An introduction to relational program verification

Suggested Citation: Gilles Barthe (2020), "An introduction to relational program verification", : Vol. xx, No. xx, pp 1–1. DOI: 10.1561/XXXXXXXXX.

Gilles Barthe

now
the essence of knowledge
Boston — Delft

Contents

An introduction to relational program verification

Gilles Barthe[1]

[1] *Max Planck Institute for Security and Privacy*

ABSTRACT

Gilles Barthe (2020), "An introduction to relational program verification", : Vol. xx, No. xx, pp 1–1. DOI: 10.1561/XXXXXXXXX.

1

An introduction to relational verification

Program verification has traditionally focused on the class of trace properties. One of the most fundamental trace property is (functional) correctness. Informally, and excluding for now the possibility that program executions may go wrong or not terminate, a program is correct if its output as expected. The notion of expected output can take several meanings: it may mean that the output is a mathematical function of its input (for instance it takes as input a list and outputs the list sorted in ascending order) or is appropriately related to the input (for instance it takes as input a list and outputs a position that holds a position of the list that holds a minimal element of that list—if such an element exists). These different meanings can be made precise using assertions. Informally, an assertion is a logical formula Φ that defines a subset of program states. Different notions of correctness can then be captured by triples of the form $c : \Phi \Rightarrow \Psi$, stating that every execution of a statement c started with an initial state which satisfies the assertion Φ, called pre-condition, concludes with a final state that satisfies the assertion Ψ, called the post-condition. Hoare Logic provides a proof system for reasoning about such triples. The proof system is compositional, i.e. a triple about statement c can be derived from properties of fragments

of c. The next chapter provides a brief account of Hoare logic.

However, many program properties from the literature escape the class of trace properties. An important class of properties that go beyond trace properties are so-called relational properties. These properties consider pairs of executions. These executions can be of the same program with different inputs or of two different programs with equal or different inputs. Arguably, the most fundamental relational property is program equivalence: two programs are equivalent iff they produce the same output when executed on the same input. Relational properties also play a very prominent role in reasoning about program security. For instance, consider a setting where variables are either public or secrets; in such a setting, we may require that programs are non-interfering, i.e. the final values of public variables should not leak information about the initial values of secret variables. This can be made more precise by requiring every two executions starting from initial states whose public values coincide terminate in final states whose public values also coincide. We can make the statement precise using relational assertions. In the same way that assertions are logical formulae Φ that define a subset of program states, relational assertions are formulae Φ that define a relation on program states. We can then consider quadruples of the form $c_1 \sim c_2 : \Phi \Rightarrow \Psi$, stating that every pair of executions of statements c_1 and c_2 started with initial states which satisfy the relational assertion Φ, called relational pre-condition, concludes with final states that satisfiy the relational assertion Ψ, called the relational post-condition. For instance, program equivalence is modelled as

$$c_1 \sim c_2 : = \Rightarrow =$$

where $=$ denotes equality of states, whereas non-interference is modelled as

$$c \sim c : =_{\mathcal{L}} \Rightarrow =_{\mathcal{L}}$$

where $=_{\mathcal{L}}$ relates pairs of states that coincide on their public variables.

Part I

Deterministic computations

2

First-order logic

3

The While programming language

We consider a core imperative language WHILE with assignments, sequencing, conditionals and while loops. We give denotational semantics for statements of the language. Moreover, we introduce instrumented semantics to reason about control flow and cost of statements. Our semantics uses elementary mathematical tools; more elegant (but equivalent) definitions of the semantics can be obtained using basic tools from domain theory, and monads.

3.1 Syntax

3.1.1 Types

Our programming language is simply typed, i.e. types are built from base types and constructors. Examples of base types are booleans and integers; examples of type constructors include lists and finite maps.

Definition 3.1 (Types)**.** Let $\mathcal{T}_0$ be a set of base types and let $\mathcal{CT}$ be a set of type constructors, such that each element $T \in \mathcal{CT}$ has an arity $k \in \mathbb{N}$. The set $\mathcal{T}$ of types is defined by the following syntax:

$$\begin{array}{rcll} \sigma & ::= & b & \text{base type} \\ & | & T(\sigma_1, \ldots, \sigma_n) & \text{type constructor} \end{array}$$

3.1.2 Expressions

Expressions of the language are deterministic and built from variables and operators. Examples of operators include the usual constants and operations for arithmetic, lists, finite maps, etc.

Definition 3.2 (Expressions)**.** Let **Op** be a set of operators. We assume that every operator $f \in \mathbf{Op}$ comes with a declaration of the form $f : \sigma_1 \times \ldots \times \sigma_n \to \tau$ that determines the number and types of elements it takes, and the type of the output. Moreover, let **Vars** be a set of variables. We assume that each variable x comes equipped with a declaration of the form $x : \sigma$ that defines its type. The set **Expr** of expressions is defined by the following syntax:

$$\begin{array}{rcll} e & ::= & x & \text{variable} \\ & | & f(e_1, \ldots, e_n) & \text{operator} \end{array}$$

The set $\mathrm{vars}(e)$ of variables of an expression e is defined inductively by the clauses:

$$\begin{array}{rcl} \mathrm{vars}(x) & = & \{x\} \\ \mathrm{vars}(f(e_1, \ldots, e_n)) & = & \bigcup_{1 \leq i \leq n} \mathrm{vars}(e_i) \end{array}$$

The substitution $e[e'/x]$ of an expression e' for a variable x in an expression e is defined inductively by the clause:

$$\begin{array}{rcl} y[e'/x] & = & \begin{cases} y & \text{if } y \neq x \\ e' & \text{if } y = x \end{cases} \\ f(e_1, \ldots, e_n)[e'/x] & = & f(e_1[e'/x], \ldots, e_n[e'/x]) \end{array}$$

We equip expressions with a type system which ensures that functions receive arguments of compatible types. The typing rules for ex-

pressions are straightforward:

$$\frac{x : \sigma}{\vdash x : \sigma}\ [\textsc{Var}]$$

$$\frac{\vdash e_1 : \sigma_1 \quad \ldots \quad \vdash e_n : \sigma_n \qquad f : \sigma_1 \times \ldots \times \sigma_n \to \tau}{\vdash f(e_1, \ldots, e_n) : \tau}\ [\textsc{Op}]$$

3.1.3 Statements

Statements are built from assignments, conditionals, loops, and sequencing.

Definition 3.3 (Statements)**.** The set **Cmd** of statements is defined by the following syntax:

$c ::=$	abort	abort
$\mid$	skip	do nothing
$\mid$	$x := e$	deterministic assignment
$\mid$	$c; c$	sequencing
$\mid$	if e then c else c	conditional
$\mid$	while e do c	while loop

Statements are equipped with a type system, which ensures that expressions are assigned to variables of compatible types and that guards of conditionals and loops are booleans. The typing rules are straightforward (Figure 3.1).

It is often useful to know which variables are read and modified by a statement. The formal definition of depending and modified variables is given below.

Definition 3.4 (Depending and modified variables)**.** The set $\mathrm{dep}(c)$ of depending variables of a statement c is defined by the clauses:

$$\begin{aligned}
\mathrm{dep}(x := e) &= \mathrm{vars}(e) \\
\mathrm{dep}(c_1; c_2) &= \mathrm{dep}(c_1) \cup \mathrm{dep}(c_2) \\
\mathrm{dep}(\text{if } e \text{ then } c_1 \text{ else } c_2) &= \mathrm{vars}(e) \cup \mathrm{dep}(c_1) \cup \mathrm{dep}(c_2) \\
\mathrm{dep}(\text{while } e \text{ do } c) &= \mathrm{vars}(e) \cup \mathrm{dep}(c)
\end{aligned}$$

$$\frac{}{\vdash \mathsf{skip}}\ [\textsc{Skip}]$$

$$\frac{\vdash x : \sigma \qquad \vdash e : \sigma}{\vdash x := e}\ [\textsc{Ass}]$$

$$\frac{\vdash c_1 \qquad \vdash c_2}{\vdash c_1; c_2}\ [\textsc{Seq}]$$

$$\frac{\vdash e : \mathbb{B} \qquad \vdash c_1 \qquad \vdash c_2}{\vdash \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2}\ [\textsc{Cond}]$$

$$\frac{\vdash e : \mathbb{B} \qquad \vdash c}{\vdash \mathsf{while}\ e\ \mathsf{do}\ c}\ [\textsc{While}]$$

Figure 3.1: Typing rules for statements

The set $\mathrm{mod}\, c$ of modified variables of a statement c is defined by the clauses:

$$\begin{array}{rcl} \mathrm{mod}(x := e) & = & \{x\} \\ \mathrm{mod}(c_1; c_2) & = & \mathrm{mod}(c_1) \cup \mathrm{mod}(c_2) \\ \mathrm{mod}(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) & = & \mathrm{mod}(c_1) \cup \mathrm{mod}(c_2) \\ \mathrm{mod}(\mathsf{while}\ e\ \mathsf{do}\ c) & = & \mathrm{mod}(c) \end{array}$$

3.2 Semantics

3.2.1 Types

We assume that each type has a set-theoretical interpretation, defined inductively over the structure of types.

Definition 3.5 (Interpretation of types)**.** Suppose given a set-theoretical interpretation $[\![b]\!] \in \mathbf{Set}$ for every base type $b \in \mathcal{T}_0$, and a set-theoretical interpretation $[\![T]\!] : \mathbf{Set}^k \to \mathbf{Set}$ for every type constructor $T \in \mathcal{CT}$ of arity k. The interpretation for types is defined inductively by the clause:

$$[\![T(\sigma_1, \ldots, \sigma_n)]\!] \quad = \quad [\![T]\!]([\![\sigma_1]\!], \ldots, [\![\sigma_n]\!])$$

3.2.2 Memories

Memories are mapping from variables to elements of the interpretation of their type.

Definition 3.6 (Memory)**.** A memory is a mapping m from variables to values, such that for every variable $x \in \mathbf{Vars}$ of type σ, we have $m(x) \in [\![\sigma]\!]$. We let $\mathbf{Mem}$ denote the set of memories.

We adopt standard notation for memory update: given a memory m, a variable x of type σ and a value $v \in [\![\sigma]\!]$, we let $m[x \leftarrow v]$ denote the unique memory such that for every variable y

$$m[x \leftarrow v](y) = \begin{cases} v & \text{if } x = y \\ m(y) & \text{otherwise} \end{cases}$$

Definition 3.7. Local equivalence of memories Let X be a set of variables. Two memories m and m' are equivalent w.r.t. X, written $m =_X m'$ iff $m(x) = m'(x)$ for every $x \in X$.

Note that we sometimes use $\complement(X)$ to denote the complement of X. Thus, $m =_{\complement(X)} m'$ iff $m(x) = m'(x)$ for every $x \notin X$.

3.2.3 Expressions

The semantics of expressions is parametrized by a memory and defined by structural induction.

Definition 3.8 (Semantics of expressions)**.** Suppose given a set-theoretical interpretation $[\![f]\!] \in [\![\sigma_1]\!] \times \ldots \times [\![\sigma_n]\!] \to [\![\tau]\!]$ for every $f : \sigma_1 \times \ldots \times \sigma_n \to \tau$. The semantics of an expression e with respect to a memory m is defined by the clauses:

$$\begin{aligned} [\![x]\!]_m &= m(x) \\ [\![f(e_1, \ldots, e_n)]\!]_m &= [\![f]\!]([\![e_1]\!]_m, \ldots, [\![e_n]\!]_m) \end{aligned}$$

The following lemma establishes the correctness of $\mathrm{vars}(e)$.

Lemma 3.1. If $m_1 =_{\mathrm{vars}(e)} m_2$ then $[\![e]\!]_{m_1} = [\![e]\!]_{m_2}$.

3.2.4 Statements

We now turn to give a denotational semantics to statements. The semantics is partial, i.e. may take a special value $\perp$ to denote non-termination.

Definition 3.9 (Semantics of statements)**.** The denotational semantics $[\![s]\!]$ of a statement s is a function that assigns to every memory $m \in \mathbf{Mem}$ an element $[\![s]\!]_m \in \mathbf{Mem}_\perp$, where $X_\perp = X \sqcup \{\perp\}$. The definition of $[\![s]\!]_m$ is given in Figure 3.2, where we use the following convention:

$$\mathsf{let}\ m' = mo\ \mathsf{in}\ g(m') = \begin{cases} m' & \text{if } mo = m \neq \perp \text{ and } g(m) = m' \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

We briefly comment on the definition of the semantics. The semantics of abort is the error memory $\perp$. The semantics of skip is simply the identity—do nothing. The semantics of a deterministic assignment is a map that takes as input an inital memory m and returns the memory obtained by updating m with the value v resulting from the evaluation e in memory m. The semantics of a sequential composition is defined as the composition of the semantics of the first and second statements. The semantics of conditional statements is straightforward: given a memory m, one evaluates the guard e of the conditional in m, and return the output $[\![c_1]\!]_m$ of the true branch if the guard evaluates to tt and the output $[\![c_2]\!]_m$ of the false branch if the guard evaluates to ff. Finally, the semantics of while loops is defined as follows: given a memory m, one evaluates the guard the guard e of the conditional in m, and returns m if the guard is false; else one repeats the process with the output $[\![c]\!]_m$ of the loop guard. The process stops when the loop guard returns false; if this never happens, then one outputs $\perp$. Formally, the clause for loops is defined as the least upper bound of its finite approximations. For this, we use the following convention: for every statement c and boolean expression e, we define the lower iterations of $\mathsf{while}_c\ i\ \mathsf{do}\ e$ by the clauses:

$$\begin{aligned} \mathsf{while}_n\ e\ \mathsf{do}\ c &= \mathsf{while}^n\ e\ \mathsf{do}\ c; \mathsf{if}\ e\ \mathsf{then\ abort} \\ \mathsf{while}^0\ e\ \mathsf{do}\ c &= \mathsf{skip} \\ \mathsf{while}^{n+1}\ e\ \mathsf{do}\ c &= \mathsf{if}\ b\ \mathsf{then}\ (c; \mathsf{while}^n\ e\ \mathsf{do}\ c) \end{aligned}$$

It is easy to see that the sequence of memories $[\![\mathsf{while}_i\ e\ \mathsf{do}\ c]\!]_m$ is increasing and thus has a least upper bound.

$$
\begin{aligned}
[\![\mathsf{abort}]\!]_m &= \bot \\
[\![\mathsf{skip}]\!]_m &= \mathsf{unit}(m) \\
[\![x := e]\!]_m &= \mathsf{unit}(m[x \leftarrow [\![e]\!]_m]) \\
[\![c_1; c_2]\!]_m &= \mathsf{let}\ m' = [\![c_1]\!]_m\ \mathsf{in}\ [\![c_2]\!]_{m'}
\end{aligned}
$$

$$
[\![\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!]_m = \begin{cases} [\![c_1]\!]_m & \text{if } [\![e]\!]_m = \mathsf{tt} \\ [\![c_2]\!]_m & \text{if } [\![e]\!]_m = \mathsf{ff} \end{cases}
$$

$$
[\![\mathsf{while}\ e\ \mathsf{do}\ c]\!]_m = \sup_{i \in \mathbb{N}} [\![\mathsf{while}_i\ e\ \mathsf{do}\ c]\!]_m
$$

Figure 3.2: Denotational semantics of statements

The following lemmas capture the correctness of dep and mod.

Lemma 3.2. If $m_1 =_{\mathrm{dep}(c)} m_2$ then $[\![c]\!]_{m_1} = [\![c]\!]_{m_2}$.

Lemma 3.3. If $[\![c]\!]_m = m'$ then $m =_{\complement(\mathrm{mod}(c))} m'$.

We conclude this section by introducing notations for terminating programs.

Definition 3.10 (Termination)**.** A statement c is terminating on initial memory m, written $c, m \Downarrow$, if $[\![c]\!]_m \neq \bot$. A statement c is terminating, written $\mathsf{term}(c)$, if $c, m \Downarrow$ for every initial memory m.

By convention a terminating program cannot return $\bot$ so abort is considered non-terminating.

3.3 Instrumented semantics

It is often useful to enrich the semantics of statements to reason about properties other than functional behavior. In this section, we instrument the denotational semantics of statements to keep track of their control-flow decisions, and of their cost.

3.3.1 Control flow

We instrument the semantics of statements to track their control-flow. The instrumented semantics returns an additional list of booleans (indicating for each control-flow statement, i.e. loop or conditional, whether execution followed the true or the false branch). Therefore the instrumented semantics of c is the function

$$[\![c]\!]^{\mathrm{cf}}_m : \mathbf{Mem} \to (\mathbf{Mem} \times \mathbb{B}^*)_\bot$$

defined by the clauses of Figure 3.3, where we use the convention for $g : \mathbf{Mem} \to (\mathbf{Mem} \times \mathbb{B}^*)_\bot$:

$$\mathsf{let}\ m = mcf\ \mathsf{in}\ g(m) = \begin{cases} (m', cf + cf') & \text{if } mcf = (m, cf) \text{ and } g(m) = (m', cf') \\ \bot & \text{otherwise} \end{cases}$$

and by abuse of notation for $g : \mathbf{Mem} \to \mathbf{Mem}_\bot$:

$$\mathsf{let}\ m' = mcf\ \mathsf{in}\ g(m) = \begin{cases} (m', cf) & \text{if } mcf = (m, cf) \text{ and } g(m) = m' \\ \bot & \text{otherwise} \end{cases}$$

We briefly comment on the clauses for conditionals and control-flow. Note that the clause for conditionals extends the list of booleans, i.e. leakage, with a new boolean to record which branch has been taken. As for the baseline semantics, the clause for loops is defined using a least fixed point.

3.3.2 Cost

We instrument the semantics of statements to track their cost. The instrumented semantics considers an extended language with a new construct $\mathsf{tick}\ k$, where $k \in \mathbb{N}$, and returns an additional number representing the execution cost of the statement. Therefore the instrumented semantics of c is the function

$$[\![c]\!]^{\mathrm{cost}}_m : \mathbf{Mem} \to (\mathbf{Mem} \times \mathbb{N})_\bot$$

defined by the clauses of Figure 3.4, where we use the following convention:

$$\mathsf{let}\ m' = mk\ \mathsf{in}\ g(m') = \begin{cases} (m', \kappa + \kappa') & \text{if } mk = (m, \kappa) \text{ and } g(m) = (m', \kappa') \\ \bot & \text{otherwise} \end{cases}$$

$$
\begin{aligned}
[\![\mathsf{skip}]\!]^{\mathrm{cf}}_m &= (m, \epsilon) \\
[\![x := e]\!]^{\mathrm{cf}}_m &= (m[x \leftarrow [\![e]\!]_m], \epsilon) \\
[\![c_1; c_2]\!]^{\mathrm{cf}}_m &= \mathsf{let}\ mcf = [\![c_1]\!]^{\mathrm{cf}}_m\ \mathsf{in}\ [\![c_2]\!]^{\mathrm{cf}}_{mcf}
\end{aligned}
$$

$$
[\![\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!]_m = \begin{cases} \mathsf{let}\ mpc = [\![c_1]\!]^{\mathrm{cf}}_m\ \mathsf{in}\ \lambda m'.(m', \mathsf{tt}) & \text{if } [\![e]\!]_m = \mathsf{tt} \\ \mathsf{let}\ mpc = [\![c_2]\!]^{\mathrm{cf}}_m\ \mathsf{in}\ \lambda m'.(m', \mathsf{ff}) & \text{if } [\![e]\!]_m = \mathsf{ff} \end{cases}
$$

$$
[\![\mathsf{while}\ e\ \mathsf{do}\ c]\!]^{\mathrm{cf}}_m = \sup_{i \in \mathbb{N}} [\![\mathsf{while}_i\ e\ \mathsf{do}\ c]\!]^{\mathrm{cf}}_m
$$

Figure 3.3: Control-Flow instrumented semantics of statements

$$
\begin{aligned}
[\![\mathsf{skip}]\!]^{\mathrm{cost}}_m &= (m, 0) \\
[\![\mathsf{skip}]\!]^{\mathrm{cost}}_m &= (m, 0) \\
[\![x := e]\!]^{\mathrm{cost}}_m &= (m[x \leftarrow [\![e]\!]_m], 0) \\
[\![\mathsf{tick}\ k]\!]^{\mathrm{cost}}_m &= (m, [\![k]\!]_m) \\
[\![c_1; c_2]\!]^{\mathrm{cost}}_m &= \mathsf{let}\ mc' = [\![c_1]\!]^{\mathrm{cost}}_m\ \mathsf{in}\ [\![c_2]\!]^{\mathrm{cost}}_{m'}
\end{aligned}
$$

$$
[\![\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!]_m = \begin{cases} [\![c_1]\!]^{\mathrm{cost}}_m & \text{if } [\![e]\!]_m = \mathsf{tt} \\ [\![c_2]\!]^{\mathrm{cost}}_m & \text{if } [\![e]\!]_m = \mathsf{ff} \end{cases}
$$

$$
[\![\mathsf{while}\ e\ \mathsf{do}\ c]\!]^{\mathrm{cost}}_m = \sup_{i \in \mathbb{N}} [\![\mathsf{while}_i\ e\ \mathsf{do}\ c]\!]^{\mathrm{cost}}_m
$$

Figure 3.4: Cost instrumented semantics of statements

4

Hoare Logic

4.1 Assertions

Assertions are first-order formulae built from program variables. The interpretation of an assertion ϕ is a subset $[\![\phi]\!]$ of memories, i.e. $[\![\phi]\!] \subseteq \mathbf{Mem}$. The interpretation of formulae is given in **??**.

4.2 Judgments

Judgments of Hoare logic are of the form

$$c : \Phi \Rightarrow \psi$$

and relate a program c with a pre-condition Φ and a post-condition ψ. Informally, a judgment is valid if for $[\![c]\!]$ maps memories satisfying the pre-condition to memories satisfying the post-condition. We focus on partial correctness:

Definition 4.1 (Valid judgment)**.** The judgment $c : \Phi \Rightarrow \psi$ is valid, written $\models c : \Phi \Rightarrow \psi$, if for every memory m such that $m \in [\![\Phi]\!]$ and $[\![c]\!]_m \neq \bot$, we have $[\![c]\!]_m \in [\![\psi]\!]$.

$$\frac{\vdash c : \Phi' \Rightarrow \Psi' \qquad \models \Phi \Longrightarrow \Phi' \qquad \models \Psi' \Longrightarrow \Psi}{\vdash c : \Phi \Rightarrow \Psi} \text{ [HL-Conseq]}$$

$$\frac{\vdash c : \Phi \Rightarrow \Psi \quad \text{vars}(\Theta) \cap \text{mod}(c) = \emptyset}{\vdash c : \Phi \wedge \Theta \Rightarrow \Psi \wedge \Theta} \text{ [HL-Frame]}$$

$$\frac{\vdash c : \Phi \wedge e = \mathsf{tt} \Rightarrow \Psi \quad \vdash c : \Phi \wedge e = \mathsf{ff} \Rightarrow \Psi}{\vdash c : \Phi \Rightarrow \Psi} \text{ [HL-Case]}$$

$$\frac{\forall x : T. \vdash c : \Phi \Rightarrow \Psi}{\vdash c : \exists x : T. \Phi \Rightarrow \Psi} \text{ [HL-Exists]}$$

Figure 4.1: Hoare Logic: structural rules

4.3 Proof system

We now present a proof system for deriving valid Hoare triples. The proof system combines structural rules, which apply independently of the shape of programs, and construct-specific rules. There is one construct-specific rule for each form of statement.

4.3.1 Structural rules

The [Conseq] rule is known as the rule of consequence, and can be used for weakening the post-condition and strengthening the pre-condition.

The [Frame] rule allows to strengthen simultaneously the pre-condition and the post-condition of a valid judgment with an assertion Θ such that $\text{vars}(\Theta)$ is disjoint from $\text{mod}(c)$, where the set $\text{vars}(\Theta)$ of free variables of Θ is defined in the usual way (**??**).

The [Case] rule allows proving a judgment by case analysis on a boolean expression e.

The [Exists] rule is similar to the [Case] rule, excepts that it considers the case where the pre-condition is an existential statement. It allows to prove a judgment by supposing the existence of a witness.

$$\vdash \mathsf{skip} : \Psi \Rightarrow \Psi \ [\text{HL-Skip}]$$

$$\frac{}{\vdash x := e : \Psi[e/x] \Rightarrow \Psi} \ [\text{HL-Assn}]$$

$$\frac{\vdash c : \Phi \Rightarrow \Theta \quad \vdash c' : \Theta \Rightarrow \Psi}{\vdash c; c' : \Phi \Rightarrow \Psi} \ [\text{HL-Seq}]$$

$$\frac{\vdash c : \Phi \wedge e \Rightarrow \Psi \qquad \vdash c' : \Phi \wedge \neg e \Rightarrow \Psi}{\vdash \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c' : \Phi \Rightarrow \Psi} \ [\text{HL-Cond}]$$

$$\frac{\vdash c : \Theta \wedge e \Rightarrow \Theta}{\vdash \mathsf{while}\ e\ \mathsf{do}\ c : \Theta \Rightarrow \Theta \wedge \neg e} \ [\text{HL-While}]$$

Figure 4.2: Proof system for Hoare logic

4.4 Soundness and relative completeness

The proof system is sound, i.e every derivable judgment is valid.

Theorem 4.1. If $\vdash c : \Phi \Rightarrow \Psi$ then $\models c : \Phi \Rightarrow \Psi$.

The converse is called relative completeness, and may hold under the assumption that the assertion language is sufficiently rich. We refer the reader to e.g. (Winskel, 1993) for a detailed proof of relative completeness.

4.5 Verification condition generation

A common approach to automating Hoare Logic is to generate a set of verification conditions (VC) whose validity entails validity of Hoare triples. These verification conditions can be computed using a precondition calculus, which traverses the program backwards and computes for every statement c and postcondition ψ a precondition ϕ and a set of verification conditions Ξ. The definition of the calculus is given in Figure 4.3. The clauses for all constructs except loops match the corresponding rule in Hoare logic; note that the set of verification conditions for compound statements is the union of the set of verification

conditions for sub-statements. The clause for loops is more interesting. Indeed, computing a precondition for loops requires a loop invariant, i.e. an assertion that is preserved by the loop body. In this chapter, we assume that loop invariants are provided by an external oracle invgen that takes as input a loop and a post-condition, and returns an assertion intended to be a loop invariant that entails (with the negation of the loop guard) the post-condition. The latter is captured by introducing a new verification condition.

Given a Hoare triple $c : \Phi \Rightarrow \Psi$, we define

$$\mathsf{VC}(c : \Phi \Rightarrow \Psi) \triangleq \{\Phi \implies \Phi_0 \mid \Phi_0 \in \mathsf{vcgen}\,(c, \Psi, \emptyset)\}$$

Verification condition generation is sound with respect to Hoare logic, i.e. validity of verification conditions entails provability of Hoare triples.

Theorem 4.2. If $\models \mathsf{VC}(c : \Phi \Rightarrow \Psi)$, then $\vdash c : \Phi \Rightarrow \Psi$.

The converse is not true, due to our simplified presentation of verification condition generation.

4.6 Examples

We consider examples which we reuse throughout the following chapters.

4.6.1 Exponentiation

The following example computes exponentiation:

$$\begin{array}{l}
\mathbf{exp}(n, k : \mathbb{N}) : \mathbb{N} \\
r := 1; \\
i := 0; \\
\mathsf{while}\ i < k\ \mathsf{do}\ r := n * r; i := i + 1; \\
\mathsf{return}\ r
\end{array}$$

We can derive the following Hoare triple:

$$\vdash \mathsf{exp} : n \geq 0 \wedge k > 1 \Rightarrow r = n^k$$

The proof follows from taking $r = n^i \wedge i \leq k$ as an invariant. One concludes with the rule of consequence.

$$\frac{}{\mathsf{vcgen}\,(\mathsf{skip}, \psi, \Xi) = (\psi, \Xi)}\ [\text{VC-Skip}]$$

$$\frac{}{\mathsf{vcgen}\,(x := e, \psi, \Xi) = (\psi[e/x], \Xi)}\ [\text{VC-Assn}]$$

$$\frac{\mathsf{vcgen}\left(c', \psi, \Xi\right) = (\psi_0, \Xi_0) \qquad \mathsf{vcgen}\,(c, \psi_0, \Xi_0) = (\phi, \Xi_1)}{\mathsf{vcgen}\left(c; c', \psi, \Xi\right) = (\phi, \Xi_1)}\ [\text{VC-Seq}]$$

$$\frac{\begin{array}{c}\mathsf{vcgen}\left(c', \psi, \Xi\right) = (\phi_1, \Xi_1) \\ \mathsf{vcgen}\left(c', \psi, \Xi\right) = (\phi_2, \Xi_2) \\ \Xi_0 = \Xi_1 \cup \Xi_2\end{array}}{\mathsf{vcgen}\left(\mathsf{if}\ b\ \mathsf{then}\ c\ \mathsf{else}\ c', \psi, \Xi\right) = (b \implies \phi_1 \wedge \neg b \implies \phi_2, \Xi_0)}\ [\text{VC-Cond}]$$

$$\frac{\begin{array}{c}\mathsf{invgen}(\mathsf{while}\ b\ \mathsf{do}\ c, \psi) = \phi \\ \mathsf{vcgen}\,(c, \phi, \Xi) = (\phi_0, \Xi_0) \\ \Xi_1 = \Xi_0 \cup \{\phi \wedge \neg b \implies \psi, \phi \wedge b \implies \phi_0\}\end{array}}{\mathsf{vcgen}\,(\mathsf{while}\ b\ \mathsf{do}\ c, \psi, \Xi) = (\phi, \Xi_1)}\ [\text{VC-While}]$$

Figure 4.3: Verification condition generation

4.6.2 Fast exponentiation

One can also consider a more efficient algorithm for computing exponentiation:

```
fastexp(n, k : ℕ) : ℕ
if k = 0 then return 1;
r := 1;
while k > 1 do
  if even(n) then
    n := n * n;
    n := n/2;
  else
    r := n * r;
    n := n * n;
    n := (n − 1)/2;
r := x * r;
return r
```

We can also derive the following Hoare triple:

$$\vdash \mathsf{fastexp} : n \geq 0 \wedge k > 1 \Rightarrow r = n^k$$

4.6.3 Sums

Our final example (shown in Figure 4.4) sums n natural numbers $p,\ p+k,\ p+2k$. We can prove different Hoare triples about this statement, e.g.

$$\vdash \mathsf{sum} : p = 1 \wedge k = 1 \wedge n \geq 0 \Rightarrow s = \frac{n(n-1)}{2}$$

This statement is proved using as loop invariant

$$s = \frac{i(i+1)}{2} \wedge i \leq n$$

More generally, one can prove

$$\vdash \mathsf{sum} : p \geq 0 \wedge k \geq 0 \wedge n \geq 0 \Rightarrow s = p \cdot n + k \cdot \frac{n(n-1)}{2}$$

$\mathbf{sum}(p, k, n : \mathbb{N}) : \mathbb{N}$
$s := 0;$
$i := 0;$
while $i < n$ do $s := s + p; p := p + k; i := i + 1;$
return s

Figure 4.4: Partial sum of integers

5

Relational Hoare Logic

Our central tool for reasoning about the relationship between two programs is Relational Hoare Logic (Benton, 2004).

5.1 Relational assertions

Relational assertions are first-order formulae whose interpretation is taken over two memories. Therefore, basic relational assertions are of the form $P(t_1, \ldots, t_n)$ where the predicate P is specified in the underlying theory, and the relational expressions $t_1, \ldots, t_n$ are built from function symbols of the underlying theory, logical variables and tagged variables of the form $x\langle 1\rangle$ and $x\langle 2\rangle$ where x is a program variable. Here the tags $\langle 1\rangle$ and $\langle 2\rangle$ are used to indicate that the interpretation of x should be taken in the first and second memory respectively. Therefore, in particular, the relational assertion $x\langle 1\rangle = x\langle 2\rangle$ captures the fact that the value of x in the left memory is equal to the value of x in the right memory.

The interpretation of a relational assertion Φ is a relation $[\![\Phi]\!] \subseteq \mathbf{Mem} \times \mathbf{Mem}$ consisting of all the set of pairs of memories (m_1, m_2) for whichΦ holds. The definition is similar (modulo the use of two memories

to interpret tagged variables) to **??**.

Notation 5.1. For every expression e, we let $e\langle 1\rangle$ and $e\langle 2\rangle$ denote the generalized expressions obtained by tagging every variable in e with a $\langle 1\rangle$ and $\langle 2\rangle$ respectively. For instance, if e is $x + y$ then $e\langle 1\rangle$ is defined as $x\langle 1\rangle + y\langle 1\rangle$.

Then, every assertion ϕ yields two relational assertions $\phi\langle 1\rangle$ and $\phi\langle 2\rangle$, with the expected relational interpretation; $[\![\phi\langle 1\rangle]\!]_{(m_1,m_2)} = [\![\phi]\!]_{m_1}$ and $[\![\phi\langle 2\rangle]\!]_{(m_1,m_2)} = [\![\phi]\!]_{m_2}$.

Notation 5.2. For every $R \subseteq A \times A$, we let $R_\perp \subseteq A_\perp \times A_\perp$ be the smallest relation such that $R_\perp \perp \perp$ and for every $a_1, a_2 \in A$, $R\ a_1\ a_2$ implies $R_\perp\ a_1\ a_2$.

5.2 Judgments

Judgments of relational Hoare logic are of the form

$$c_1 \sim c_2 : \Phi \Rightarrow \Psi$$

and relate two programs, c_1 and c_2, w.r.t. a pre-condition Φ and a post-condition Ψ. Informally, a judgment is valid if for every pair of memories m_1 and m_2 related by Φ, the memories $[\![c_1]\!]_{m_1}$ and $[\![c_2]\!]_{m_2}$ are related by $\Psi_\perp$.

Definition 5.1 (Valid judgment)**.** The judgment $c_1 \sim c_2 : \Phi \Rightarrow \Psi$ is valid, written $\models c_1 \sim c_2 : \Phi \Rightarrow \Psi$, if for every pair of memories (m_1, m_2) such that $(m_1, m_2) \in [\![\Phi]\!]$, we have $([\![c_1]\!]_{m_1}, [\![c_2]\!]_{m_2}) \in [\![\Psi]\!]_\perp$.

Note that the notion of valid judgment enforces co-termination. If $\models c_1 \sim c_2 : \Phi \Rightarrow \Psi$ is valid and $(m_1, m_2) \in [\![\Phi]\!]$, then $c_1, m_1 \Downarrow$ iff $c_2, m_2 \Downarrow$.

5.3 Proof system

We now present a proof system for deriving Hoare quadruples. The proof rules are split into three groups:

structural rules: these rules can be applied independently of the shape of the programs;

2-sided rules: these rules requires that the two programs have a specific and corresponding shape (for instance, the two programs must be a deterministic assignment; or the two programs must be a conditional statement). There is one single rule for each form of statement;

1-sided rules: these rules requires that one of the two programs has a specific shape. There are two rules per for each form of statement; a left rule requiring that the left statement of the judgment has the desired shape, and a right rule requiring that the right statement has the desired shape.

5.3.1 Structural rules

Figure 5.1 presents structural rules. We briefly comment on each rule.

The [Hoare] rule allows to derive relational Hoare judgments from Hoare judgments of individual programs.

The [False] rule states that arbitrary programs are related, when the precondition is provably false. The rule is derivable from the [Hoare] and the [False] rule from Hoare logic (the rule is admissible, and can be proved by induction on the structure of the statement).

The [Conseq] rule is similar to the one from Hoare Logic, and can be used for weakening the post-condition and strengthening the pre-condition.

The [Frame] rule allows to strengthen simultaneously the pre-condition and the post-condition of a valid judgment with an assertion Θ, provided the variables modified by the two statements of the judgment are disjoint from $\mathrm{vars}(\Theta)$, where the definition of vars is similar to **??**[1].

The [Case] rule is similar to the one from Hoare Logic, and allows proving a judgment by case analysis on the value of a boolean expression e.

The [Exists] rule is is similar to the one from Hoare Logic, and allows to prove a judgment whose precondition is an existential by supposing the existence of a witness.

[1]The soundness of the rule relies on the relational counterpart of **??**: $(m_1, m_2) =_{\mathrm{vars}(\Phi)} (m_1', m_2')$ implies $[\![\Phi]\!]_{(m_1,m_2)}$ iff $[\![\Phi]\!]_{(m_1',m_2')}$.

The [Trans] rule introduces an intermediate program c to relate two programs c_1 and c_2. It requires proving judgments for c_1 and c, and for c and c_2. The pre- and post-condition of the conclusion use relation composition, defined by

$$(R \circ S)(x, z) \triangleq \exists y.\ R(x, y) \wedge S(y, z)$$

The [Struct] rule allows replacing programs by provably equivalent programs. The rule depends on an auxiliary judgment of the form $\Phi \models c \equiv c'$, where Φ is a relational assertion, and which states that c and c' are equivalent (i.e. yield equal memories) for every two pairs of memories that satisfy Φ. We leave the proof system for program equivalence unspecified.

5.3.2 Two-sided rules

Figure 5.2 presents two-sided rules. We briefly comment on each rule.

The [Assn] rule states that an assertion is valid after two assignments, if the original pairs of memories satisfies the assertion obtained by substituting in place of the variables being assigned the (tagged) expressions of the assignments.

The [Seq] rule for sequential composition requires that the left and right statements are sequential compositions and that there exists an intermediate assertion Θ that is a valid postcondition for the first statements and a valid precondition for the second statements. This rule is the relational counterpart of the rule for sequential composition in Hoare logic.

The [Cond] rule considers two conditional statements that execute in lockstep. Specifically, it requires that the pre-condition Φ implies that the guards of the two conditional statements are logically equivalent. The premises of the rule ensure that that both the true branches of the statements and the false branches of the statements are related by pre-condition Φ (strengthened by the guard of the conditionals or their negation) and the same post-condition Ψ. Therefore the two conditional statements are related by the pre-condition Φ and the post-condition Ψ.

$$\frac{\begin{array}{c}\vdash c_1 : \Phi_1 \Rightarrow \Psi_1 \\ \vdash c_2 : \Phi_2 \Rightarrow \Psi_2 \\ \forall m_1 \in [\![\Phi_1]\!].\ c_1, m_1 \Downarrow \\ \forall m_2 \in [\![\Phi_2]\!].\ c_2, m_2 \Downarrow\end{array}}{\vdash c_1 \sim c_2 : \Phi_1\langle 1\rangle \wedge \Phi_2\langle 2\rangle \Rightarrow \Psi_1\langle 1\rangle \wedge \Psi_2\langle 2\rangle}\ [\text{RHL-HOARE}]$$

$$\frac{\mathsf{term}(c_1) \qquad \mathsf{term}(c_2)}{\vdash c_1 \sim c_2 : \bot \Rightarrow \Psi}\ [\text{RHL-FALSE}]$$

$$\frac{\vdash c_1 \sim c_2 : \Phi' \Rightarrow \Psi' \qquad \models \Phi \Longrightarrow \Phi' \qquad \models \Psi' \Longrightarrow \Psi}{\vdash c_1 \sim c_2 : \Phi \Rightarrow \Psi}\ [\text{RHL-CONSEQ}]$$

$$\frac{\begin{array}{c}\vdash c_1 \sim c_2 : \Phi \Rightarrow \Psi \\ \mathrm{vars}(\Theta) \cap (\mathrm{mod}(c_1)\langle 1\rangle \cup \mathrm{mod}(c_2)\langle 2\rangle) = \emptyset\end{array}}{\vdash c_1 \sim c_2 : \Phi \wedge \Theta \Rightarrow \Psi \wedge \Theta}\ [\text{RHL-FRAME}]$$

$$\frac{\vdash c_1 \sim c_2 : \Phi \wedge e = \mathsf{tt} \Rightarrow \Psi \qquad \vdash c_1 \sim c_2 : \Phi \wedge e = \mathsf{ff} \Rightarrow \Psi}{\vdash c_1 \sim c_2 : \Phi \Rightarrow \Psi}\ [\text{RHL-CASE}]$$

$$\frac{\forall x : T.\ \vdash c_1 \sim c_2 : \Phi \Rightarrow \Psi}{\vdash c_1 \sim c_2 : \exists x : T.\ \Phi \Rightarrow \Psi}\ [\text{RHL-EXISTS}]$$

$$\frac{\vdash c_1 \sim c : \Phi \Rightarrow \Psi \qquad \vdash c \sim c_2 : \Phi' \Rightarrow \Psi'}{\vdash c_1 \sim c_2 : \Phi \circ \Phi' \Rightarrow \Psi \circ \Psi'}\ [\text{RHL-TRANS}]$$

$$\frac{\vdash c_1 \sim c_2 : \Phi \Rightarrow \Psi \qquad \Phi \models c_1 \equiv c_1' \qquad \Phi \models c_2 \equiv c_2'}{\vdash c_1' \sim c_2' : \Phi \Rightarrow \Psi}\ [\text{RHL-STRUCT}]$$

Figure 5.1: Structural rules

$$\vdash \mathsf{skip} \sim \mathsf{skip} : \Psi \Rightarrow \Psi \ [\text{RHL-Skip}]$$

$$\frac{\vdash c_1 \sim c_2 : \Phi \Rightarrow \Theta \quad \vdash c_1' \sim c_2' : \Theta \Rightarrow \Psi}{\vdash c_1; c_1' \sim c_2; c_2' : \Phi \Rightarrow \Psi} \ [\text{RHL-Seq}]$$

$$\frac{}{\vdash x_1 := e_1 \sim x_2 := e_2 : \Psi[e_1\langle 1\rangle / x_1\langle 1\rangle][e_2\langle 2\rangle / x_2\langle 2\rangle] \Rightarrow \Psi} \ [\text{RHL-Assn}]$$

$$\frac{\begin{array}{c}\models \Phi \implies e_1\langle 1\rangle = e_2\langle 2\rangle \\ \vdash c_1 \sim c_2 : \Phi \wedge e_1\langle 1\rangle \Rightarrow \Psi \\ \vdash c_1' \sim c_2' : \Phi \wedge \neg e_1\langle 1\rangle \Rightarrow \Psi\end{array}}{\vdash \mathsf{if}\ e_1\ \mathsf{then}\ c_1\ \mathsf{else}\ c_1' \sim \mathsf{if}\ e_2\ \mathsf{then}\ c_2\ \mathsf{else}\ c_2' : \Phi \Rightarrow \Psi} \ [\text{RHL-Cond}]$$

$$\frac{\models \Theta \implies e_1\langle 1\rangle = e_2\langle 2\rangle \qquad \vdash c_1 \sim c_2 : \Theta \wedge e_1\langle 1\rangle \Rightarrow \Theta}{\vdash \mathsf{while}\ e_1\ \mathsf{do}\ c_1 \sim \mathsf{while}\ e_2\ \mathsf{do}\ c_2 : \Theta \Rightarrow \Theta \wedge \neg e_1\langle 1\rangle} \ [\text{RHL-While}]$$

Figure 5.2: Two-sided rules

The [While] rule considers two while loops that execute in lockstep. Specifically, it requires that there exists a relational loop invariant Θ that is initially valid and preserved by one iteration of the two loop bodies, and such that the loop guards are equivalent for any two memories satisfying the invariant. Upon termination, i.e. in the output distributions, both loop guards are false and the loop invariant is valid.

In the remainder of this monograph, we shall informally refer to two-sided relational Hoare logic as the set of 2-sided rules plus all structural rules except [Struct].

5.3.3 One-sided rules

Figure 5.3 presents left rules (right rules are similar). In all cases, except for the rule for conditionals, the program on the right is a skip statement.

The [Assg-L] rule states that an assertion Ψ is a valid post-condition, if the initial pair of memories satisfy the assertion $\Psi[e\langle 2\rangle / x\langle 1\rangle]$.

The [Cond-L] rule considers a conditional statement on the left

$$\frac{}{\vdash x_1 := e_1 \sim \mathsf{skip} : \Psi[e_1\langle 1\rangle / x_1\langle 1\rangle] \Rightarrow \Psi}\ [\text{RHL-Assg-L}]$$

$$\frac{\vdash c_1 \sim c_2 : \Phi \wedge e_1\langle 1\rangle \Rightarrow \Psi \quad \vdash c_1' \sim c_2 : \Phi \wedge \neg e_1\langle 1\rangle \Rightarrow \Psi}{\vdash \mathsf{if}\ e_1\ \mathsf{then}\ c_1\ \mathsf{else}\ c_1' \sim c_2 : \Phi \Rightarrow \Psi}\ [\text{RHL-Cond-L}]$$

$$\frac{\vdash c_1 \sim \mathsf{skip} : \Theta \wedge e_1\langle 1\rangle \Rightarrow \Theta \quad \forall m_1\ m_2, (m_1, m_2) \in \Theta \Rightarrow \mathsf{while}\ e_1\ \mathsf{do}\ c_1, m_1 \Downarrow}{\vdash \mathsf{while}\ e_1\ \mathsf{do}\ c_1 \sim \mathsf{skip} : \Theta \Rightarrow \Theta \wedge \neg e_1\langle 1\rangle}\ [\text{RHL-While-L}]$$

Figure 5.3: One-sided (left) rules

and an arbitrary statement to the right. It performs a case analysis on the guard of the conditional statement and matches its true and false branch against the right statement.

The [While-L] rule requires each iteration of the loop body preserves an invariant Θ, and that the loop is almost surely terminating. If the initial pair of memories satisfy Θ, then upon termination, i.e. in the output distributions, the loop guard is false and the loop invariant is valid.

Note that there is no one-sided rule for sequential composition.

Remark 5.1. The proof system is not minimal, in the sense that some rules can be derived from others. For instance, Barthe *et al.* (2017) prove that, given a sufficiently strong proof system for structural equivalence on programs, all one-sided rules except [While-L] can be derived from their two-sided counterpart. Similarly, all uses of the one-sided rule for loops [While-L] can be avoided by proving the loop in Hoare logic and using the rule [Hoare] afterwards.

5.3.4 Soundness and relative completeness

The proof system is *sound*, in the sense that the conclusions of all the proof rules are valid judgments, provided the premises of the rules are valid judgments and the side-conditions, if any, hold.

Proposition 5.1. Every derivable judgment $\vdash c_1 \sim c_2 : \Phi \Rightarrow \Psi$ is valid, i.e. for every pair of memories (m_1, m_2), $(m_1, m_2) \in [\![\Phi]\!]$ implies $([\![c_1]\!]_{m_1}, [\![c_2]\!]_{m_2}) \in [\![\Psi]\!]_\perp$.

The proof is by induction on the structure of derivations. The only interesting case is for loops. We prove by induction on i that $(m_1, m_2) \in [\![\Theta]\!]$ implies

$$([\![(\mathsf{if}\ e_1\ \mathsf{then}\ c_1\ \mathsf{else}\ \mathsf{skip})^i]\!]_{m_1}, [\![(\mathsf{if}\ e_2\ \mathsf{then}\ c_2\ \mathsf{else}\ \mathsf{skip})^i]\!]_{m_2}) \in [\![\Theta]\!]_\perp$$

Therefore by definition of the denotational semantics for loops, it follows that $(m_1, m_2) \in [\![\Theta]\!]$ implies

$$([\![\mathsf{while}\ e_1\ \mathsf{do}\ c_1]\!]_{m_1}, [\![\mathsf{while}\ e_2\ \mathsf{do}\ c_2]\!]_{m_2}) \in ([\![\Theta]\!] \wedge \neg e_1\langle 1\rangle)_\perp$$

The converse of soundness, also called *relatively complete* by anology with Hoare logic, may hold for terminating programs, provided the assertion language is sufficiently rich. In Section 5.5, we also show that relational Hoare logic is relatively complete w.r.t. Hoare logic, in a precise mathematical sense.

5.4 Verification condition generation

The verification condition generation algorithm for relational judgments takes as inputs two statements c_1 and c_2 and two relational assertions Φ and Ψ and returns a set of relational verification conditions whose validity entails the validity of the relational judgment $\models c_1 \sim c_2 : \Phi \Rightarrow \Psi$.

The natural strategy for computing relational preconditions is to traverse both programs in lockstep. The lockstep rules for relational preconditions are given in Figure 5.4. Note that each rule requires that the two programs have the same top-level construct, so that in effect relational preconditions are only defined for pairs of programs that are structurally equivalent.

The most interesting cases are the computation of preconditions for conditionals and loops. They both require the guards of the two statements to be equivalent, so that relational preconditions are non-trivial only for pairs of programs that have the same control flow.

Additionally, the rule for loops requires the existence of a loop invariant, which is provided by an external oracle rinvgen that takes as input two loops and a relational post-condition, and a relational assertion intended to be a relational loop invariant that entails equivalence of guard and (with the negation of the loop guard) the post-condition.

Given a Hoare quadruple $c_1 \sim c_2 : \Phi \Rightarrow \Psi$, we define

$$\mathsf{RVC}(\models c_1 \sim c_2 : \Phi \Rightarrow \Psi) \triangleq \{\Phi \implies \Phi_0 \mid \Phi_0 \in \mathsf{rvcgen}\,(c_1, c_2, \Psi, \emptyset)\}$$

Verification condition generation is sound with respect to Hoare logic, i.e. validity of verification conditions entails provability of Hoare triples.

Theorem 5.1. If $\models \mathsf{RVC}(\models c_1 \sim c_2 : \Phi \Rightarrow \Psi)$, then $\models c_1 \sim c_2 : \Phi \Rightarrow \Psi$.

The converse is not true, i.e. the afore-defined algorithm for relational verification condition generation is incomplete; in particular, it does not provide any support for one-sided reasoning. This incompleteness can be resolved by introducing additional rules for one-sided reasoning, such as:

$$\frac{\begin{array}{c}\mathsf{rvcgen}\left(\begin{array}{c} c_1 \\ c_2 \end{array}, \psi, \Xi\right) = (\phi, \Xi) \\ \mathsf{rvcgen}\left(\begin{array}{c} c_1' \\ c_2 \end{array}, \psi, \Xi\right) = (\phi', \Xi') \\ \phi_0 \triangleq b_1\langle 1\rangle \implies \phi \wedge \neg b_1\langle 1\rangle \implies \phi'\end{array}}{\mathsf{rvcgen}\left(\begin{array}{c} \mathsf{if}\ b_1\ \mathsf{then}\ c_1\ \mathsf{else}\ c_1' \\ c_2 \end{array}, \psi, \Xi\right) = (\phi_0, \Xi \cup \Xi')}\ [\textsc{Rvc-Cond-L}]$$

and carefully controlling their application so that the computation of the precondition remains deterministic. However, lockstep computation of precondition suffices for many purposes.

5.5 Comparison with product programs

Another common method to prove relational properties for a pair of programs c_1 and c_2 is to build a product program c that emulates the behavior of the two programs. In this section, we review basic constructions of product programs and formalize their connections with relational Hoare logic.

For the sake of simplicity, we henceforth assume that programs c_1 and c_2 operate (i.e. read and write) on syntactically disjoint sets $\mathbf{Vars}_1$ and $\mathbf{Vars}_2$ of variables, i.e. $\mathrm{dep}(c_1) \cup \mathrm{mod}(c_1) \subseteq \mathbf{Vars}_1$ and $\mathrm{dep}(c_2) \cup \mathrm{mod}(c_2) \subseteq \mathbf{Vars}_2$, with $\mathbf{Vars}_1 \cup \mathbf{Vars}_2 = \emptyset$. Under this assumption, and restricting our attention to relational assertions built from tagged variables $x_1\langle 1\rangle$ and $x_2\langle 2\rangle$, where $x_1 \in \mathbf{Vars}_1$ and $x_2 \in \mathbf{Vars}_2$ respectively, we can view every relational assertion as a standard assertion.

5.5.1 Sequential product program

The simplest form of product program is sequential composition, often known as self-composition (**BartheDR04**).

Definition 5.2 (Sequential product)**.** The sequential product of programs c_1 and c_2 is the program $c_1; c_2$.

Because the two programs operate on disjoint set of variables, it is immediate that their sequential composition emulates their behavior, whenever both programs terminate. This is formalized by the following proposition.

Proposition 5.2 (Soundness and completeness of sequential product)**.** For every memory m, we have:

- $c_1; c_2, m \Downarrow$ iff $c_1, m \Downarrow$ and $c_2, m \Downarrow$;
- if $c_1; c_2, m \Downarrow$, then $[\![c_1; c_2]\!]_m =_{\mathbf{Vars}_1} [\![c_1]\!]_m$ and $[\![c_1; c_2]\!]_m =_{\mathbf{Vars}_2} [\![c_2]\!]_m$.

Moreover, two statements satisfy a relational Hoare specification iff their sequential product program verifies the same specification viewed as a Hoare triple.

Proposition 5.3. The following are equivalent:

- $\models c_1 \sim c_2 : \Phi \Rightarrow \Psi$ is valid in relational Hoare logic;
- $\models c_1; c_2 : \Phi \Rightarrow \Psi$ is valid in Hoare logic;

assuming that for every $(m_1, m_2) \in [\![\Phi]\!]$, we have $c_1, m_1 \Downarrow$ and $c_2, m_2 \Downarrow$.

Furthermore, it is easy to prove that provability of the sequential product in Hoare logic entails provability of the two programs in relational Hoare logic. Indeed, assuming provability of the product programin Hoare logic, observe that there exists an intermediate assertion Θ such that $\vdash c_1 : \Phi \Rightarrow \Theta$ and $\vdash c_2 : \Theta \Rightarrow \Psi$. One can prove by induction on c that $\vdash c : \Phi \Rightarrow \Theta$ implies $\vdash c \sim \mathsf{skip} : \Phi \Rightarrow \Theta$ and $\vdash \mathsf{skip} \sim c : \Phi \Rightarrow \Theta$ to conclude—both relational statements assume the canonical interpretation of assertions as relational assertions. It is more challenging to give a method to transfrom derivations in relational Hoare logic into derivations for the sequential product program in Hoare logic. One can give a non-constructive argument based on relative completeness but the key point is that some proofs are easier relationally.

5.5.2 Synchronous product program

Another simple form of self-composition is the synchronous product program, which forces programs to execute in lockstep. The basic idea is developed by Zaks and Pnueli (2008), under the name cross-product. Here we follow the presentation of (Barthe *et al.*, 2011; Barthe *et al.*, 2016a) and consider an extended programming language with a construct $\mathsf{assert}\ e$. The denotational semantics of $\mathsf{assert}\ e$ is given by the clause:

$$[\![\mathsf{assert}\ e]\!]_m = \begin{cases} m & \text{if } [\![e]\!]_m = \mathsf{tt} \\ \bot & \text{if } [\![e]\!]_m = \mathsf{ff} \end{cases}$$

Definition 5.3 (Synchronous product)**.** A program c is the synchronous product of programs c_1 and c_2, if the judgment $c_1 \times c_2 \to c$ can be derived inductively by the clauses of Figure 5.5.

Note that the definition of synchronous products is partial, i.e. product programs may not exist. However, synchronous products are unique whenever they exist. Moreover, the synchronous product of two statements that are provably related by a relational Hoare specification is always defined, and verifies this same specification—viewed as a Hoare triple. The converse also holds.

Proposition 5.4. The following are equivalent:

- $\vdash c_1 \sim c_2 : \Phi \Rightarrow \Psi$ is derivable in two-sided relational Hoare logic;
- there exists c such that $c_1 \times c_2 \to c$ and $\vdash c : \Phi \Rightarrow \Psi$ is derivable in Hoare logic.

Barthe *et al.* (2016a) extend the rules of synchronous product programs to match the 1-sided rules of relational Hoare logic, and show that the correspondence from Proposition 5.4 extends to this richer setting.

5.5.3 Relational Hoare Logic with explicit product programs

Derivations in relational Hoare logic implicitly construct a product program. This can be captured by considering a proof system with judgments of the form

$$\models c_1 \sim c_2 : \Phi \Rightarrow \Psi \rightsquigarrow c$$

The proof rules of the logic are given in Figure 5.6—rules are only given for some structural rules. This proof system is equivalent to the method proposed in (Barthe *et al.*, 2016a) to combine construction of the product program and verification in Hoare logic. Note that the rules from Figure 5.6 are a special case of the proof system from (Barthe *et al.*, 2017).

$$\frac{}{\mathsf{rvcgen}\left(\begin{array}{c}\mathsf{skip}\\ \mathsf{skip}\end{array}, \psi, \Xi\right) = (\psi, \Xi)}\ [\text{RVC-Skip}]$$

$$\frac{}{\mathsf{rvcgen}\left(\begin{array}{c}x_1 := e_1\\ x_2 := e_2\end{array}, \psi, \Xi\right) = (\psi[e_1\langle 1\rangle/x_1\langle 1\rangle][e_2\langle 2\rangle/x_2\langle 2\rangle], \Xi)}\ [\text{RVC-Ass}]$$

$$\frac{\mathsf{rvcgen}\left(\begin{array}{c}c_1'\\ c_2'\end{array}, \psi, \Xi\right) = (\psi_0, \Xi_0) \qquad \mathsf{rvcgen}\left(\begin{array}{c}c_1\\ c_2\end{array}, \psi_0, \Xi_0\right) = (\phi, \Xi_1)}{\mathsf{rvcgen}\left(\begin{array}{c}c_1; c_1'\\ c_2; c_2'\end{array}, \psi, \Xi\right) = (\phi, \Xi_1)}\ [\text{RVC-Seq}]$$

$$\frac{\begin{array}{c}\mathsf{rvcgen}\left(\begin{array}{c}c_1\\ c_2\end{array}, \psi, \Xi\right) = (\phi, \Xi_1)\\ \mathsf{rvcgen}\left(\begin{array}{c}c_1'\\ c_2'\end{array}, \psi, \Xi\right) = (\phi', \Xi_2)\end{array} \qquad \phi_0 \triangleq b_1\langle 1\rangle = b_2\langle 2\rangle \wedge b_1\langle 1\rangle \implies \phi \wedge \neg b_1\langle 1\rangle \implies \phi'}{\mathsf{rvcgen}\left(\begin{array}{c}\mathsf{if}\ b_1\ \mathsf{then}\ c_1\ \mathsf{else}\ c_1'\\ \mathsf{if}\ b_2\ \mathsf{then}\ c_2\ \mathsf{else}\ c_2'\end{array}, \psi, \Xi\right) = (\phi_0, \Xi_1 \cup \Xi_2)}\ [\text{RVC-Cond}]$$

$$\frac{\begin{array}{c}\mathsf{rinvgen}(\mathsf{while}\ b_1\ \mathsf{do}\ c_1, \mathsf{while}\ b_2\ \mathsf{do}\ c_2, \psi) = \phi\\ \phi \implies b_1\langle 1\rangle = b_2\langle 2\rangle\\ \mathsf{rvcgen}\left(\begin{array}{c}c_1\\ c_2\end{array}, \phi, \Xi\right) = (\phi_0, \Xi_0)\end{array} \qquad \Xi_1 \triangleq \Xi_0 \cup \{\phi \wedge \neg b_1\langle 1\rangle \implies \psi, \phi \wedge b_1\langle 1\rangle \implies \phi_0\}}{\mathsf{rvcgen}\left(\begin{array}{c}\mathsf{while}\ b_1\ \mathsf{do}\ c_1\\ \mathsf{while}\ b_2\ \mathsf{do}\ c_2\end{array}, \psi, \Xi\right) = (\phi, \Xi_1)}\ [\text{RVC-While}]$$

Figure 5.4: Verification condition generation

$$\frac{}{\mathsf{skip} \times \mathsf{skip} \to \mathsf{skip}}\ (\text{P-SKIP})$$

$$\frac{}{(x_1 := e_1) \times (x_2 := e_2) \to x_1 := e_1; x_2 := e_2}\ (\text{P-ASS})$$

$$\frac{c_1 \times c_2 \to c \quad c_1' \times c_2' \to c'}{c_1; c_1' \times c_2; c_2' \to c; c'}\ (\text{P-SEQ})$$

$$\frac{c_1 \times c_2 \to c \quad c_1' \times c_2' \to c'}{\begin{array}{c}(\mathsf{if}\ b_1\ \mathsf{then}\ c_1\ \mathsf{else}\ c_1') \\ \times \\ (\mathsf{if}\ b_2\ \mathsf{then}\ c_2\ \mathsf{else}\ c_2')\end{array} \to \mathsf{assert}\ b_1 = b_2; \mathsf{if}\ b_1\ \mathsf{then}\ c\ \mathsf{else}\ c'}\ (\text{P-COND})$$

$$\frac{c_1 \times c_2 \to c \quad c' \triangleq c; \mathsf{assert}\ b_1 = b_2}{\begin{array}{c}(\mathsf{while}\ b_1\ \mathsf{do}\ c_1) \\ \times \\ (\mathsf{while}\ b_2\ \mathsf{do}\ c_2)\end{array} \to \mathsf{assert}\ b_1 = b_2; \mathsf{while}\ b_1\ \mathsf{do}\ c'}\ (\text{P-WHILE})$$

Figure 5.5: Synchronous product programs

$$\frac{\begin{array}{c}\vdash c_1 : \Phi_1 \Rightarrow \Psi_1 \\ \vdash c_2 : \Phi_2 \Rightarrow \Psi_2 \\ \forall m_1 \in [\![\Phi_1]\!].\ c_1, m_1 \Downarrow \\ \forall m_2 \in [\![\Phi_2]\!].\ c_2, m_2 \Downarrow\end{array}}{\vdash \begin{array}{c} c_1 \\ \sim \\ c_2 \end{array} : \Phi_1\langle 1\rangle \wedge \Phi_2\langle 2\rangle \Rightarrow \Psi_1\langle 1\rangle \wedge \Psi_2\langle 2\rangle \rightsquigarrow c_1; c_2}\ [\textsc{xRHL-Hoare}]$$

$$\frac{\vdash \begin{array}{c} c_1 \\ \sim \\ c_2 \end{array} : \Phi' \Rightarrow \Psi' \rightsquigarrow c \qquad \models \Phi \Longrightarrow \Phi' \qquad \models \Psi' \Longrightarrow \Psi}{\vdash \begin{array}{c} c_1 \\ \sim \\ c_2 \end{array} : \Phi \Rightarrow \Psi \rightsquigarrow c}\ [\textsc{xRHL-Conseq}]$$

$$\frac{\vdash \begin{array}{c} c_1 \\ \sim \\ c_2 \end{array} : \Phi \wedge e \Rightarrow \Psi \rightsquigarrow c \qquad \vdash \begin{array}{c} c_1 \\ \sim \\ c_2 \end{array} : \Phi \wedge \neg e \Rightarrow \Psi \rightsquigarrow c'}{\vdash \begin{array}{c} c_1 \\ \sim \\ c_2 \end{array} : \Phi \Rightarrow \Psi \rightsquigarrow \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c'}\ [\textsc{xRHL-Case}]$$

$$\frac{\vdash \begin{array}{c} c_1 \\ \sim \\ c_2 \end{array} : \Phi \Rightarrow \Theta \rightsquigarrow c \qquad \vdash \begin{array}{c} c_1' \\ \sim \\ c_2' \end{array} : \Theta \Rightarrow \Psi \rightsquigarrow c'}{\vdash \begin{array}{c} c_1; c_1' \\ \sim \\ c_2; c_2' \end{array} : \Phi \Rightarrow \Psi \rightsquigarrow c; c'}\ [\textsc{xRHL-Seq}]$$

$$\frac{\begin{array}{c}\models \Phi \implies e_1\langle 1\rangle = e_2\langle 2\rangle \\ \vdash \begin{array}{c} c_1 \\ \sim \\ c_2 \end{array} : \Phi \wedge e_1\langle 1\rangle \Rightarrow \Psi \rightsquigarrow c \\ \vdash \begin{array}{c} c_1' \\ \sim \\ c_2' \end{array} : \Phi \wedge \neg e_1\langle 1\rangle \Rightarrow \Psi \rightsquigarrow c'\end{array}}{\vdash \begin{array}{c} \mathsf{if}\ e_1\ \mathsf{then}\ c_1\ \mathsf{else}\ c_1' \\ \sim \\ \mathsf{if}\ e_2\ \mathsf{then}\ c_2\ \mathsf{else}\ c_2' \end{array} : \Phi \Rightarrow \Psi \rightsquigarrow \mathsf{if}\ e_1\ \mathsf{then}\ c\ \mathsf{else}\ c'}\ [\textsc{xRHL-Cond}]$$

6

Equivalence and robustness

This chapter illustrates the use of relational Hoare logic for proving equivalence of programs. In particular, we show how relational Hoare logic can be used for translation validation, i.e. showing that a program is equivalent to its optimized form. We then show how relational Hoare logic can be used for proving so-called program robustness.

6.1 Simple examples

We start with a simple example that illustrates the benefits of relational reasoning. We consider monotonicity of **sum** from Figure 4.4. Monotonicity in the first argument is captured by the judgement:

$$\textbf{sum} \sim \textbf{sum} : p\langle 1\rangle \leq p\langle 2\rangle \wedge k\langle 1\rangle = k\langle 2\rangle \wedge n\langle 1\rangle = n\langle 2\rangle \Rightarrow s\langle 1\rangle \leq s\langle 2\rangle$$

Informally, the validity of the judgment can be derived from the validity of the logical formula

$$p_1 \leq p_2 \implies p_1 \cdot n + k \cdot \frac{n(n-1)}{2} \leq p_2 \cdot n + k \cdot \frac{n(n-1)}{2}$$

The validity of the judgment can be proved using Relational Hoare Logic, with relational loop invariant

$$s\langle 1\rangle \leq s\langle 2\rangle \wedge k\langle 1\rangle = k\langle 2\rangle \wedge n\langle 1\rangle = n\langle 2\rangle$$

Of course, we can also prove the validity of the statement using the rule of consequence and the [HOARE] rule to connect with the Hoare judgments from Chapter 4. However, as previously noted, proving the correctness of **sum** in Hoare logic requires quadratic invariants.

Similarly, one can prove that **sum** is monotonic in its second argument. However, it is more challenging to prove that **sum** is monotonic in its third argument:

$$\mathbf{sum} \sim \mathbf{sum} : n\langle 1\rangle \leq n\langle 2\rangle \wedge p\langle 1\rangle = p\langle 2\rangle \wedge k\langle 1\rangle = k\langle 2\rangle \Rightarrow s\langle 1\rangle \leq s\langle 2\rangle$$

even though the validity of the statement can be argued informally in the same way as for the first and second arguments. The problem is due to the fact that the logic does not provide 2-sided rules to reason about loops that perform different numbers of iterations. This limitation can be alleviated to some extent using the [RHL-STRUCT] rule, or by adding a more general rule which allows loops to perform different numbers of iterations, as in (Barthe *et al.*, 2017). For instance, one can add the following axiom:

$$\mathsf{while}\ b\ \mathsf{do}\ c \equiv \mathsf{while}\ b \wedge b'\ \mathsf{do}\ c; \mathsf{while}\ b\ \mathsf{do}\ c$$

and use the [RHL-STRUCT] rule to prove that **sum** is monotonic in its last argument.

A similar problem arises when trying to prove:

$$\mathbf{sum} \sim \mathbf{sum} : p\langle 1\rangle = 0 \wedge p\langle 2\rangle = 1 \wedge k\langle 1\rangle = k\langle 2\rangle \wedge n\langle 1\rangle = n\langle 2\rangle + 1 \Rightarrow s\langle 1\rangle = s\langle 2\rangle$$

because the loops will do a different number of iterations—even though the additional iteration adds 0 and thus has the effect of a skip. This can be resolved by adding the following axiom:

$$\mathsf{while}\ b\ \mathsf{do}\ c \equiv \mathsf{if}\ b\ \mathsf{then}\ (c; \mathsf{while}\ b\ \mathsf{do}\ c)$$

Nevertheless, it is not always possible to use 2-sided rules for simplifying relational proofs. For instance, proving equivalence between exp and fastexp is best proved using [RHL-HOARE], and showing that the two programs compute exponentiation.

6.2 Translation validation

[GB1]: Induction variable strength reduction, loop unswitching, code sinking

6.3 More examples with a general rule for loops

An alternative to the [Struct] rule is to use a rule for loops that does not require the two programs to make the same number of iterations. We only present a simplified rule, and refer the reader to (Barthe *et al.*, 2017) for the most general rule:

$$\frac{\begin{array}{c}\Theta \implies (e_1 \vee e_2) = e \\ \Theta \wedge e \implies \oplus\{p_0, p_1, p_2\} \\ \Theta \wedge p_0 \wedge e \implies e_1 = e_2 \\ \Theta \wedge p_1 \wedge e \implies e_1 \\ \Theta \wedge p_2 \wedge e \implies e_2 \\ \mathsf{term}(\mathsf{while}\ (e_1 \wedge p_1)\ \mathsf{do}\ c_1) \\ \mathsf{term}(\mathsf{while}\ (e_2 \wedge p_2)\ \mathsf{do}\ c_2) \\ \models \mathsf{if}\ e_1\ \mathsf{then}\ c_1 \sim \mathsf{if}\ c_2\ \mathsf{then}\ e_2 : \Theta \wedge p_0 \Rightarrow \Theta \\ \models c_1 \sim \mathsf{skip} : \Theta \wedge e_1 \wedge p_1 \Rightarrow \Theta \\ \models \mathsf{skip} \sim c_2 : \Theta \wedge e_2 \wedge p_2 \Rightarrow \Theta\end{array}}{\models \mathsf{while}\ e_1\ \mathsf{do}\ c_1 \sim \mathsf{while}\ e_2\ \mathsf{do}\ c_2 : \Theta \Rightarrow \Theta \wedge \neg e_1 \wedge \neg e_2}$$

The rule interleaves synchronous and asynchronous executions of the loop bodies, as reflected by its last three premises. The first set of premises defines the conditions under which interleavings must be considered. The first premise specifies an expression e, which may mention variables from both sides, that holds true exactly when at least one of the guards is true. Next, the next premise states that whenever e is valid, exactly one of the tests p_0, p_1, and p_2 must hold—this is captured by the notation $\oplus\{p_0, p_1, p_2\}$. These tests must satisfy some additional conditions, given in the third, fourth, and fifth premises, and guide the analysis of the loop bodies. If p_0 holds, then both guards should be equal and we can execute the two sides one iteration, preserving the loop invariant Θ. If p_1 holds and the right loop has not terminated yet, then the left loop also has not terminated yet (i.e., e_2 holds), we

may execute the left loop one iteration. If p_2 holds and the left loop has not terminated yet (i.e., e_1 holds), then the right loop also has not terminated yet and we may execute the right loop one iteration. The sixth and seven premises deal with termination. Note that some condition on termination is needed for soundness of the logic: if the left loop terminates while the right loop does not terminates, it is impossible to relate the two loops. So, we require that the first and second loops are terminating assuming p_1 and p_2 respectively. This ensures that there are only finitely many steps where we execute the left or right loop separately.

Using this new rule, we can deal with several new examples.

6.4 Sensitivity

7

Information flow and program counter security

This chapter explores applications of relational Hoare logic to information flow security and program counter security.

7.1 Information flow

One of the main motivations for the development of relational Hoare logic is information flow security. The main goal of information flow security is to establish that programs do not reveal any confidential information during execution. In this chapter, we consider a basic information flow policy called $(\mathcal{X}, \mathcal{Y})$-non-interference. Informally, $(\mathcal{X}, \mathcal{Y})$-non-interference considers a setting in which confidential information is initially stored in variables in $\mathsf{C}(\mathcal{X})$, and where an attacker can observe the final values of variables in $\mathcal{Y}$. The guarantee is that the attacker will not learn anything about the secrets.

Definition 7.1 (Non-interference)**.** A statement c is $(\mathcal{X}, \mathcal{Y})$ non-interfering, written $\mathsf{NI}_{\mathcal{X},\mathcal{Y}}(c)$, iff for every memories m_1, m_2, m_1' and m_2' such that $[\![c]\!]_{m_1} = m_1'$ and $[\![c]\!]_{m_2} = m_2'$, we have

$$m_1 =_{\mathcal{X}} m_2 \implies m_1' =_{\mathcal{Y}} m_2'$$

$$
\begin{aligned}
\mathsf{dep}^{\#}_{\mathsf{skip}}(\mathcal{Y}) &= \mathcal{Y} \\
\mathsf{dep}^{\#}_{c;c'}(\mathcal{Y}) &= \mathsf{dep}^{\#}_{c}(\mathsf{dep}^{\#}_{c'}(\mathcal{Y})) \\
\mathsf{dep}^{\#}_{x:=e}(\mathcal{Y}) &= \mathcal{Y}[e/x] \\
\mathsf{dep}^{\#}_{\mathsf{if}\ b\ \mathsf{then}\ c\ \mathsf{else}\ c'}(\mathcal{Y}) &= \begin{cases} \mathcal{Y} & \text{if} (\mathrm{mod}(c) \cup \mathrm{mod}(c')) \cap \mathcal{Y} = \emptyset \\ \mathrm{vars}(b) \cup \mathsf{dep}^{\#}_{c}(\mathcal{Y}) \cup \mathsf{dep}^{\#}_{c'}(\mathcal{Y}) & \text{otherwise} \end{cases} \\
\mathsf{dep}^{\#}_{\mathsf{while}\ b\ \mathsf{do}\ c}(\mathcal{Y}) &= \begin{cases} \mathcal{Y} & \text{if} \, \mathrm{mod}(c) \cap \mathcal{Y} = \emptyset \\ \mathcal{Y}' & \text{otherwise and } \mathcal{Y}, \mathrm{vars}(b), \mathsf{dep}^{\#}_{c}(\mathcal{Y}') \subseteq \mathcal{Y}' \end{cases}
\end{aligned}
$$

where

$$
\mathcal{Y}[e/x] = \begin{cases} \mathcal{Y} & \text{if } x \notin \mathcal{Y} \\ \mathcal{Y} \setminus \{x\} \cup \mathrm{vars}(e) & \text{if } x \in \mathcal{Y} \end{cases}
$$

Figure 7.1: Backwards information flow analysis

We note that our definition of non-interference only considers terminating executions—for this reason it is generally called termination-insensitive non-interference in the literature.

7.1.1 Non-interference from relational Hoare logic

It immediately follows from the soundness of relational Hoare logic that it can be used for proving non-interference. More generally, one can characterize non-interference using Hoare quadruples.

Proposition 7.1 (Non-interference as Hoare quadruples)**.**

- If $\models c \sim c : =_{\mathcal{X}} \Rightarrow =_{\mathcal{Y}}$ then $\mathsf{NI}_{\mathcal{X},\mathcal{Y}}(c)$.
- Conversely, if $\mathsf{NI}_{\mathcal{X},\mathcal{Y}}(c)$ and $\mathsf{term}(c)$ then $\models c \sim c : =_{\mathcal{X}} \Rightarrow =_{\mathcal{Y}}$.

7.1.2 Automating non-interference proofs

In practice, it is often desirable to automate non-interference proofs using a variant of the precondition calculus, or a forward algorithm similar to symbolic evaluation. We review the backwards approach here.

The backwards analysis takes as input a statement c and a set of variables $\mathcal{Y}$ and returns another set of variables $\mathsf{dep}_c^\#(\mathcal{Y})$ such that

$$\models c \sim c : =_{\mathsf{dep}_c^\#(\mathcal{Y})} \Rightarrow =_{\mathcal{Y}}$$

The rules are given in Figure 7.1. We comment on some of the most interesting rules.

The rule for assignment distinguishes whether the assigned variable x is in the set $\mathcal{Y}$ or not. In the former case, it is removed from the set and the set of free variables of the assigned expression is added instead. In the latter case, the set is not modified.

The rule for while loops requires the existence of a set $\mathcal{Y}'$ such that

$$\mathcal{Y} \cup \mathrm{vars}(b) \cup \mathsf{dep}_c^\#(\mathcal{Y}' \cup \mathrm{vars}(b))) \subseteq \mathcal{Y}'$$

Informally, this requires that $=_{\mathcal{Y}'}$ is an invariant for the loop and entails the equivalence of guards; moreover, it must also satisfy $\mathcal{Y} \subseteq \mathcal{Y}'$ so that $=_{\mathcal{Y}}$ can be established as a valid post-condition using the rule of consequence.

Such a set $\mathcal{Y}'$ always exists; take $\mathcal{Y}'$ to be the set of all variables. However, it is desirable to select the smallest possible set $\mathcal{Y}'$; one way to compute $\mathcal{Y}'$ is as follows: first, set $\mathcal{Y}_0 = \mathcal{Y} \cup \mathrm{vars}(b)$ and compute $\mathcal{Y}_1 = \mathsf{dep}_c^\#(\mathcal{Y}_0)$. If $\mathcal{Y}_1 \subseteq \mathcal{Y}_0$, then set $\mathcal{Y}' = \mathcal{Y}_0$. Else, set $\mathcal{Y}_1 = \mathcal{Y}_0 \cup \mathcal{Y}_1$ and repeat the process until hitting some k such that $\mathcal{Y}_{k+1} \subseteq \mathcal{Y}_k$. If such a k is found before some fixed number of iterations K, set $\mathcal{Y}' = \mathcal{Y}_k$. Else, set $\mathcal{Y}'$ to be the set of all variables.

The correctness of $\mathsf{dep}^\#$ is stated w.r.t. relational Hoare Logic.

Proposition 7.2 (Correctness of $\mathsf{dep}^\#$)**.** For every statement c and set of variables $\mathcal{Y}$, the judgment

$$\vdash c \sim c : =_{\mathsf{dep}_c^\#(\mathcal{Y})} \Rightarrow =_{\mathcal{Y}}$$

is derivable in relational Hoare logic and hence

$$\models c \sim c : =_{\mathsf{dep}_c^\#(\mathcal{Y})} \Rightarrow =_{\mathcal{Y}}$$

The analysis is incomplete, for several reasons. For example, the rule for assignments requires equivalence for the variables of the assigned

expression; this is too strong in practice, e.g. if $e \triangleq x - x + y$ or $e = 0 \times x + y$. Similarly, the rule for conditionals requires equivalence for the variables of the guard, even if no variables in $\mathcal{Y}$ is modified by the branches of the statements. Finally, the iterative process to compute the set of variables for a loop may not converge within the prescribed number of iterations, and may thus lead to an overly conservative set of variables.

7.1.3 Embedding information flow typing

Information flow security is often proved using a type system. Here we consider a simple type system inspired from (Volpano and Smith, 1997). Let $\mathcal{H}$ be a subset of confidential variables, and let $\mathcal{L} = \complement(\mathcal{H})$. We say that an expression e is high, written $\vdash e : H$, if $\mathrm{vars}(e) \cap \mathcal{H} \neq \emptyset$, and low, written $\vdash e : L$, otherwise. The typing rules are of the form $\vdash c : \tau$ cmd, where $\tau \in \{H, L\}$. We assume that $L \leq H$.

The typing rules are given in Figure 7.2. The subtyping rule [IF-Sub] allows to view a statement of type H cmd as a statement of type L cmd. The rule [IF-Ass] for assignments requires that the security level of the variable is higher than the level of the assigned expression. The rule [IF-Seq] for sequential composition can be used to compose typable statements of the same level. The rule [IF-Cond] for conditionals requires that the security level of the branches is equal (or with the subtyping rule higher) than the security level of the guard. Finally, the rule [IF-While] for loops requires that the security level of the loop bodyis equal (or with the subtyping rule higher) than the security level of the guard.

Lemma 7.1.

- If $\vdash c : H$ cmd then $\mathrm{mod}(c) \subseteq \mathcal{H}$.
- If $\vdash c : L$ cmd then $\mathsf{dep}_c^{\#}(\mathcal{L}) \subseteq \mathcal{L}$.

Both are proved by induction on the derivation. As a corollay, we recover soundness of the type system.

Proposition 7.3 (Embedding information flow typing)**.** If $\vdash c : \tau$ cmd, then $\mathsf{NI}_{\mathcal{L},\mathcal{L}}(c)$.

$$\frac{\vdash x : \tau \quad \vdash e : \tau' \quad tau' \leq \tau}{\vdash x := e : \tau \ \mathsf{cmd}} \ [\text{IF-Ass}]$$

$$\frac{\vdash c : \tau \ \mathsf{cmd} \qquad \vdash c' : \tau \ \mathsf{cmd}}{\vdash c; c' : \tau \ \mathsf{cmd}} \ [\text{IF-Seq}]$$

$$\frac{\vdash b : \tau \qquad \vdash c : \tau \ \mathsf{cmd} \qquad \vdash c' : \tau \ \mathsf{cmd}}{\vdash \mathsf{if}\ b\ \mathsf{then}\ c\ \mathsf{else}\ c' : \tau \ \mathsf{cmd}} \ [\text{IF-Cond}]$$

$$\frac{\vdash b : \tau \qquad \vdash c : \tau \ \mathsf{cmd}}{\vdash \mathsf{while}\ b\ \mathsf{do}\ c : \tau \ \mathsf{cmd}} \ [\text{IF-While}]$$

$$\frac{\vdash c : H \ \mathsf{cmd}}{\vdash \mathsf{while}\ b\ \mathsf{do}\ c : L \ \mathsf{cmd}} \ [\text{IF-Sub}]$$

Figure 7.2: Information flow typing

The proof of the embedding follows from the correctness of $\mathsf{dep}^{\#}$ (Proposition 7.2).

7.2 Program counter security

A common practice to minimize the risks of side-channel attacks against security implementations is to write programs that do not branch on secrets. Molnar *et al.* (2005) formalize this measure by introducing a simple but useful model, which they coin the program counter security model. The program counter security model is based on the instrumented semantics of programs from Section 3.3.1. In addition to returning the program's output, the instrumented semantics also tracks the sequence of control-flow decisions taken by the program during execution. This sequence of booleans represents the information leaked to the adversary during execution. A program is then said to be secure in the program counter security model if its leakage does not depend on secrets.

Definition 7.2 (Program counter security)**.** A statement c is program counter secure w.r.t. a relational assertion Φ iff for every pair of memories $(m_1, m_2) \in [\![\Phi]\!]$ such that $[\![c_1]\!]^{\mathrm{cf}}_{m_1} = (m_1', \ell_1)$ and $[\![c_2]\!]^{\mathrm{cf}}_{m_2} = (m_2', \ell_2)$, we have $\ell_1 = \ell_2$.

As a special case, we say that a statement c is program counter secure iff it is program counter secure w.r.t. the relational assertion $\top$.

To reason about program counter security, we instrument programs with ghost code that keeps track of their leakage, and we use Relational Hoare Logic to prove that programs are secure in the program counter security model. The soundness and (relative) completeness of the approach is captured by the following statements.

Proposition 7.4 (Verification of program counter security)**.** Let Φ be a relational assertion and let c be a statement. Let c^ℓ denote its instrumentation with a ghost variable ℓ that tracks control-flow. Then the following are equivalent:

1. c is program counter secure w.r.t. Φ;
2. $\vdash c^\ell \sim c^\ell : \Phi \Rightarrow \ell\langle 1\rangle = \ell\langle 2\rangle$ is derivable in relational Hoare Logic;
3. $\models \mathsf{RVC}(c^\ell \sim c^\ell : \Phi \Rightarrow \ell\langle 1\rangle = \ell\langle 2\rangle)$;

The proof of relative completeness of the methods was first established in Almeida *et al.*, 2016 for a more complex property called cryptographic constant-time. In the remainder of this chapter, we will consider various relaxations of program counter security. Rather than defining these relaxations in terms of the denotational semantics, we will informally justify how to model them directly in relational Hoare logic.

Interestingly, it is also possible to prove automatically program counter security w.r.t. $=_{\mathcal{X}}$ using a mild modification of $\mathsf{dep}^{\#}$ from the previous section. The modification is shown in Figure 7.3. We have:

Proposition 7.5. If $\mathsf{dep}^{\mathsf{cf}}_{c^\ell}(\{\ell\}) \subseteq \mathcal{X}$ then

$$\vdash c \sim c : =_{\mathcal{X}} \Rightarrow =_{\{\ell\}}$$

is derivable in relational Hoare logic and therefore c is program counter secure w.r.t. $=_{\mathcal{X}}$.

$$
\begin{aligned}
\mathsf{dep}^{\mathsf{cf}}_{\mathsf{skip}}(\mathcal{Y}) &= \mathcal{Y} \\
\mathsf{dep}^{\mathsf{cf}}_{c;c'}(\mathcal{Y}) &= \mathsf{dep}^{\mathsf{cf}}_{c}(\mathsf{dep}^{\mathsf{cf}}_{c'}(\mathcal{Y})) \\
\mathsf{dep}^{\mathsf{cf}}_{x:=e}(\mathcal{Y}) &= \mathcal{Y}[e/x] \\
\mathsf{dep}^{\mathsf{cf}}_{\mathsf{if}\ b\ \mathsf{then}\ c\ \mathsf{else}\ c'}(\mathcal{Y}) &= \mathrm{vars}(b) \cup \mathsf{dep}^{\mathsf{cf}}_{c}(\mathcal{Y}) \cup \mathsf{dep}^{\mathsf{cf}}_{c'}(\mathcal{Y}) \\
\mathsf{dep}^{\mathsf{cf}}_{\mathsf{while}\ b\ \mathsf{do}\ c}(\mathcal{Y}) &= \mathcal{Y}' \text{ if } \mathcal{Y}, \mathrm{vars}(b), \mathsf{dep}^{\mathsf{cf}}_{c}(\mathcal{Y}') \subseteq \mathcal{Y}'
\end{aligned}
$$

where

$$
\mathcal{Y}[e/x] = \begin{cases} \mathcal{Y} & \text{if } x \notin \mathcal{Y} \\ \mathcal{Y} \setminus \{x\} \cup \mathrm{vars}(e) & \text{if } x \in \mathcal{Y} \end{cases}
$$

Figure 7.3: Program counter security analysis

7.2.1 Secure comparison

Checking equality between secret values, e.g. passwords, is a basic routine used to implement access control and many other security mechanisms. However, the naive implementation of equality checking for fixed-length bitstrings is not secure in the program counter security model. Indeed, consider the following algorithm, which operates on two bitstrings s_1 and s_2 of length n, instrumented with a list ℓ that models leakage:

naivecompare$(s_1 : \{0,1\}^n, s_2 : \{0,1\}^n) : \{0,1\}$
$\ell := \epsilon$;
$i := 0$;
while $i < n$ do
  $\ell := 1 :: \ell$;
  if $s_1[i] = s_2[i]$ then $\ell := 1 :: \ell; i := i + 1$ else $\ell := 0 :: l$; return 0;
$\ell := 0 :: \ell$;
return 1

Program counter security of **naivecompare** is equivalent to requiring that the value of ℓ is equal for any two executions with arbitrarily different values for s_1 and s_2. But one can see that algorithm returns as soon as it encounters an index i where the two bitstrings differ and

$\mathbf{sm}(x : G, k : \{0,1\}^n) : G$
$\ell := \epsilon;$
$y := 1;$
$i := 0;$
while $i < n$ do
  if $k[i] = 1$ then $\ell := 1 :: \ell; y := y^2 \times x$ else $\ell := 0 :: l; y := y^2;$
  $i := i + 1$
$\ell := 0 :: \ell;$
return y

Figure 7.4: Square and multiply

hence the length of ℓ is equal to $2i$. Therefore the program is not secure.

This issue is addressed by the next algorithm, which performs all comparisons of individual bits:

$\mathbf{compare}(s_1 : \{0,1\}^n, s_2 : \{0,1\}^n) : \{0,1\}$
$\ell := \epsilon;$
$r := 1;$
$i := 0;$
while $i < n$ do
  $\ell := 1 :: \ell;$
  if $s_1[i] = s_2[i]$ then $\ell := 1 :: \ell$ else $\ell := 0 :: l; r := 0;$
$\ell := 0 :: \ell;$
  $i := i + 1$
return r

Program counter security of **compare** is proved by deriving the RHL judgment in the 2-sided fragment of relational Hoare logic:

$$\vdash \mathbf{compare} \sim \mathbf{compare} : \top \Rightarrow \ell\langle 1\rangle = \ell\langle 2\rangle$$

which can be proved by a straightforward application of two-sided rules.

7.2.2 Square-and-always multiply

Next, we consider the square and multiply algorithm to exponentiate in a group. The algorithm takes as input a group element x and a

$\mathbf{sam}(x : G, k : \{0,1\}^n) : G$
$\ell := \epsilon;$
$y := 1;$
$i := 0;$
while $i < n$ do
  $\ell := 1 :: \ell;$
  $y_1 := y^2;$
  $y_2 := y_1 \times x;$
  $y := (1 - k[i]) \cdot y_1 + k[i] \cdot y_2;$
  $i := i + 1$
$\ell := 0 :: \ell;$
return y

Figure 7.5: Square-and-always-multiply

fixed length bitstring k, and computes x^k. The code of the algorithm is shown in Figure 7.4. In this implementation, leakage depends on k and thus the program is not secure in the program counter security model with respect to k. This can be addressed by using a square and always multiply algorithm, as shown in Figure 7.5, were $b \cdot x$ is a scalar multiplication, with $0 \cdot x = 0$ and $1 \cdot x = x$.

Program counter security of **sam** is established by proving the RHL judgment in two-sided relational Hoare logic:

$$\vdash \mathbf{sam} \sim \mathbf{sam} : \top \Rightarrow \ell\langle 1\rangle = \ell\langle 2\rangle$$

which can be proved by a straightforward application of two-sided rules.

7.2.3 Finding smallest value in array and insertion sort

As a simple example, consider the task of computing the minimum value in an array of secret values:

$$
\begin{array}{l}
\mathbf{min}(a : \mathbb{Z}[n]) : \mathbb{Z} \\
\ell := \epsilon; \\
y := a[0]; \\
i := 1; \\
\mathsf{while}\ i < n\ \mathsf{do} \\
\quad \ell := 1 :: \ell; \\
\quad \mathsf{if}\ a[i] < y\ \mathsf{then}\ \ell := 1 :: \ell; y := a[i]\ \mathsf{else}\ \ell := 0 :: \ell; \\
\quad i := i + 1 \\
\ell := 0 :: \ell; \\
\mathsf{return}\ y
\end{array}
$$

The program is not secure in the program counter security model. Indeed, the leakage of the 2-elements arrays $[0, 1]$ and $[[1, 0]$ differ. However, we can define a relational assertion Φ which relates any two arrays $a, a' : \mathbb{Z}[n]$ such that for every $0 \leq i, j < n$, $a[i] < a[j]$ iff $a'[i] < a'[j]$. Indeed, one can prove in the 2-sided fragment of relational Hoare logic:

$$\vdash \mathbf{min} \sim \mathbf{min} : \Phi \Rightarrow \ell\langle 1\rangle = \ell\langle 2\rangle$$

The proof uses as relational loop invariant:

$$\forall i \leq k < n.\ a[i]\langle 1\rangle < y\langle 1\rangle \Leftrightarrow a[i]\langle 2\rangle < y\langle 2\rangle$$

In a similar way, consider the following program for insertion sort from Figure 7.6. Insertion sort is not secure in the program counter model. However, it is Φ-secure. This can be established by proving the RHL judgment in the 2-sided fragment of relational Hoare logic:

$$\vdash \mathbf{isort} \sim \mathbf{isort} : \Phi \Rightarrow \ell\langle 1\rangle = \ell\langle 2\rangle$$

7.2.4 Square and multiply

One may further weaken the security notion to require that leakages are related by some relational assertion Ψ. Informally, Ψ captures how much is leaked by program execution. In practice, this weaker notion of

```
isort(a : ℤ[n]) : ℤ[n]
ℓ := ϵ;
i := 1;
while i < n do
    ℓ := 1 :: ℓ;
    j := i;
    while j > 0 ∧ a[j − 1] > a[j] do
        ℓ := 1 :: ℓ;
        x := a[j];
        a[j] := a[j − 1];
        a[j − 1] := x;
        j := j − 1;
    end while
    ℓ := 0 :: ℓ;
    i := i + 1;
end while
ℓ := 0 :: ℓ
return a
```

Figure 7.6: Insertion sort

security holds for related input states, so we require that for every two executions starting from states related by Φ, leakages are related by Ψ.

For instance, let us return to the square and multiply program from Figure 7.4. Now consider a malicious observer that is able to count the number of multiplications and additions performed by the program. It is easy to see that such an observer cannot distinguish between two executions whose leakage is equal up to permutation. Therefore, we define the relation $l_1 \equiv_\pi l_2$ to hold whenever the two lists are equal up to permutation. We can prove:

$$\vdash \mathbf{sm} \sim \mathbf{sm} : \mathsf{hw}(k\langle 1\rangle) = \mathsf{hw}(k\langle 1\rangle) \Rightarrow \ell\langle 1\rangle \equiv_\pi \ell\langle 2\rangle$$

The proof is carried in relational Hoare logic with sound instances of the [Struct] rule for program counter security. Specifically, we use notion of program equivalences which allow to split and unroll loops and to replace guards by equivalent guards. This allows to transform a loop that performs n iterations into a loop that performs $i - 1$ iterations, followed by the execution of the i and $i + 1$ iterations, and then a loop that performs the remaining iterations. We prove

$$\models \mathbf{sm} \sim \mathbf{sm} : \mathsf{swap}(j, k\langle 1\rangle, k\langle 2\rangle) \Rightarrow \ell\langle 1\rangle \equiv_\pi \ell\langle 2\rangle$$

where $\mathsf{swap}(j, k, k')$ is a shorthand for

$$k[j] = k'[j+1] \wedge k[j+1] = k'[i] \wedge \forall j' \notin \{j, j+1\}, k[j'] = k'[j']$$

We use the key property:

$$l_1 \equiv_\pi l_2 \Longrightarrow a :: a' :: l_1 \equiv_\pi a' :: a :: l_2$$

From the latter, we can apply structural rules, concretely [Exists] and [Trans] to conclude.

8

Relative cost

9

Cartesian Hoare Logic

Part II

Probabilistic computations

10

Probability sub-distributions

10.1 Distributions

This section introduces the necessary background on probability theory and distributions. For simplicity, we only consider discrete sub-distributions.

Definition 10.1 (Sub-distributions)**.** A (discrete) sub-distribution over a set A is a function $\mu : A \to [0, 1]$ such that the *mass* $|\mu| = \sum_{a \in A} \mu(a)$ of μ is defined and verifies $|\mu| \leq 1$. In particular, the *support* of a sub-distribution μ, defined as $\mathrm{supp}(\mu) = \{a \in A \mid \mu(a) > 0\}$, is necessarily countable, i.e. finite or countably infinite. The set of sub-distributions over A is denoted by $\mathbb{D}(A)$.

We often use the following terminology. Sub-distributions of mass 1 are called *(full) distributions*; other sub-distributions are called *proper sub-distributions*.

We next consider events. Events over a set A are subsets of A. The probability of an event E w.r.t. a sub-distribution μ, written as $\mathbb{P}_\mu[E]$, is defined as $\sum_{x \in E} \mu(x)$. We almost exclusively use the following basic facts about events:

- for every $a \in A$, $\mathbb{P}_\mu[\{a\}] = \mu(a)$ and $0 \leq \mathbb{P}_\mu[\{a\}] \leq 1$;

- for every $E \subseteq A$, $0 \leq \mathbb{P}_\mu[E] \leq 1$;
- $\mathbb{P}_\mu[\emptyset] = 0$;
- $\mathbb{P}_\mu[A] = 1$ if μ is a full distribution;
- for every events E and F, $E \subseteq F$ implies $\mathbb{P}_\mu[E] \leq \mathbb{P}_\mu[F]$;
- for every events E and F, $\mathbb{P}_\mu[E \cup F] = \mathbb{P}_\mu[E] + \mathbb{P}_\mu[F] - \mathbb{P}_\mu[E \cap F]$.

The statistical distance between two distributions $\mu_1, \mu_2 \in \mathbb{D}(A)$ by the clause

$$\mathsf{TV}(\mu_1, \mu_2) = \max_{E \subseteq A} \left| \mathbb{P}_{\mu_1}[E] - \mathbb{P}_{\mu_2}[E] \right|$$

10.2 Monadic structure of distributions

Lawvere Lawvere, 1962 and Giry **Gry82** were among the first to note that distributions can be given a monadic structure. A similar construction can be given for the discrete sub-distributions that we consider.

Definition 10.2 (Monadic structure of sub-distributions)**.** The unit of the monad is the Dirac distribution, which assigns to every $x \in A$ the *Dirac distribution* $\mathbb{1}_x$ defined by the clause:

$$\mathbb{P}_{\mathsf{unit}(x)}[\{y\}] \triangleq \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}$$

The monadic composition takes as input a sub-distribution $\mu \in \mathbb{D}(A)$ and a mapping $M : A \to \mathbb{D}(B)$ and yields a sub-distribution $\mathsf{let}\ a = \mu\ \mathsf{in}\ M\ a \in \mathbb{D}(B)$ defined by the clause:

$$\mathbb{P}_{\mathsf{let}\ a=\mu\ \mathsf{in}\ M\ a}[\{b\}] \triangleq \sum_{a \in \mathrm{supp}(\mu)} \mu(a) \cdot \mathbb{P}_{M(a)}[\{b\}]$$

it is the image distribution of μ along M.

10.3 The partial order of sub-distributions

Sub-distributions are partially ordered by the pointwise inequality inherited from reals: given any two sub-distributions $\mu, \mu' \in \mathbb{D}(A)$, we

have $\mu \leq \mu'$ iff $\mu(a) \leq \mu'(a)$ for every $a \in A$. Moreover, two sub-distributions are equal iff they assign the same value (i.e., probability) to each element in their domain: given any two sub-distributions $\mu, \mu' \in \mathbb{D}(A)$, we have $\mu = \mu'$ iff $\mu(a) = \mu'(a)$ for every $a \in A$. Note that $\mu = \mu'$ iff $\mu \leq \mu'$ and $\mu' \leq \mu$.

We use two key facts about the partial order. First, equality and inequality coincide for full distributions. This is used in some examples to prove the existence of identity couplings.

Lemma 10.1. Let $\mu, \mu' \in \mathbb{D}(A)$.

- If $\mu \leq \mu'$ then $|\mu| \leq |\mu'|$.
- If $|\mu| = 1$ and $\mu \leq \mu'$ then $\mu = \mu'$.

Second, every increasing sequence of sub-distributions converges to its supremum. This is a simple consequence of the Monotone Convergence Theorem for the reals.

Proposition 10.1 (Limit distribution)**.** Let $(\mu_i)_{i \in \mathbb{N}} \in \mathbb{D}(A)$ be an increasing family of sub-distributions, i.e. for every $i \in \mathbb{N}$, we have $\mu_i \leq \mu_{i+1}$ or more explicitly for every $a \in A$:

$$\mathbb{P}_{x \sim \mu_i}[x = a] \leq \mathbb{P}_{x \sim \mu_{i+1}}[x = a]$$

Then the sequence $\mathbb{P}_{x \sim \mu_i}[x = a]$ has a limit in $[0, 1]$ for every $a \in A$. The limit distribution of $(\mu_i)_i$, written $\lim_{i \to \infty} \mu_i$, is defined by the clause:

$$\mathbb{P}_{x \sim \lim_{i \to \infty} \mu_i}[x = a] = \lim_{i \to \infty} \mathbb{P}_{x \sim \mu_i}[x = a]$$

for every $a \in A$. It verifies $\lim_i(\mu_i) = \mu_\infty$.

Formally the limit of the μ_i is defined w.r.t. statistical distance.

10.4 Expectation

The expectation $\mathbb{E}_\mu[f]$ of a function $f : A \to \mathbb{R}^+$ w.r.t. a sub-distribution $\mu \in \mathbb{D}(A)$ is defined as $\sum_{x \in A} \mu(x) f(x)$ when this sum exists, and $+\infty$ otherwise.

[GB2]: add properties as needed

11

The pWhile programming language

We now introduce the pWhile language, a probabilistic extension of the While language from Chapter 3. The two languages only differ in that pWhile features an instruction to sample from a distribution.

11.1 Syntax

We briefly describe how the set of types, expressions, and statements are defined.

11.1.1 Types

The definition of types is similar to the one for While. We assume given a distinguished constructor that maps every type σ in $\mathcal{T}$ to a type $\mathsf{D}(\sigma)$ of (discrete) distributions over σ.

11.1.2 Expressions

We distinguish between expressions, whose definition is similar to the one for While, and distribution expressions. For the latter, we assume given a set of distribution operators. Typical examples of distribution operators include uniform distributions and product distributions.

Definition 11.1 (Distribution expressions)**.** Let **DOp** be a set of distribution operators. We assume that every operator $f \in \mathbf{DOp}$ comes with a declaration of the form $f : \sigma'_1 \times \ldots \times \sigma'_{n'} \times \mathsf{D}(\sigma_1) \times \ldots \times \mathsf{D}(\sigma_n) \to \mathsf{D}(\tau)$ that determines the number and types of elements it takes, and the type of the output. The set **DExpr** of distribution expressions is defined by the following syntax:

$$d ::= | \quad f(e_1, \ldots, e_{n'}, d_1, \ldots, d_n)$$

Each distribution expression d has a distribution type of the form $\mathsf{D}(\sigma)$ where $\sigma \in \mathcal{T}$.

11.1.3 Statements

Statements are defined similarly to While, to the exception of two statements for sampling from a distribution and aborting, i.e. returning the empty distribution.

Definition 11.2 (Statements)**.** The set **Cmd** of statements is defined by the following syntax:

$c ::=$	abort	abort
$\mid$	skip	noop
$\mid$	$x := e$	deterministic assignment
$\mid$	$x \xleftarrow{\$} d$	probabilistic assignment
$\mid$	$c; c$	sequencing
$\mid$	$\mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c$	conditional
$\mid$	$\mathsf{while}\ e\ \mathsf{do}\ c$	while loop

The typing rules for statements are extended in a straightforward way (Figure 11.1). The definition of depending and modified variables is also extended in the obvious way.

Definition 11.3 (Depending and modified variables)**.** The set $\mathrm{dep}(c)$ of depending variables of a statement c is extended by the clauses:

$$\begin{aligned} \mathrm{dep}(x \xleftarrow{\$} d) &= \mathrm{vars}(d) \\ \mathrm{dep}(\mathsf{abort}) &= \emptyset \end{aligned}$$

$$\frac{}{\vdash \mathsf{abort}} \ [\textsc{Abort}]$$

$$\frac{\vdash x : \sigma \qquad \vdash d : \mathbb{D}(\sigma)}{\vdash x \xleftarrow{\$} d} \ [\textsc{Rand}]$$

Figure 11.1: Typing rules for statements

The set $\mathrm{mod}(c)$ of modified variables of a statement c is extended by the clauses:

$$\begin{array}{rcl} \mathrm{mod}(x \xleftarrow{\$} d) & = & \{x\} \\ \mathrm{mod}(\mathsf{abort}) & = & \emptyset \end{array}$$

11.2 Semantics

11.2.1 Types

The interpretation of types from Chapter 3 is extended with the clause:

$$[\![\mathsf{D}(\sigma)]\!] = \mathbb{D}([\![\sigma]\!])$$

11.2.2 Expressions

For simplicity, we require that every distribution expression d is intrepreted as a full distribution (rather than a sub-distribution) over the interpretation of its type.

Definition 11.4 (Semantics of expressions)**.** Suppose given a set-theoretical interpretation $[\![f]\!] \in [\![\sigma_1]\!] \times \ldots \times [\![\sigma_{n'}]\!] \times \mathbb{D}[\![\sigma_1]\!] \times \ldots \times \mathbb{D}[\![\sigma_n]\!] \to \mathbb{D}[\![\tau]\!]$ for every $f : \sigma_1 \times \ldots \times \sigma_{n'} \times \mathsf{D}(\sigma_1) \times \ldots \times \mathsf{D}(\sigma_n) \to \mathsf{D}(\tau)$. The semantics of a distribution expression e with respect to a memory m is defined by the clauses:

$$[\![f(e_1, \ldots, e_{n'}, d_1, \ldots, d_n)]\!]_m \quad = \quad [\![f]\!]([\![e_1]\!]_m, \ldots, [\![e_n]\!]_m, [\![d_1]\!]_m, \ldots, [\![d_n]\!]_m)$$

11.2.3 Statements

We now turn to give a denotational semantics to statements.

Definition 11.5 (Semantics of statements)**.** The denotational semantics $[\![s]\!]$ of a statement s is a function that assigns to every memory $m \in \mathbf{Mem}$ a sub-distribution $[\![s]\!]_m \in \mathbb{D}(\mathbf{Mem})$. The definition of $[\![s]\!]_m$ is given in Figure 11.2.

The semantics of abort is the constant function that maps every initial memory to the null sub-distribution, and the semantics of skip is the function that maps every memory m to the Dirac distribution $\mathbb{1}_m$.

The semantics of a deterministic assignment is a map that takes as input an inital memory m and returns the Dirac distribution $\mathbb{1}_{m[x \leftarrow v]}$, where $m[x \leftarrow v]$ is the memory obtained by updating m with the value v resulting from the evaluation e in memory m.

The semantics of a probabilistic assignment is defined in a similar way. Concretely, the semantics of a probabilistic assignment is a map that takes as input an inital memory m, evaluates the distribution expression d in m, samples v from the resulting distribution and returns the Dirac distribution $\mathbb{1}_{m[x \leftarrow v]}$.

The semantics of a sequential composition is defined as the monadic composition of the semantics of the first and second statements.

The semantics of conditional statements is straightforward: given a memory m, one evaluates the guard e of the conditional in m, and return the output sub-distribution $[\![c_1]\!]_m$ of the true branch if the guard evaluates to true and the output sub-distribution $[\![c_2]\!]_m$ of the false branch if the guard evaluates to false.

The semantics of while loops defined as the limit of its lower approximations; the correctness of the definition relies on the existence of limit distributions (Proposition 10.1).

It is interesting to note that the sequence $[\![\mathsf{while}^i\ e\ \mathsf{do}\ c]\!]_m$ is not converging. However, whenever $|[\![\mathsf{while}\ e\ \mathsf{do}\ c]\!]_m| = 1$, the limit of $[\![\mathsf{while}^i\ e\ \mathsf{do}\ c]\!]_m$ exists, and coincides with the limit of $[\![\mathsf{while}_i\ e\ \mathsf{do}\ c]\!]_m$. The proof of this fact is based on the observation that lower approximations are below approximations, so that the limit of their weight is equal to 1, and on Theorem 10.1. This is used e.g. in the program logic of (Barthe *et al.*, 2018) to prove soundness for the rule for loops.

The following lemmas capture the correctness of dep and mod.

Lemma 11.1. If $m_1 =_{\mathrm{dep}(c)} m_2$ then $[\![c]\!]_{m_1} = [\![c]\!]_{m_2}$.

$$
\begin{aligned}
[\![\mathsf{abort}]\!]_m &= \mathbb{0} \\
[\![\mathsf{skip}]\!]_m &= \mathsf{unit}(m) \\
[\![x := e]\!]_m &= \mathsf{unit}(m[x \leftarrow [\![e]\!]_m]) \\
[\![x \xleftarrow{\$} d]\!]_m &= \mathsf{let}\ v = [\![d]\!]_m\ \mathsf{in}\ \mathsf{unit}(m[x \leftarrow v]) \\
[\![c_1; c_2]\!]_m &= \mathsf{let}\ \mu = [\![c_1]\!]_m\ \mathsf{in}\ [\![c_2]\!]_\mu \\
[\![\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!]_m &= \begin{cases} [\![c_1]\!]_m & \text{if } [\![e]\!]_m = \mathsf{tt} \\ [\![c_2]\!]_m & \text{if } [\![e]\!]_m = \mathsf{ff} \end{cases} \\
[\![\mathsf{while}\ e\ \mathsf{do}\ c]\!]_m &= \sup_{i \in \mathbb{N}}\ \mu_i \\
\text{where } \mu_i &= \left(\mathsf{let}\ \mu = [\![(\mathsf{if}\ e\ \mathsf{then}\ c)^i]\!]_m\ \mathsf{in}\ [\![\mathsf{if}\ e\ \mathsf{then}\ \mathsf{abort}]\!]_\mu\right)
\end{aligned}
$$

Figure 11.2: Denotational semantics of statements

Lemma 11.2. If $[\![c]\!]_m = \mu$ and $m' \in \mathrm{supp}(\mu)$ then $m =_{\mathbb{C}(\mathrm{mod}(c))} m'$.

11.3 Termination

Probabilistic programs exhibit a rich range of termination behaviors. Almost sure termination is an important case when programs terminate with probability 1.

Definition 11.6 (Almost sure termination)**.** A statement c is almost surely terminating, written $\mathsf{ast}(c)$, if $|[\![c]\!]_m| = 1$ for every initial memory m.

It is easy to see that every loop-free program is almost surely terminating; in fact, it satisfies a stronger property, called certain termination, i.e. there exists $n \in \mathbb{N}$ so that the program completes its execution in less than n steps on arbitrary inputs. Proving almost sure termination for more general classes of programs can be achieved by a number of means, including weakest preexpectation calculus (McIver *et al.*, 2018) and martingale-based reasoning (Chakarov and Sankaranarayanan, 2013; Fioriti and Hermanns, 2015; Chatterjee *et al.*, 2017).

11.4 Further reading

The semantics of probabilistic programs has been studied extensively, generally in the more broader context of continuous distributions and often for programs that combine probabilities and non-determinism. A landmark work of Kozen (1981) uses Banach fixpoint theorem and measure-theoretical tools to give a denotational semantics for a purely probabilistic language similar to ours. The probabilistic powerdomain of Jones and Plotkin (1989) is the canonical formalism for the denotational semantics of programs with both probabilities and non-determinism. Our semantics is intended to be more elementary, in the sense that it focuses on discrete sub-distributions and that it only uses the Monotone Convergence Theorem.

[GB3]: TODO: Aumann, QBS, conditioning

12

Union Bound Logic

We use the Union Bound logic from Barthe *et al.* (2016b) for bounding the probability of events on output sub-distributions of probabilistic programs. The logic is based on the union bound, a very simple but effective tool from probability theory.

12.1 Judgments and validity

Judgments are of the form $\models_\beta c : \phi \Rightarrow \psi$ where c is a statement, ϕ, ψ are assertions and $\beta \in [0, 1]$ is a constant.

Informally, a judgment is valid if the probability of $\neg\psi$ in $[\![c]\!]_m$ is upper bounded by β for every memory m that satisfies the precondition ϕ.

Definition 12.1 (Valid judgment)**.** The judgment $\models_\beta c : \phi \Rightarrow \psi$ is valid if $\mathbb{P}_{[\![c]\!](m)}[\neg\psi] \leq \beta$ for every memory m such that $[\![\phi]\!]_m$ holds.

The proof rules for UBHL judgments include structural rules (Figure 12.1), and rules for each construct (Figure 12.2).

We briefly discuss the rules. The [False] rule allows us to conclude that false holds with probability at most 0 in the final memory.

$$[\textsc{False}]\ \frac{}{\models_1 c : \psi \Rightarrow \bot}$$

$$[\textsc{Conseq}]\ \frac{\models \phi' \implies \phi \qquad \models \psi \implies \psi' \qquad \beta \leq \beta' \qquad \models_\beta c : \phi \Rightarrow \psi}{\models_{\beta'} c : \phi' \Rightarrow \psi'}$$

$$[\textsc{Frame}]\ \frac{\mathrm{vars}(\psi) \cap \mathrm{mod}(c) = \emptyset}{\models_0 c : \psi \Rightarrow \psi}$$

$$[\textsc{Case}]\ \frac{\models_\beta c : \phi_1 \Rightarrow \psi \qquad \models_\beta c : \phi_2 \Rightarrow \psi}{\models_\beta c : \phi_1 \vee \phi_2 \Rightarrow \psi}$$

$$[\textsc{Exists}]\ \frac{\forall x : T.\ \models_\beta c : \phi \Rightarrow \psi}{\models_\beta c : \exists x : T.\, \phi \Rightarrow \psi}$$

$$[\textsc{And}]\ \frac{\models_{\beta_1} c : \phi \Rightarrow \psi_1 \qquad \models_{\beta_2} c : \phi \Rightarrow \psi_2}{\models_{\beta_1+\beta_2} c : \psi \Rightarrow \psi_1 \wedge \psi_2}$$

Figure 12.1: Structural UBHL proof rules

$$[\textsc{Skip}]\ \frac{}{\models_0 \mathsf{skip} : \psi \Rightarrow \psi} \qquad [\textsc{Assn}]\ \frac{}{\models_0 x := e : \psi[e/x] \Rightarrow \psi}$$

$$[\textsc{Rand}]\ \frac{\forall m.\ [\![\phi]\!]_m \implies \mathbb{P}_{[\![x \xleftarrow{\$} d]\!]_m}[\neg\psi] \leq \beta}{\models_\beta x \xleftarrow{\$} d(e) : \phi \Rightarrow \psi}$$

$$[\textsc{Seq}]\ \frac{\models_\beta c : \phi \Rightarrow \theta \qquad \models_{\beta'} c' : \theta \Rightarrow \psi}{\models_{\beta+\beta'} c; c' : \phi \Rightarrow \psi}$$

$$[\textsc{Cond}]\ \frac{\models_\beta c : \phi \wedge e \Rightarrow \psi \qquad \models_\beta c' : \phi \wedge \neg e \Rightarrow \psi}{\models_\beta \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c' : \phi \Rightarrow \psi}$$

$$[\textsc{While-CT}]\ \frac{\models_\beta c : \theta \wedge e \Rightarrow \theta \qquad \models_0 (\mathsf{if}\ e\ \mathsf{then}\ c)^n : \phi \Rightarrow \neg e}{\models_{n\beta} \mathsf{while}\ e\ \mathsf{do}\ c : \theta \wedge \phi \Rightarrow \theta \wedge \neg e}$$

$$[\textsc{While-AST}]\ \frac{\models_0 c : \theta \wedge e \Rightarrow \theta \qquad \models_\beta c : \theta \wedge e \Rightarrow \neg e \qquad \beta \neq 1}{\models_0 \mathsf{while}\ e\ \mathsf{do}\ c : \theta \Rightarrow \theta \wedge \neg e}$$

Figure 12.2: Non-structural ubHL proof rules

The [Conseq] rule allows strengthening the pre-condition, weakening the post-condition, and increasing the index—this corresponds to allowing a possibly higher probability of failure.

The frame rule [Frame] preserves assertions that do not mention variables modified by the command. The conjunction rule [And] is another instance of the union bound, allowing us to combine two post-conditions while adding up the failure probabilities. The case rule [Case] is the dual of [And] and takes the maximum failure probability among two post-conditions when taking their disjunction. Finally,

The rule for random sampling [Rand] allows us to assume a proposition ψ about the random sample provided that ψ fails with probability at most β. This is a semantic condition which we introduce as an axiom for each primitive distribution.

The remaining rules are similar to the standard Hoare logic rules, with special handling for the index. The sequence rule [Seq] states that the failure probabilities of the two commands add together; this is simply the union bound internalized in our logic. The conditional rule [If] assumes that the indices for the two branch judgments are equal—which can always be achieved via weakening—keeping the same index for the conditional. Roughly, this is because only one branch of the conditional is executed. The loop rule [While] simply accumulates the failure probability β throughout the iterations; the side conditions ensure that the loop terminates in at most k iterations except with probability $k \cdot \beta$.

12.2 Soundness and completeness

The logic is sound: if the premises of a proof rule are valid, and the side-conditions, if any, hold, then the conclusions of the proof rule are valid.

Proposition 12.1. Every derivable judgment $\models_\beta c : \phi \Rightarrow \psi$ is valid, i.e. for every memory m, $[\![\phi]\!]_m$ implies $\mathbb{P}_{[\![c]\!]_m}[\neg\psi] \leq \beta$.

Unsurprisingly, the proof system is incomplete: there are valid judgments that cannot be proved using the logic. Indeed, the union bound principle is a simple tool, and it induces non-optimal bounds. Concentration inequalities Dubhashi and Panconesi, 2009 is an active field of research that studies advanced methods for improving these bounds, and that is not captured by ubHL.

12.3 Examples

[GB4]: Add dice sampling and other examples

12.4 Further reading

There is a long line of research, spanning more than four decades, on program logics for reasoning about general probabilistic properties

both for purely probabilistic programs and for programs that combine probabilities and non-deterministic choice. Kozen (1985) develops a propositional dynamic logic for a purely probabilistic language; this work has later been extended in many directions. Particularly, a long line of work by McIver and Morgan, summarized in (McIver and Morgan, 2005), develops a weakest pre-expectation calculus for a language with both probabilities and non-determinism. Their work has also been extended in many directions, with applications to security and to complexity analysis. We recommend the reader to consult Kaminski, 2019 for a detailed account of the latter.

13

Probabilistic couplings

13.1 Definition

Informally, a probabilistic coupling for two sub-distributions $\mu_1 \in \mathbb{D}(A_1)$ and $\mu_2 \in \mathbb{D}(A_2)$ is a sub-distribution $\mu \in \mathbb{D}(A_1 \times A_2)$ which lets the two sub-distributions share some randomness. Formally, the definition of probabilistic coupling requires that the left and right projections of μ (also known as its marginals, and defined formally below), coincide with μ_1 and μ_2 respectively.

Definition 13.1 (Marginal sub-distributions)**.** The first and second marginals of a sub-distribution $\mu \in \mathbb{D}(A_1 \times A_2)$ are the two sub-distributions $\pi_1(\mu) \in \mathbb{D}(A_1)$ and $\pi_2(\mu) \in \mathbb{D}(A_2)$ given by

$$\pi_1(\mu)(a_1) = \sum_{a_2 \in A_2} \mu(a_1, a_2) \qquad \pi_2(\mu)(a_2) = \sum_{a_1 \in A_1} \mu(a_1, a_2)$$

For our purposes, we are interested in probabilistic couplings that lie within a given relation, to be thought as the post-condition of the coupling.

Definition 13.2 (Probabilistic couplings)**.** Let $R \subseteq A_1 \times A_2$ be a binary relation. Let $\mu_1 \in \mathbb{D}(A_1)$ and $\mu_2 \in \mathbb{D}(A_2)$. A R-coupling for μ_1 and μ_2

is a discrete sub-distribution $\mu \in \mathbb{D}(A_1 \times A_2)$ such that the following conditions are satisfied:

- **marginals:** $\pi_1(\mu) = \mu_1$ and $\pi_2(\mu) = \mu_2$
- **support:** $\mathrm{supp}(\mu) \subseteq R$

We write $\mu \blacktriangleleft_R \langle \mu_1 \,\&\, \mu_2 \rangle$ whenever μ is a R-coupling of μ_1 and μ_2.

For many purposes, it suffices to know the existence of a R-coupling. This leads to the definition of R-lifting.

Definition 13.3 (Probabilistic lifting of a relation)**.** Let $R \subseteq A_1 \times A_2$ be a binary relation. The probabilistic lifting of the relation R is the binary relation $R^\sharp$ over $\mathbb{D}(A_1) \times \mathbb{D}(A_2)$ such that for every $\mu_1 \in \mathbb{D}(A_1)$ and $\mu_2 \in \mathbb{D}(A_2)$, $(\mu_1, \mu_2) \in R^\sharp$ iff there exists μ such that $\mu \blacktriangleleft_R \langle \mu_1 \,\&\, \mu_2 \rangle$.

13.2 Basic properties

The following is an important consequence of the definition of R-couplings.

Lemma 13.1. If $\mu \blacktriangleleft_R \langle \mu_1 \,\&\, \mu_2 \rangle$ then $|\mu| = |\mu_1| = |\mu_2|$.

Proof. We do the proof only for $\mu_1 = \mu$

$$\begin{array}{rcll} |\mu_1| & = & \sum_{a_1 \in A_1} \mu_1(a_1) & \text{(by definition)} \\ & = & \sum_{a_1 \in A_1} \pi_1(\mu)(a_1) & \text{(marginals)} \\ & = & \sum_{a_1 \in A_1} \sum_{a_2 \in A_2} \mu(a_1, a_2) & \text{(by definition)} \\ & = & \sum_{(a_1,a_2) \in A_1 \times A_2} \mu(a_1, a_2) & \\ & = & |\mu| & \text{(by definition)} \end{array}$$

□

Conversely, $\top$-couplings always exist for pairs of sub-distributions whose mass coincide.

Lemma 13.2 (Existence of $\top$-couplings)**.** Let $\mu_1 \in \mathbb{D}(A_1)$ and $\mu_2 \in \mathbb{D}(A_2)$. If $|\mu_1| = |\mu_2|$ then there exists $\mu \in \mathbb{D}(A_1 \times A_2)$ such that $\mu \blacktriangleleft_\top \langle \mu_1 \,\&\, \mu_2 \rangle$.

Proof. We need to exhibit a sub-distribution μ over $A_1 \times A_2$ with the desirable properties. It suffices to take a rescaling of the product distribution:

$$\mu(a_1, a_2) = \frac{\mu_1(a_1) \cdot \mu_2(a_2)}{|\mu_1|}$$

□

There exist many $\top$-couplings for probability sub-distributions with the same mass; one interesting instance is the *optimal coupling*, when $A_1 = A_2$ (see Exercise **??**).

On the other hand, $\bot$-couplings do not exist, except for the null sub-distributions.

Lemma 13.3 (Non-existence of $\bot$-couplings)**.** Let $\mu_1 \in \mathbb{D}(A_1)$ and $\mu_2 \in \mathbb{D}(A_2)$. If $\mu \blacktriangleleft_\bot \langle \mu_1 \,\&\, \mu_2 \rangle$ then $|\mu_1| = |\mu_2| = 0$.

Proof. The sub-distribution μ over $A_1 \times A_2$ must necessarily be the null sub-distribution. By the marginal conditions both μ_1 and μ_2 must also be the null sub-distributions. □

The next lemma shows that R-couplings are preserved under weakenings.

Lemma 13.4 (Monotonocity of couplings)**.** Let $\mu_1 \in \mathbb{D}(A_1)$ and $\mu_2 \in \mathbb{D}(A_2)$ and $\mu \in \mathbb{D}(A_1 \times A_2)$. Moreover let $R, S \subseteq A_1 \times A_2$ such that If $R \subseteq S$. If $\mu \blacktriangleleft_R \langle \mu_1 \,\&\, \mu_2 \rangle$ then $\mu \blacktriangleleft_S \langle \mu_1 \,\&\, \mu_2 \rangle$.

Proof. The marginal conditions follow immediately from the assumption, the support condition follows by transitivity of $\subseteq$. □

Conversely, one can strengthen the relation of the coupling, under suitable conditions.

Lemma 13.5 (Strengthening couplings)**.** Let $\mu_1 \in \mathbb{D}(A_1)$ and $\mu_2 \in \mathbb{D}(A_2)$ and $\mu \in \mathbb{D}(A_1 \times A_2)$. Moreover let $R, S \subseteq A_1 \times A_2$ such that $R \cap (\mathrm{supp}(\mu_1) \times \mathrm{supp}(\mu_2)) \subseteq S$. If $\mu \blacktriangleleft_R \langle \mu_1 \,\&\, \mu_2 \rangle$ then $\mu \blacktriangleleft_S \langle \mu_1 \,\&\, \mu_2 \rangle$.

Proof. The marginal conditions follow immediately from the assumption. For the support condition, the marginal conditions entail that $\mu(a_1, a_2) = 0$ whenever $a_1 \notin \mathrm{supp}(\mu_1)$ or $a_2 \notin \mathrm{supp}(\mu_2)$, so $\mathrm{supp}(\mu) \subseteq (\mathrm{supp}(\mu_1) \times \mathrm{supp}(\mu_2))$. □

The following lemma shows that couplings are not closed under conjunction of relations.

Proposition 13.1. There exists $R, S \subseteq A_1 \times A_2$ and $\mu_1 \in \mathbb{D}(A_1)$ and $\mu_2 \in \mathbb{D}(A_2)$ such that $(\mu_1, \mu_2) \in R^\sharp$ and $(\mu_1, \mu_2) \in S^\sharp$ but $(\mu_1, \mu_2) \notin (R \cap S)^\sharp$.

Proof. Consider $A_1 = A_2 = \{0, 1\}$ and $\mu_1 = \mu_2 = \mathcal{U}_{\{0,1\}}$ be the uniform distribution over booleans. Moreover, let R be the diagonal relation, i.e. for every $(x_1, x_2) \in \{0, 1\} \times \{0, 1\}$, $((x_1, x_2) \in R$ iff $x_1 = x_2$ and let S be its complement, i.e. for every $(x_1, x_2) \in \{0, 1\} \times \{0, 1\}$, $(x_1, x_2) \in S$ iff $x_1 \neq x_2$. We have $\mu \blacktriangleleft_R \langle \mu_1 \,\&\, \mu_2 \rangle$ and $\mu' \blacktriangleleft_S \langle \mu_1 \,\&\, \mu_2 \rangle$, where μ and μ' are defined by the clauses:

$$\mu(x_1, x_2) = \begin{cases} \frac{1}{2} & \text{if } x_1 = x_2 \\ 0 & \text{otherwise} \end{cases} \qquad \mu'(x_1, x_2) = \begin{cases} \frac{1}{2} & \text{if } x_1 \neq x_2 \\ 0 & \text{otherwise} \end{cases}$$

However, there exists no sub-distribution μ'' such that $\mu'' \blacktriangleleft_{R \cap S} \langle \mu_1 \,\&\, \mu_2 \rangle$, since $R \cap S = \emptyset$, and therefore the above would entail $|\mu''| = 0$, which contradicts $\pi_i(\mu'') = \mu_i$ as $|\mu_i| = 1$. □

13.3 Bijective couplings

A common strategy to establish a R-coupling, with $R \subseteq A_1 \times A_2$, between two sub-distributions μ_1 and μ_2, is to exhibit a mapping $h : A_1 \to A_2$ that satisfies the following three conditions:

bijectivity: for every $a_1, a_1' \in \mathrm{supp}(\mu_1)$, if $h(a_1) = h(a_1')$ then $a_1 = a_1'$, and for every $a_2 \in \mathrm{supp}(\mu_2)$, there exists $a_1 \in \mathrm{supp}(\mu_1)$ such that $h(a_1) = a_2$;

graph inclusion: for every $a_1 \in \mathrm{supp}(\mu_1)$, $(a_1, h(a_1)) \in R$;

equal probabilities: for every $a \in A_1$,

$$\mathbb{P}_{x_1 \sim \mu_1}[x_1 = a] = \mathbb{P}_{x_2 \sim \mu_2}[x_2 = h(a)]$$

(In particular, this entails that $h(a) \in \mathrm{supp}(\mu_2)$ for every $a \in \mathrm{supp}(\mu_1)$.)

We write $h \blacktriangleleft_R \langle \mu_1 \,\&\, \mu_2 \rangle$ whenever the above conditions are satisfied.

Proposition 13.2. If $h \blacktriangleleft_R \langle \mu_1 \,\&\, \mu_2 \rangle$ then $\mu \blacktriangleleft_R \langle \mu_1 \,\&\, \mu_2 \rangle$, where

$$\mathbb{P}_{x \sim \mu}[x = (a_1, a_2)] = \begin{cases} \mathbb{P}_{x_1 \sim \mu_1}[x_1 = a_1] & \text{if } h(a_1) = a_2 \\ 0 & \text{otherwise} \end{cases}$$

Proof. By inspection. □

There are examples of couplings that cannot be established using bijective couplings. Consider for instance the uniform distributions over bitstrings of length ℓ and $\ell + \ell'$ respectively. The two distributions are R-coupled for the relation R defined by the clause $(x, y) \in R$ iff $x = [y]_\ell$, but there is no bijective coupling between the two distributions.

13.4 From couplings to probabilistic inequalities

R-couplings enjoy many important properties. In what follows, we summarize the properties that are of immediate importance for our context.

Proposition 13.3 (Fundamental theorem of R-couplings)**.** Let $E_1 \subseteq A_1$ and $E_2 \subseteq A_2$. Let $R \subseteq A_1 \times A_2$, such that for every $(a_1, a_2) \in A_1 \times A_2$, $(a_1, a_2) \in R$ implies $a_1 \in E_1 \implies a_2 \in E_2$. If $(\mu_1, \mu_2) \in R^\sharp$ then

$$\mathbb{P}_{\mu_1}[E_1] \leq \mathbb{P}_{\mu_2}[E_2]$$

Proof. Let μ such that $\mu \blacktriangleleft_R \langle \mu_1 \,\&\, \mu_2 \rangle$. By the support property of couplings, we know that $\mathrm{supp}(\mu) \subseteq R$, and hence for every $(x_1, x_2) \in A$ such that $\mu(x_1, x_2) \neq 0$, we have $x_1 \notin E_1$ or $x_2 \in E_2$. It follows that $\mathbb{P}_{(x_1,x_2)\sim\mu}[x_1 \in E_1] \leq \mathbb{P}_{(x_1,x_2)\sim\mu}[x_2 \in E_2]$. By the marginal property of couplings, it follows that $\mathbb{P}_{\mu_1}[E_1] \leq \mathbb{P}_{\mu_2}[E_2]$. □

This theorem has many useful corollaries, which we list below. Many of the corollaries are named after their applications in cryptographic proofs.

Corollary 13.6 (Bridging step)**.** Let $E_1 \subseteq A_1$ and $E_2 \subseteq A_2$. Let $R \subseteq A_1 \times A_2$, such that for every $(a_1, a_2) \in A_1 \times A_2$, $(a_1, a_2) \in R$ implies $a_1 \in E_1 \Leftrightarrow a_2 \in E_2$. If $(\mu_1, \mu_2) \in R^\sharp$ then

$$\mathbb{P}_{\mu_1}[E_1] = \mathbb{P}_{\mu_2}[E_2]$$

In many cases, the two events E_1 and E_2 of interest are the same.

Corollary 13.7 (Bridging step from equivalence relation)**.** Let $A_1 = A_2 = A$ and $R \subseteq A \times A$ be an equivalence relation. If $(\mu_1, \mu_2) \in R^\sharp$ then

$$\mathbb{P}_{\mu_1}[E] = \mathbb{P}_{\mu_2}[E]$$

for every $E \subseteq A$ such that $a_1 \in E \wedge (a_1, a_2) \in R$ implies $a_2 \in E$ for every $a_1, a_2 \in A$.

The next corollary involves two events E_1 and E_2 whose probability we want to connect, and a third event F, called a failure event.

Corollary 13.8 (Failure event)**.** Let $E_1 \subseteq A_1$ and $E_2, F \subseteq A_2$. Define $R \subseteq A_1 \times A_2$ such that for every $(a_1, a_2) \in A_1 \times A_2$, $(a_1, a_2) \in R$ implies $a_1 \in E_1 \implies a_2 \in E_2 \cup F$. If $(\mu_1, \mu_2) \in R^\sharp$ then

$$\mathbb{P}_{\mu_1}[E_1] - \mathbb{P}_{\mu_2}[E_2] \leq \mathbb{P}_{\mu_2}[F]$$

Proof. By proposition 13.3 we have $\mathbb{P}_{\mu_1}[E_1] \leq \mathbb{P}_{\mu_2}[E_2 \cup F]$ and we have $\mathbb{P}_{\mu_2}[E_2 \cup F] \leq \mathbb{P}_{\mu_2}[E_2] + \mathbb{P}_{\mu_2}[F]$, the conclusion follows trivially. □

The next corollary provides a symmetric version of the previous corollary.

Corollary 13.9 (Delayed failure event)**.** Let $E_1, F_1 \subseteq A_1$ and $E_2, F_2 \subseteq A_2$. Define $R \subseteq A_1 \times A_2$ such that for every $(a_1, a_2) \in A_1 \times A_2$, $(a_1, a_2) \in R$ implies $a_1 \in E_1 \cap \neg F_1 \Leftrightarrow a_2 \in E_2 \cap \neg F_2$. If $(\mu_1, \mu_2) \in R^\sharp$ then

$$\left|\mathbb{P}_{\mu_1}[E_1] - \mathbb{P}_{\mu_2}[E_2]\right| \leq \max(\mathbb{P}_{\mu_1}[F_1], \mathbb{P}_{\mu_2}[F_2])$$

Proof. By corollary Theorem 13.6, we have $\mathbb{P}_{\mu_1}[E_1 \cap \neg F_1] = \mathbb{P}_{\mu_2}[E_2 \cap \neg F_2]$, by elementary reasoning:

$$\begin{aligned}
\left|\mathbb{P}_{\mu_1}[E_1] - \mathbb{P}_{\mu_2}[E_2]\right| &= \left| \begin{array}{l} \mathbb{P}_{\mu_1}[E_1 \cap F_1] + \mathbb{P}_{\mu_1}[E_1 \cap \neg F_1] \\ -\mathbb{P}_{\mu_2}[E_2 \cap F_2] - \mathbb{P}_{\mu_2}[E_2 \cap \neg F_2] \end{array} \right| \\
&= \left|\mathbb{P}_{\mu_1}[E_1 \cap F_1] - \mathbb{P}_{\mu_2}[E_2 \cap F_2]\right| \\
&\leq \max(\mathbb{P}_{\mu_1}[E_1 \cap F_1], \mathbb{P}_{\mu_2}[E_2 \cap F_2]) \\
&\leq \max(\mathbb{P}_{\mu_1}[F_1], \mathbb{P}_{\mu_2}[F_2])
\end{aligned}$$

□

The following variant is also useful and proved in a similar way.

Corollary 13.10. Let S be a relation such that $(a_1, a_2) \in S$ iff $a_1 \in F_1 \Leftrightarrow a_2 \in F_2$ and $a_1 \notin F_1 \implies a_1 \in E_1 \Leftrightarrow a_2 \in E_2$. Prove that

$$\left|\mathbb{P}_{\mu_1}[E_1] - \mathbb{P}_{\mu_2}[E_2]\right| \leq \mathbb{P}_{\mu_1}[F_1]$$

and

$$\left|\mathbb{P}_{\mu_1}[E_1] - \mathbb{P}_{\mu_2}[E_2]\right| \leq \mathbb{P}_{\mu_1}[F_2]$$

The following proposition states that lifting of equality on the underlying set coincides with equality of distributions, and is useful to fall back on standard probability reasoning when proving existence of couplings.

Proposition 13.4 (Equality coupling)**.** If $A_1 = A_2 = A$, then $(\mu_1, \mu_2) \in =^\sharp$, then $(\mu_1, \mu_2) \in =^\sharp$ iff $\mu_1 = \mu_2$, i.e. for every $E \subseteq A$,

$$\mathbb{P}_{\mu_1}[E] = \mathbb{P}_{\mu_2}[E]$$

Proof. The direct implication follows from the Fundamental Theorem of R-Couplings. The existence of an identity coupling is shown by taking as witness the sub-distribution μ such that

$$\mu(a, b) = \begin{cases} \mu_1(a) & \text{if } a = b \\ 0 & \text{otherwise} \end{cases}$$

□

In addition, the following lemma provides a method to decompose a proof that two distributions are related by an equality coupling into a proof that the two distributions are related by pointwise equality.

Proposition 13.5 (Pointwise equality coupling)**.** For every $\mu_1, \mu_2 \in \mathbb{D}(A)$, $\mu_1 = \mu_2$ iff $(\mu_1, \mu_2) \in R_a^\sharp$ for every $a \in A$, where $(x_1, x_2) \in R_a$ iff $x_1 = a \Leftrightarrow x_2 = a$. If moreover μ_1 is a full-distribution, then it suffices that $(\mu_1, \mu_2) \in S_a^\sharp$ for every $a \in A$, where $(x_1, x_2) \in S_a$ iff $x_1 = a \Rightarrow x_2 = a$.

Proof. The direct implication is trivial, so we consider the reverse implication. By Theorem 13.6, it follows that for every $a \in A$, we have $\mathbb{P}_{x \sim \mu_1}[x = a] = \mathbb{P}_{x \sim \mu_2}[x = a]$, hence the two sub-distributions are equal.

In case μ_1 is a full distribution, $(\mu_1, \mu_2) \in S^\sharp_a$ implies $\mathbb{P}_{x\sim\mu_1}[x = a] \leq \mathbb{P}_{x\sim\mu_2}[x = a]$, from which we can again conclude equality of distributons. □

The above proposition can be generalized to accommodate a failure event.

Our final result is useful to establish that an event has probability exactly $\frac{1}{2}$, assuming that the underlying distribution is full.

Proposition 13.6. Let $\mu \in \mathbb{D}(A)$ be a full-distribution, and $E \subseteq A$. Let $(x_1, x_2, R) \in$ iff $x_1 \in E \Leftrightarrow x_2 \notin E$. If $(\mu, \mu) \in R^\sharp$ then $\mathbb{P}_\mu[E] = \frac{1}{2}$.

Proof. It follows from the Fundamental Theorem of R-Couplings that $\mathbb{P}_\mu[E] = \mathbb{P}_\mu[\neg E]$. Since μ is a full-distribution, it follows that $\mathbb{P}_\mu[E] = \frac{1}{2}$. □

13.5 Closure properties

Couplings are closed under convex combinations.

Lemma 13.11 (Convex combinations of couplings)**.** Let $R \subseteq A_1 \times A_2$, and let I be a finite set. Let $(\mu^i_1)_{i\in I} \in \mathbb{D}(A_1)$, $(\mu^i_2)_I \in \mathbb{D}(A_2)$ and $(\mu^i)_{i\in I} \in \mathbb{D}(A_1 \times A_2)$ such that $\mu^i \blacktriangleleft_R \langle \mu^i_1 \,\&\, \mu^i_2 \rangle$ for every $i \in I$. For every $(p^i)_{i\in I} \in [0, 1]$ such that $\sum_{i\in I} p_i \leq 1$, we have

$$\sum_{i\in I} p^i \mu^i \blacktriangleleft_R \langle \sum_{i\in I} p^i \mu^i_1 \,\&\, \sum_{i\in I} p^i \mu^i_2 \rangle$$

where $\sum_{i\in I} p^i \mu^i$ is the sub-distribution defined by the clause $(\sum_{i\in I} p^i \mu^i)(a) = \sum_{i\in I} p^i \mu^i(a)$.

Couplings are closed under relation composition.

Lemma 13.12. Let $R \subseteq A_1 \times A_2$ and $S \subseteq A_2 \times A_3$. Let $\mu_1 \in \mathbb{D}(A_1)$, $\mu_2 \in \mathbb{D}(A_2)$, and $\mu_3 \in \mathbb{D}(A_3)$. Assume that $\mu \blacktriangleleft_R \langle \mu_1 \,\&\, \mu_2 \rangle$ and $\mu' \blacktriangleleft_S \langle \mu_2 \,\&\, \mu_3 \rangle$. Then there exists $\mu'' \in \mathbb{D}(A_1 \times A_3)$ such that $\mu'' \blacktriangleleft_{R\circ S} \langle \mu_1 \,\&\, \mu_3 \rangle$.

Proof. Take

$$\mu''(a_1, a_3) = \sum_{a_2} \frac{\mu(a_1, a_2) \cdot \mu'(a_2, a_3)}{\mu_2(a_2)}$$

where by convention $\frac{0}{0} = 0$. □

Moreover, couplings are closed under monadic unit and monadic composition. The following establishes that Dirac distributions of elements related by a relation R are R-coupled.

Lemma 13.13. Let $R \subseteq A_1 \times A_2$ and let $(a_1, a_2) \in R$. Then $\mathbb{1}_{(a_1,a_2)} \blacktriangleleft_R \langle \mathbb{1}_{a_1} \,\&\, \mathbb{1}_{a_2} \rangle$.

The next theorem establishes that couplings are closed under sequential composition.

Theorem 13.14 (Sequential composition of couplings)**.** Let $R \subseteq A_1 \times A_2$ and $S \subseteq B_1 \times B_2$. Let $\mu_1 \in \mathbb{D}(A_1)$, $\mu_2 \in \mathbb{D}(A_2)$, $M_1 : A_1 \to \mathbb{D}(B_1)$ and $M_2 : A_2 \to \mathbb{D}(B_2)$. Assume that:

- $\mu \blacktriangleleft_R \langle \mu_1 \,\&\, \mu_2 \rangle$;
- $M(a_1, a_2) \blacktriangleleft_S \langle M_1(a_1) \,\&\, M_2(a_2) \rangle$ for every $(a_1, a_2) \in A_1 \times A_2$ such that $(a_1, a_2) \in R$.

Then $\mathsf{let}\ a = \mu\ \mathsf{in}\ M\ a \blacktriangleleft_S \langle \mathsf{let}\ a_1 = \mu_1\ \mathsf{in}\ M_1\ a_1 \,\&\, \mathsf{let}\ a_2 = \mu_2\ \mathsf{in}\ M_2\ a_2 \rangle$.

13.6 Strassen's Theorem and limits of couplings

(Strassen, 1965) gives an elegant characterization of the existence of R-couplings. The following specializes his result to our simpler setting.

Theorem 13.15 (Strassen's theorem)**.** Let $R \subseteq A_1 \times A_2$, $\mu_1 \in \mathbb{D}(A_1)$ and $\mu_2 \in \mathbb{D}(A_2)$ such that $|\mu_1| = |\mu_2| = 1$. Then $(\mu_1, \mu_2) \in R^\sharp$ iff $\mathbb{P}_{\mu_1}[X] \leq \mathbb{P}_{\mu_2}[R(X)]$ for every $X \subseteq A_1$, where $R(X) = \{a_2 \in A_2 \mid \exists a_1 \in A_1.\, a_1 \in X \wedge (a_1, a_2) \in R\}$ is the image of X under R.

By rescaling, the characterization readily extends to sub-distributions μ_1 and μ_2 such that $|\mu_1| = |\mu_2|$.

Strassen's Theorem is useful to provide alternative proofs to several statements of this chapter. Another useful consequence of Strassen's Theorem is that existence of R-liftings carries to limits distributions.

Proposition 13.7 (Limits of R-liftings)**.** Let $(\mu_n)_{n\in\mathbb{N}} \in \mathbb{D}(A_1)$ and $(\nu_n)_{n\in\mathbb{N}} \in \mathbb{D}(A_2)$ be increasing sequences of sub-distributions over A_1 and A_2 respectively. Let $R \subseteq A_1 \times A_2$. If for any $n \in \mathbb{N}$, $(\mu_n, \nu_n) \in R^\sharp$ then $(\mu_\infty, \nu_\infty) \in R^\sharp$, where $\mu_\infty = \lim_{i\in\mathbb{N}} \mu_i$ and $\nu_\infty = \lim_{i\in\mathbb{N}} \nu_i$.

Proof. By Strassen's Theorem, it suffices to show that for every $X \subseteq A_1$, we have $\mu_\infty(X) \leq \nu_\infty(R(X))$ or equivalently, $\lim_{i\in\mathbb{N}} \mu_i(X) \leq \lim_{i\in\mathbb{N}} \nu_i(R(X))$. The latter follows immediately from the fact that for every i, $\mu_i(X) \leq \nu_i(R(X))$, which follows from our hypothesis by Strassen's Theorem. □

Note that a direct proof based on witnesses is more difficult, because the limit of couplings of two increasing sequences of sub-distributions may not exist. Indeed, consider the increasing sequences of sub-distributions over booleans:

$$\mu_i(\mathsf{tt}) = \mu_i(\mathsf{ff}) = \nu_i(\mathsf{tt}) = \nu_i(\mathsf{ff}) = 1/2 - 1/2^i$$

Then we can define ρ_i as follows:

- if i is odd, then $\rho_i(\mathsf{tt}, \mathsf{tt}) = \rho_i(\mathsf{ff}, false) = 1/2 - 1/2^i$ and $\rho_i(\mathsf{ff}, \mathsf{tt}) = \rho_i(\mathsf{tt}, false) = 0$
- if i is even, then $\rho_i(\mathsf{tt}, \mathsf{tt}) = \rho_i(\mathsf{ff}, false) = 0$ and $\rho_i(\mathsf{ff}, \mathsf{tt}) = \rho_i(\mathsf{tt}, false) = 1/2 - 1/2^i$

Since ρ_i and ρ_{i+1} have disjoint support for every i, their limit does not exist.

13.7 An inductive characterization of R-liftings

R-liftings admit the following inductive relation:

$$\frac{(a_1, a_2) \in R}{\mathbb{1}_{(a_1,a_2)} \blacktriangleleft_R \langle \mathbb{1}_{a_1} \,\&\, \mathbb{1}_{a_2} \rangle}\ [\textsc{Dirac}]$$

$$\frac{\forall i \in I.\, \mu^i \blacktriangleleft_R \langle \mu_1^i \,\&\, \mu_2^i \rangle}{\sum_{i\in I} p^i \mu^i \blacktriangleleft_R \langle \sum_{i\in I} p^i \mu_1^i \,\&\, \sum_{i\in I} p^i \mu_2^i \rangle}\ [\textsc{Convex}]$$

Proposition 13.8 (Inductive characterization of R-couplings)**.** $\mu \blacktriangleleft_R \langle \mu_1 \;\&\; \mu_2 \rangle$ iff the validity of the coupling can be established using the inductive relation above.

Note that the soundness of the proof system uses the fact that couplings are closed under convex combinations.

13.8 Further reading

[GB5]: TODO: Lindvall, Thorisson, Peres, etc

14

Probabilistic Relational Hoare Logic

15

Probabilistic non-interference

16

Probabilistic product programs

Part III

Adversarial computations

17

Adversaries

18

The PRF/PRP Switching Lemma

19

Encryption

20

Signatures

Part IV

Epilogue

21

Conclusion

References

Almeida, J. B., M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi (2016). "Verifying Constant-Time Implementations". In: *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. Ed. by T. Holz and S. Savage. USENIX Association. 53–70. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida.

Barthe, G., J. M. Crespo, and C. Kunz (2011). "Relational Verification Using Product Programs". In: *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*. Ed. by M. J. Butler and W. Schulte. Vol. 6664. *Lecture Notes in Computer Science*. Springer. 200–214. DOI: 10.1007/978-3-642-21437-0_17. URL: https://doi.org/10.1007/978-3-642-21437-0%5C_17.

Barthe, G., J. M. Crespo, and C. Kunz (2016a). "Product programs and relational program logics". *J. Log. Algebr. Meth. Program.* 85(5): 847–859. DOI: 10.1016/j.jlamp.2016.05.004. URL: https://doi.org/10.1016/j.jlamp.2016.05.004.

Barthe, G., T. Espitau, M. Gaboardi, B. Grégoire, J. Hsu, and P. Strub (2018). "An Assertion-Based Program Logic for Probabilistic Programs". In: *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. Ed. by A. Ahmed. Vol. 10801. *Lecture Notes in Computer Science*. Springer. 117–144. DOI: 10.1007/978-3-319-89884-1_5. URL: https://doi.org/10.1007/978-3-319-89884-1%5C_5.

Barthe, G., M. Gaboardi, B. Grégoire, J. Hsu, and P. Strub (2016b). "A Program Logic for Union Bounds". In: *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*. Ed. by I. Chatzigiannakis, M. Mitzenmacher, Y. Rabani, and D. Sangiorgi. Vol. 55. *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. 107:1–107:15. DOI: 10.4230/LIPIcs.ICALP.2016.107. URL: http://dx.doi.org/10.4230/LIPIcs.ICALP.2016.107.

Barthe, G., B. Grégoire, J. Hsu, and P. Strub (2017). "Coupling proofs are probabilistic product programs". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Ed. by G. Castagna and A. D. Gordon. ACM. 161–174. URL: http://dl.acm.org/citation.cfm?id=3009896.

Benton, N. (2004). "Simple relational correctness proofs for static analyses and program transformations". In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*. Ed. by N. D. Jones and X. Leroy. ACM. 14–25. DOI: 10.1145/964001.964003. URL: http://doi.acm.org/10.1145/964001.964003.

Chakarov, A. and S. Sankaranarayanan (2013). "Probabilistic Program Analysis with Martingales". In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. Ed. by N. Sharygina and H. Veith. Vol. 8044. *Lecture Notes in Computer Science*. Springer. 511–526. DOI: 10.1007/978-3-642-39799-8_34. URL: https://doi.org/10.1007/978-3-642-39799-8%5C_34.

Chatterjee, K., P. Novotný, and D. Zikelic (2017). "Stochastic invariants for probabilistic termination". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Ed. by G. Castagna and A. D. Gordon. ACM. 145–160. URL: http://dl.acm.org/citation.cfm?id=3009873.

Dubhashi, D. P. and A. Panconesi (2009). "Concentration of measure for the analysis of randomized algorithms".

Fioriti, L. M. F. and H. Hermanns (2015). "Probabilistic Termination: Soundness, Completeness, and Compositionality". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. Ed. by S. K. Rajamani and D. Walker. ACM. 489–501. DOI: 10.1145/2676726.2677001. URL: https://doi.org/10.1145/2676726.2677001.

Jones, C. and G. D. Plotkin (1989). "A Probabilistic Powerdomain of Evaluations". In: *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*. IEEE Computer Society. 186–195. DOI: 10.1109/LICS.1989.39173. URL: http://dx.doi.org/10.1109/LICS.1989.39173.

Kaminski, B. L. (2019). "Advanced weakest precondition calculi for probabilistic programs". *PhD thesis*. RWTH Aachen University, Germany. URL: http://publications.rwth-aachen.de/record/755408.

Kozen, D. (1981). "Semantics of Probabilistic Programs". *J. Comput. Syst. Sci.* 22(3): 328–350. DOI: 10.1016/0022-0000(81)90036-2. URL: http://dx.doi.org/10.1016/0022-0000(81)90036-2.

Kozen, D. (1985). "A Probabilistic PDL". *J. Comput. Syst. Sci.* 30(2): 162–178. DOI: 10.1016/0022-0000(85)90012-1. URL: http://dx.doi.org/10.1016/0022-0000(85)90012-1.

Lawvere, F. W. (1962). "The category of probabilistic mappings". *preprint*.

McIver, A. and C. Morgan (2005). *Abstraction, Refinement and Proof for Probabilistic Systems. Monographs in Computer Science*. Springer. ISBN: 978-0-387-40115-7. DOI: 10.1007/b138392. URL: http://dx.doi.org/10.1007/b138392.

McIver, A., C. Morgan, B. L. Kaminski, and J. Katoen (2018). "A new proof rule for almost-sure termination". *PACMPL*. 2(POPL): 33:1–33:28. DOI: 10.1145/3158121. URL: https://doi.org/10.1145/3158121.

Molnar, D., M. Piotrowski, D. Schultz, and D. A. Wagner (2005). "The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks". In: *Information Security and Cryptology - ICISC 2005, 8th International Conference, Seoul, Korea, December 1-2, 2005, Revised Selected Papers*. Ed. by D. Won and S. Kim. Vol. 3935. *Lecture Notes in Computer Science*. Springer. 156–168. DOI: 10.1007/11734727_14. URL: https://doi.org/10.1007/11734727%5C_14.

Strassen, V. (1965). "The existence of probability measures with given marginals". *The Annals of Mathematical Statistics*: 423–439. URL: http://projecteuclid.org/euclid.aoms/1177700153.

Volpano, D. M. and G. Smith (1997). "A Type-Based Approach to Program Security". In: *TAPSOFT'97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France, April 14-18, 1997, Proceedings*. Ed. by M. Bidoit and M. Dauchet. Vol. 1214. *Lecture Notes in Computer Science*. Springer. 607–621. DOI: 10.1007/BFb0030629. URL: https://doi.org/10.1007/BFb0030629.

Winskel, G. (1993). *The formal semantics of programming languages: an introduction*.

Zaks, A. and A. Pnueli (2008). "CoVaC: Compiler Validation by Program Analysis of the Cross-Product". In: *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings*. Ed. by J. Cuéllar, T. S. E. Maibaum, and K. Sere. Vol. 5014. *Lecture Notes in Computer Science*. Springer. 35–51. DOI: 10.1007/978-3-540-68237-0_5. URL: https://doi.org/10.1007/978-3-540-68237-0%5C_5.

Index