found in Table 4.

Observe that $[\![\_]\!]^e$ has to be extended to auxiliary variables. Therefore, we stipulate $[\![n]\!]^e \eta\, \sigma = \eta(n)$ and assume that the definitional equations for $[\![\_]\!]^e$ in Table 2 have been changed accordingly.

A particular subset of downward-closed assertions Equality between procedures ($\leq_{\mathsf{com}}$) is problematic as it is not a downward-closed predicate which will be

Table 5. Syntax of pure assertions without undesired comparisons of commands

$\phi, \in \mathsf{BAsrt}$	$::= \mathit{false} \mid \phi \wedge \phi \mid \neg\phi \mid \forall n.\, \phi \mid \tau?\, e \mid e \leq_{\mathsf{nat}} e \mid e \leq_{\mathsf{bool}} e$
$P \in \mathsf{DClAsrt}$	$::= \phi \mid \forall n.\, P \mid x \leq_{\mathsf{com}} n \mid x \leq_{\mathsf{com}} \text{`}s\text{'} \mid \text{`}s\text{'} \leq_{\mathsf{com}} \text{`}s\text{'} \mid P \wedge P \mid P \vee P$

important for our semantics. We therefore identify a particular sublanguage of assertions, DClAsrt, which are more restrictive with respect to comparisons of commands, in particular they do not admit equality on procedures. To define DClAsrt, we introduce "basic assertions" BAsrt first, which do not use any comparison between expressions of type com at all. The exact definitions can be found in Table 5.

Assertions like $x =_{\mathsf{com}} \text{`}s\text{'}$, i.e. $x \leq_{\mathsf{com}} \text{`}s\text{'} \wedge \text{`}s\text{'} \leq_{\mathsf{com}} x$, are not in DClAsrt as $\text{`}s\text{'} \leq_{\mathrm{com}} x$ is *not* in DClAsrt. The assertions in DClAsrt all satisfy three conditions explained in Lemma 1 below. These properties will turn out to be crucial to obtain a semantics that validates the proof rules introduced in the next section.

Lemma 1. *For any assertion* $P \in \mathsf{DClAsrt}$ *its semantics* $(\!|P|\!) \in \mathsf{Env} \to \mathcal{P}(\mathsf{St})$ *has the following properties:*

1. $(\!|P|\!)\,\eta$ *is a downward closed predicate for all* $\eta \in \mathsf{Env}$.
2. $(\!|P|\!)$ *is monotonic in its (procedure environment) argument.*
3. $(\!|P|\!)\,\eta\,\sigma$ *implies* $(\!|P|\!)\,\hat{e}(\eta)\;\hat{e}(\sigma)$ *for all* $e \in \mathsf{Cl} \rightharpoonup \mathsf{Cl}$, $\sigma \in \mathsf{St}$, *and* $\eta \in \mathsf{Env}$ *where* $\hat{e}(\eta)$ *is defined analogously to* $\hat{e}(\sigma)$ *with the only difference that the variables used in* Env *are* AuxVar *whereas those used in* St *are* Var.

Proof. We only have to consider assertions on commands. Assertions on basic types trivially fulfill the requirements since BVal is ordered discretely. Since $\forall$, $\wedge$, and $\vee$ preserve the conditions above we only need to show each of them for the three comparisons on commands (and the type check assertion com?e which trivially fulfills all conditions). (1) and (2) are immediate by definition of the assertions. For (3) we show the interesting case: $[\![x \leq_{\mathsf{com}} n]\!]\,\eta\,\sigma$ iff $\sigma(x) \sqsubseteq \eta(n)$ (†). Now $[\![x \leq_{\mathsf{com}} n]\!]\,\hat{e}(\eta)\,\hat{e}(\sigma)$ iff $\hat{e}(\sigma)(x) = e(\sigma(x)) \sqsubseteq \hat{e}(\eta)(n) = e(\eta(n))$ which follows from (†) as e is monotonic.

4 Proof Rules

First of all, the standard rules for assignment (A), composition (S), conditional (I), weakening (W), and no operation (ϵ) are in use as presented in Fig. 1.

New rules are needed to deal with stored procedures as outlined in Fig. 2. The run-rule (R) is canonical for non-recursive procedure calls. Rule (H) is like (R) for cases where the code in a variables is not known but described by an auxiliary variable. Finally, the recursion rule (μ) is used for stored procedures that are (mutually) recursive. This is necessary as rule (R) is not able to get rid of the circular reference to the procedure. Rule (μ) is able to do just that analogously to the standard procedure rule (*proc*) mentioned in the introduction.

$$(A)\ \frac{}{\{P[e/x]\}\, x := e\, \{P\}} \qquad (S)\ \frac{\{P\}\, C_1\, \{R\} \quad \{R\}\, C_2\, \{Q\}}{\{P\}\, C_1; C_2\, \{Q\}}$$

$$(I)\ \frac{\{b \wedge P\}\, C_1\, \{Q\} \quad \{\neg b \wedge P\}\, C_2\, \{Q\}}{\{P\}\, \mathsf{if}\ b\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\, \{Q\}} \qquad (W)\ \frac{P \Rightarrow P' \quad \{P'\}\, C\, \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\, C\, \{Q\}}$$

$$(\epsilon)\ \frac{}{\{P\}\, \mathsf{nop}\, \{P\}}$$

Fig. 1. Standard Rules

$$(R)\ \frac{\{P \wedge x \leq \text{`}C\text{'}\}\, C\, \{Q\}}{\{P \wedge x \leq \text{`}C\text{'}\}\, \mathsf{run}\, x\, \{Q\}} \qquad Q \in \mathsf{DClAsrt}$$

$$(H)\ \frac{}{\{P \wedge x \leq p\}\, p\, \{Q\} \vdash \{P \wedge x \leq p\}\, \mathsf{run}\, x\, \{Q\}} \qquad Q \in \mathsf{DClAsrt}$$

$$(\mu)\ \frac{\bigwedge_{1 \leq i \leq n} \ \{P_1\}\, p_1\, \{Q_1\} \ldots \{P_n\}\, p_n\, \{Q_n\} \vdash \{P_i\}\, C_i\, \{Q_i\}}{\bigwedge_{1 \leq i \leq n} \ \{P_i[\boldsymbol{C}/\boldsymbol{p}]\}\, C_i\, \{Q_i[\boldsymbol{C}/\boldsymbol{p}]\}} \qquad \forall 1 \leq i \leq n.\ P_i, Q_i \in \mathsf{DClAsrt}$$

Fig. 2. New Rules for Stored Procedures

In fact, by using first (R) and then (μ), one obtains the derived rule stated below (left). For comparison the standard recursive procedure rule is repeated next to it (right).

$$\frac{\{P \wedge x \leq p\}\, p\, \{Q\} \vdash \{P \wedge x \leq p\}\, C\, \{Q\}}{\{P \wedge x \leq \text{`}C\text{'}\}\, \mathsf{run}\, x\, \{Q\}} \qquad (proc)\ \frac{\begin{array}{c}\mathrm{rec}\ p \Leftarrow C[p] \\ \{P\}\, p\, \{Q\} \quad \vdash \{P\}\, C\, \{Q\}\end{array}}{\{P\}\, p\, \{Q\}}$$

Whereas for $(proc)$ the definition of the procedure p is separate, for stored procedure x one needs to use an auxiliary variable p to denote the content of x during execution of its body which may change x.

Note that throughout the rest of the paper we simply write $\leq$ instead of $\leq_{\mathrm{com}}$ when one of its arguments is obviously of type com. The necessity of the side conditions for (μ), (R), and (H) will become clear when we discuss the soundness of these rules.

5 Sample Derivations

We present some sample derivations to demonstrate how the proof rules above are to be used.

Example 2. A derivation for a specification of our introductory example of a non-terminating loop, $\{true\}\, x := \text{`}\mathsf{run}\ x\text{'}\, ;\, \mathsf{run}\, x\, \{false\}$, is outlined in Figure 3.

Example 3. Because of recursion through the store we can simulate a while loop $\mathsf{while}\ B\ \mathsf{do}\ C\ \mathsf{od}$ as $z := \text{`}\mathsf{if}\ B\ \mathsf{then}\ C; \mathsf{run}\, z\, \mathsf{else}\, \mathsf{nop}\text{'}\ ;\ \mathsf{run}\, z$. Of course, program variable z is not supposed to occur in C. When doing the proof it becomes clear

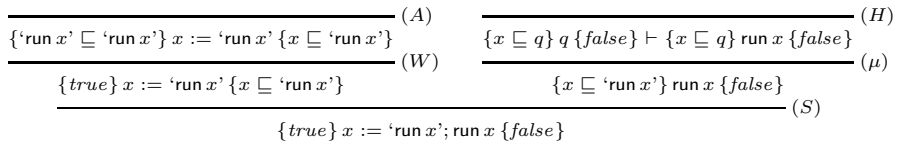


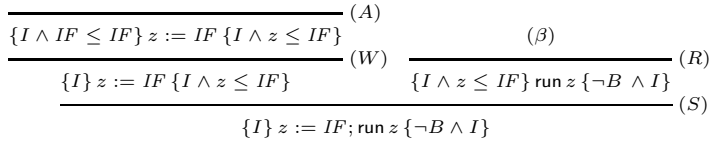
Fig. 3. Derivation for Example 2

that it is enough that z is not altered by C. The standard rule for while and its derived equivalent for the encoding in $\mathcal{L}$ read as follows:

$$\frac{\{B \wedge I\}\, C\, \{I\}}{\{I\}\, \mathsf{while}\, B\, \mathsf{do}\, C\, \mathsf{od}\, \{\neg B \wedge I\}} \qquad \frac{\{B \wedge I \wedge z \leq p\}\, C\, \{I \wedge z \leq p\}}{\{I\}\, \mathsf{while}\, B\, \mathsf{do}\, C\, \mathsf{od}\, \{\neg B \wedge I\}}$$

The encoded form has to state that z is invariant, i.e. that the content of this cell is not changed by the body of the while statement. This is expressed using an auxiliary variable $p \in \mathsf{AuxVar}$.

For the proof assume that $(*)$ $\{B \wedge I \wedge z \leq p\}\, C\, \{I \wedge z \leq p\}$ and let IF abbreviate the expression (of command type) ‘$\mathsf{if}\ B\ \mathsf{then}\ C; \mathsf{run}\ z\ \mathsf{else}\ \mathsf{nop}$’. The first part of the derivation is straightforward (see Fig. 4). The remaining open goal $\{I \wedge z \leq IF\}\, \mathsf{run}\, z\, \{\neg B \wedge I\}$ is then derived in (β) in the second prooftree of Fig. 4. The application of the recursion rule (μ) will introduce the hypothesis $(\dagger)$ $\{I \wedge z \leq p\}\, p\, \{\neg B \wedge I\}$ to be used at the top of subtree (β).



where (β) is the following prooftree

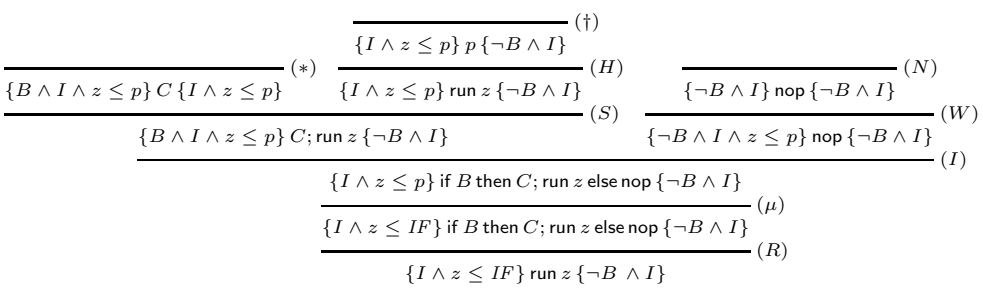


Fig. 4. Derivation for Example 3

Table 6. Semantics of Triples

$(\eta,\sigma) \models \{P\}\, C\, \{Q\}$	$\Leftrightarrow \forall \sigma' \in \mathsf{St}.\ (\!|P|\!)\,\eta\,\sigma \wedge [\![C]\!]\,\sigma = \sigma' \Rightarrow (\!|Q|\!)\,\eta\,\sigma'$
$(\eta,\sigma) \models \{P\}\, p\, \{Q\}$	$\Leftrightarrow \forall \sigma' \in \mathsf{St}.\ (\!|P|\!)\,\eta\,\sigma \wedge \eta(p)(\sigma) = \sigma' \Rightarrow (\!|Q|\!)\,\eta\,\sigma'$
$\models \{P_1\}\, p_1\, \{Q_1\}, \ldots, \{P_n\}\, p_n\, \{Q_n\}$	$\Leftrightarrow \forall \eta \in \mathsf{Env}. (\forall \sigma \in \mathsf{St}.\ \bigwedge_{1\le i\le n} (\eta,\sigma) \models \{P_i\}\, p_i\, \{Q_i\})$
$\vdash \{P\}\, C\, \{Q\}$	$\Rightarrow \forall \sigma \in \mathsf{St}.\ (\eta,\sigma) \models \{P\}\, C\, \{Q\}$

Example 4. This example shows how procedures can modify themselves so that different invocations of x behave differently. Let

$$S \equiv \text{`}\mathsf{if}\ z{=}0\ \mathsf{then\ nop\ else}\ (y := y + z; z := z{-}1; \mathsf{run}\ x)\text{'}$$

and consider the program $x := \text{`}y := y{+}1; x := S\text{'}; \mathsf{run}\ x; \mathsf{run}\ x$ for which we can derive the following annotations:

$\{y = n \wedge z = m\}$
$x := \text{`}y := y{+}1; x := S\text{'}$ (A),(W)
$\{y = n \wedge z = m \wedge x \le \text{`}y := y{+}1; x := S\text{'}\}$
$\mathsf{run}\ x$ (R),(A),(S),(W)
$\{y = n{+}1 \wedge z = m \wedge x \le S\}$
$\mathsf{run}\ x$ (R),(μ),(I),(S),(A),(W),(ϵ)
$\{y = n{+}1 + \sum_1^m i \wedge z = 0 \wedge x \le S\}$

6 Soundness

Pre-condition P and post-condition Q of Hoare triples do not only depend on the store but also on the values for the auxiliary variables in environment Env. We write $(\eta,\sigma) \models \{P\}\, C\, \{Q\}$ meaning that $\{P\}\, C\, \{Q\}$ is valid in σ and η.

Definition 5. The semantics of Hoare triples for commands, for closure variables, and for commands in context, resp., is given in Table 6. Note that triples express *partial* correctness.

Now we are in a position to formally prove the soundness of the new rules.[2]

Theorem 6. *The (run-)rules (R) and (H) are sound.*

Proof. Let C be a command. Assume that (1) $\models \{P \wedge x \le \text{`}C\text{'}\}\, C\, \{Q\}$. We have to show $\models \{P \wedge x \le \text{`}C\text{'}\}\, \mathsf{run}\, x\, \{Q\}$. Therefore, assume that $\eta \in \mathsf{Env}$, and $\sigma \in \mathsf{St}$ such that (2) $(\!|P \wedge x \le \text{`}C\text{'}|\!)\,\eta\,\sigma$ and that (3) $[\![\mathsf{run}\ x]\!]\sigma = \sigma'$. It remains to show that $(\!|Q|\!)\,\eta\,\sigma'$. Define (4) $\sigma'' := [\![C]\!]\,\sigma$. From (1), (2) and (4) we obtain $(\!|Q|\!)\,\eta\,\sigma''$. Since $(\!|Q|\!)$ is downward-closed by Lemma 1(2) and $Q \in \mathsf{DClAsrt}$, it suffices to prove that $\sigma' \sqsubseteq \sigma''$. But

$$\sigma' =_{(3)} [\![\mathsf{run}\ x]\!]\sigma = \sigma(x)(\sigma) \sqsubseteq_{(2)} [\![C]\!]\,\sigma =_{(4)} \sigma'' \quad .$$

The proof for (H) is carried out analogously.

[2] Soundness of the old rules is standard.

In the next proof we will make use of a binary operation $+ : \mathsf{Env} \times \mathsf{Cl}^A \to \mathsf{Env}$ that represents "overwriting of environments" where $A \subseteq \mathsf{AuxVar}$. Accordingly, $(\eta{+}\delta)(n) = \eta(n)$ if $n \in A$ and $\delta(n)$ otherwise.

Theorem 7. *The rule (μ) is correct.*

Proof. Let $(\!|P_i|\!), (\!|Q_i|\!) \subseteq \mathsf{Env} \times \mathsf{St}$ for $1{\leq}i{\leq}n$ be the denotations of the predicates in the assertions of (μ). We basically follow the ideas of [16] where similar arguments were used to prove the correctness of the object introduction rule of [2]. For arbitrary $\eta \in \mathsf{Env}$ let $A_\eta \subseteq \mathsf{Cl}^{\{p_1,\ldots,p_n\}} \times \mathsf{Cl}^{\{p_1,\ldots,p_n\}}$ be defined as follows:

$$\begin{aligned} A_\eta(\psi,\phi) \equiv \forall 1{\leq}j{\leq}n.\ \forall \sigma{\in}\mathsf{St}.\\ (\!|P_j|\!)\,(\eta{+}\psi)\,\sigma \wedge \phi(p_j)(\sigma){\downarrow} \Rightarrow (\!|Q_j|\!)\,(\eta{+}\psi)\ (\phi(p_j)(\sigma)) \end{aligned}$$

Let $p_i \mapsto [\![C_i]\!]$ be the environment in $\mathsf{Cl}^{\{p_1,\ldots,p_n\}}$ that assigns $[\![C_i]\!]$ to p_i for $1{\leq}i{\leq}n$. Then verifying rule (μ) amounts to showing that for arbitrary $\eta \in \mathsf{Env}$:

$$\begin{aligned} (\dagger) \qquad \forall \phi{\in}\mathsf{Cl}^{\{p_1,\ldots,p_n\}}.\ A_\eta(\phi,\phi) \Rightarrow A_\eta(\phi, p_i \mapsto [\![C_i]\!])\\ \text{implies}\\ A_\eta(p_i \mapsto [\![C_i]\!], p_i \mapsto [\![C_i]\!]) \end{aligned}$$

since in general $\forall \phi{\in}\mathsf{Env}.\, R\,\phi$ is equivalent to $\forall \phi{\in}\mathsf{Env}.\, \forall \psi{\in}\mathsf{Cl}^{\{p_1,\ldots,p_n\}}.\, R\,(\phi{+}\psi)$. Let S be a predicate on $\mathsf{Cl}^{\{p_1,\ldots,p_n\}}$ such that for all $\phi \in \mathsf{Cl}^{\{p_1,\ldots,p_n\}}$

$$(1) \qquad S(\phi) \iff \forall \psi{\in}\mathsf{Cl}^{\{p_1,\ldots,p_n\}}.\ S(\psi) \Rightarrow A_\eta(\psi,\phi)$$

from which it follows that for any ϕ

$$(2) \qquad S(\phi) \Rightarrow A_\eta(\phi,\phi).$$

Now from (2) and assumption (†) it follows that

$$(3) \qquad S(\phi) \Rightarrow A_\eta(\phi, p_i \mapsto [\![C_i]\!])$$

i.e. $S(p_i \mapsto [\![C_i]\!])$ due to (1) as ϕ was arbitrary such that by (1) again we obtain $A_\eta(p_i \mapsto [\![C_i]\!], p_i \mapsto [\![C_i]\!])$ as desired. The existence of an appropriate S in (1) is shown in Lemma 8.

Lemma 8. *There is a $S \subseteq \mathsf{Env}$ such that $S(\phi) \iff \forall \rho{\in}\mathsf{Env}.\ S(\rho) \Rightarrow A_\eta(\rho,\phi)$ for any $\eta \in \mathsf{Env}$.*

Proof. Let $\mathcal{L}$ denote the admissible subsets of $\mathsf{Cl}^{\{p_1,\ldots,p_n\}}$ ordered by $\subseteq$ and

$$\Phi : \mathcal{L}^{\mathsf{op}} \to \mathcal{L} : S \mapsto \{\, \phi{\in}\mathsf{Cl}^{\{p_1,\ldots,p_n\}} \mid \forall \rho{\in}\mathsf{Cl}^{\{p_1,\ldots,p_n\}}.\ S(\rho) \Rightarrow A_\eta(\rho,\phi) \,\}$$

for which it is necessary that $A_\eta(\rho,-)$ specifies an admissible subset of $\mathsf{Cl}^{\{p_1,\ldots,p_n\}}$ for all $\rho \in \mathsf{Cl}^{\{p_1,\ldots,p_n\}}$ which follows from Lemma 1(1). For guaranteeing existence (and uniqueness) of such an S by Pitts' Theorem [13] we have to show that $e : S_1 \subseteq S_2$ implies $F(e,e) : \Phi(S_2) \subseteq \Phi(S_1)$ for all (idempotent) $e \sqsubseteq \mathsf{id}_{\mathsf{Cl}}$ where for $X, Y \subseteq \mathsf{Cl}^{\{p_1,\ldots,p_n\}}$ one defines $e : X \subseteq Y$ iff $\forall x{\in}X.\, \hat{e}(x){\downarrow} \Rightarrow \hat{e}(x){\in}Y$. Suppose $e \sqsubseteq \mathsf{id}_{\mathsf{Cl}}$ with $e : S_1 \subseteq S_2$ and $\Phi(S_2)(\phi)$. We have to show $\Phi(S_1)(\widehat{F(e,e)}(\phi))$. For this to be possible we need that $(\!|P_j|\!)$ and $(\!|Q_j|\!)$ satisfy all three properties of Lemma 1.

7 Loose Ends and Related Work

Our programming language $\mathcal{L}$ contains only constant command expressions. Self-modifying programs would also modify commands, in other words, use operations on commands.

Consider, for example, an operation $\mathsf{make\_com}$ that concatenates two commands returning a new one, such that modifications like $x := \mathsf{make\_com}(x, y)$ become possible. To allow for such expressions one needs to axiomatise $\mathsf{make\_com}$ algebraically, i.e. the underlying first-order logic on data types needs to be enriched by axioms like $\mathsf{make\_com}(\text{`}s_1\text{'}, \text{`}s_2\text{'}) = \text{`}s_1; s_2\text{'}$ and a monotonicity axiom for $\mathsf{make\_com}$ w.r.t. $\leq_{\mathsf{com}}$.

Procedures with parameters have not been discussed here but are certainly an issue. We expect, however, that they are not more difficult to cope with than in ordinary Hoare-calculus with recursive procedures. In particular, if parameters are passed by value.

The logic presented is not modular as all code must be known in advance and must be carried around in assertions. In [5], a calculus for objects was suggested that uses nested Hoare triples in order to state properties of programs in the store rather than referring to their explicit code without a hint on a denotational semantics nor a soundness proof. The meaning of such a "specification" triple in our setting would be a recursively defined predicate:

$$\begin{aligned}(\sigma, \eta) \models \{P\}\, x\, \{Q\} \Leftrightarrow \forall \sigma', \sigma'' \in \mathsf{St}.\\ (\sigma', \eta) \models \{P\}\, x\, \{Q\} \wedge (\!|P|\!)\, \sigma' \wedge \sigma(x)(\sigma') = \sigma'' \Rightarrow (\!|Q|\!)\, \sigma''\end{aligned}$$

Unfortunately, such a recursively defined semantics of triples, $\models$, does not allow for a separate specification for mutual recursive procedures (there is no "Bekic Lemma"). This suggests that it is highly unlikely that there is a modular logic for $\mathcal{L}$. On the other hand, in [2, 15] it has been shown that modularity can be achieved for Abadi and Leino's object logic in a setting where assignments to procedure variables are disallowed. It is unclear yet, to which extent this restriction can be relaxed. In [17] we have discussed an idea for a relaxation but it will have to be tested on a fully-fledged soundness proof where frame properties are to be shown.

Alternative models for higher-order store have been suggested by [3, 10, 11] using games or locally boolean domains. It might be worthwhile to consider the domain equation for CI within such locally boolean domains (or equivalently the category of sequential algorithms). This would already provide us with a fully abstract model for higher-order store in the sense of Idealized Algol.

Another open question is relative completeness of our logic. It is no problem to express weakest liberal preconditions, at least when extending the assertion language such that it can express arbitrary inductively defined predicates. The hard problem is to show, by induction on S, that $\{\mathrm{wlp}(C, P)\}\, C\, \{P\}$ is derivable for all post-conditions P.

The results of this paper, however, are the first step to get a grip on higher-order store from a programming logic point of view.

Acknowledgements We would like to thank Cristiano Calcagno, Peter O'Hearn, and Jan Schwinghammer for discussions and comments on this work. We are grateful to the anonymous referees for their suggestions to improve readability.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer Verlag, 1996.
2. M. Abadi and K. R. M. Leino. A logic of object-oriented programs. In Nachum Dershowitz, editor, *Verification: Theory and Practice*, pages 11–41. Springer, 2004.
3. S. Abramsky, K. Honda, and G. McCusker. A fully abstract game semantics for general references. In *LICS '98: Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science*, page 334. IEEE Computer Society, 1998.
4. K.R. Apt. Ten Years of Hoare's Logic: A Survey – Part I. *TOPLAS*, 3(4):431–483, 1981.
5. C. Calcagno and P.W. O'Hearn. A logic for objects. Slides of a Talk, 2001.
6. A.V. Hense. Wrapper semantics of an object-oriented programming language with state. In *Proceedings Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 548–568, Berlin, 1991. Springer.
7. C.A.R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12:576–583, 1969.
8. K. Honda, N. Yoshida, and M. Berger. An observationally complete program logic for imperative higher-order functions. In *20th Symp. on Logics in Computer Science, LICS*. To appear in IEEE, 2005.
9. S.N. Kamin and U.S. Reddy. Two semantic models of object-oriented languages. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 464–495. The MIT Press, 1994.
10. J. Laird. A categorical semantics of higher-order store. *Electronic notes in Theoretical Computer Science*, 69, 2002.
11. J. Laird. Locally boolean domains. 2004. Submitted.
12. P.W. O'Hearn, J.C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, volume 2142 of *Logic in Computer Science*, pages 1–19, Berlin, 2001. Springer.
13. A. M. Pitts. Relational properties of domains. *Information and Computation*, 127:66–90, 1996.
14. B. Reus. Modular semantics and logics of classes. In *Computer Science Logic*, volume 2803 of *Lecture Notes in Computer Science*, pages 456–469, Berlin, 2003. Springer.
15. B. Reus and J. Schwinghammer. Denotational semantics for Abadi and Leino's logic of objects. In Mooly Sagiv, editor, *European Symposium on Programming*, volume 3444 of *Lecture Notes in Computer Science*, pages 263–278, Berlin, 2005. Springer.
16. B. Reus and Th. Streicher. Semantics and logics of objects. In *Proceedings of the 17th Symp. Logic in Computer Science*, pages 113–122, 2002.
17. B. Reus and Th. Streicher. Semantics and logic of object calculi. *TCS*, 316:191–213, 2004.