Reverse Hoare Logic*

Edsko de Vries and Vasileios Koutavas

Trinity College Dublin, Ireland
{Edsko.de.Vries,Vasileios.Koutavas}@cs.tcd.ie

Abstract. We present a novel Hoare-style logic, called Reverse Hoare Logic, which can be used to reason about state reachability of imperative programs. This enables us to give natural specifications to randomized (deterministic or nondeterministic) algorithms. We give a proof system for the logic and use this to give simple formal proofs for a number of illustrative examples. We define a weakest postcondition calculus and use this to show that the proof system is sound and complete.

1 Introduction

Hoare Logic [12] is a popular method for proving properties of imperative programs. The following specification for sort is a classical example:

$$\{\top\}\ \mathsf{sort}(\mathtt{a})\ \{\mathrm{sorted}(\mathtt{a}) \wedge \mathtt{a} \in \Pi(\mathtt{old}(\mathtt{a}))\} \tag{1}$$

The empty *precondition* of this *Hoare triple* says that sort makes no assumptions about its input; the *postcondition* says that array a will be sorted and a permutation ($\Pi(\mathtt{old}(\mathtt{a}))$) of the original ("old") value of a after sort terminates.

For other algorithms, especially randomized ones, it is not so clear what the right specification is. For instance, consider an algorithm to shuffle the elements of an array. Certainly, shuffle should generate a permutation of the array, but

$$\{\top\}\ \mathsf{shuffle}(\mathtt{a})\ \{\mathtt{a} \in \Pi(\mathtt{old}(a))\} \tag{2}$$

is an incomplete specification of shuffle at best. In fact, clearly sort satisfies specification (2) too, but for most purposes sort would be a badly behaved implementation of shuffle! A better specification would require that shuffle can generate *all* permutations. If we allow so-called *logic variables* in the triples we might give the Hoare triple (schema)

$$\{\alpha \in \Pi(\mathtt{a})\}\ \mathsf{shuffle}(\mathtt{a})\ \{\mathtt{a} = \alpha\} \tag{3}$$

Unfortunately, although an abstract implementation of shuffle based on nondeterministic choice ($\sqcup$) such as

$$\bigsqcup_{\alpha \in \Pi(\mathtt{a})} \mathtt{a} := \alpha \tag{4}$$

satisfies specification (3), real implementations of shuffle that rely on a pseudo-random number generator do not: the permutation they generate will be dictated by the state of the random number generator.

* This research was supported by SFI project SFI 06 IN.1 1898.

We can model a random number generator as a stream of random numbers available through some global variable $\mathtt{r}$. This suggests a Hoare triple of the form

$$\{\alpha \in \Pi(\mathtt{a}), \mathtt{r} = ?\}\ \mathsf{shuffle}(\mathtt{a})\ \{\mathtt{a} = \alpha\} \tag{5}$$

but it is unclear what we should put at the location of (?). We would like to state that every permutation can be generated by *some* (unspecified) choice of random number stream, but unfortunately we are forced to be more precise than that. Choose some enumeration of permutations, and let $\Pi_\iota(\alpha)$ be the ιth permutation of α. Then we could give the following specification:

$$\{\alpha = \Pi_\iota(\mathtt{a}), \mathtt{r} = \rho_\iota\}\ \mathsf{shuffle}(\mathtt{a})\ \{\mathtt{a} = \alpha\} \tag{6}$$

Specification (6) says that if shuffle is executed in an initial state with random number stream ρ_ι, then it will generate the ιth permutation. This is however rather unsatisfactory. We cannot define ρ without detailed knowledge of the algorithm; an alternative shuffling program would require a different mapping ρ' and would not satisfy the above specification. In fact, we would prefer not to have to mention $\mathtt{r}$ in the specification at all so that we do not have to adjust the specification when we refine an abstract, non-deterministic, implementation of shuffle to a real one that relies on some global state.

The main problem is that Hoare Logic uses a universal quantification over *initial* states ("for all initial states satisfying the precondition..."), while shuffle is more naturally specified using a universal quantification over *final* states ("for all permutations..."). This is precisely the purpose of Reverse Hoare Logic, in which we can give the following specification for shuffle:

$$\langle \mathtt{a} = \alpha \rangle\ \mathsf{shuffle}(\mathtt{a})\ \langle \mathtt{a} \in \Pi(\alpha) \rangle \tag{7}$$

or, using an operator "$\mathtt{new}$" dual to "$\mathtt{old}$", without logic variables:

$$\langle \mathtt{new}(\mathtt{a}) \in \Pi(\mathtt{a}) \rangle\ \mathsf{shuffle}(\mathtt{a})\ \langle \top \rangle \tag{8}$$

The *reverse triple* (7) is satisfied when all final states in which $\mathtt{a}$ is a permutation of α are reachable by executing shuffle in some initial state in which $\mathtt{a}$ has value α. It states precisely that shuffle can generate all permutations, exposes none of the implementation so that we can give this specification without reference to the algorithm, and can be used for many algorithms, regardless of whether they use random number streams or non-determinism.

Reverse triples are statements about the reachability of all "good" final states but say nothing about "bad" states. They are essentially dual to Hoare triples, which are statements about "bad" states not being reachable but do not guarantee that "good" states are reachable. This duality between the logics can also be observed in certain proof rules, such as the rule for consequence, where implications are reversed, and in the proof of completeness, which requires the definition of a *weakest postcondition*, rather than a weakest precondition, calculus. We believe that the two logics complement each other and their combination can express a complete specification for shuffle.

Since we are interested in reachability, we are interested in the *existence* of paths: a triple $\langle P \rangle\ c\ \langle Q \rangle$ is satisfied when for each final state σ' satisfying Q there is a state σ satisfying P such program c, when started in state σ, *can* terminate in state σ'. In this sense, Reverse Hoare Logic is a total logic, although the above triple does not require that c never diverges.

We make the following contributions in this paper:

1. We define Reverse Hoare Logic, a logic in which we can naturally express reachability specifications for imperative programs. Equivalent specifications in Hoare Logic would be less abstract and more difficult to define.
2. We give a proof system (Sect. 3) which can be used to prove the validity of reverse triples, without appealing to the underlying model of the logic. Some of the rules are familiar from Hoare Logic but others are different in subtle and sometimes surprising ways; for instance, the rule for loops requires the loop variant to *increase* rather than decrease.
3. We show the usefulness of the proof system by using it to derive admissible rules for complex commands, and use it to give an elegant proof for the specification of shuffle (Sect. 4).
4. We show that the proof system is sound and complete (Sect. 6). The completeness proof is based on a *weakest postcondition* calculus, which is also useful in showing the invalidity of reverse triples (Sect. 5).

2 Definitions

We use a standard imperative language with local definitions and choice ($\sqcup$); its big-step semantics is given in Fig. 1 as a mapping from states to states. Throughout this paper we will use $(\sigma, x \mapsto n)$ to denote the *extension* of σ with variable x (i.e., $x \notin \operatorname{dom} \sigma$) and $(\sigma \dagger x \mapsto n)$ to denote the extension *or* update of σ. We assume the Barendregt convention: all bound variables are assumed to be different to all other variables, and we identify programs up to alpha-renaming.

We let e range over arithmetic expressions, n over natural numbers, and let b range over boolean expressions. We use $[\![e]\!]_\sigma$ to denote the evaluation of expression e in state σ; we leave the exact definition of the syntax for arithmetic expressions and their evaluation relation open.

We use a standard first-order infinitary assertion language based on $L_{\omega_1,\omega}$ with the satisfaction relation shown in Fig. 1. To make the technical development smoother we will allow for logic variables, ranged over by ι, in expressions in the assertion language. We use e to range over these "extended" expressions too, as the intended meaning will be clear from the context. The value of these variables is given by an *interpretation* I, a mapping from logic variables to values. We can define existentials and universals in the assertion language as syntactic sugar:

$$\forall \ell \in L \cdot P \stackrel{\text{def}}{=} \bigwedge_{\ell \in L} P \qquad \exists \ell \in L \cdot P \stackrel{\text{def}}{=} \bigvee_{\ell \in L} P \tag{9}$$

Likewise, we will use $P \wedge Q$ and $P \vee Q$ to denote binary conjunctions and disjunctions, respectively. A formula P is *valid* iff $\sigma \vDash^I P$ for all states σ and interpretations I.

Operational Semantics

$$\frac{}{\sigma \xrightarrow{\texttt{skip}} \sigma} \qquad \frac{}{\sigma \xrightarrow{x:=e} \sigma \dagger x \mapsto [\![e]\!]_\sigma}$$

$$\frac{(\sigma, x \mapsto [\![e]\!]_\sigma) \xrightarrow{c} (\sigma', x \mapsto n')}{\sigma \xrightarrow{\texttt{local } x=e \texttt{ in } c} \sigma'} \qquad \frac{\sigma \xrightarrow{c_0} \sigma' \qquad \sigma' \xrightarrow{c_1} \sigma''}{\sigma \xrightarrow{c_0;c_1} \sigma''}$$

$$\frac{\sigma \vDash b \qquad \sigma \xrightarrow{c_0} \sigma'}{\sigma \xrightarrow{\texttt{if } b \texttt{ then } c_0 \texttt{ else } c_1} \sigma'} \qquad \frac{\sigma \vDash \neg b \qquad \sigma \xrightarrow{c_1} \sigma'}{\sigma \xrightarrow{\texttt{if } b \texttt{ then } c_0 \texttt{ else } c_1} \sigma'}$$

$$\frac{\sigma \vDash \neg b}{\sigma \xrightarrow{\texttt{while } b \texttt{ do } c} \sigma} \qquad \frac{\sigma \vDash b \qquad \sigma \xrightarrow{c} \sigma' \qquad \sigma' \xrightarrow{\texttt{while } b \texttt{ do } c} \sigma''}{\sigma \xrightarrow{\texttt{while } b \texttt{ do } c} \sigma''}$$

$$\frac{\sigma \xrightarrow{c_0} \sigma'}{\sigma \xrightarrow{c_0 \sqcup c_1} \sigma'} \qquad \frac{\sigma \xrightarrow{c_1} \sigma'}{\sigma \xrightarrow{c_0 \sqcup c_1} \sigma'}$$

Satisfaction Relation

$$\begin{array}{ll}
\sigma \vDash^I \top & \\
\sigma \vDash^I (e_0 = e_1) & \text{if } [\![e_0]\!]_{\sigma,I} = [\![e_1]\!]_{\sigma,I} \\
\sigma \vDash^I (e_0 \leq e_1) & \text{if } [\![e_0]\!]_{\sigma,I} \leq [\![e_1]\!]_{\sigma,I} \\
\sigma \vDash^I \neg P & \text{if not } \sigma \vDash^I P \\
\sigma \vDash^I P \Rightarrow Q & \text{if } \sigma \vDash^I P \text{ implies } \sigma \vDash^I Q
\end{array}$$

$$\begin{array}{ll}
\sigma \vDash^I \bigwedge_{\ell \in L} P_\ell & \text{if } \sigma \vDash^I P_\ell \text{ for all } \ell \in L \\
\sigma \vDash^I \bigvee_{\ell \in L} P_\ell & \text{if } \sigma \vDash^I P_\ell \text{ for some } \ell \in L
\end{array}$$

Fig. 1. Operational Semantics and Satisfaction of Assertions

Definition 1 (Reverse Hoare validity). *We write* $\vDash^I \langle P \rangle\ c\ \langle Q \rangle$ *iff*

$$\forall \sigma' \vDash^I Q \cdot \exists \sigma \vDash^I P \cdot \sigma \xrightarrow{c} \sigma'$$

We write $\vDash \langle P \rangle\ c\ \langle Q \rangle$ *iff* $\vDash^I \langle P \rangle\ c\ \langle Q \rangle$*, for all* I*.*

3 Program Logic

In order to abstract away from states and interpretations, we introduce a program logic, shown in Fig. 2, which gives proof rules for each of the constructs in the language. We will show in Sect. 6 that these rules are sound and complete; in this section, we explain the rules and give examples.

Rules Skip and Assn are familiar from Hoare Logic. We can think of the existential ι in the postcondition as the old value of x. Here is a simple example:

$$\dfrac{\dfrac{}{\left\langle \mathtt{x} = \iota' \right\rangle\ \mathtt{x} := \mathtt{x} + 1\ \left\langle \exists \iota \in \mathbb{Z} \cdot \iota = \iota' \wedge \mathtt{x} = \iota + 1 \right\rangle}\ \textsc{Assn}}{\left\langle \mathtt{x} = \iota' \right\rangle\ \mathtt{x} := \mathtt{x} + 1\ \left\langle \mathtt{x} = \iota' + 1 \right\rangle}\ \textsc{Con} \tag{10}$$

We use rule Con to bring the postcondition into the right form; note the direction of the implications! This is a consequence of the underlying semantics of reverse Hoare triples.

$$\frac{}{\langle P\rangle\ x := e\ \langle \exists \iota \in L \cdot P(^{\iota}/_{x}) \wedge x = e(^{\iota}/_{x})\rangle}\ \textsc{Assn}\ \text{(where } L \text{ is the type of } x\text{)}$$

$$\frac{\langle P \wedge x = e\rangle\ c\ \langle Q\rangle \qquad x \notin \operatorname{fn} P}{\langle P\rangle\ \texttt{local}\ x = e\ \texttt{in}\ c\ \langle \exists x \in L \cdot Q\rangle}\ \textsc{Local}\ \text{(where } L \text{ is the type of } x\text{)}$$

$$\frac{\langle P_\iota \wedge b\rangle\ c\ \langle P_{\iota+1}\rangle}{\langle P_0\rangle\ \texttt{while}\ b\ \texttt{do}\ c\ \langle \neg b \wedge \exists \iota \in \mathbb{N} \cdot P_\iota\rangle}\ \textsc{While}\ (\iota \text{ fresh})$$

$$\frac{}{\langle P\rangle\ \texttt{skip}\ \langle P\rangle}\ \textsc{Skip} \qquad \frac{\langle P\rangle\ c_0\ \langle Q\rangle \qquad \langle Q\rangle\ c_1\ \langle R\rangle}{\langle P\rangle\ c_0; c_1\ \langle R\rangle}\ \textsc{Seq}$$

$$\frac{\langle P \wedge b\rangle\ c_0\ \langle Q\rangle}{\langle P\rangle\ \texttt{if}\ b\ \texttt{then}\ c_0\ \texttt{else}\ c_1\ \langle Q\rangle}\ \textsc{Then} \qquad \frac{\langle P \wedge \neg b\rangle\ c_1\ \langle Q\rangle}{\langle P\rangle\ \texttt{if}\ b\ \texttt{then}\ c_0\ \texttt{else}\ c_1\ \langle Q\rangle}\ \textsc{Else}$$

$$\frac{\langle P\rangle\ c_0\ \langle Q\rangle}{\langle P\rangle\ c_0 \sqcup c_1\ \langle Q\rangle}\ \textsc{Left} \qquad \frac{\langle P\rangle\ c_1\ \langle Q\rangle}{\langle P\rangle\ c_0 \sqcup c_1\ \langle Q\rangle}\ \textsc{Right}$$

$$\frac{P' \Rightarrow P \quad \langle P'\rangle\ c\ \langle Q'\rangle \quad Q \Rightarrow Q'}{\langle P\rangle\ c\ \langle Q\rangle}\ \textsc{Con} \qquad \frac{\left(\forall \ell \in L \cdot \langle P\rangle\ c\ \langle Q_\ell\rangle\right)}{\langle P\rangle\ c\ \left\langle \bigvee_{\ell \in L} Q_\ell \right\rangle}\ \textsc{Split}$$

$$\frac{\langle P\rangle\ c\ \langle Q\rangle}{\langle P \wedge R\rangle\ c\ \langle Q \wedge R\rangle}\ \textsc{Frame}\ \text{(no variable occurring free in } R \text{ is modified by } c\text{)}$$

Fig. 2. Reverse Hoare Rules

In Hoare Logic, $\{P(^{e}/_{x})\}\ x := e\ \{P\}$ is a popular alternative rule for assignment. The naive translation of this rule to $\langle P(^{e}/_{x})\rangle\ x := e\ \langle P\rangle$ is neither sound nor complete; for instance, it allows us to derive $\langle \top\rangle\ \mathtt{x} := 2\ \langle \top\rangle$, which is invalid: it says that any state at all (including one where $\mathtt{x} \neq 2$) is reachable by executing $\mathtt{x} := 2$, which is clearly not true. On the other hand, $\langle \top\rangle\ \mathtt{x} := \mathtt{y}\ \langle \mathtt{x} = 2\rangle$ cannot be derived using this rule, while this is a valid reverse Hoare triple.

The rule for the introduction of a local variable x hides x from the postcondition and requires that x must have the specified initial value in the initial state. Here is an example:

$$\frac{\dfrac{\dfrac{}{\langle \mathtt{x} = \mathtt{y}\rangle\ \mathtt{z} := \mathtt{x}\ \langle \exists \iota \in \mathbb{Z} \cdot \mathtt{x} = \mathtt{y} \wedge \mathtt{z} = \mathtt{x}\rangle}\ \textsc{Assn}}{\langle \mathtt{x} = \mathtt{y}\rangle\ \mathtt{z} := \mathtt{x}\ \langle \mathtt{x} = \mathtt{y} \wedge \mathtt{z} = \mathtt{y}\rangle}\ \textsc{Con}}{\dfrac{\langle \top\rangle\ \texttt{local}\ \mathtt{x} = \mathtt{y}\ \texttt{in}\ \mathtt{z} := \mathtt{x}\ \langle \exists \mathtt{x} \in \mathbb{Z} \cdot \mathtt{x} = \mathtt{y} \wedge \mathtt{z} = \mathtt{y}\rangle}\ \textsc{Local}}{\langle \top\rangle\ \texttt{local}\ \mathtt{x} = \mathtt{y}\ \texttt{in}\ \mathtt{z} := \mathtt{x}\ \langle \mathtt{z} = \mathtt{y}\rangle}\ \textsc{Con} \tag{11}$$

Rule Local is particularly useful in our setting because all global variables assigned to by the program need to be mentioned in the postcondition. For example, the triple

$$\langle \top \rangle \ \mathtt{x} := \mathtt{y}; \mathtt{z} := \mathtt{x} \ \langle \mathtt{z} = \mathtt{y} \rangle \tag{12}$$

is *not* valid, since final states in which $\mathtt{x} \neq \mathtt{y}$ are not reachable by executing $\mathtt{x} := \mathtt{y}; \mathtt{z} := \mathtt{x}$.

The rule for sequential composition is as expected; the rule for conditions is more interesting. Rule Then can be used if all states that satisfy the postcondition can be reached by executing the true-branch of the conditional; rule Else is the analogous rule for the false-branch of the conditional. Typically, rule Split will first be used to partition the set of final states into those that can be reached by the true-branch and those that can be reached by the false-branch. For example:

$$\dfrac{\dfrac{\dfrac{\vdots}{\langle b \rangle \ \mathtt{x} := 1 \ \langle \mathtt{x} = 1 \rangle}}{\langle \top \rangle \ \ldots \ \langle \mathtt{x} = 1 \rangle}\ \textsc{Then} \quad \dfrac{\dfrac{\vdots}{\langle \neg b \rangle \ \mathtt{x} := 2 \ \langle \mathtt{x} = 2 \rangle}}{\langle \top \rangle \ \ldots \ \langle \mathtt{x} = 2 \rangle}\ \textsc{Else}}{\langle \top \rangle \ \mathtt{if}\ b\ \mathtt{then}\ \mathtt{x} := 1\ \mathtt{else}\ \mathtt{x} := 2 \ \langle \mathtt{x} = 1 \vee \mathtt{x} = 2 \rangle}\ \textsc{Split} \tag{13}$$

The rules for non-deterministic choice are similar to the rules for conditionals. It remains to explain the rule for the loop construct. Like in standard Hoare logics for total correctness [1] we have to provide a *loop variant* P, a predicate over natural numbers. Unlike in standard Hoare logics, however, we have to show that P holds for a smaller number *before* the loop body. Consider the following example:

$$\langle \mathtt{x} = 0 \rangle \ \mathtt{while}\ \mathtt{b}\ \mathtt{do}\ \mathtt{x} := \mathtt{x} + 1 \sqcup \mathtt{b} := \bot \ \langle \neg \mathtt{b} \rangle \tag{14}$$

Intuitively, specification (14) is valid: any state which satisfies $\neg\mathtt{b}$, in particular, any state where $\mathtt{x}$ has some value n, can be reached from a state in which $\mathtt{x} = 0$ (pick a state where $\mathtt{b} = \top$) by executing $\mathtt{x} := \mathtt{x} + 1$ the first n iterations through the loop, followed by one iteration executing $\mathtt{b} := \bot$. We can prove the validity of this program by picking the loop variant[1]

$$\phi_n \stackrel{\text{def}}{=} (\mathtt{x} = n \wedge \mathtt{b}) \vee (\mathtt{x} = n - 1 \wedge \neg \mathtt{b}) \tag{15}$$

The proof is given in Fig. 3.

4 Case Studies

In this section we prove admissible proof rules for complex commands using the program logic and use them when we formally prove that shuffle can generate any permutation of the input array. The proof rules are summarized in Fig. 4.

[1] To simplify the example we assume $0 - 1 = 0$ for natural numbers.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\langle \mathtt{x} = \iota \wedge \mathtt{b}\rangle\ \mathtt{x} := \mathtt{x}+1\ \langle \mathtt{x} = \iota + 1 \wedge \mathtt{b}\rangle}\,\textsc{Left}}{\langle \mathtt{x} = \iota \wedge \mathtt{b}\rangle \ldots \langle \mathtt{x} = \iota + 1 \wedge \mathtt{b}\rangle}\,\textsc{Con}}{\langle \phi_\iota \wedge \mathtt{b}\rangle \ldots \langle \mathtt{x} = \iota + 1 \wedge \mathtt{b}\rangle} \quad \dfrac{\dfrac{\dfrac{\vdots}{\langle \mathtt{x} = \iota \wedge \mathtt{b}\rangle\ \mathtt{b} := \bot\ \langle \mathtt{x} = \iota \wedge \neg\mathtt{b}\rangle}\,\textsc{Right}}{\langle \mathtt{x} = \iota \wedge \mathtt{b}\rangle \ldots \langle \mathtt{x} = \iota \wedge \neg\mathtt{b}\rangle}\,\textsc{Con}}{\langle \phi_\iota \wedge \mathtt{b}\rangle \ldots \langle \mathtt{x} = \iota \wedge \neg\mathtt{b}\rangle}\,\textsc{Split}}{\dfrac{\langle \phi_\iota \wedge \mathtt{b}\rangle\ \mathtt{x} := \mathtt{x}+1 \sqcup \mathtt{b} := \bot\ \langle \phi_{\iota+1}\rangle}{\langle \phi_0\rangle \ldots \langle \neg\mathtt{b} \wedge \exists \iota \in \mathbb{N} \cdot \phi_\iota\rangle}\,\textsc{While}}\,\textsc{Con}}{\langle \mathtt{x} = 0\rangle\ \mathtt{while\ b\ do\ x} := \mathtt{x}+1 \sqcup \mathtt{b} := \bot\ \langle \neg\mathtt{b}\rangle}\,\textsc{Con}$$

Fig. 3. Loop Example (ϕ_n defined in equation (15))

4.1 Picking random numbers

Assuming the availability of a stream of random numbers $n_1 : n_2 : \ldots$ through global variable $\mathtt{r}$, picking a random number boils down to

$$\Big(x := \mathtt{rnd}[0, e]\Big) \stackrel{\text{def}}{=} \Big(x := \mathrm{head}(\mathtt{r}); \mathtt{r} := \mathrm{tail}(\mathtt{r})\Big) \tag{16}$$

The proof that this satisfies the expected specification is an easy exercise. Let Q be the post-condition $\exists \iota \cdot P(^{\iota}/_{x}) \wedge e \leq x \leq e'$. Observe that

$$\dfrac{\dfrac{\dfrac{}{\left\langle \begin{array}{l} P \\ \wedge\ \exists \rho, \iota \cdot e \leq \iota \leq e' \\ \wedge\ \mathtt{r} = \iota : \rho \end{array} \right\rangle \ldots \left\langle \begin{array}{l} \exists \iota' \cdot P(^{\iota'}/_{x}) \\ \wedge\ \exists \rho, \iota \cdot e \leq \iota \leq e' \\ \wedge\ \mathtt{r} = \iota : \rho \\ \wedge\ x = \mathrm{head}(\mathtt{r}) \end{array} \right\rangle}\,\textsc{Assn}}{\langle P \wedge \exists \rho, \iota \cdot e \leq \iota \leq e' \wedge \mathtt{r} = \iota : \rho\rangle \ldots \langle Q \wedge \exists \rho \cdot \mathtt{r} = x : \rho\rangle}\,\textsc{Con}}{\langle P\rangle\ x := \mathrm{head}(\mathtt{r})\ \langle Q \wedge \exists \rho \cdot \mathtt{r} = x : \rho\rangle}\,\textsc{Con} \tag{17}$$

Moreover,

$$\dfrac{\dfrac{\dfrac{}{\langle Q \wedge \exists \rho \cdot \mathtt{r} = x : \rho\rangle\ \mathtt{r} := \mathrm{tail}(\mathtt{r})\ \langle \exists \rho' \cdot Q \wedge \exists \rho \cdot \rho' = x : \rho \wedge \mathtt{r} = \mathrm{tail}(\rho')\rangle}\,\textsc{Assn}}{\langle Q \wedge \exists \rho \cdot \mathtt{r} = x : \rho\rangle\ \mathtt{r} := \mathrm{tail}(\mathtt{r})\ \langle Q \wedge \exists \rho \cdot \mathtt{r} = \rho\rangle}\,\textsc{Con}}{\langle Q \wedge \exists \rho \cdot \mathtt{r} = x : \rho\rangle\ \mathtt{r} := \mathrm{tail}(\mathtt{r})\ \langle Q\rangle}\,\textsc{Con} \tag{18}$$

The required specification follows by combining (17) and (18) using Seq. We leave the generalization to $x := \mathtt{rnd}[e, e']$ as an (easy) exercise for the reader.

4.2 Arrays

We deal with arrays by adopting the approach taken in Hoare Logic [13]: we consider an expression language that contains expressions $a[e]$ and $a \dagger e \mapsto e'$ for array indexing and override, respectively.

We define two derived commands to update the value of an array at a particular index and two swap to elements:

$$\Big(a[e] := e'\Big) \stackrel{\text{def}}{=} \Big(a := a \dagger e \mapsto e'\Big)$$
$$\Big(\texttt{swap}\ a[e, e']\Big) \stackrel{\text{def}}{=} \Big(\texttt{local}\ x = a[e]\ \texttt{in}\ a[e] := a[e']; a[e'] := x\Big)$$

The associated proof rules Upd and Swap are shown in Fig. 4. As syntactic conventions, we use a to range over array valued expressions, $\texttt{a}$ for array valued program variables, and α for array valued logic variables.

4.3 Iteration

We introduce a `for` loop as syntactic sugar; for simplicity, we fix the lower bound:

$$\Big(\texttt{for}\ x\ \texttt{in}\ [0, e)\ \texttt{do}\ c\Big) \stackrel{\text{def}}{=} \Big(\texttt{local}\ x = 0\ \texttt{in}\ \texttt{while}\ x < e\ \texttt{do}\ c; x := x + 1\Big) \quad (19)$$

Rule For is simpler than While, because termination is guaranteed; hence, we only need to provide a loop *invariant* P, rather than a loop variant.[2]

Proving For is a good exercise. Let $\phi_n \stackrel{\text{def}}{=} P \wedge x = n \leq e$. We derive the rule as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\langle P\rangle\ c\ \langle P(^{x+1}/_x)\rangle}{\langle P \wedge x = i < e\rangle\ c\ \langle P(^{x+1}/_x) \wedge x = i < e\rangle}\textsc{Frame} \quad \nabla}{\langle P \wedge x = i < e\rangle\ c; x := x + 1\ \langle P \wedge x = i + 1 \leq e\rangle}\textsc{Seq}}{\langle P \wedge x = i \leq e\rangle\ c; x := x + 1\ \langle P \wedge x = i + 1 \leq e\rangle}\textsc{Con}}{\langle \phi_0\rangle\ \texttt{while}\ x < e\ \texttt{do}\ \ldots\ \langle \neg(x < e) \wedge \exists i \cdot \phi_i\rangle}\textsc{While}}{\langle P \wedge x = 0\rangle\ \texttt{while}\ x < e\ \texttt{do}\ \ldots\ \langle \neg(x < e) \wedge x \leq e \wedge P\rangle}\textsc{Con}}{\langle P \wedge x = 0\rangle\ \texttt{while}\ x < e\ \texttt{do}\ \ldots\ \langle x = e \wedge P\rangle}\textsc{Con}}{\langle P(^0/_x)\rangle\ \texttt{local}\ x = 0\ \texttt{in}\ \ldots\ \langle \exists x \cdot x = e \wedge P\rangle}\textsc{Local}}{\langle P(^0/_x)\rangle\ \texttt{for}\ x\ \texttt{in}\ [0, e)\ \texttt{do}\ c\ \langle P(^e/_x)\rangle}\textsc{Con} \quad (20)$$

where ∇ is a derivation with conclusion

$$\big\langle P(^{x+1}/_x) \wedge x = i < e\big\rangle\ x := x + 1\ \langle P \wedge x = i + 1 \leq e\rangle \quad (21)$$

4.4 Shuffle

We are now in a position to prove that shuffle can generate any permutation. We give the Fisher-Yates implementation of shuffle as follows:

$$\texttt{for x in } [0, |\texttt{a}|)\ \texttt{do local y} = 0\ \texttt{in y} := \texttt{rnd}[\texttt{x}, |\texttt{a}| - 1]; \texttt{swap a}[\texttt{x}, \texttt{y}] \quad (22)$$

[2] P truly is a loop invariant; although we require $P(^{x+1}/_x)$ after c, this means that P will be true again after x is incremented.

$$\frac{\langle P\rangle\ c\ \langle P({}^{x+1}\!/_{x})\rangle}{\langle P({}^{0}\!/_{x})\rangle\ \texttt{for}\ x\ \texttt{in}\ [0,e)\ \texttt{do}\ c\ \langle P({}^{e}\!/_{x})\rangle}\ \textsc{For}\quad \begin{pmatrix} c \text{ cannot change } x \text{ or any variable} \\ \text{appearing in } e \end{pmatrix}$$

$$\frac{x \notin e, e'}{\langle P\rangle\ x := \texttt{rnd}[e,e']\ \langle \exists \iota \cdot P({}^{\iota}\!/_{x}) \wedge e \le x \le e'\rangle}\ \textsc{Rnd}$$

$$\frac{a \notin e}{\langle P\rangle\ a[e] := e'\ \langle \exists \alpha \in [\mathbb{Z}] \cdot P({}^{\alpha}\!/_{a}) \wedge a = \alpha \dagger e \mapsto e'({}^{\alpha}\!/_{a})\rangle}\ \textsc{Upd}$$

$$\frac{a \notin e, e'}{\langle P\rangle\ \texttt{swap}\ a[e,e']\ \langle \exists \alpha \in [\mathbb{Z}] \cdot P({}^{\alpha}\!/_{a}) \wedge a = \alpha \dagger e \leftrightarrow e'\rangle}\ \textsc{Swap}$$

Fig. 4. Derived Proof Rules

In order to prove shuffle correct, we need a lemma that says that any permutation of an array α of size $|\alpha|$ can be generated by first swapping the first element with an element $0 \le m < |\alpha|$, then the second element with an element $1 \le m' < |\alpha|$, etc. Formally:

Lemma 1 (Permutations). *Define an indexed predicate π_n as follows.*

$$\begin{array}{lll} a'\ \pi_0\ a & \textit{iff} & a' = a \\ a'\ \pi_{n+1}\ a & \textit{iff} & \exists m \cdot a'\ \pi^m_{n+1}\ a \end{array}$$

where $a'\ \pi^m_{n+1}\ a = \exists a'' \cdot n \le m < |a| \wedge \Big(a' = a'' \dagger n \leftrightarrow m\Big) \wedge a''\ \pi_n\ a$.
Then $a' \in \Pi(a)$ *iff* $a'\ \pi_{|a|}\ a$.

We can use $\mathtt{a}\ \pi_{\mathtt{x}}\ \alpha$ as our loop invariant[3] and give the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\nabla_1 \qquad \nabla_2}{\langle \mathtt{a}\ \pi_{\mathtt{x}}\ \alpha \wedge \mathtt{y} = 0\rangle\ \ldots\ \left\langle \mathtt{a}\ \pi^{\mathtt{y}}_{\mathtt{x}+1}\ \alpha\right\rangle}\ \textsc{Seq}}{\langle \mathtt{a}\ \pi_{\mathtt{x}}\ \alpha\rangle\ \texttt{local}\ \mathtt{y} = 0\ \texttt{in}\ \ldots\ \left\langle \exists \mathtt{y} \cdot \mathtt{a}\ \pi^{\mathtt{y}}_{\mathtt{x}+1}\ \alpha\right\rangle}\ \textsc{Local}}{\langle \mathtt{a}\ \pi_{\mathtt{x}}\ \alpha\rangle\ \texttt{local}\ \mathtt{y} = 0\ \texttt{in}\ \ldots\ \langle \mathtt{a}\ \pi_{\mathtt{x}+1}\ \alpha\rangle}\ \textsc{Con}}{\left\langle (\mathtt{a}\ \pi_{\mathtt{x}}\ \alpha)({}^{0}\!/_{\mathtt{x}})\right\rangle\ \texttt{for}\ \mathtt{x}\ \texttt{in}\ [0,|\mathtt{a}|)\ \texttt{do}\ \ldots\ \left\langle (\mathtt{a}\ \pi_{\mathtt{x}}\ \alpha)({}^{|\mathtt{a}|}\!/_{\mathtt{x}})\right\rangle}\ \textsc{For}}{\langle \mathtt{a} = \alpha\rangle\ \mathsf{shuffle}(\mathtt{a})\ \langle \mathtt{a} \in \Pi(\alpha)\rangle}\ \textsc{Con}\ (\text{Lem. 1}) \tag{23}$$

where ∇_1 is a derivation with conclusion

$$\langle \mathtt{a}\ \pi_{\mathtt{x}}\ \alpha \wedge \mathtt{y} = 0\rangle\ \mathtt{y} := \texttt{rnd}[\mathtt{x}, |\mathtt{a}| - 1]\ \langle \mathtt{a}\ \pi_{\mathtt{x}}\ \alpha \wedge \mathtt{x} \le \mathtt{y} < |\mathtt{a}|\rangle \tag{24}$$

[3] Technically the subscript to π must be a natural number; we can regard this loop invariant as syntactic sugar for $\bigvee_{n \in \mathbb{N}} \mathtt{a}\ \pi_n\ \alpha \wedge x = n$.

which can be proved using Rnd and Con. Derivation ∇_2 is given by

$$\dfrac{\dfrac{}{\left\langle \mathtt{a}\ \pi_{\mathtt{x}}\ \alpha \wedge \mathtt{x} \leq \mathtt{y} < |\mathtt{a}| \right\rangle\ \mathtt{swap\ a[x,y]}\ \left\langle \begin{array}{l} \exists \alpha'.\ \alpha'\ \pi_{\mathtt{x}}\ \alpha \\ \quad \wedge\ (\mathtt{a} = \alpha' \dagger \mathtt{x} \leftrightarrow \mathtt{y}) \\ \quad \wedge\ \mathtt{x} \leq \mathtt{y} < |\mathtt{a}| \end{array} \right\rangle}\ \textsc{Swap}}{\langle \mathtt{a}\ \pi_{\mathtt{x}}\ \alpha \wedge \mathtt{x} \leq \mathtt{y} < |\mathtt{a}| \rangle\ \mathtt{swap\ a[x,y]}\ \left\langle \mathtt{a}\ \pi^{\mathtt{y}}_{\mathtt{x}+1}\ \alpha \right\rangle}\ \textsc{Con} \tag{25}$$

The difficulty of this proof is on par with the difficulty of proofs of comparable properties in Hoare Logic, for instance proving that sort is correct. As in Hoare Logic proofs, the key step in the proof is identifying the loop invariant.

5 Weakest Postcondition Calculus

In Sect. 3 we presented a program logic for deriving reverse Hoare triples. In this section we present a weakest-postcondition calculus, shown in Fig. 5, which computes $\mathrm{wpo}(P, c)$, the *weakest* postcondition given a precondition P and program c. We then have that $\langle P \rangle\ c\ \langle Q \rangle$ is a valid triple if and only if $Q \Rightarrow \mathrm{wpo}(P, c)$. As we shall see in Sect. 6, the calculus is also an essential ingredient in proving completeness of the program logic.

Moreover, a weakest postcondition calculus gives us a tool for proving that triples are *not* valid. For instance, in Sect. 3 we claimed that triple (12)

$$\langle \top \rangle\ \mathtt{x} := \mathtt{y}; \mathtt{z} := \mathtt{x}\ \langle \mathtt{z} = \mathtt{y} \rangle$$

was invalid. Using our calculus we can prove this formally. We compute

$$\begin{aligned} &\mathrm{wpo}(\top, \mathtt{x} := \mathtt{y}; \mathtt{z} := \mathtt{x}) = \mathrm{wpo}(\mathrm{wpo}(\top, \mathtt{x} := \mathtt{y}), \mathtt{z} := \mathtt{x}) \\ &= \mathrm{wpo}(\mathtt{x} = \mathtt{y}, \mathtt{z} := \mathtt{x}) = (\mathtt{x} = \mathtt{y} \wedge \mathtt{z} = \mathtt{x}) = (\mathtt{x} = \mathtt{y} \wedge \mathtt{z} = \mathtt{y}) \end{aligned}$$

Since $\mathtt{z} = \mathtt{y} \not\Rightarrow \mathtt{x} = \mathtt{y} \wedge \mathtt{z} = \mathtt{y}$ (it is easy to find a counterexample) we can conclude that this triple is indeed invalid, and the weakest postcondition tells us precisely what is missing: the condition on x.

6 Soundness and Completeness

We first show that any reverse Hoare triple derivable by the proof system is valid. We rely on two auxiliary lemmas, which we state without proof. The first is a standard substitution lemma:

Lemma 2. *$\sigma \vDash^I P({}^{n}/_{x})$ iff $(\sigma \dagger x \mapsto n) \vDash^I P$.*

The second states that predicates are preserved by programs that do not modify any of the variables mentioned in the predicate:

Lemma 3. *If $\sigma' \vDash^I P$, $\sigma \xrightarrow{c} \sigma'$ and no variable occurring free in P is modified by c, then $\sigma \vDash^I P$.*

$$
\begin{aligned}
\mathrm{wpo}(P, \texttt{skip}) &= P \\
\mathrm{wpo}(P, x := e) &= \exists \ell \in L \cdot P(^{\ell}/_{x}) \wedge x = e(^{\ell}/_{x}) \\
\mathrm{wpo}(P, c_0; c_1) &= \mathrm{wpo}(\mathrm{wpo}(P, c_0), c_1) \\
\mathrm{wpo}(P, \texttt{if } b \texttt{ then } c_0 \texttt{ else } c_1) &= \mathrm{wpo}(P \wedge b, c_0) \vee \mathrm{wpo}(P \wedge \neg b, c_1) \\
\mathrm{wpo}(P, c_0 \sqcup c_1) &= \mathrm{wpo}(P, c_0) \vee \mathrm{wpo}(P, c_1) \\
\mathrm{wpo}(P, \texttt{local } x = e \texttt{ in } c) &= \exists x \in L \cdot \mathrm{wpo}(P \wedge x = e, c) \\
\mathrm{wpo}(P, \texttt{while } b \texttt{ do } c) &= \neg b \wedge \bigvee_{n \in \mathbb{N}} \Upsilon_{b,c,P}(n)
\end{aligned}
$$

where

$$
\begin{aligned}
\Upsilon_{b,c,P}(0) &= P \\
\Upsilon_{b,c,P}(n+1) &= \mathrm{wpo}(\Upsilon_{b,c,P}(n) \wedge b, c)
\end{aligned}
$$

Fig. 5. Weakest Postcondition Calculus

We can now state and prove soundness.

Theorem 1 (Soundness). *If* $\vdash \langle P \rangle\ c\ \langle Q \rangle$ *then* $\vDash \langle P \rangle\ c\ \langle Q \rangle$.

Proof. By induction on the derivation of $\langle P \rangle\ c\ \langle Q \rangle$. The cases for Skip, Seq, Then, Else, Left, Right, Con and Split are straightforward. We give details for the other cases.

1. Case $\dfrac{}{\langle P \rangle\ x := e\ \left\langle \exists i \cdot P(^{i}/_{x}) \wedge x = e(^{i}/_{x}) \right\rangle}$ Assn.
   Pick an I, σ' such that $\sigma' \vDash^I \exists i \cdot P(^{i}/_{x}) \wedge x = e(^{i}/_{x})$ i.e. $\exists n \cdot \sigma' \vDash^I P(^{n}/_{x}) \wedge x = e(^{n}/_{x})$. Choose $\sigma = (\sigma' \dagger x \mapsto n)$. By Lem. 2 we have $\sigma' \vDash P(^{n}/_{x})$ iff $\sigma', x \mapsto n \vDash P$. Finally, since $\sigma \dagger x \mapsto [\![e]\!]_{\sigma} = (\sigma' \dagger x \mapsto n) \dagger x \mapsto [\![e]\!]_{\sigma' \dagger x \mapsto n} = \sigma' \dagger x \mapsto [\![e]\!]_{\sigma' \dagger x \mapsto n} = \sigma' \dagger x \mapsto [\![e(^{n}/_{x})]\!]_{\sigma'} = \sigma'$, we have $\sigma \xrightarrow{x:=e} \sigma'$.
2. Case $\dfrac{\langle P \wedge x = e \rangle\ c\ \langle Q \rangle \qquad x \notin \mathrm{fn}\, P}{\langle P \rangle\ \texttt{local } x = e \texttt{ in } c\ \langle \exists x \cdot Q \rangle}$ Local.
   Pick an I, σ' such that $\sigma' \vDash^I \exists x \cdot Q$. We have $\sigma' \vDash^I \exists x \cdot Q$ i.e. $\exists n' \cdot \sigma' \vDash^I Q(^{n'}/_{x})$ i.e. $\exists n' \cdot (\sigma', x \mapsto n') \vDash^I Q$. By the induction hypothesis there exist a σ_0 such that $\sigma_0 \xrightarrow{c} (\sigma', x \mapsto n')$ and $\sigma_0 \vDash^I P \wedge x = e$. Hence, there must be some $\sigma \vDash^I P$ such that $\sigma_0 = \sigma, x \mapsto [\![e]\!]_{\sigma_0} = \sigma, x \mapsto [\![e]\!]_{\sigma}$ (since $x \notin e, P$). The proof is completed by $\dfrac{(\sigma, x \mapsto [\![e]\!]_{\sigma}) \xrightarrow{c} (\sigma', x \mapsto n')}{\sigma \xrightarrow{\texttt{local } x=e \texttt{ in } c} \sigma'}$.
3. Case $\dfrac{\langle P_i \wedge b \rangle\ c\ \langle P_{i+1} \rangle}{\langle P_0 \rangle\ \texttt{while } b \texttt{ do } c\ \langle \neg b \wedge \exists i \cdot P_i \rangle}$ While.
   Pick an interpretation I and a state σ' such that $\sigma' \vDash^I \neg b \wedge \exists i \cdot P_i$ i.e. $\exists n \cdot \sigma' \vDash^I \neg b \wedge P_n$. If $n = 0$ we take $\sigma = \sigma'$ and we're done. Otherwise we use the premise to obtain a series of states which satisfy $P(n') \wedge b$ for a decreasing n' until we reach a state that satisfies $P(0)$.
4. Case $\dfrac{\langle P \rangle\ c\ \langle Q \rangle}{\langle P \wedge R \rangle\ c\ \langle Q \wedge R \rangle}$ Frame.

Pick an interpretation I and a state $\sigma' \vDash^I Q \wedge R$. Clearly $\sigma' \vDash^I Q$ so that by the premise there exist an $\sigma \vDash^I P$ where $\sigma \xrightarrow{c} \sigma'$; $\sigma \vDash^I R$ follows from Lem. 3.

We prove relative completeness[4] in two steps. We establish that our weakest-postcondition calculus calculates the largest set of states that can be reached by a precondition and a program, and then use this result to prove completeness.

Definition 2 (Weakest postcondition). *The weakest postcondition of a precondition P and program c, with respect to an interpretation I, is defined by*

$$wpo^I[\![P, c]\!] \stackrel{def}{=} \{\sigma' \mid \exists \sigma \cdot \sigma \vDash^I P \wedge \sigma \xrightarrow{c} \sigma'\}$$

The calculus presented in Sect. 5 characterizes exactly this set. This property is sometimes referred to as the *expressivity* of the assertion language. To prove it, we first give a characterization of Υ:

Lemma 4 (Υ). *Given a program c such that*

$$\forall \sigma', P, I \cdot \sigma' \in wpo^I[\![P, c]\!] \text{ iff } \sigma' \vDash^I wpo(P, c)$$

and given boolean predicate b and precondition P we have $\forall n, \sigma' \cdot$

$$\sigma' \vDash^I \Upsilon_{b,c,P}(n) \text{ iff } \exists \sigma_{0 \le i \le n} \cdot \sigma_n = \sigma' \wedge \sigma_{0 \le i < n} \vDash b \wedge \sigma_0 \vDash^I P \wedge \sigma_{0 \le i < n} \xrightarrow{c} \sigma_{i+1}$$

Proof. By induction on n.

Proposition 1. $\forall \sigma' \cdot \sigma' \in wpo^I[\![P, c]\!]$ *iff* $\sigma' \vDash^I wpo(P, c)$.

Proof. By induction on c, similar to the proof of Theorem 1. The case for loops relies on Lem. 4.

It remains to show that the weakest postcondition is always derivable in the program logic; completeness of the logic then follows.

Proposition 2. $\forall c, P$ *we have* $\vdash \langle P \rangle\ c\ \langle wpo(P, c) \rangle$.

Proof. By induction on c. The cases for Skip and Assn are immediate. The other cases are detailed below.

1. Case $c = \texttt{local}\ x = e\ \texttt{in}\ c_0$.

$$\frac{\langle P \wedge x = e \rangle\ c_0\ \langle \mathrm{wpo}(P \wedge x = e, c) \rangle}{\langle P \rangle\ \texttt{local}\ x = e\ \texttt{in}\ c_0\ \langle \exists x \cdot \mathrm{wpo}(P \wedge x = e, c) \rangle}\ \textsc{Local}$$

2. Case $c = c_0; c_1$.

$$\frac{\langle P \rangle\ c_0\ \langle \mathrm{wpo}(P, c_0) \rangle \quad \langle \mathrm{wpo}(P, c_0) \rangle\ c_1\ \langle \mathrm{wpo}(\mathrm{wpo}(P, c_0), c_1) \rangle}{\langle P \rangle\ c_0; c_1\ \langle \mathrm{wpo}(\mathrm{wpo}(P, c_0), c_1) \rangle}\ \textsc{Seq}$$

[4] Also known as completeness in the sense of Cook [1, Sect. 2.8].

3. Case $c = \texttt{if}\ b\ \texttt{then}\ c_0\ \texttt{else}\ c_1$.

$$\dfrac{\dfrac{\langle P \wedge b\rangle\ c_0\ \langle \mathrm{wpo}(P \wedge b, c_0)\rangle}{\langle P\rangle\ \ldots\ \langle \mathrm{wpo}(P \wedge b, c_0)\rangle}\ \textsc{Then} \quad \dfrac{\langle P \wedge \neg b\rangle\ c_1\ \langle \mathrm{wpo}(P \wedge \neg b, c_1)\rangle}{\langle P\rangle\ \ldots\ \langle \mathrm{wpo}(P \wedge \neg b, c_1)\rangle}\ \textsc{Else}}{\langle P\rangle\ \texttt{if}\ b\ \texttt{then}\ c_0\ \texttt{else}\ c_1\ \langle \mathrm{wpo}(P \wedge b, c_0) \vee \mathrm{wpo}(P \wedge \neg b, c_1)\rangle}\ \textsc{Split}$$

4. Case $c = \texttt{while}\ b\ \texttt{do}\ c_0$.

$$\dfrac{\dfrac{\dfrac{\langle \Upsilon_{b,c_0,P}(n) \wedge b\rangle\ c_0\ \langle \mathrm{wpo}(\Upsilon_{b,c_0,P}(n) \wedge b, c_0)\rangle}{\langle \Upsilon_{b,c_0,P}(n) \wedge b\rangle\ c_0\ \langle \Upsilon_{b,c_0,P}(n+1)\rangle}\ \textsc{Con}}{\langle \Upsilon_{b,c_0,P}(0)\rangle\ \texttt{while}\ b\ \texttt{do}\ c_0\ \left\langle \neg b \wedge \bigvee_n \Upsilon_{b,c_0,P}(n) \right\rangle}\ \textsc{While}}{\langle P\rangle\ \texttt{while}\ b\ \texttt{do}\ c_0\ \langle \neg b \wedge \exists i \cdot \Upsilon_{b,c_0,P}(i)\rangle}\ \textsc{Con}$$

5. Case $c = c_0 \sqcup c_1$.

$$\dfrac{\dfrac{\langle P\rangle\ c_0\ \langle \mathrm{wpo}(P, c_0)\rangle}{\langle P\rangle\ c_0 \sqcup c_1\ \langle \mathrm{wpo}(P, c_0)\rangle}\ \textsc{Left} \quad \dfrac{\langle P\rangle\ c_1\ \langle \mathrm{wpo}(P, c_0)\rangle}{\langle P\rangle\ c_0 \sqcup c_1\ \langle \mathrm{wpo}(P, c_0)\rangle}\ \textsc{Right}}{\langle P\rangle\ c_0 \sqcup c_1\ \langle \mathrm{wpo}(P, c_0) \vee \mathrm{wpo}(P, c_1)\rangle}\ \textsc{Split}$$

Lemma 5. *If* $\vDash \langle P\rangle\ c\ \langle Q\rangle$ *then* $Q \Rightarrow \mathit{wpo}(P, c)$.

Proof.

$$\begin{array}{lll} \vDash \langle P\rangle\ c\ \langle Q\rangle & \text{iff } \forall I, \sigma' \vDash^I Q \cdot \exists \sigma \vDash^I P \wedge \sigma \xrightarrow{c} \sigma' & \text{iff } \forall I, \sigma' \vDash^I Q \cdot \sigma' \in \mathrm{wpo}^I[\![P, c]\!] \\ & \text{iff } \forall I, \sigma' \vDash^I Q \cdot \sigma' \vDash^I \mathrm{wpo}(P, c) & \text{iff } Q \Rightarrow \mathrm{wpo}(P, c) \end{array}$$

Theorem 2 (Completeness). *If* $\vDash \langle P\rangle\ c\ \langle Q\rangle$ *then* $\vdash \langle P\rangle\ c\ \langle Q\rangle$.

Proof. Follows from rule Con and Prop. 2.

7 Related Work

Hoare Logic was introduced by Hoare in 1969 [12], and has since grown into a very active research field. Ten years after its invention a survey already spanned two papers [1, 2]. Recent work has introduced the concept of *separation* into the logic to deal with aliasing [25], extended it to functional languages [15, 16] and embedded Hoare logic in type theory [22].

As already discussed, however, Hoare logics are not suitable for state reachability specifications, with the exception of probabilistic Hoare logics [23, 18, 10, 5, 21]. A probabilistic specification for shuffle might say that the probability of the generation of any permutation is greater than zero; indeed, it might say that every permutation is equally likely. Although such exact guarantees cannot be proven using Reverse Hoare Logic, proofs in Reverse Hoare Logic are simpler than proofs in probabilistic Hoare logics. The assertion language in Reverse Hoare Logic is a familiar first order logic, and reasoning does not involve manipulating probabilities or state distributions.

The notions of "weakest postcondition" and "strongest precondition" appear only occasionally in the literature, mostly in the context of incomplete knowledge. For instance, when using dynamic binding in object oriented programming, the caller can only assume the weakest postcondition, i.e. the postcondition of the method in the top of the inheritance hierarchy [11]. Similar situations arise with web services [19] and contracts [24], and in artificial intelligence when omitting information in an attempt to simplify a search domain [3, 26]. In UTP these notions are used to define recursion in the theory of "designs" [27].

Dynamic logic [8, 9] is a multi-modal logic; for any program c the formula $\langle c \rangle P$ is satisfied if c can terminate in a state satisfying P, and the formula $[c]P$ is satisfied if all states that c terminates in satisfy P (there might not be any). The Hoare triple $\{\alpha \in \Pi(a)\}$ shuffle $\{a = \alpha\}$ can be expressed as

$$\forall \alpha \cdot \alpha \in \Pi(a) \to \langle \mathsf{shuffle}(\mathsf{a}) \rangle (a = \alpha) \tag{26}$$

in Dynamic Logic. Like in Hoare Logic, however, an implementation of shuffle which relies on a global random number stream would not satisfy specification (26). The more general treatment of qualifiers in Dynamic Logic means that in Dynamic Logic we do not have to be quite as precise as in Hoare Logic:

$$\forall \alpha, \iota \cdot \exists \rho \cdot \alpha = \pi_i(a), \mathtt{r} = \rho \to \langle \mathsf{shuffle}(\mathsf{a}) \rangle (a = \alpha) \tag{27}$$

Specification (27) is better than the Hoare triple (6) as we do not have to specify the precise relation between permutations and random number streams, so that this specification is satisfied by more implementations of shuffle. Nevertheless, this specification still exposes an implementation detail (the reliance on r).

One might envision extending Dynamic Logic with a reverse modality to obtain a "Reverse Dynamic Logic". This would certainly be of interest. Finally, we remark that a lot of the literature on Dynamic Logic is concerned with the treatment of "totality" in the presence of non-determinism; this is less relevant in our setting, since we are interested in reachability.

If we have an inverse operation c^{-1} on programs such that $\sigma \xrightarrow{c^{-1}} \sigma'$ exactly when $\sigma' \xrightarrow{c} \sigma$, it is immediate from the definitions of validity of total Hoare triples (with "for all initial... exists final..." semantics) that $\langle P \rangle\ c\ \langle Q \rangle$ exactly when $\{Q\}\ c^{-1}\ \{P\}$. Reverse Hoare Logic is thus related to the old idea of program inversion [6, 28], but we avoid the need for computing program inverses. From another perspective, Reverse Hoare Logic provides a way to prove Hoare specifications of inverse programs without having to compute the inverse.

Hoare-style "contracts" (pre- and post-conditions) are often implemented as runtime checks in programming languages such as Eiffel [20]. It is not obvious how to check "reverse contracts" at runtime. However, static verification of contracts is slowly becoming a reality through tools such as ESC/Java2 [7] and Spec# [4]. It should not be too difficult to extend these tools to support verification of reverse contracts too, although we have not attempted to do so. Some of these tools do not support logic variables, amplifying the need for a reverse logic to reason about reachability.[5]

8 Conclusions

Reverse Hoare Logic can be used to give and prove reachability specifications. Compared to the alternatives, the specifications can be more abstract than in Hoare logic, and the corresponding proofs are simpler than when using a probabilistic logic. Reverse Hoare Logic naturally gives rise to the concept of a weakest postcondition, which we have used to show that the proof system is complete.

It would be worthwhile to attempt to combine standard and reverse Hoare logic, yielding a logic in which we can express both the reachability of good states and the non-reachability of bad states. An extension to functional languages, especially higher-order ones, would also be of interest.

The use of an infinitary logic enabled an elegant definition of the weakest postcondition calculus and hence made the completeness proof easier. However, a formulation using a finitary logic would be useful for an implementation of Reverse Hoare Logic in an automatic theorem prover.

Finally, it would be interesting to look at adaptation in the context of Reverse Hoare Logic. We believe that a simple adaptation rule such as equation (6) in [17] is sound for Reverse Hoare Logic (*mutatis mutandis*), but it is unclear at present if Reverse Hoare Logic can be made adaptation complete.

Acknowledgements We would like to thank Colm Bhandal for an insightful discussion on the expressivity of Hoare triples in the presence of non-determinism, and to Hugh Gibbons for providing valuable references.

References

1. Apt, K.R.: Ten years of Hoare's logic: A survey—part I. ACM Trans. Program. Lang. Syst. 3, 431–483 (1981)
2. Apt, K.R.: Ten years of Hoare's logic: A survey—part II: Nondeterminism. Theoretical Computer Science 28(1-2), 83 – 109 (1983)
3. Bacchus, F., Yang, Q.: Downward refinement and the efficiency of hierarchical problem solving. Artificial Intelligence 71(1), 43 – 100 (1994)
4. Barnett, M., Deline, R., Fähndrich, M., Jacobs, B., Leino, K.R., Schulte, W., Venter, H.: The spec# programming system: Challenges and directions. In: Meyer, B., Woodcock, J. (eds.) Verified Software: Theories, Tools, Experiments, pp. 144–152. Springer-Verlag, Berlin, Heidelberg (2008)
5. Chadha, R., Cruz-Filipe, L., Mateus, P., Sernadas, A.: Reasoning about probabilistic sequential programs. Theor. Comput. Sci. 379, 142–165 (July 2007)
6. Chen, W., Udding, J.T.: Program inversion: more than fun! Sci. Comput. Program. 15, 1–13 (November 1990)

[5] In some tools such as ESC/Java2 we can use "ghost variables" (ordinary variables appearing only in specifications and proofs) to express reachability properties; e.g. for the nondeterministic shuffle we can write $\{\gamma \in \Pi(\mathtt{a})\}$ shuffle $\{\mathtt{a} = \gamma \wedge \gamma = \mathtt{old}(\gamma)\}$. Again, deterministic implementations of shuffle do not satisfy this specification. Note that techniques to remove ghost variables (*cf.* [14]) do not apply here.

7. Cok, D., Kiniry, J.: ESC/Java2: Uniting ESC/Java and JML. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.L., Muntean, T. (eds.) Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, Lecture Notes in Computer Science, vol. 3362, pp. 108–128. Springer Berlin / Heidelberg (2005)
8. Goldblatt, R.: Logics of time and computation. Center for the Study of Language and Information, Stanford, CA, USA (1987)
9. Harel, D.: Logics of programs: Axiomatics and descriptive power. Tech. rep., Massachusetts Institute of Technology, Cambridge, MA, USA (1978)
10. den Hartog, J.: Verifying probabilistic programs using a Hoare like logic. In: Thiagarajan, P., Yap, R. (eds.) ASIAN, LNCS, vol. 1742, pp. 790–790. Springer (1999)
11. Heyer, T.: Semantic Inspection of Software Artifacts From Theory to Practice. Ph.D. thesis, Linköping University (2001)
12. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12, 576–580 (October 1969)
13. Hoare, C.A.R., Wirth, N.: An axiomatic definition of the programming language PASCAL. Acta Informatica 2, 335–355 (1973)
14. Hofmann, M., Pavlova, M.: Elimination of ghost variables in program logics. In: Proceedings of the 3rd conference on Trustworthy global computing. pp. 1–20. TGC'07, Springer-Verlag, Berlin, Heidelberg (2008)
15. Honda, K., Yoshida, N., Berger, M.: An observationally complete program logic for imperative higher-order functions. In: LICS. pp. 270–279. IEEE (2005)
16. Kanig, J., Filliâtre, J.C.: Who: a verifier for effectful higher-order programs. In: ACM SIGPLAN Workshop on ML. pp. 39–48. ACM, New York, NY, USA (2009)
17. Kleymann, T.: Hoare logic and auxiliary variables. Formal Aspects of Computing 11, 541–566 (1999), http://dx.doi.org/10.1007/s001650050057, 10.1007/s001650050057
18. Kozen, D.: A probabilistic PDL. J. Comp. and Sys. Sc. 30(2), 162 – 178 (1985)
19. Kumar, A., Srivastava, B., Mittal, S.: Information modeling for end to end composition of semantic web services. In: Gil, Y., Motta, E., Benjamins, V., Musen, M. (eds.) ISWC, LNCS, vol. 3729, pp. 476–490. Springer (2005)
20. Meyer, B.: Object-oriented software construction (2nd ed.). Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1997)
21. Morgan, C., McIver, A., Seidel, K.: Probabilistic predicate transformers. ACM Trans. Program. Lang. Syst. pp. 325–353 (1996)
22. Nanevski, A., Morrisett, G., Birkedal, L.: Polymorphism and separation in Hoare Type Theory. SIGPLAN Not. 41, 62–73 (September 2006)
23. Ramshaw, L.H.: Formalizing the analysis of algorithms. Ph.D. thesis, Stanford University (1979)
24. Reussner, R., Poernomo, I., Schmidt, H.: Reasoning about software architectures with contractually specified components. In: CBSQ. LNCS, vol. 2693, pp. 287–325. Springer (2003)
25. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS. pp. 55–74. IEEE Computer Society, Washington, DC, USA (2002)
26. ten Teije, A., van Harmelen, F.: Characterising approximate problem solving: by partially fulfilled pre- and postconditions. In: ECAI. CEUR-WS, vol. 16, pp. 78–82 (1998)
27. Woodcock, J., Cavalcanti, A.: A tutorial introduction to designs in Unifying Theories of Programming. In: Boiten, E.A., Derrick, J., Smith, G. (eds.) IFM, LNCS, vol. 2999, pp. 40–66. Springer (2004)
28. von Wright, J.: Program inversion in the refinement calculus. Information Processing Letters 37(2), 95 – 100 (1991)