Lecture Notes
Hoare Logic and Variations: Probabilistic, Relational, Probabilistic+Relational

Assaf Kfoury

March 26, 2018

Contents

I start with a quick presentation of classical Hoare Logic (HL) in Section 1, the study of which has extended over nearly five decades – and this is why I call it ‘classical’. I present enough of HL to make explicit connections with the more recent logics, *relational Hoare Logic* (RHL) in Section 2, *probabilistic Hoare Logic* (pHL) in Section 4, and *probabilistic Relational Hoare Logic* (pRHL) in Section 5.

The material on classical HL is found in several excellent textbooks ([10, 12, 15, 16] among several other), although the presentation here reflects my own slant and emphasis of what should be remembered about HL. The material on RHL, pHL, and pRHL, is not in any textbook, as far as I know; I have collected it, and also simplified it for pedagogical reasons, from several research articles.

1 Classical Hoare Logic

A Hoare Logic combines a *programming language* (PL) and a formal logic, the latter being typically a fragment of *first-order logic* (FOL). Formulas of a Hoare Logic are usually written as triples of the form $\{\,\varphi\,\}\ P\ \{\,\psi\,\}$ where P is a well-formed program or program phrase in the programming language, and φ (called a *pre-condition*) and ψ (called a *post-condition*) are well-formed formulas of FOL.

There is no single 'Hoare Logic'. There is a different one for every choice of PL and every choice of a FOL fragment. We here choose PL to be the language of WHILE-programs, a very simple imperative language often used for pedagogical purposes. In our definitions below, we allow pre-conditions φ and post-conditions ψ to be formulas of FOL in general, but in all the examples and exercises, it will suffice to consider quantifier-free φ and ψ. Informally, the intended meaning of a triple $\{\,\varphi\,\}\ P\ \{\,\psi\,\}$ is:

> *If execution of P terminates when started from a state satisfying φ,*
> *then P terminates in a state satisfying ψ.*

Another informal way of saying the same thing is: *When P is started from a state satisfying φ, either P diverges or P terminates in a state satisfying ψ.* The triple $\{\,\varphi\,\}\ P\ \{\,\psi\,\}$ is sometimes called a *Hoare triple* and sometimes a *partial correctness assertion* (PCA); the reason for the latter appellation is explained later. We prefer the expression 'Hoare triple' to make explicit the contrast with 'Hoare quadruple' which later designates a formula of RHL or pRHL.[1]

1.1 An Imperative Programming Language: WHILE

We precede the formal definition of WHILE-programs with motivational examples.

Example 1. The following is a small WHILE-program:

```
    $y := 1;$
    $z := 0;$
    while $\neg(x = z)$ do
        $z := z + 1;$
        $y := y * z$
    od
```

The program is simple enough that we can correctly say that, if a non-negative integer n is initially assigned to variable x, then the program computes the factorial $n!$ and stores it in variable y. And so, we may call this program fact.

There are pre-conditions, *i.e.*, conditions on the input states, that will guarantee that fact operates correctly. One condition is that the initial value n stored in variable x cannot be fractional, which can be guaranteed by declaring all variables to be of type int (and we assume such type declarations are done somewhere else in the program). Another condition is that n should not be a negative integer. When such a pre-condition is satisfied, the output state is guaranteed to specify that $n!$ is stored in

[1] The conventions for writing Hoare triples vary. Some like to write $\varphi\ \{P\}\ \psi$ instead of $\{\,\varphi\,\}\ P\ \{\,\psi\,\}$, for example in [12]. Others invent a somewhat unusual notation as in $(\!|\,\varphi\,|\!)P\ (\!|\psi|\!)$, for example in [10]. In all cases, the idea is to clearly separate the pre-condition φ and the post-condition ψ from the inserted code P.

variable y. We can therefore write the following Hoare triple:

$$\{\, x \geqslant 0 \,\}\ \mathsf{fact}\ \{\, y = x! \,\}$$

which asserts that if execution of fact terminates when started at a state satisfying $x \geqslant 0$, then fact ends in a state satisfying $y = x!$. □

Example 2. The following is a small WHILE-program, call it foo:

$$\begin{array}{l} y := 0\ ; \\ \textbf{while}\ \ y * y < x\ \ \textbf{do}\ \ y := y + 1\ \ \textbf{od}\ ; \\ y := y - 1 \end{array}$$

Given an integer $x > 0$, foo computes the largest integer whose square is less than x and stores it in y. Given $x \leqslant 0$, foo returns -1 in the variable y. So, appropriate Hoare triples involving foo are:

$$\{\, x \leqslant 0 \,\}\ \mathsf{foo}\ \{\, y = -1 \,\} \qquad \text{and} \qquad \{\, x > 0 \,\}\ \mathsf{foo}\ \{\, y^2 < x \,\}$$

and there are many others. In general, we prefer a Hoare triple that says more about the program's behavior. Of the preceding two, we prefer the second, because it says more about foo's computation, though it does not say that the returned value in y is 'the largest integer whose square is less than x'. So, a more precise Hoare triple is:

$$\{\, x > 0 \,\}\ \mathsf{foo}\ \{\, (y^2 < x) \wedge (y+1)^2 \geqslant x \,\}$$

In general, a Hoare triple is most informative if its pre-condition is *weakest*, *i.e.*, it imposes the fewest possible restrictions on input states. And the Hoare triple is also most informative if its post-condition is *strongest*, *i.e.*, it expresses the most precise properties or restrictions satisfied by output states. □

Definition 3 (*Syntax of* WHILE *Programs*)**.** Let x range over a countable set of variables and n range over all the numerals $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$. The syntax of *integer expressions* is given by an extended BNF definition:[2]

$$E ::= n \mid x \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 * E_2 \mid \cdots$$

The ellipsis in the preceding line indicates that other standard forms of integer expressions may be added according to need. The syntax of *Boolean expressions* is given by an extended BNF definition:

$$B ::= \texttt{true} \mid \texttt{false} \mid \neg B \mid B_1 \vee B_2 \mid B_1 \wedge B_2 \mid E_1 = E_2 \mid E_1 < E_2 \mid \cdots$$

Program expressions or *commands* are specified by an extended BNF definition:

$$C ::= \texttt{skip} \mid x := E \mid C_1 ; C_2 \mid \textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \textbf{fi} \mid \textbf{while}\ B\ \textbf{do}\ C\ \textbf{od}$$

Following standard practice for easier reading, we use indentation liberally when we write the text of a WHILE-program. We may insert in-line comments by preceding them with '//'. □

[2]We are careful in distinguishing the syntax of WHILE programs from their semantics later in this section. So, for example, you should think that the numeral '2' is a constant symbol which is later interpreted as the number *two*, the binary function symbol '+' is later interpreted as *addition*, etc.

1.2 Formal Proof Rules of Classical HL

Classical Hoare Logic consists of Hoare triples, by which we specify the input-output behavior of programs, and the axioms and inference rules for deriving valid triples.

$$\vdash \{\,\varphi\,\}\ \texttt{skip}\ \{\,\varphi\,\} \qquad \text{[skip]}$$

$$\vdash \{\,\psi[x \mapsto E]\,\}\ x := E\ \{\,\psi\,\} \qquad \text{[assignment]}$$

$$\frac{\vdash \{\,\varphi\,\}\ C_1\ \{\,\theta\,\} \qquad \vdash \{\,\theta\,\}\ C_2\ \{\,\psi\,\}}{\vdash \{\,\varphi\,\}\ C_1; C_2\ \{\,\psi\,\}} \qquad \text{[sequencing]}$$

$$\frac{\vdash \{\,\varphi \wedge B\,\}\ C_1\ \{\,\psi\,\} \qquad \vdash \{\,\varphi \wedge \neg B\,\}\ C_2\ \{\,\psi\,\}}{\vdash \{\,\varphi\,\}\ \textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \textbf{fi}\ \{\,\psi\,\}} \qquad \text{[conditional]}$$

$$\frac{\vdash \{\,\varphi \wedge B\,\}\ C\ \{\,\varphi\,\}}{\vdash \{\,\varphi\,\}\ \textbf{while}\ B\ \textbf{do}\ C\ \textbf{od}\ \{\,\varphi \wedge \neg B\,\}} \qquad \text{[while]}$$

$$\frac{\models \varphi' \to \varphi \qquad \vdash \{\,\varphi\,\}\ C\ \{\,\psi\,\} \qquad \models \psi \to \psi'}{\vdash \{\,\varphi'\,\}\ C\ \{\,\psi'\,\}} \qquad \text{[weakening]}$$

Figure 1: Inference rules of Classical HL.

As with any formal logic, we assess the proof rules for what they are set out to accomplish by defining a formal semantics for the logic. The relationship between proof rules and formal semantics is stated by means of:

- *Soundness*: Every formula derivable by the proof rules is true w.r.t. the semantics.
- *Completeness*: Every true formula w.r.t. the semantics is derivable by the proof rules.

If the axioms (*i.e.*, the initial formulas) are valid (*i.e.*, true), then *soundness* means that the proof rules preserves validity (*i.e.*, truth) of formulas. *Soundness* is a minimum requirement for the proof rules of any formal logic. *Completeness* may or may not be achieved, but if it is, then we have an exact match between proof rules and formal semantics. Hence, before we discuss the soundness and completeness of the proof rules of Classical HL, we need to define its formal semantics.[3]

1.3 Formal Semantics of Classical HL

In a Hoare formula $\{\,\varphi\,\}\ P\ \{\,\psi\,\}$, the formal meaning of the program P may be defined in one of several ways, all mutually equivalent (in some sense that can be made precise), but not all equally convenient for all situations. They can be classified into two general groups, *operational* and *denotational*, and each of these two include several varieties.[4]

[3] I tend to use the words 'true' and 'truth' where others use the words 'valid' and 'validity'. Taken in a technical sense, these words require a precise definition of the formal semantics.

[4] This is an interesting topic to pursue independently, but not for this handout, which is well covered in several textbooks. The following is a very broad classification:

- *Operational Semantics*: The meaning of the program P is explained in terms of the execution of a hypothetical

By contrast, the formal meaning of the pre- and post-conditions φ and ψ in $\{\,\varphi\,\}\ P\ \{\,\psi\,\}$ is straightforward and without alternatives to choose from, resulting from a model theory of *first-order logic* that has simpler norms and conventions.

Depending on the approach one chooses to define the formal semantics of P, *operational* or *denotational* and in one of their respective varieties, there is a corresponding definition of the formal semantics of $\{\,\varphi\,\}\ P\ \{\,\psi\,\}$. In this handout, we do not give a survey of approaches to the formal semantics of WHILE programs; it suffices to use one of them and we choose, most conveniently, a *denotational* approach – except in one place where we need to invoke an *operational* approach, but without dwelling on it (see Remark 14).

Let $\mathcal{V}$ be the set of variables. A *state* is a map from $\mathcal{V}$ to $\mathbb{Z}$. We use the letter σ, appropriately decorated if need be, to denote a state. If σ is a state, $x \in \mathcal{V}$, and $n \in \mathbb{Z}$, we write $\sigma[x \mapsto n]$ to denote the state-update function; that is, for every $v \in \mathcal{V}$:

$$\sigma[x \mapsto n](v) \;\triangleq\; \begin{cases} n & \text{if } v = x\ , \\ \sigma(v) & \text{otherwise.} \end{cases}$$

Let Σ be the set of *states*. If E is an integer expression, we write $[\![E]\!] : \Sigma \to \mathbb{Z}$ for the interpretation of E, which maps every $\sigma \in \Sigma$ to a value in $\mathbb{Z}$. Likewise, if B is a Boolean expression, we write $[\![B]\!] : \Sigma \to \mathbb{B}$ for the interpretation of B.

Exercise 4. Provide the details in the definitions of $[\![E]\!] : \Sigma \to \mathbb{Z}$ and $[\![B]\!] : \Sigma \to \mathbb{B}$. *Hint*: You have to assign an interpretation for each of the cases in the BNF definitions of integer expressions and Boolean expressions. You can limit your answer to the cases explicitly mentioned in the BNF definitions for E and B in Definition 3. □

The interpretation $[\![C]\!]$ of a command C is a little more complicated. It turns out to be a *partial* function $[\![C]\!] : \Sigma \rightharpoonup \Sigma$, which may or may not be *total*, to account for the fact that execution of a WHILE-program may diverge. We first define $[\![C]\!]_{\text{rel}}$ as a relation between states, *i.e.*, $[\![C]\!]_{\text{rel}} \subseteq \Sigma \times \Sigma$, and then show that this relation $[\![C]\!]_{\text{rel}}$ is in fact the desired partial function $[\![C]\!] : \Sigma \rightharpoonup \Sigma$, because it turns out that if $(\sigma, \sigma_1), (\sigma, \sigma_2) \in [\![C]\!]_{\text{rel}}$ then $\sigma_1 = \sigma_2$. If $X, Y \subseteq \Sigma \times \Sigma$ are binary relation on states, we write $X \circ Y$ to denote their *composition*:

$$X \circ Y \triangleq \{\, (\sigma, \sigma') \in \Sigma \times \Sigma \mid \text{there is } \sigma'' \in \Sigma \text{ such that } (\sigma, \sigma'') \in Y \text{ and } (\sigma'', \sigma') \in X \,\}.$$

Note that Y is applied first and X is applied second, even though they appear in the reverse order in

computer on which P is run, and this execution may in turn be defined in one of two ways:

1. *Small-Step Operational Semantics*, a framework for describing the execution of P as an iterative sequence of small computational steps, definable in one of two styles:

   (a) *Structural Operational Semantics*, which takes the form of a set of inference rules defining the allowed transitions of a composite piece of syntax in terms of the transitions of its constituent parts, or

   (b) *Reduction Operational Semantics*, based on the prior definition of what are called *reduction contexts*, each such context being a program with a hole where a subterm ready to be executed can be plugged.

2. *Big-Step Operational Semantics*, the central idea of which is to evaluate P by recursively evaluating its subterms and then combining the results.

- *Denotational Semantics*: The meaning of P is obtained by first attaching a mathematical function to each atomic component of P, and then finally to P itself, by successive syntax-directed functional compositions.

Both forms of semantics have their purpose: the *operational* is closer to actual implementations of programming languages, the *denotational* is more mathematical at it invokes notions of category theory and domain theory.

'$X \circ Y$'. By induction on the syntax of commands:

$$\begin{aligned}
[\![\texttt{skip}]\!]_{\text{rel}} &\triangleq \{\,(\sigma,\sigma) \mid \sigma \in \Sigma\} \\
[\![x := E]\!]_{\text{rel}} &\triangleq \{\,(\sigma,\sigma[x \mapsto n]) \mid \sigma \in \Sigma \text{ and } n = [\![E]\!]\sigma\,\} \\
[\![C_1; C_2]\!]_{\text{rel}} &\triangleq [\![C_2]\!]_{\text{rel}} \circ [\![C_1]\!]_{\text{rel}} \\
[\![\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}]\!]_{\text{rel}} &\triangleq \{\,(\sigma,\sigma') \mid [\![B]\!]\sigma = \texttt{true} \text{ and } (\sigma,\sigma') \in [\![C_1]\!]_{\text{rel}}\,\} \\
&\quad\;\cup\; \{\,(\sigma,\sigma') \mid [\![B]\!]\sigma = \texttt{false} \text{ and } (\sigma,\sigma') \in [\![C_2]\!]_{\text{rel}}\,\}
\end{aligned}$$

There are some subtleties in defining $[\![\textbf{while } B \textbf{ do } C \textbf{ od}]\!]_{\text{rel}}$, suggested by the expected equivalence between the two program phrases:

$$\underbrace{\textbf{while } B \textbf{ do } C \textbf{ od}} \quad \text{and} \quad \textbf{if } B \textbf{ then } \; C; \underbrace{\textbf{while } B \textbf{ do } C \textbf{ od}} \;\; \textbf{else } \texttt{skip} \textbf{ fi}$$

where the second phrase is obtained from the first by unwinding the loop once. Hence, if $R \subseteq \Sigma \times \Sigma$ is the denotation of $[\![\textbf{while } B \textbf{ do } C \textbf{ od}]\!]_{\text{rel}}$ as a relation between states, which is yet to be defined, then we would like the following equality (§) to hold:

$$(\S) \qquad \boxed{R \;=\; \Big\{\,(\sigma,\sigma') \;\Big|\; [\![B]\!]\sigma = \texttt{true} \text{ and } (\sigma,\sigma') \in R \circ [\![C]\!]_{\text{rel}} \Big\} \;\cup\; \Big\{\,(\sigma,\sigma) \;\Big|\; [\![B]\!]\sigma = \texttt{false} \Big\}}$$

Note that R appears on both sides of (§). We can therefore view (§) as an equation to be solved for the unknown R. The right-hand side of (§) can be written as a function $\mathcal{F}$ of the unknown R, namely:

$$\mathcal{F}(R) \;\triangleq\; \{\,(\sigma,\sigma') \mid [\![B]\!]\sigma = \texttt{true} \text{ and } (\sigma,\sigma') \in R \circ [\![C]\!]_{\text{rel}}\,\} \;\cup\; \{\,(\sigma,\sigma) \mid [\![B]\!]\sigma = \texttt{false}\,\}.$$

Note carefully that $\mathcal{F}$ is a function from relations to relations, $\mathcal{F} : 2^{\Sigma\times\Sigma} \to 2^{\Sigma\times\Sigma}$, *i.e.*, if $A \subseteq \Sigma \times \Sigma$ is a specific relation between states (not an unknown relation), then $\mathcal{F}(A) \subseteq \Sigma \times \Sigma$ is another specific relation between states. Solving (§) means solving a *fixpoint equation* for the unknown R:

$$(\S) \qquad \boxed{R = \mathcal{F}(R)}$$

If there is a specific $\widetilde{R} \subseteq \Sigma \times \Sigma$ such that $\widetilde{R} = \mathcal{F}(\widetilde{R})$, we say $\widetilde{R}$ is a *fixpoint solution* of equation (§).

Remark 5. Fixpoint equations are familiar to you from freshman calculus. Consider, for example, the equation $x = f(x)$ where $f(x) \triangleq x^2 - 2x + 2$; it has two fixpoint solutions $\{1, 2\}$ (which are usually called the *roots* of the equation in calculus). Such an equation may or may not have integer solutions; *e.g.*, when $f(x) \triangleq x^2 - 3x + 10$, the two roots of $x = f(x)$ are imaginary numbers.[5] Other fixpoint equations $x = f(x)$ may have one, or two, ..., or infinitely many solutions that are integers.

What is new in the fixpoint equation $R = \mathcal{F}(R)$ above is that a solution $\widetilde{R}$, if it exists, is not a number but a relation – and not any relation, but a binary relation between states, *i.e.*, $\widetilde{R} \subseteq \Sigma \times \Sigma$, where states in Σ are vectors or tuples (all of the same dimension, possibly infinite). And just as an equation $x = f(x)$ can be solved for x by an iterative process of successive approximations (under reasonable assumptions about the function f), so too the equation $R = \mathcal{F}(R)$ can be solved by a process of successive approximations. $\square$

[5] Specifically, if you want to check it out, the two roots are $2 - i\sqrt{6}$ and $2 + i\sqrt{6}$.

We give a couple of examples to give some intuition for how a fixpoint equation $R = \mathcal{F}(R)$ can be solved. In what follows, if $\sigma_1, \sigma_2 \in \Sigma$ are vectors over the integers of the same finite or infinite dimension, and $\mathtt{iop}$ is one of the standard binary operation on integers $\{+, \mathsf{div}, \bmod, \times, \ldots\}$ in infix position, then $\sigma_1 \mathbin{\mathtt{iop}} \sigma_2$ is pointwise application of $\mathtt{iop}$ to corresponding entries in σ_1 and σ_2:

$$\sigma_1 \mathbin{\mathtt{iop}} \sigma_2 \triangleq \big\langle\, \sigma_1(1) \mathbin{\mathtt{iop}} \sigma_2(1),\ \sigma_1(2) \mathbin{\mathtt{iop}} \sigma_2(2),\ \ldots \,\big\rangle$$

where $\sigma_i = \big\langle \sigma_i(1), \sigma_i(2), \ldots \big\rangle$. We extend this operation to two sets of states $A_1, A_2 \subseteq \Sigma$ by defining:

$$A_1 \mathbin{\mathtt{iop}} A_2 \triangleq \left\{ \sigma_1 \mathbin{\mathtt{iop}} \sigma_2 \,\middle|\, \sigma_1 \in A_1 \text{ and } \sigma_2 \in A_2 \right\}$$

and similarly to two binary relations between states $R_1, R_2 \subseteq \Sigma \times \Sigma$.

Example 6. Consider the following set A of integer pairs:

$$A \triangleq \left\{ \big\langle n, 1 + (n \mathbin{\mathsf{div}} 2) \cdot 2 \big\rangle \,\middle|\, n \in \mathbb{Z} \text{ and } n \geqslant 0 \right\} \ \subseteq\ \mathbb{Z} \times \mathbb{Z}$$

Some of the pairs in A are $\{ \langle 0,1\rangle, \langle 1,1\rangle, \langle 2,3\rangle, \langle 3,3\rangle, \langle 4,5\rangle, \langle 5,5\rangle, \ldots\}$. An example of a fixpoint equation for an unknown relation $R \subseteq \mathbb{Z} \times \mathbb{Z}$ is the following:

$$(\heartsuit) \quad R \ =\ \mathcal{F}(R) \quad \text{where } \mathcal{F}(R) \ \triangleq\ A \cup \big(R \mathbin{\mathsf{div}} \langle 2,1\rangle\big) \cdot \langle 2,1\rangle$$

Note carefully how the function $\mathcal{F} : 2^{\mathbb{Z}\times\mathbb{Z}} \to 2^{\mathbb{Z}\times\mathbb{Z}}$ is defined: If $R \subseteq \mathbb{Z} \times \mathbb{Z}$, then for every $\langle x, y\rangle \in R$ the action of $\mathcal{F}$ is to compute the pair:

$$\big(\langle x, y\rangle \mathbin{\mathsf{div}} \langle 2,1\rangle\big) \cdot \langle 2,1\rangle = \langle (x \mathbin{\mathsf{div}} 2) \cdot 2, (y \mathbin{\mathsf{div}} 1) \cdot 1\rangle = \langle (x \mathbin{\mathsf{div}} 2) \cdot 2, y\rangle$$

and place the resulting pair in $\mathcal{F}(R)$. It is easy to check that A is a fixpoint solution of equation $(\heartsuit)$, but it is not the only fixpoint solution. For arbitrary non-empty and possibly infinite sets $X, Y \subseteq \mathbb{Z}$, consider the following set $B_{X,Y}$ of integer pairs:

$$B_{X,Y} \triangleq \bigcup \left\{ \{ \langle 2i, j\rangle, \langle 2i+1, j\rangle \} \,\middle|\, i \in X \text{ and } j \in Y \right\}$$

It is readily checked that, for every $X, Y \subseteq \mathbb{Z}$, the set of pairs $A \cup B_{X,Y}$ is again a fixpoint solution of equation $(\heartsuit)$.

There are therefore infinitely many fixpoint solutions of $(\heartsuit)$, one for every choice of X and Y. Of these infinitely many solutions, there is one that is the *least* or smallest fixpoint, namely, the set of pairs A defined in the opening paragraph of the current example.

It turns out that the least fixpoint solution A is the denotation $[\![P]\!]_{\text{rel}}$ of the following WHILE-program P over a single variable $\{x\}$ which is used for both input and output, thus allowing us to take input states and output states to be each a single integer:

```
if $x < 0$ then diverge
        else  $x := 1 + (x \mathbin{\mathsf{div}} 2) \cdot 2$;
              while even($x$) do skip od
fi
```

where '`diverge`' is a shorthand for '**while** `true` **do** `skip` **od**'. It is easy to check, by inspection here, that A describes the input-output relation of program P and $A = [\![P]\!]_{\text{rel}}$. □

In the preceding example we started from a specific set A of integer pairs, then defined a fixpoint equation ($\heartsuit$) for which A is the least solution, and finally defined a WHILE-program whose denotation is A. In Example 7, 10, and 12, we go in the reverse order: We start from a WHILE-program, then define a fixpoint equation whose least solution is the denotation of the program.

Example 7. Consider the WHILE-program fact in Example 1. Since all the variables occurring in fact are $\{x, y, z\}$ we can take every state σ to be a tuple of dimension 3, *i.e.*, the set of states is $\Sigma = \mathbb{Z}^3$, with the understanding that if $\sigma = \langle m, n, p\rangle$ then m, n, and p are the integers assigned to x, y, and z, in that order, *i.e.*, $\sigma(1)$, $\sigma(2)$, and $\sigma(3)$ are the integers stored in x, y, and z, respectively.

By inspection, we first define the denotations of the simpler phrases in fact, namely, the initial two instructions '$y := 1; z := 0$' and the two instructions in the body of the loop '$z := z + 1; y := y * z$', which we call $\widetilde{R}_0$ and $\widetilde{R}_1$, respectively:

$$\begin{aligned}\widetilde{R}_0 &= [\![\, y := 1;\ z := 0 \,]\!]_{\text{rel}}\\ &= \{\, (\sigma, \sigma') \mid \sigma = \langle m, n, p\rangle,\ \sigma' = \langle m, 1, 0\rangle \,\}\end{aligned}$$

$$\begin{aligned}\widetilde{R}_1 &= [\![\, z := z + 1;\ y := y * z \,]\!]_{\text{rel}}\\ &= \{\, (\sigma, \sigma') \mid \sigma = \langle m, n, p\rangle,\ \sigma' = \langle m, n * (p + 1), p + 1\rangle \,\}\end{aligned}$$

The denotation $\widetilde{R}_2$ of the **while-do** loop is a solution of the following fixpoint equation:

$$(\diamondsuit) \quad R = \mathcal{F}(R) \quad \text{where}$$

$$\begin{aligned}\mathcal{F}(R) \triangleq\ &\{\, (\sigma, \sigma') \mid \sigma = \langle m, n, p\rangle,\ \sigma' = \langle m', n', p'\rangle,\ [\![\neg(x = z)]\!]\sigma = \texttt{true},\ (\sigma, \sigma') \in R \circ \widetilde{R}_1 \,\}\\ &\cup \{\, (\sigma, \sigma) \mid \sigma = \langle m, n, p\rangle,\ [\![\neg(x = z)]\!]\sigma = \texttt{false} \,\}\\ =\ &\{\, (\sigma, \sigma') \mid \sigma = \langle m, n, p\rangle,\ \sigma' = \langle m', n', p'\rangle,\ m \neq p,\ (\sigma, \sigma') \in R \circ \widetilde{R}_1 \,\}\\ &\cup \{\, (\sigma, \sigma) \mid \sigma = \langle m, n, p\rangle,\ m = p \,\}\end{aligned}$$

We delay for a moment the question of how to systematically compute the least fixpoint solution $\widetilde{R}_2$ of an equation such as ($\diamondsuit$). The **while-do** loop is simple enough so that, by inspection, we conjecture:

1. $\widetilde{R}_2 = \{\, (\sigma, \sigma) \mid \sigma = \langle m, n, p\rangle,\ m = p \,\} \cup$
2. $\quad \{\, (\sigma, \sigma') \mid \sigma = \langle m, n, p\rangle,\ \sigma' = \langle m, n * m!/p!, m\rangle,\ m > p \geqslant 0 \,\} \cup$
3. $\quad \{\, (\sigma, \sigma') \mid \sigma = \langle m, n, p\rangle,\ \sigma' = \langle m, 0, m\rangle,\ m \geqslant 0 > p \,\} \cup$
4. $\quad \{\, (\sigma, \sigma') \mid \sigma = \langle m, n, p\rangle,\ \sigma' = \langle m, n * (-p - 1)!/(-m)!, m\rangle,\ 0 > m > p,\ m - p \text{ is even} \,\} \cup$
5. $\quad \{\, (\sigma, \sigma') \mid \sigma = \langle m, n, p\rangle,\ \sigma' = \langle m, -n * (-p - 1)!/(-m)!, m\rangle,\ 0 > m > p,\ m - p \text{ is odd} \,\}$

where $x!$ is the factorial function which is only defined on the natural numbers.[6] Note that we only consider $m \geqslant p$ in our conjectured $\widetilde{R}_2$, because if $m < p$, the **while-do** loop diverges.

We now check that $\widetilde{R}_2$ is indeed a fixpoint solution of ($\diamondsuit$), which turns out to be the least fixpoint solution. For the case when $m = p$ corresponding to line 1 in the definition of $\widetilde{R}_2$, it is easy to see that all pairs of the form $\big(\langle m, n, p\rangle, \langle m, n, p\rangle\big)$ are in both $\widetilde{R}_2$ and $\mathcal{F}(\widetilde{R}_2)$.

Consider next the case when $m > p \geqslant 0$, which we divide into two subcases: $m > p + 1 > 0$ and $m = p + 1 > 0$. Consider the first of these two subcases. For every $\sigma = \langle m, n, p\rangle$ with $m > p + 1 > 0$

[6]Some have extended the factorial function to other kinds of numbers (negative integers, fractional numbers, imaginary numbers). This should not be our concern here.

and every σ', we have the following sequence of equivalences:

1. $(\sigma, \sigma') \in \mathcal{F}(\widetilde{R}_2)$ iff $(\sigma, \sigma') \in \widetilde{R}_2 \circ \widetilde{R}_1$
   by right-hand side of $(\diamondsuit)$

2. iff $(\sigma, \sigma'') \in \widetilde{R}_1$ and $(\sigma'', \sigma') \in \widetilde{R}_2$ with $\sigma'' = \langle m, n * (p+1), p+1 \rangle$
   by the computed $\widetilde{R}_1$

3. iff $(\sigma, \sigma'') \in \widetilde{R}_1$ and $(\sigma'', \sigma') \in \widetilde{R}_2$ with $\sigma' = \langle m, n * (p+1) * m!/(p+1)!, m \rangle$
   by the conjectured $\widetilde{R}_2$ [*by multiplying by* $(p+1)/(p+1)$ *in* σ']

4. iff $(\sigma, \sigma'') \in \widetilde{R}_1$ and $(\sigma'', \sigma') \in \widetilde{R}_2$ with $\sigma' = \langle m, n * m!/p!, m \rangle$
   by cancelling $(p+1)/(p+1)$ in σ' [*for some* σ'']

5. iff $(\sigma, \sigma') \in \widetilde{R}_2$ with $\sigma' = \langle m, n * m!/p!, m \rangle$
   by the conjectured $\widetilde{R}_2$

We can read these equivalences top-down or bottom-up, and for each equivalence we give a reason. When reading them bottom-up, the reason for the equivalences on lines 3 and 4 are inserted in italics between square brackets.

We next consider the second subcase $m = p + 1 > 0$. For every $\sigma = \langle m, n, p \rangle$ with $m = p + 1 > 0$ and every σ', we have the following sequence of equivalences:

1. $(\sigma, \sigma') \in \mathcal{F}(\widetilde{R}_2)$ iff $(\sigma, \sigma') \in \widetilde{R}_2 \circ \widetilde{R}_1$
   by right-hand side of $(\diamondsuit)$

2. iff $(\sigma, \sigma'') \in \widetilde{R}_1$ and $(\sigma'', \sigma') \in \widetilde{R}_2$ with $\sigma'' = \langle m, n * (p+1), p+1 \rangle$
   by the computed $\widetilde{R}_1$

3. iff $(\sigma, \sigma'') \in \widetilde{R}_1$ and $(\sigma'', \sigma') \in \widetilde{R}_2$ with $\sigma' = \sigma'' = \langle m, n * (p+1), m \rangle$
   by the conjectured $\widetilde{R}_2$ when $m = p + 1$

4. iff $(\sigma, \sigma'') \in \widetilde{R}_1$ and $(\sigma'', \sigma') \in \widetilde{R}_2$ with $\sigma' = \sigma'' = \langle m, n * m!/p!, m \rangle$
   because $n * (p+1) = n * m!/p!$ when $m = p + 1$

5. iff $(\sigma, \sigma') \in \widetilde{R}_2$ with $\sigma' = \langle m, n * m!/p!, m \rangle$
   by the conjectured $\widetilde{R}_2$

We are not yet done with proving $\widetilde{R}_2 = \mathcal{F}(\widetilde{R}_2)$. We have to prove it again for the remaining cases:

(a) $m \geqslant 0 > p$, corresponding to line 3 in the definition of $\widetilde{R}_2$,
(b) $0 > m > p$ with $(m - p)$ even, corresponding to line 4 in the definition of $\widetilde{R}_2$,
(c) $0 > m > p$ with $(m - p)$ odd, corresponding to line 5 in the definition of $\widetilde{R}_2$.

This is done in a totally similar manner to the previous cases and I leave them as an exercise.

Finally, the denotation of the program fact is the composition $\widetilde{R}_2 \circ \widetilde{R}_0$. Since

$$\widetilde{R}_0 = \{ (\sigma, \sigma') \mid \sigma = \langle m, n, p \rangle,\ \sigma' = \langle m, 1, 0 \rangle \},$$

only lines 1 and 2 in the definition of $\widetilde{R}_2$ apply in the composition $\widetilde{R}_2 \circ \widetilde{R}_0$. A little computation shows:

$$[\![\mathsf{fact}]\!]_{\text{rel}} = \widetilde{R}_2 \circ \widetilde{R}_0 = \{ (\sigma, \sigma') \mid \sigma = \langle 0, n, p \rangle,\ \sigma' = \langle 0, 1, 0 \rangle \} \cup \{ (\sigma, \sigma') \mid \sigma = \langle m, n, p \rangle,\ \sigma' = \langle m, m!, m \rangle,\ m > 0 \}$$

where we simplified $n * m!/p!$ (in line 2 in the definition of $\widetilde{R}_2$) to $m!$ because $n = 1$ and $p = 0$. □

Exercise 8. Provide the details proving the equality $\widetilde{R}_2 = \mathcal{F}(\widetilde{R}_2)$ for cases (a), (b), and (c), at the end of Example 7. □

Exercise 9. Consider the following WHILE-program P over the variables $\{x, y, z\}$:

```
y := 1;
if x < z then  skip
         else  while ¬(x = z) do
                     z := z + 1;
                     y := y * z
               od
fi
```

There are two parts in this exercise:

1. Determine the denotation $[\![P]\!]_{\text{rel}} \subseteq \Sigma \times \Sigma$ where $\Sigma = \mathbb{Z}^3$.
2. Let $\psi \triangleq (x \geqslant 0) \wedge (y = x!)$. Define a *weakest pre-condition* φ which makes the Hoare triple $\{\,\varphi\,\}\ P\ \{\,\psi\,\}$ true. Justify your answer.

Hint: Observe that the **while-do** loop in the program P in this exercise is identical to the **while-do** loop in the program fact in Example 1 and Example 7. □

Example 10. The following is a very simple WHILE-program P over the single variable $\{x\}$:

```
while x > 10 do x := x + 1 od
```

There is only one variable x in the program and we can take the set of states $\Sigma = \mathbb{Z}$. We want to determine the denotation of P, $[\![P]\!]_{\text{rel}} \subseteq \mathbb{Z} \times \mathbb{Z}$. It is clear that on any input integer m assigned to x, if $m > 10$ then P diverges, and if $m \leqslant 10$ then P converges and returns the same m stored in x. Let $\widetilde{R}$ be the denotation of the body of the **while-do** loop, which is:

$$\widetilde{R} \triangleq [\![x := x + 1]\!]_{\text{rel}} = \{\, \langle m, m+1 \rangle \mid m \in \mathbb{Z} \,\}$$

The denotation $[\![P]\!]_{\text{rel}}$ of P is a fixpoint solution of the equation $R = \mathcal{F}(R)$ where:

$$\begin{aligned}\mathcal{F}(R) &\triangleq \Big\{ \langle m, n \rangle \;\Big|\; m > 10 \text{ and } \langle m, n \rangle \in R \circ \widetilde{R} \Big\} \cup \Big\{ \langle m, m \rangle \;\Big|\; m \leqslant 10 \Big\} \\ &= \Big\{ \langle m, n \rangle \;\Big|\; m > 10, \text{ there is } p \in \mathbb{Z} \text{ s.t. } \langle m, p \rangle \in \widetilde{R} \text{ and } \langle p, n \rangle \in R \Big\} \cup \Big\{ \langle m, m \rangle \;\Big|\; m \leqslant 10 \Big\}\end{aligned}$$

Claim: There are infinitely many solutions of the fixpoint equation $R = \mathcal{F}(R)$:

- $X \triangleq \{\, \langle m, m \rangle \mid m \leqslant 10 \,\}$ is a fixpoint of $R = \mathcal{F}(R)$.
- For every $k \in \mathbb{Z}$, the relation $Y_k \triangleq X \cup \{\, \langle m, k \rangle \mid m > 10 \,\}$ is a fixpoint of $R = \mathcal{F}(R)$.

We leave the verification of this claim as an exercise. Clearly, X is the *least* of these fixpoint solutions, and corresponds to the actual behavior of P. □

Exercise 11. Verify the claim at the end of Example 10. □

Example 12. Consider the following WHILE-program P over the single variable $\{x\}$:

```
while $x \geqslant 0$ do
        if  $x$ is even  then $x := x + 1$ else $x := (x + 3)/2$ fi
od
```

Because there is only one variable x in the program, we can take the set of states $\Sigma = \mathbb{Z}$. The denotation of P, which is yet to be defined, is therefore a binary relation on integers $[\![P]\!]_{\text{rel}} \subseteq \mathbb{Z} \times \mathbb{Z}$.

Call $\widetilde{R}$ the denotation of the conditional statement in P, which is also the body of the **while-do** loop. It is easy to see that, by inspection:

$$\begin{aligned}\widetilde{R} &\triangleq [\![\textbf{if } \ x \text{ is even } \ \textbf{then } x := x + 1 \textbf{ else } x := (x + 3)/2 \textbf{ fi}]\!]_{\text{rel}} \\ &= \{\, \langle m, m + 1 \rangle \mid m \text{ even} \,\} \cup \{\, \langle m, (m + 3)/2 \rangle \mid m \text{ odd} \,\}\end{aligned}$$

The denotation $[\![P]\!]_{\text{rel}}$ of the full program is a fixpoint solution of the equation $R = \mathcal{F}(R)$ where:

$$\begin{aligned}\mathcal{F}(R) &\triangleq \left\{ \langle m, n \rangle \;\middle|\; m \geqslant 0 \text{ and } \langle m, n \rangle \in R \circ \widetilde{R} \right\} \cup \left\{ \langle m, m \rangle \;\middle|\; m < 0 \right\} \\ &= \left\{ \langle m, n \rangle \;\middle|\; m \geqslant 0, \text{ there is } p \in \mathbb{Z} \text{ s.t. } \langle m, p \rangle \in \widetilde{R} \text{ and } \langle p, n \rangle \in R \right\} \cup \left\{ \langle m, m \rangle \;\middle|\; m < 0 \right\}\end{aligned}$$

Claim: There are infinitely many solutions of the fixpoint equation $R = \mathcal{F}(R)$ above, namely:

- $X \triangleq \{\, \langle m, m \rangle \mid m < 0 \,\}$ is a fixpoint of $R = \mathcal{F}(R)$.
- For every $k \in \mathbb{Z}$, the relation $Y_k \triangleq X \cup \{\, \langle m, k \rangle \mid m \geqslant 0 \,\}$ is a fixpoint of $R = \mathcal{F}(R)$.

We leave the verification of this claim as an exercise. Observe that X is the *least fixpoint solution* since $X \subseteq Y_k$ for every $k \in \mathbb{Z}$. □

Exercise 13. There are two parts:

1. Verify the claim at the end of Example 12.
2. Prove P in Example 12 converges for every $x = m < 0$ and diverges for every $x = m \geqslant 0$.

Based on these two parts, conclude that the correct denotation of P is $[\![P]\!]_{\text{rel}} = X$. □

Remark 14. Examples 6, 7, 10, and 12, show that in the presence of WHILE-loops, the denotation $[\![P]\!]_{\text{rel}} \subseteq \Sigma \times \Sigma$ of a program P is given by the least fixpoint solution of an equation of the form $R = \mathcal{F}(R)$, and in each case the *least fixpoint* rather than any fixpoint matches the actual behavior of the program. The *denotational semantics* of WHILE-programs by itself does not provide the means for preferring the least fixpoint. We need to invoke the *operational semantics* of WHILE-programs, in any of its several variants, in order to justify that the least fixpoint is the one to choose, *i.e.*, the one that matches the semantics obtained by using an operational approach. □

1.4 Soundness and Completeness of Classical HL

A first-order formula φ of arithmetic is a first-order formula over the signature of arithmetic, *i.e.*, φ uses constant symbols, function symbols, and relation symbols of arithmetic $\{0, 1, +, -, \times, \ldots, \leqslant, \ldots\}$.

In this handout, a state is a map from the set $\mathcal{V}$ of variables to the set $\mathbb{Z}$ of integers, and Σ is the set of all states. We write $[\![\varphi]\!]$ for the set of states satisfying φ:

$$[\![\varphi]\!] \triangleq \{\, \sigma \in \Sigma \mid \sigma \models \varphi \}$$

In general, $[\![\varphi]\!] \subseteq \Sigma$. If $[\![\varphi]\!] = \Sigma$, we say that φ is always *true*, *i.e.*, satisfied by every state.

Definition 15 (*Valid Hoare Triples*)**.** The Hoare triple $\{\,\varphi\,\}\ C\ \{\,\psi\,\}$ is *true* (*i.e.*, *valid*), written $\models \{\,\varphi\,\}\ C\ \{\,\psi\,\}$, iff for all states $\sigma, \sigma' \in \Sigma$ it holds that if $\sigma \in [\![\varphi]\!]$ and $(\sigma, \sigma') \in [\![C]\!]$ then $\sigma' \in [\![\psi]\!]$. $\square$

A minimum requirement for a proof system, the set of axioms and inference rules for deriving formulas, is that it be *sound*. This property is satisfied by the proof system presented in Section 1.2.

Theorem 16 (Soundness of Classical HL)**.** *Let $\{\,\varphi\,\}\ C\ \{\,\psi\,\}$ be a Hoare triple. If $\vdash \{\,\varphi\,\}\ C\ \{\,\psi\,\}$ then $\models \{\,\varphi\,\}\ C\ \{\,\psi\,\}$.*

Proofs for Theorem 16 are in several standard textbooks, in particular in [16, 12].

We say a proof system is *effective* if it can be automated, *i.e.*, its axioms and inference rules can be used mechanically to derive formulas by strict pattern matching of syntactic expressions. In this sense, the proof system in Section 1.2 is not effective, the culprit being the rule '[weakening]' which mentions among its premises the validity (not the formal derivability) of two first-order formulas, namely $\models \varphi' \to \varphi$ and $\models \psi \to \psi'$.

The completeness of a proof system is the converse implication: Every true formula can be formally derived by the proof system. It turns out that a proof system for Hoare Logic cannot be complete in this absolute sense, if the proof system is to be effective.

Gödel's Incompleteness Theorem says there is no effective proof system for first-order formulas of arithmetic, *i.e.*, a proof system that can formally derive all the first-order formulas that are true in the standard model of arithmetic, whose universe is the set $\mathbb{N}$ of natural numbers, not $\mathbb{Z}$. However, this result implies there is no effective proof system that can formally derive all the first-order formulas that are true in the model whose universe is $\mathbb{Z}$ equipped with the usual operations of arithmetic. From this, deep consequences follow for Hoare Logic.

Proposition 17. *There is no effective proof system for Hoare triples, in the sense that the set of all formally derivable Hoare triples coincide with the set of all true Hoare triples.*

Proof. There are two different ways of proving this result. First, for an arbitrary first-order formula ψ of arithmetic, we have that ψ is true iff the Hoare triple $\{\,\texttt{true}\,\}\ \texttt{skip}\ \{\,\psi\,\}$ is true, by our definitions above. We can thus reduce the existence of an effective proof system for Hoare triples to the existence of an effective proof system for first-order arithmetic. The latter is impossible, by Gödel's Incompleteness Theorem, which implies the desired result.

The second way is to consider all Hoare triples of the form $\{\,\texttt{true}\,\}\ P\ \{\,\texttt{false}\,\}$ where P ranges over all While-programs. Such a triple is true iff P diverges for all input states. If we had an effective proof system for Hoare triples, we would have a computable method for deciding that a While-program P diverges on all input states. The undecidability of the Halting Problem says that this is not possible. $\square$

Inspite of the preceding proposition, we do have a *relative completeness* result. This means that if we have an oracle to decide the truth of formulas of first-order arithmetic – specifically, the truth of the premises $\models \varphi' \to \varphi$ and $\models \psi \to \psi'$ in the rule '[weakening]' – then the proof system in Section 1.2 is *complete relative to this oracle.*

Theorem 18 (Relative Completeness of Classical HL)**.** *Let* $\{\,\varphi\,\}\ C\ \{\,\psi\,\}$ *be a Hoare triple. If we have* $\models \{\,\varphi\,\}\ C\ \{\,\psi\,\}$*, then we also have* $\vdash \{\,\varphi\,\}\ C\ \{\,\psi\,\}$ *by the rules of Section 1.2.*

Once again, keep in mind that the proof system in Section 1.2 is *not* effective – or is effective relative to the existence of an oracle that decides the truth of any first-order formula of arithmetic. Put differently still, the proof system in Section 1.2 is complete if we take all first-order formulas of arithmetic that are true as *axioms*, *i.e.*, formulas whose truth we take for granted and do not need to formally establish.

Just as for Theorem 16, proofs for Theorem 18 are in several textbooks, in particular in [16, 12].

1.5 Extensions of Classical Hoare Logic

Several useful extensions of Classical HL have been proposed and studied over the years. Among those are the following, which will be discussed in lecture and the homework exercises. They require each careful adaptation or extension of the proof rules of Section 1.2 and formal semantics of Section 1.3:

- Straightforward extensions of the WHILE Language: **for**-*loops*, **repeat-until**-*loops*, *arrays.*
- Non-trivial, but still easy, extensions of the WHILE Language: *concurrency*, *non-determinism.*

There are other important extensions, separate from the preceding and from those later in this handout, which are outside the scope of CS 512 this semester. Among these I include:

- Non-trivial and tricky extension of the WHILE Language: *procedure calls.*
- *Separation logic*, an extension of Hoare Logic to reason about *shared mutable data structures.*

Separation Logic is relatively new, starting in the early 2000's. Work on WHILE programs augmented with *procedure calls* goes back to the early years of Hoare Logic, as they come with many different variations – depending on many features (recursive or non-recursive, with or without global variables, with or without parameters, call-by-value or call-by-result or call-by-value-result, etc.).

2 Relational Hoare Logic (RHL)

RHL is a very simple variation on classical Hoare Logic. A judgement of classical HL asserts something about a single command (or a single program). A judgement of RHL asserts something about two commands (or two programs).

In the case of classical HL, we deal with assertions that denote predicates on states (this is what *pre-conditions* and *post-conditions* are), and judgements that say that a command (or a program) terminating in a state satisfying a pre-condition will yield a state satisfying a post-condition (this is what a *Hoare triple* says). In the case of RHL, we compare two commands (or two programs) according to whether they map a given *pre-relation* into a given *post-relation.* Pre-relations and post-relations are binary relations on pairs of states which are here expressed as quantifier-free formulas of first-order logic, over variables tagged with $\langle 1\rangle$ or $\langle 2\rangle$, if need be, to indicate which of the two states in a pair they refer to. Other more advanced accounts of RHL allow quantifiers in the pre-relations and post-relations, although always in restricted ways so as not to encounter some of the intractable (or even undecidable) questions of full first-order logic.

But what justifies the invention of RHL as another logic of programs? One obvious reason is that it is common to specify a program by its relationship to another program. For example, when a compiler optimizes an input program, the optimized program and the original program must be equivalent. For another example, consider a client of an abstract data type which has two different implementations; we may want to specify that a client is insensitive to the choice of the implementation, or that a client with one implementation is (observationally) equivalent to another client with the other implementation. There will be further justification after we introduce pHL in Section 4, and later combine pHL and RHL to obtain pRHL in Section 5.

Classical Hoare Logic does not provide the means for specifying how two programs are related, at least directly. Hoare triples $\{\,\varphi\,\}\;P\;\{\,\psi\,\}$ are good for specifying the input-output relation of a single command (or a single program), but not for the equivalence between two programs – although there are roundabout ways of using Hoare triples for analyzing program equivalence under some restrictions.

Relational Hoare Logic was precisely invented to compensate for this lack or weakness. The central concept in RHL is what we may call a *Hoare quadruple*, which is written as:

$$\{\,\Phi\,\}\;C_1 \sim C_2\;\{\,\Psi\,\}$$

where C_1 and C_2 are commands or programs (here in the WHILE language), and Φ and Ψ are binary relations on states.[7] Informally, the intended meaning of such a quadruple is the following:

When executions of C_1 and C_2 are started from Φ-related states,
either they both diverge or they both terminate in Ψ-related states.

Various qualifications can be added to this informal meaning; one such qualification is to require that, during execution, both C_1 and C_2 access only memory cells (or variables) that the pre-relation Φ guarantees to exist. Below are a few simple examples to make some of these ideas more concrete.

[7]Not all authors have adopted the same style in writing Hoare quadruples. The researcher who first introduced RHL wrote $C_1 \sim C_2 : \Phi \Rightarrow \Psi$, see [6]. Others have written a Hoare quadruple as $\Phi\,{}^{C_1}_{C_2}\,\Psi$, *e.g.*, in [17].

Also, some write 'pre-condition' and 'post-condition' where we write 'pre-relation' and 'post-relation', respectively. We prefer to keep these appellations separate, with the former used in the context of classical HL and the latter used in RHL. We try to use lower-case Greek letters $\varphi, \psi, \ldots$ to name 'pre-conditions' and 'post-conditions', and upper-case Greek letters $\Phi, \Psi, \ldots$ to name 'pre-relations' and 'post-relations'.

Example 19. Below are two tiny program phrases, P_1 and P_2, each consisting of two instructions over the same set of variables $\{x, y, z\}$:

```
$P_1$                    $P_2$
$y := x + 1;$            $z := x + 2;$
$z := y + 1;$            $y := z - 1;$
```

A state here is an assignment of integers $\langle m, n, p\rangle \in \mathbb{Z}^3$ to $\langle x, y, z\rangle$. For $i = 1, 2$, we view P_i as the code for a *state transformer*, *i.e.*, the code for a function $[\![P_i]\!]$ from $\mathbb{Z}^3$ to $\mathbb{Z}^3$.

We want to write a Hoare quadruple asserting that P_1 and P_2 are equivalent. More precisely, since the values of y and z in the initial state do not affect the final state, we want to write a Hoare quadruple asserting that if P_1 and P_2 are started at initial states whose x-components are equal, then P_1 and P_2 stop in final states that are equal. Our proposed Hoare quadruple is:

$$\{\,\Phi\,\}\; P_1 \sim P_2\; \{\,\Psi\,\} \quad \text{where} \quad \begin{aligned}\Phi &\triangleq (x\langle 1\rangle = x\langle 2\rangle) \quad \text{and}\\ \Psi &\triangleq \Big([\![P_1]\!]\big\langle x\langle 1\rangle, y\langle 1\rangle, z\langle 1\rangle\big\rangle = [\![P_2]\!]\big\langle x\langle 2\rangle, y\langle 2\rangle, z\langle 2\rangle\big\rangle\Big)\end{aligned}$$

In the pre-relation and post-relation Φ and Ψ,[8] we tagged the variables with $\langle i\rangle$ as in $\langle x\langle i\rangle, y\langle i\rangle, z\langle i\rangle\rangle$ to distinguish the state on which $[\![P_i]\!]$ operates, for $i = 1, 2$. You should understand this Hoare quadruple as saying: *When executions of P_1 and P_2 start from states whose x-components are equal, either they both diverge or they return states whose respective x-, y-, and z-components are equal.* □

If the programs P_1 and P_2 that we want to compare use disjoint sets of variables, there is no need to tag their respective variables with $\langle 1\rangle$ and $\langle 2\rangle$ in the pre-relation Φ and post-relation Ψ. In such a case, we say that P_1 and P_2 are *separable*. The resulting syntax of Hoare quadruples is somewhat lighter and easier to read, as illustrated by the next example.

Example 20. Below are two program phrases, P_1 and P_2, the first over variables $\{k, n, x, y\}$ and the second over variables $\{k', n', x', y'\}$, *i.e.*, P_1 and P_2 are *separable*:

```
$P_1$                        $P_2$
$k := 0;$                    $k' := 0;$
while $k < n$ do             $x' := y' + 1;$
    $x := y + 1;$            while $k' < n'$ do
    $k := k + x;$                $k' := k' + x';$
od                           od
```

P_2 is obtained from P_1 by a typical form of compiler optimization, *invariant hoisting*: in this case, the invariant instruction '$x := y + 1$' is taken out of the loop. We have renamed the variables in P_2 as $\{k', n', x', y'\}$ to avoid using the tags $\langle 1\rangle$ and $\langle 2\rangle$ in the Hoare quadruple, which we can write as:

$$\{\,\Phi\,\}\; P_1 \sim P_2\; \{\,\Psi\,\} \quad \text{where} \quad \begin{aligned}\Phi &\triangleq \big((n = n') \wedge (y = y')\big) \quad \text{and}\\ \Psi &\triangleq \big((k = k') \wedge (n = n') \wedge (x = x') \wedge (y = y')\big)\end{aligned}$$

The pre-relation Φ only requires that the n-component and the y-component of P_1's initial state and P_2's initial state be the same, because the initial values of the other two variables $\{k, x\}$ have no effect on the final state. The post-relation Ψ requires that P_1 and P_2 return the same final state. □

[8] We cheated in the way Ψ is written! Ψ is supposed to be a first-order formula, and first-order logic does not allow interpreted functions, here $[\![P_1]\!]$ and $[\![P_2]\!]$, to appear in well-formed formulas. Our lapsus in defining Ψ is to make explicit that Ψ is to be satisfied *after* P_1 and P_2 terminate. It suffices to define $\Psi \triangleq \big((x\langle 1\rangle = x\langle 2\rangle) \wedge (y\langle 1\rangle = y\langle 2\rangle) \wedge (z\langle 1\rangle = z\langle 2\rangle)\big)$.

In general, a Hoare quadruple $\{\,\Phi\,\}\ C_1 \sim C_2\ \{\,\Psi\,\}$ asserts a relationship between distinct C_1 and C_2. A special case is when C_1 and C_2 are the same, as illustrated in the next example.

Example 21. Let C be the two instructions in sequence:

$$x := 5; \quad y := 8;$$

which can be part of a larger program that uses variables $\{x, y, z\}$. We may write the Hoare quadruple:

$$\Big\{\,\underbrace{z\langle 1\rangle = z\langle 2\rangle}_{\Phi}\,\Big\}\ C \sim C\ \Big\{\,\underbrace{(x\langle 1\rangle = x\langle 2\rangle = 5)\ \wedge\ (y\langle 1\rangle = y\langle 2\rangle = 8)\ \wedge\ (z\langle 1\rangle = z\langle 2\rangle)}_{\Psi}\,\Big\}$$

which is indeed true, after a moment of thought. □

The next example illustrates how to translate Hoare triples of classical HL into Hoare quadruples of RHL and, thus, how to view the latter as an extension of the former.

Example 22. This is a continuation of Example 1, where we defined WHILE-program fact and the Hoare triple $\{\,\varphi\,\}\ \mathsf{fact}\ \{\,\psi\,\}$ where $\varphi \triangleq (x \geqslant 0)$ and $\psi \triangleq (y = x!)$. The program fact is defined over the variables $\{x, y, z\}$. As a state-transformer, fact defines a function $[\![\mathsf{fact}]\!]$ from $\mathbb{Z}^3$ to $\mathbb{Z}^3$. Consider now the Hoare quadruple:

$$\{\,\Phi\,\}\ \mathsf{fact} \sim \mathsf{fact}\ \{\,\Psi\,\} \quad \text{where} \quad \begin{aligned} \Phi &\triangleq \Big(\big(x\langle 1\rangle \geqslant 0\big) \wedge \big(x\langle 2\rangle \geqslant 0\big)\Big) \quad \text{and} \\ \Psi &\triangleq \Big(\big(y\langle 1\rangle = x\langle 1\rangle!\big) \wedge \big(y\langle 2\rangle = x\langle 2\rangle!\big)\Big) \end{aligned}$$

where the pre-relation Φ is obtained from the pre-condition φ by tagging the variables in φ with $\langle 1\rangle$ and $\langle 2\rangle$, and the post-relation Ψ is obtained from the post-condition ψ by tagging the variables in ψ with $\langle 1\rangle$ and $\langle 2\rangle$. The tag $\langle 1\rangle$ qualifies the variables that the copy of fact on the *left* of '$\sim$' acts on, and the tag $\langle 2\rangle$ qualifies the variables that the copy of fact on the *right* of '$\sim$' acts on.

On reflection, it is easy to see that the Hoare triple $\{\,\varphi\}\ \mathsf{fact}\ \{\,\psi\,\}$ is true iff the Hoare quadruple $\{\,\Phi\,\}\ \mathsf{fact} \sim \mathsf{fact}\ \{\,\Psi\,\}$ is true. □

2.1 Formal Proof Rules of RHL

To facilitate our presentation, we henceforth assume that, in a Hoare quadruple $\{\,\Phi\,\}\ C_1 \sim C_2\ \{\,\Psi\,\}$, the commands or programs C_1 and C_2 are *separable*, in the sense explained before Example 20. If need be, we rename variables *ad lib* in our presentation below in order to keep the two programs *separable*, although in actual implementations of the rules such renaming may be unwieldy and tricky.

Relational Hoare Logic consists of Hoare quadruples, by which we specify how two programs are related, and the axioms and inference rules for deriving valid quadruples. We start with a version which has been called *Minimal Relational Hoare Logic* (Minimal RHL) [2]. We use the same notational conventions that are used in Section 1.2 for Classical HL.

$$\vdash \{\,\Phi\,\}\ \mathsf{skip} \sim \mathsf{skip}\ \{\,\Phi\,\} \qquad \text{[skip]}$$

$$\vdash \{\,\Psi[x_1 \mapsto E_1][x_2 \mapsto E_2]\,\}\ \ x_1 := E_2 \sim x_2 := E_2\ \ \{\,\Psi\,\} \qquad \text{[assignment]}$$

$$\frac{\vdash \{\,\Phi\,\}\ C_1 \sim C_2\ \{\,\Theta\,\} \qquad \vdash \{\,\Theta\,\}\ C_1' \sim C_2'\ \{\,\Psi\,\}}{\vdash \{\,\Phi\,\}\ C_1; C_1' \sim C_2; C_2'\ \{\,\Psi\,\}} \qquad \text{[sequencing]}$$

$$\frac{\models \Phi \to (B_1 = B_2) \qquad \vdash \{\,\Phi \wedge B_1\,\}\ C_1 \sim C_2\ \{\,\Psi\,\} \qquad \vdash \{\,\Phi \wedge \neg B_1\,\}\ C_1' \sim C_2'\ \{\,\Psi\,\}}{\vdash \{\,\Phi\,\}\ \mathbf{if}\ B_1\ \mathbf{then}\ C_1\ \mathbf{else}\ C_1'\ \mathbf{fi} \sim \mathbf{if}\ B_2\ \mathbf{then}\ C_2\ \mathbf{else}\ C_2'\ \mathbf{fi}\ \{\,\Psi\,\}} \qquad \text{[conditional]}$$

$$\frac{\vdash \{\,\Phi \wedge B_1\,\}\ C_1 \sim C_2\ \{\,\Phi\,\} \qquad \models \Phi \to (B_1 = B_2)}{\vdash \{\,\Phi\,\}\ \mathbf{while}\ B_1\ \mathbf{do}\ C_1\ \mathbf{od} \sim \mathbf{while}\ B_2\ \mathbf{do}\ C_2\ \mathbf{od}\ \{\,\Phi\,\}} \qquad \text{[while]}$$

$$\frac{\models \Phi' \to \Phi \qquad \vdash \{\,\Phi\,\}\ C_1 \sim C_2\ \{\,\Psi\,\} \qquad \models \Psi \to \Psi'}{\vdash \{\,\Phi'\,\}\ C_1 \sim C_2\ \{\,\Psi'\,\}} \qquad \text{[weakening]}$$

Figure 2: Inference rules of Minimal RHL.

Minimal RHL requires that both commands, C_1 and C_2, in a Hoare quadruple $\{\,\Phi\,\}\ C_1 \sim C_2\ \{\,\Psi\,\}$ execute in lockstep and that both must have the same shape. As a result, it is not possible to derive a Hoare quadruple as simple as $\{\,\Phi\,\}\ C; \texttt{skip} \sim C\ \{\,\Psi\,\}$. To obviate this weakness, we can extend Minimal RHL to what has been called *Core Relational Hoare Logic* (Core RHL) [2], which introduces rules that allow for the separate analysis of the C_1 and C_2. The extra rules for Core RHL are shown in Figure 3.

$$\frac{\vdash \{\,\Phi[x \mapsto E]\,\}\ \ \texttt{skip} \sim C\ \ \{\,\Psi\,\}}{\vdash \{\,\Phi\,\}\ \ x := E \sim C\ \ \{\,\Psi\,\}} \qquad \text{[assignment-L]}$$

$$\frac{\vdash \{\,\Phi[x \mapsto E]\,\}\ \ C \sim \texttt{skip}\ \ \{\,\Psi\,\}}{\vdash \{\,\Phi\,\}\ \ C \sim x := E\ \ \{\,\Psi\,\}} \qquad \text{[assignment-R]}$$

$$\frac{\vdash \{\,\Phi \wedge B\,\}\ \ C_1 \sim C\ \ \{\,\Psi\,\} \qquad \vdash \{\,\Phi \wedge \neg B\,\}\ \ C_2 \sim C\ \ \{\,\Psi\,\}}{\vdash \{\,\Phi\,\}\ \ \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi} \sim C\ \ \{\,\Psi\,\}} \qquad \text{[conditional-L]}$$

$$\frac{\vdash \{\,\Phi \wedge B\,\}\ \ C \sim C_1\ \ \{\,\Psi\,\} \qquad \vdash \{\,\Phi \wedge \neg B\,\}\ \ C \sim C_2\ \ \{\,\Psi\,\}}{\vdash \{\,\Phi\,\}\ \ C \sim \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}\ \ \{\,\Psi\,\}} \qquad \text{[conditional-R]}$$

Figure 3: Additional inference rules for Core RHL.

One more rule for RHL is shown in Figure 4, called [self-composition], which is added to the rules of Core RHL to obtain what I call *Extended Core Relational Hoare Logic* (Extended Core RHL). The justification for [self-composition] is not immediately obvious; we come back to this when we discuss the formal semantics of RHL.[9]

Observe carefully that the premise in [self-composition] is a Hoare triple $\{\,\Phi\,\}\ C_1; C_2\ \{\,\Psi\,\}$, not a Hoare quadruple. In the comments preceding Examples 21 and 22, we explain how to view a Hoare triple as

[9]The idea of 'self-composition' was first introduced in [3, 4]. The rule [self-composition] here is an adaptation and formalization of that idea in [2].

a Hoare quadruple, by using the tags $\langle 1\rangle$ and $\langle 2\rangle$. Specifically here, to avoid the use of tags, we can view the triple $\{\,\Phi\,\}\; C_1; C_2\; \{\,\Psi\,\}$ as the quadruple:

$$\{\,\Phi \wedge \Phi'\,\}\;\; C_1; C_2 \sim C_1'; C_2'\;\; \{\,\Psi \wedge \Psi'\,\}$$

after appropriate renaming of variables to keep the two copies of the program, $C_1; C_2$ and $C_1'; C_2'$, separable and where Φ' and Ψ' are Φ and Ψ after this renaming of variables, respectively.

$$\frac{\vdash \{\,\Phi\,\}\;\; C_1; C_2\;\; \{\,\Psi\,\}}{\vdash \{\,\Phi\,\}\;\; C_1 \sim C_2\;\; \{\,\Psi\,\}} \qquad \text{[self-composition]}$$

Figure 4: One more rule for Extended Core RHL.

2.2 Formal Semantics of RHL

3 Notions of Probability

4 Probabilistic Hoare Logic (pHL)

4.1 A Probabilistic Imperative Programming Language: pWhile

Starting from a given initial state (*i.e.*, contents of memory) an imperative program returns at most one state. The returned state is entirely determined by the program and the initial state. This is no longer the case when we deal with probabilistic programs. Running the same probabilistic program several times, starting with the same initial state, the returned states may be different. The distribution of returned states can be represented by a random variable, whose value is a probability distribution over the set of possible states.

An example of a randomized expression is $x + \mathsf{random}(10)$ where the subexpression $\mathsf{random}(10)$ returns with uniform distribution an integer $k \in \{0, \dots, 10\}$. Another example is $(x < y) \vee \mathsf{flip}$ where the subexpression flip returns the Boolean `true` or `false`, each with probability $1/2$ (in the case of a fair coin). If the expression $(x < y)$ evaluates to `true`, then $(x < y) \vee \mathsf{flip}$ returns `true` with probability $= 1$; and if $(x < y)$ evaluates to `false`, then $(x < y) \vee \mathsf{flip}$ returns `true` with probability $= 1/2$.

Randomization may be also introduced by using a *probabilistic choice* between two deterministic instructions; for example, if C is the following probabilistic choice:

$$C \;\triangleq\; \Big(\, (x := 1) \oplus_{1/4} (x := 5) \,\Big)$$

then starting C from the state $\langle w, x, y, z\rangle = \langle 30, 30, 30, 30\rangle$ returns the state $\langle w, x, y, z\rangle = \langle 30, 1, 30, 30\rangle$ with probability $= 1/4$ and the state $\langle w, x, y, z\rangle = \langle 30, 5, 30, 30\rangle$ with probability $= 3/4$.

A remark is in order regarding: *randomized* versus *non-deterministic.* These correspond to two forms of *possible event* or *possible outcome* in the execution of programs, which have been called the *non-deterministic form* and the *probabilistic form.* In the former, events are either possible or impossible, with no further distinction. In the latter, events occur according to a probability distribution. It is tempting to equate 'possible' in the *non-deterministic form* with 'nonzero probability' in the *probabilistic form*, but this correspondence goes only so far. This is illustrated in the next example, which uses the randomized prim op flip.

Example 23. The following is a pWhile program, call it HeadsOrTails, where flip is a randomized primitive operator which returns the Boolean `true` with probability p, and the Boolean `false` with probability $1 - p$, where $0 < p < 1$:

```
heads := true;
tails := false;
x := flip;
while  x = heads  do  x := flip  od
```

HeadsOrTails always terminates or, more precisely, terminates with probability $= 1$ because the probability that flip always returns `true` and that the program does not terminate is $\lim_{n\to\infty} p^n = 0$. However, non-deterministically, HeadsOrTails does not always terminate, since one possible execution path is indeed infinite, when flip is allowed to always return `true`. □

The next example gives an idea of how we may want to write a Hoare triple in the presence of probabilistic behavior. The example uses the prim op random defined in the opening paragraph of this section.

Example 24. Let C be the single instruction $y := x + 2$. A Hoare triple in standard HL may be:

$$\{\, x = 1 \,\}\; C \;\{\, y = 3 \,\}$$

which asserts that if an initial state maps x to 1 and we start execution of C from that state, then the resulting state maps y to 3, assuming this execution terminates (which it obviously does!). Let now C' be the single instruction $y := x + \mathsf{random}(2)$. A Hoare triple in the logic pHL, which is yet to be defined, may be written as:

$$\Big\{\, \mathtt{Pr}\big(x = 1\big) \geqslant 3/4 \Big\}\; C' \;\Big\{\, \mathtt{Pr}\big(y = 3\big) \geqslant 1/4 \Big\}$$

which asserts that if an initial state maps x to 1 with probability $\geqslant 3/4$, and we start execution of C' from that state, then the resulting state maps y to 3 with probability $\geqslant 1/4$, assuming this execution terminates. A moment of thought shows that this Hoare triple of pHL is valid. $\square$

A simple but less trivial example of a randomized program follows. To understand its input-output behavior requires a careful probability analysis.

Example 25. The following is a pWhile program, call it factA, a variation on While program fact in Example 1:

```
$y := 1;$
$z := 0;$
while flip do             // substitute ‘flip’ for ‘$\neg(x = z)$’ in program fact in Example 1
    $z := z + 1;$
    $y := y * z;$
od
```

This program is simple enough that we can analyze it by inspection. Let $\langle x, y, z\rangle = \langle k, \ell, m\rangle$ be the initial state and $\langle x, y, z\rangle = \langle k', \ell', m'\rangle$ be the final state, right before and right after the execution of factA, respectively, where $k, \ell, m, k', \ell', m' \in \mathbb{Z}$.

First, note $k = k'$, since variable x is not updated at any step of the execution. Moreover, the initial $\{\ell, m\}$ have no effect on the final $\{\ell', m'\}$. The latter are not uniquely determined and obey a probability distribution. The loop in factA is executed as many times as the primitive operator flip returns `true` in consecutive iterations before flip returns `false` for the first time. If the loop is iterated $n \geqslant 0$ times, this means the first n Booleans returned by flip are `true` and the $(n + 1)$-st is `false`.

The table below shows the possible values of the final state $\langle k', \ell', m'\rangle$ in the first column, the probability with which each final state occurs in the second column, and the corresponding number n of loop iterations in the third column.

$\langle k', \ell', m'\rangle$	probability	n (number of loop iterations)
$\langle k, 1, 0\rangle$	$1/2^1$	0
$\langle k, 1, 1\rangle$	$1/2^2$	1
$\langle k, 2, 2\rangle$	$1/2^3$	2
$\langle k, 6, 3\rangle$	$1/2^4$	3
$\cdots$	$\cdots$	$\cdots$
$\langle k, i!, i\rangle$	$1/2^{(i+1)}$	i
$\cdots$	$\cdots$	$\cdots$

To compute the probabilities in the second column above we can use a simple inductive reasoning:

- The first `flip` is `false` with probability 1/2 (inducing 0 loop iterations) and `true` with probability 1/2 (inducing 1 or more loop iterations).
- Assuming that the first `flip` is `true` (an outcome which occurs with probability 1/2), the second `flip` is `false` with probability 1/2 (inducing 0 loop iterations beyond the first) and `true` with probability 1/2 (inducing 1 or more loop iterations beyond the first).
- Assuming that the first and second `flip` are `true` (an outcome which occurs with probability $1/2^2$), the third `flip` is `false` with probability 1/2 (inducing 0 loop iterations beyond the first and second), and `true` with probability 1/2 (inducing 1 or more loop iterations beyond the first and second).
- More generally, assuming that the first $i \geqslant 1$ values of `flip` are `true` (an outcome which occurs with probability $1/2^i$), the $(i+1)$-st `flip` is `false` with probability 1/2 (inducing 0 loop iterations beyond the first i iterations), and `true` with probability 1/2 (inducing 1 or more loop iterations beyond the first i iterations).

If our reasoning is correct, then the sum of the probabilities in the second column should add to 1, corresponding to a (full) probability distribution among all possible outcomes. So, here, we need to verify that the sum S of the reciprocals of powers of 2 is 1:

$$S = \sum_{i \geqslant 1} 1/2^i = 1.$$

This is indeed the case.[10] □

Definition 26 (*Syntax of Probabilistic* WHILE *Programs*)**.** This extends the syntax of WHILE-programs as given in Definition 3. There are basically two approaches to doing this.

Approach 1: This uses the definitions of *integer expressions* E and *Boolean expressions* B exactly as given in Definition 3, and then extends the BNF for *commands* to include an additional case:

$$C ::= \texttt{skip} \;\big|\; x := E \;\big|\; C_1 ; C_2 \;\big|\; \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi} \;\big|\; \textbf{while } B \textbf{ do } C \textbf{ od} \;\big|\; C_1 \oplus_\rho C_2$$

The new case is the *probabilistic choice* $C_1 \oplus_\rho C_2$ where ρ is a probability in the open interval $(0,1)$. The interpretation of $C_1 \oplus_\rho C_2$ is a probabilistic decision resulting in the execution of C_1 with probability ρ and the execution of C_2 with probability $1-\rho$. This is the approach in [7, 8, 9], among other reports.

Approach 2: An alternative is to introduce randomized primitive operators and extend the syntax of *integer expressions* and *Boolean expressions* accordingly. For example, this approach adds a case to the BNF for *integer expressions*:

$$E ::= n \;\big|\; x \;\big|\; E_1 + E_2 \;\big|\; E_1 - E_2 \;\big|\; E_1 * E_2 \;\big|\; \cdots \;\big|\; \texttt{iop}_{\,k}(E_1, \ldots, E_k)$$

The new case in the preceding BNF is $\texttt{iop}_{\,k}(E_1, \ldots, E_k)$ where $\texttt{iop}_{\,k}$ is a randomized primitive operator of arity $k \geqslant 0$. An example of such a prim op $\texttt{iop}_{\,k}$ when $k = 1$ is random in Example 24.

[10] There are different ways of proving this. One way is to observe that S is the sum of an infinite geometric series with a common ratio less than 1, and then use the formula for such a sum. Another way is to define $S_j \triangleq \sum_{1 \leqslant i \leqslant j} 1/2^i$ first, for every $j \geqslant 1$, so that:

$$2^j \cdot S_j = 2^j \Big(\sum_{1 \leqslant i \leqslant j} 1/2^i \Big) = \sum_{1 \leqslant i \leqslant j} 2^j/2^i = \sum_{1 \leqslant i \leqslant j} 2^{j-i} = \sum_{1 \leqslant i \leqslant j-1} 2^i = 2^j.$$

Hence, $2^j \cdot S_j = 2^j$, which implies $S_j = 1$ for every $j \geqslant 1$, so that $S = \lim_{j\to\infty} S_j = 1$.

Similarly, a new case may be added to the BNF of *Boolean expressions* involving randomized primitive operators denoted $\texttt{bop}_{\ell}$, each with its own arity $\ell \geqslant 0$. An example of a prim op $\texttt{bop}_{\ell}$ with $\ell = 0$ is flip in Examples 23 and 25. The second approach is adopted in [1, 5, 11, 13, 14], among other reports.

Each of these two approaches involves somewhat different technical issues, and each can simulate at least parts of the other (also depending on the kind of randomized prim ops that are available). And, of course, it is also possible to combine both approaches. $\square$

5 Probabilistic Relational Hoare Logic (pRHL)

A judgment of RHL has the form:

$$\vdash \{\,\Phi\,\}\; C_1 \sim C_2 \;\{\,\Psi\,\}$$

where C_1 and C_2 are commands (or program phrases) in the language of While programs, and Φ and Ψ are relations on states (or contents of memories).

A judgment of pRHL has exactly the same form, except that C_1 and C_2 are now in the language of pWhile programs. Evaluation of a pWhile command w.r.t. an initial state returns a sub-distribution over states. Hence, giving a meaning to a pRHL judgment requires interpreting post-relations over sub-distributions. To this end, pRHL relies on a lifting operator $\mathcal{L}$ which transforms a *binary relation on states* into a *binary relation on sub-distributions* over states.

Lifting can be used to define validity of a pRHL judgment: For any two pWhile commands C_1 and C_2, and binary relations Φ and Ψ on states, the judgment $\vdash \{\,\Phi\,\}\; C_1 \sim C_2 \;\{\,\Psi\,\}$ is valid if for every pair of states s_1 and s_2, it holds that $s_1 \Phi s_2$ implies $(\llbracket C_1 \rrbracket s_1)\; \mathcal{L}(\Psi)\; (\llbracket C_2 \rrbracket s_2)$.

References

[1] Gilles Barthe, Juan Manuel Crespo, Benjamin Grégoire, César Kunz, and Santiago Zanella Béguelin. Computer-Aided Cryptographic Proofs. In L. Beringer and A. Felty, editors, *Interactive Theorem Proving*, pages 11–27. Springer, 2012.

[2] Gilles Barthe, Juan Manuel Crespo, and Csar Kunz. Product Programs and Relational Program Logics. *Journal of Logical and Algebraic Methods in Programming*, 85(5 Part 2):847–859, 2016.

[3] Gilles Barthe, Pedro R. D'argenio, and Tamara Rezk. Secure Information Flow by Self-composition. In R. Foccardi, editor, *Proc. 17th IEEE Computer Security Foundations Workshop*, pages 100–114. IEEE Press, 2004.

[4] Gilles Barthe, Pedro R. D'argenio, and Tamara Rezk. Secure Information Flow by Self-composition. *Mathematical Structures in Comp. Sci.*, 21(6):1207–1252, December 2011.

[5] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Probabilistic Relational Hoare Logics for Computer-Aided Security Proofs. In J. Gibbons and P. Nogueira, editors, *Mathematics of Program Construction*, pages 1–6. Springer, 2012.

[6] Nick Benton. Simple Relational Correctness Proofs for Static Analyses and Program Transformations. In *Proceedings of 31st ACM Symposium on Principles of Programming Languages*, pages 14–25, New York, USA, 2004. ACM.

[7] Ricardo Corin and Jerry den Hartog. A Probabilistic Hoare-style Logic for Game-based Cryptographic Proofs. In *Proceedings of 33rd Int'l Conf. on Automata, Languages and Programming*, pages 252–263. Springer-Verlag, 2006.

[8] J. I. den Hartog and E. P. de Vink. Verifying Probabilistic Programs Using a Hoare Like Logic. *Int'l Journal of Foundations of Computer Science*, 13(03):315–340, 2002.

[9] J. I. den Hartog. Verifying Probabilistic Programs Using a Hoare Like Logic. In *Proc. 5th Asian Computing Science Conference on Advances in Computing Science*, pages 113–125. Springer-Verlag, 1999.

[10] Michael Huth and Mark Ryan. *Logic in Computer Science.* Cambridge University Press, 2011.

[11] Dexter Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3):328 – 350, 1981.

[12] John C. Mitchell. *Foundations for Programming Languages.* MIT Press, Cambridge, MA, USA, 1996.

[13] Robert Rand and Steve Zdancewic. VPHL: A Verified Partial-Correctness Logic for Probabilistic Programs. *Electronic Notes in Theoretical Computer Science*, 319:351 – 367, 2015. 31st Conf. on Math. Foundations of Programming Semantics.

[14] Tetsuya Sato. Approximate Relational Hoare Logic for Continuous Random Samplings. *Electronic Notes in Theoretical Computer Science*, 325:277 – 298, 2016. 32nd Conf. on Math. Foundations of Programming Semantics.

[15] Robert D. Tennent. *Specifying Software: A Hands-On Introduction.* Cambridge University Press, 2002.

[16] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, Cambridge, MA, USA, 1993.

[17] Hongseok Yang. Relational Separation Logic. *Theoretical Computer Science*, 375(1-3):308–334, April 2007.