Coupling proofs are probabilistic product programs
Gilles Barthe, Benjamin Grégoire, Justin Hsu, Pierre-Yves Strub

To cite this version:
Gilles Barthe, Benjamin Grégoire, Justin Hsu, Pierre-Yves Strub. Coupling proofs are probabilistic product programs. POPL 2017 - Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, Jan 2017, Paris, France. pp.161-174, 10.1145/3009837.3009896 . hal-01649028

HAL Id: hal-01649028
https://hal.archives-ouvertes.fr/hal-01649028
Submitted on 27 Nov 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
We address both problems as follows.

Couplings are powerful mathematical tools for reasoning about probabilistic processes. Recent developments in formal verification identify a close connection between couplings and pRHL, a relational logic for probabilistic programs. However, existing work using pRHL is inherently incomplete, and is not able to capture advanced forms of couplings such as shift couplings. We address both problems as follows.

First, we define an extension of pRHL, called \(\times\)pRHL, which explicitly constructs the coupling in a pRHL derivation of the form of a probabilistic product program that simulates two correlated runs of the original program. Existing verification tools for probabilistic programs can then be directly applied to the probabilistic product to prove quantitative properties of the coupling. Second, we equip \(\times\)pRHL with a new rule for while loops, where reasoning can freely mix synchronized and unsynchronized loop iterations. Our proof rule can capture examples of shift couplings, and the logic is relatively complete for deterministic programs.

We show soundness of \(\times\)pRHL and use it to analyze two classes of examples. First, we verify rapid mixing using different tools from coupling: standard coupling, shift coupling, and path coupling, a compositional principle for combining local couplings into a global coupling. Second, we verify (approximate) equivalence between a source and an optimized program for several instances of loop optimizations from the literature.

1. Introduction
The coupling method [41, 42] is an elegant mathematical tool for analyzing the relationship between probabilistic processes. Informally, couplings correlate outputs of two probabilistic processes by specifying how corresponding sampling statements are correlated; reasoning about the correlated processes can then imply interesting properties of the original processes.

A classical application of couplings is showing that two probabilistic processes converge in distribution. Consider, for instance, a symmetric simple random walk over \(\mathbb{Z}\); starting from some initial position \(p\), the process repeatedly samples a value \(s\) uniformly in \([-1, 1]\) and updates its position to \(p + s\). As the process evolves, the distribution on position spreads out from its initial position, and converges to a limit distribution that is the same for all initial positions. This property can be proved by constructing a coupling where the probability that the two coupled walks end in the same position approaches \(1\) as we run process.

Beyond merely showing convergence, typically we are interested in how quickly the processes converge. The rate of convergence, sometimes measured by the mixing time of a process, is highly important [41]. A common use of probabilistic processes is to efficiently sample from a distribution that approximates a complicated distribution (e.g., Metropolis et al. [43]). For instance, the Glauber dynamics [33] approximately samples from the uniform distribution on proper colorings of a graph—a hard distribution to compute—by maintaining a coloring and randomly re-coloring a single vertex at a time. More generally, the Glauber dynamics is an example of Markov Chain Monte-Carlo (MCMC), a family of techniques that underlie many computational simulations in science, and machine learning algorithms for performing probabilistic inference (see Brooks et al. [22] for a survey). The mixing time determines how many steps we need to run the process before we are approximately sampling from the target distribution.

Recently, Barthe et al. [41] noticed a close connection between couplings and probabilistic relational Hoare logic (pRHL), originally designed for reasoning about the computational security of cryptographic constructions [5]. Namely, every valid derivation in pRHL implies the existence of a coupling of the output distributions of the two programs. Using this observation, Barthe et al. [12] verify pRHL judgments that imply properties of random walks, the balls-in-bins process, and the birth-death process. Barthe et al. [13] extend the connection to approximate couplings in approximate probabilistic relational Hoare logic (apRHL), introduced for reasoning about differentially private computations [7], and exploit this connection to prove differential privacy of several examples that had previously been beyond the reach of formal verification.

While pRHL is a useful tool for constructing couplings, it has two limitations. First, it cannot directly reason about the two coupled processes. This poses a problem for proving mixing and convergence properties. For instance, Barthe et al. [12] prove that if a certain property \(P\) holds on the coupled samples, then two random walks meet under the coupling. By a theorem about random walks, this means that the (total variation) distance between the two distributions is at most the probability of \(P\) under the coupling. However, we do not know what this probability is or how it grows as we run the random walk for more iterations, since pRHL cannot reason about the coupled process. We run into similar difficulties if we try to prove convergence using path coupling [23], a general construction that shows fast convergence by upper-bounding the expected distance when we make a transition from two coupled states. While we can express the transition function in code, pRHL cannot reason about expectations.
Second, pRHL cannot express some natural classes of couplings because the rule for while loops requires that both loops execute in lockstep. For instance, shift couplings allow the two processes to meet at a random time shift, e.g., the first process at time $t_1$ could track the second process at some other time $t_2 = t_1 + \delta$. This incompleteness of pRHL is also a limitation for more standard applications of pRHL to program analysis, like validation of loop transformations, and more generally for any optimization that alters the control flow of the programs.

We address these problems as follows. First, we deepen the connection between pRHL and couplings with an observation that is reminiscent of proofs-as-programs: not only do pRHL judgements correspond to couplings, but pRHL proofs encode a probabilistic product program that constructs the distribution witnessing the coupling. This program is similar to existing product program constructions in that it simulates two program executions with a single program. However, the probabilistic product also coordinates the samplings in the two executions, as specified by the coupling encoded in the original pRHL derivation. Second, we propose a general rule for while loops; our rule subsumes several existing rules and is sufficiently expressive to capture several examples of shift coupling. The resulting system, which we call product pRHL or $\times$ pRHL for short, has several benefits.

First, we obtain a simple, algorithmic procedure to construct the probabilistic product given a pRHL derivation. The product directly simulates the coupled processes, so we can prove quantitative properties about probabilities or expected values of this coupling using existing probabilistic verification techniques. Moreover, intermediate assertions in a pRHL derivation to be soundly transported to the probabilistic product in $\times$ pRHL derivations. For instance, we may prove synchronized control flow in pRHL, and directly assume this fact in the probabilistic product. Many facts are often practically easier to manipulate in pRHL, since pRHL works purely on non-probabilistic assertions.

Second, we obtain a powerful logic that can reason about many examples from the coupling literature, especially shift couplings, and from the translation validation literature, especially loop optimizations. On the foundational side, we prove that our logic is relatively complete for deterministic programs.

**Summary of contributions** We make the following contributions.

- We present a proof-relevant program logic $\times$ pRHL that extracts a probabilistic product program from a valid derivation, and prove (using the Coq proof assistant) that the logic is sound.
- We propose new rules for loops and random sampling in $\times$ pRHL.
- We also prove relative completeness for deterministic programs.
- We demonstrate several applications of $\times$ pRHL to showing convergence of probabilistic processes; several of these examples use path coupling, a compositional tool for constructing couplings which bounds convergence in terms of expected properties of the coupling, and shift coupling, a generalization of coupling where the two processes are allowed to meet at different times. We also show how to validate some common loop transformations.

## 2. Preliminaries

We begin by giving a bird’s eye view on probabilistic couplings, which take output distributions from two probabilistic processes and place them in the same probabilistic space.

In the following, we will work with sub-distributions over discrete (finite or countable) sets.

**Definition 1.** A discrete sub-distribution over a set $A$ is completely defined by its mass function $\mu : A \to [0, 1]$ such that $\sum_{a \in A} \mu(a)$ is defined and bound by 1. The quantity $\sum_{a \in A} \mu(a) \in [0, 1]$ is called the weight of $\mu$ and denoted by $|\mu|$. The support $\text{supp}(\mu)$ of $\mu$ is defined as $\{x \in A | \mu(x) \neq 0\}$ and is discrete (i.e., countable) by construction. A distribution is a sub-distribution with weight 1. The probability of an event $P$ w.r.t. $\mu$, written $\Pr_{\mu}[P]$ (or $\Pr_{\mu}[P]$ when $\mu$ is clear from the context), is defined as $\sum_{\{x \in A | P(x)\}} \mu(x)$.

One can equip distributions with a monadic structure using the Dirac distributions $\delta_x$ and distribution expectation $\mathbb{E}_{x \sim \mu}[M(x)]$ for the bind, where

$$\mathbb{E}_{x \sim \mu}[M(x)] : x \mapsto \sum_a \mu(a) \cdot M(a)(x).$$

When working with sub-distributions over tuples, the probabilistic versions of the usual projections on tuples are called marginals. For distributions over pairs, we define the first and second marginals $\pi_1(\mu)$ and $\pi_2(\mu)$ of a distribution $\mu$ over $A \times B$ by $\pi_1(\mu)(a) = \sum_{b \in B} \mu(a, b)$ and $\pi_2(\mu)(b) = \sum_a \mu(a, b)$. For a distribution $\mu$ over memories and a set of variables $X \subseteq \text{Vars}$, we define the X-marginal distribution of $\mu$ as

$$\pi_X(\mu)(\bar{x}) = \sum_{\bar{v} \in \text{Vars} \setminus X} \mu(\bar{x}, \bar{v}).$$

We can also construct a sub-distribution over tuples from two distributions: the product sub-distribution of sub-distributions $\mu_1$ and $\mu_2$ over $A$ and $B$ respectively is defined by

$$(\mu_1 \times \mu_2)(a, b) = \mu_1(a) \cdot \mu_2(b).$$

We are now ready to formally define coupling.

**Definition 2.** Two sub-distributions $\mu_1, \mu_2$ resp. over $A$ and $B$ are said to be coupled with witness $\mu \in \text{Distr}(A \times B)$, written $\mu \bowtie (\mu_1 \times \mu_2)$, if $\pi_1(\mu) = \mu_1$ and $\pi_2(\mu) = \mu_2$.

For $\Psi \subseteq A \times B$, we write $\mu \bowtie_{\Psi} (\mu_1 \times \mu_2)$ if $\mu \bowtie (\mu_1 \times \mu_2)$ and moreover $\text{supp}(\mu) \subseteq \Psi$. We will often abuse notation and call $\mu$ a coupling of $\mu_1$ and $\mu_2$.

As an example, suppose that $A = B$ and $\mu_1 = \mu_2$ are the uniform distributions. Then, any bijection $f : A \to A$ gives a coupling of $\mu_1$ and $\mu_2$: we call the resulting coupling $D_r(A)$ so that $D_r(A) \bowtie (\mu_1 \times \mu_2)$. The coupling $D_r(A)$ assigns positive probability only to pairs $(v, f(v))$ with $v \in A$.

For another example, suppose again that $A = B$ and $\mu_1 = \mu_2$, but the distributions are not necessarily uniform. Then, the identity map $i : A \to A$ always gives a coupling of $\mu_1$ and $\mu_2$, correlating samples from both distributions to be the same. We will write $D^r(\mu_1)$ for the resulting coupling, so that $D^r(\mu_1) \bowtie (\mu_1 \times \mu_2)$. If $\mu_1$ is the uniform distribution we will sometime write $D^r(A)$ instead of $D^r(\mu_1)$. Note that $D^r(\mu_1)$ assigns positive probability only to pairs $(v, v)$ with $\mu_1(v), \mu_2(v) \neq 0$.

To reason about convergence of probabilistic processes, we will use the total variation distance on distributions (also known as statistical distance).

**Definition 3.** Let $\mu_1$ and $\mu_2$ be sub-distributions over a countable set $A$. The total variation (TV) distance between $\mu_1$ and $\mu_2$ is defined by:

$$\text{TV}(\mu_1, \mu_2) = \frac{1}{2} \sum_{a \in A} |\mu_1(a) - \mu_2(a)|.$$
Theorem 4 (Fundamental theorem of couplings (e.g., [42])). Let \( \mu_1 \) and \( \mu_2 \) be distributions over a countable set. Then for any coupling \( \mu \) of \( \mu_1, \mu_2 \), we have
\[
TV(\mu_1, \mu_2) \leq \Pr(x \neq x').
\]

3. Product programs

3.1 Language

We will work with a core, probabilistic imperative language with a command for random sampling from primitive distributions. The set of commands is defined as follows:

\[
C ::= \text{skip} \quad \text{noop} \\
| \text{abort} \quad \text{ab} \quad \text{do} \text{skip} \\
| X \leftarrow E \text{ deterministic assignment} \\
| X \uplus D(E) \text{ probabilistic assignment} \\
| C; C \quad \text{sequencing} \\
| \text{if } E \text{ then } C \text{ else } C \quad \text{conditional} \\
| \text{while } E \text{ do } C \quad \text{while loop}
\]

Here, \( X \) is a set of variables, \( E \) is a set of (deterministic) expressions, and \( D \) is a set of distribution expressions. Variables and expressions are typed, ranging over booleans, integers, lists, etc. The expression grammar is entirely standard, and we omit it.

We will use several shorthands for commands: if \( e \) then \( c \) for if \( e \) then \( c \) else \( skip \) for the looping command while \( \top \) do \( skip \), and \( e^{c,k} \) for the k-fold composition of \( c \) restricted to \( e \). Formally,
\[
e^{c,k} \triangleq \begin{cases} i = 0; & \text{if } c \text{ then } e; i++ \\
 & \text{where } i \text{ is a fresh variable. We allow } k \text{ to be an arbitrary expression; when } k \text{ is a constant}, \\
\end{cases}
\]

for the simple notion of program equivalence introduced in Figure 2.

The denotational semantics of programs is adapted from the seminal work of Kozen [36]. We first interpret every ground type \( T \) as a set \([T]\); other constructors \( C \) are interpreted as functions \([C]\) that respect their arities. The semantics of expressions and distribution expressions is then parameterized by a state \( m \), and is defined in the usual way. Last, commands are interpreted as a function from memories to sub-distributions over memories, where memories are finite maps from variables to values. Formally, we let \( \text{State}_X \) denote the set of memories over the finite set of variables \( X \). Moreover, we use \( \uplus \) to denote the standard disjoint union on finite maps, so \( \uplus : \text{State}_{X_1} \times \text{State}_{X_2} \rightarrow \text{State}_{X_1 \uplus X_2} \) for disjoint finite sets \( X_1 \) and \( X_2 \). The interpretation of \( c \), written \([c]_m\), is a function from \( \text{State}_X \) to \( \text{Distr}(\text{State}_X) \), where \( \text{Vars}(c) = X \). The definition of \([c]_m\) is given in Figure 1.

Last, for any predicate over memories \( \Phi \), we say that a command is \( \Phi \)-lossless iff for any memory \( m \) s.t. \( \Phi(m) \), the weight of \([c]_m\) is equal to 1. A command is lossless if it is \( \top \)-lossless, i.e. if \([c]_m\) = 1 for every memory \( m \).

3.2 Proof system

Our proof system manipulates judgments of the form:
\[
\{ \Phi \} \quad c_1 \quad c_2 \quad \{ \Psi \} \quad \leadsto c
\]

where \( c_1, c_2 \) and \( c \) are statements over disjoint variables, and \( \Phi \) and \( \Psi \) are assertions over the variables of \( c_1 \) and \( c_2 \). We will not fix a specific syntax for assertions, but one natural choice is first-order formulas over the program variables of both programs.

Informally, the judgment is valid if \( c \) is a probabilistic product program for \( c_1 \) and \( c_2 \) under the pre-condition \( \Phi \), i.e. for every initial memory \( m = m_1 \uplus m_2 \) satisfying the pre-condition \( \Phi \), the sub-distribution \([c]_m \) is a coupling for \([c_1]_{m_1} \) and \([c_2]_{m_2} \), and moreover \( \text{supp}([c]_m) \) only contains states that satisfy \( \Psi \).

Definition 5 (Valid judgment).

- Two commands \( c_1 \) and \( c_2 \) are separable iff their variables are disjoint, i.e. \( \text{Vars}(c_1) \cap \text{Vars}(c_2) = \emptyset \).

A judgment \( \{ \Phi \} \quad c_1 \quad c_2 \quad \{ \Psi \} \:\leadsto c \) is valid iff the commands \( c_1 \) and \( c_2 \) are separable and for every memory \( m = m_1 \uplus m_2 \) such that \( m \vdash \Phi \), we have
\[
[c]_m \leftrightarrow \Phi \left( ([c_1]_{m_1}) \& ([c_2]_{m_2}) \right).
\]

For comparison, judgments in pRHL are of the form
\[
\{ \Phi \} \quad c_1 \quad c_2 \quad \{ \Psi \}
\]

and assert that for every initial memory \( m = m_1 \uplus m_2 \) such that \( m \vdash \Phi \), there exists a coupling \( \mu \) such that
\[
\mu \vdash \Psi \left( ([c_1]_{m_1}) \& ([c_2]_{m_2}) \right).
\]

In contrast, our notion of judgment is proof-relevant, since the derivation of the judgment guides the product construction. We briefly comment on several rules in our system, presented in Fig. 3. The [CONSEQ] rule is similar to the rule of consequence, and reflects that validity is preserved by weakening the post-condition and strengthening the pre-condition; none of the programs is modified in this case.

The [FALSE] rule corresponds to the fact that under a false precondition nothing needs to be proved.

The [CASE] rule allows proving a judgment by case analysis; specifically, the validity of a judgment with pre-condition \( \Phi \) can be established from the validity of two judgments, one where the pre-condition is strengthened with \( e \) and the other where the pre-condition is strengthened with \( \neg e \). The product program is of the form \( \text{if } e \text{ then } c \text{ else } c' \), where \( e \) and \( c' \) correspond to the product programs built from the proofs of the first and second premises respectively.

The [STRUCT] rule allows replacing programs by provably equivalent programs. The rules for proving program equivalence are given in Figure 4, and manipulate judgments of the form \( \Phi \vdash c \equiv c' \), where \( \Phi \) is a relational assertion. We only introduce equations that are needed for recovering derived rules, striving to keep the notion of structural equivalence as simple as possible.

The [ASSG] rule corresponds to the pRHL rule for assignments; in this case, the product program is simply the sequential composition of the two assignments.

The [RAND] rule informally takes a coupling between the two distributions used for sampling in the left and right program, and produces a product program that draws the pair of correlated samples from the coupling. Since our language supports sampling from distribution expressions and not only distributions, the rule asks for the existence of a coupling for each interpretation of the distribution expressions under a valuation satisfying the pre-condition \( \Phi \) of the judgment. Furthermore, note that the rule requires that every element in the support of the coupling validates the post-condition. This is similar to pRHL, and is a natural requirement given the notion of valid judgment. However, the rule is strictly more general than the corresponding pRHL rule, which is restricted to the case where \( \Psi \) is the graph of some bijection.

The [WHILE] rule for while loops constructs a product program that interleaves synchronous and asynchronous executions of the loop bodies. The first premise establishes that \( k_1 \) and \( k_2 \) are strictly
positive when the loop invariant $\Psi$ holds. Then, we specify 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 notation $\oplus\{p_0, p_1, p_2\}$ indicates that exactly one of the tests $p_0$, $p_1$, and $p_2$ must hold. These predicates guide the construction of the product program. If $p_0$ holds, then both guards should be equal and we can execute the two sides $k_1$ and $k_2$ iterations respectively, preserving the loop invariant $\Psi$. If $p_1$ holds and the right loop has not terminated yet, then the left loop also has not terminated yet and we may execute the left loop one iteration. If $p_2$ holds and the left loop has not terminated yet, then the right loop also has not terminated yet and we may execute the right loop one iteration.

The last pair of premises deal with termination. Note that some condition on termination is needed for soundness of the logic; if the left loop terminates with probability 1 while the right loop terminates with probability 0 (i.e., never), it is impossible to construct a valid product program since there is no distribution on pairs that has first marginal with weight 1 and second marginal with weight 0. So, we require that the first and second loops are lossless assuming $p_1$ and $p_2$ respectively. This ensures that with probability 1, there are only finitely many steps where we execute the left or right loop separately. Note that the product program may still fail to terminate with positive probability since there may be infinite sequences of iterations where $p_0$ holds so both loops advance, but both programs will yield sub-distributions with the same weight so we can still find a coupling.

With these premises, the construction of the product program is straightforward. At the top level, the program continues as long as $e$ is true, i.e., while at least one of the two loops can make progress. Each iteration, it performs a case analysis on the three predicates. If $p_0$ holds, then we execute the product from running the two loops $k_1$ and $k_2$ iterations respectively. If $p_1$ holds, then we execute the product from running the left loop one iteration; if $p_2$ then we execute the product from running the right loop one iteration.

### Derived rules

Presentations of relational Hoare logic often include one-sided rules, which are based on the analysis of a single program (rather than two). By reasoning about program equivalence, we can derive all one-sided rules and the two-sided rules of conditionals and loops from pRHL, presented in Fig. 2 within our system.

Proposition 6. All the rules in Figure 4 (and their symmetric version) are derived rules.

We briefly comment on some of the derived rules. The [ASSG-L] rule is the one-sided rule for assignment. It can be derived from its two-sided counterpart, the [ASSG] rule, using the [STRUCT] rule.

\[ [\text{skip}]_m = 1_m \]
\[ [x \leftarrow c]_m = 1_{m[x := c]} \]
\[ [c_1 
arrow c_2]_m = \{1 - 1\} [c_2]_m \]
\[ [\text{abort}]_m = 0 \]
\[ [x \not \equiv g]_m = \mathbb{E}_{x \sim g} [1_{m[x := e]}] \]
\[ [\text{if } e \text{ then } s_1 \text{ else } s_2]_m = [e]_m [c_1]_m \text{ else } [c_2]_m \]

Figure 1. Interpretation of commands

\[ \Phi \vdash c \equiv c \]
\[ \Phi \vdash c_1 \equiv c_2 \]
\[ \Phi \vdash c \equiv \{1_x \equiv \text{skip} \} \]
\[ \Phi \vdash (x_1, x_2) \not \equiv \mu_1 \times \mu_2 \equiv x_1 \not \equiv \mu_1; x_2 \not \equiv \mu_2 \]
\[ \Phi \quad \Rightarrow \quad x = e \]
\[ \Phi \vdash e \equiv \text{skip} \]
\[ \Phi \vdash \top \equiv c \equiv [\text{if } e \text{ then } c \text{ else } c'] \equiv c' \]
\[ \Phi \vdash c_1 \equiv \{1 \} \equiv c_2 \]
\[ \Phi \vdash c_1; c_2 \equiv c_1; c_2 \equiv c_1; c_2 \]
\[ \Phi \vdash e \equiv \{1 \} \equiv e \]
\[ \Phi \vdash \text{while } e \text{ do } c \equiv \text{while } e \text{ do } c \]

Figure 2. Equivalence rules

The [RAND-L] rule is the one-sided rule for random sampling. It can be derived from its two-sided counterpart, the [RAND] rule, using the [STRUCT] rule and the fact that the product distribution $\mu_1 \times \mu_2$ is a valid coupling of proper distributions $\mu_1$ and $\mu_2$.

We have one-sided and two-sided rules for conditionals. The [COND-L] rule is the one-sided version; it can be derived from the [CASE] and [STRUCT] rules. The [COND-S] rule is the two-sided version; the rule assumes that the two guards of the conditional statements are synchronized, so that one must only need to reason about the cases where both statements enter the true branch, and the case where both statements enter the false branch. It can be derived from the [CASE], [STRUCT], and [FALSE] rules.

We also have one-sided and two-sided rules for loops. The [WHILE-L] rule corresponds to the pRHL one-sided rule for while loops; it can be derived from the [WHILE] rule by setting $p_0 = \top$, $p_0 = p_2 = \perp$, and $k_1 = k_2 = 1, e = c_1, c_2 = \perp, c_2 = \text{skip}$ and using the [STRUCT] and [FALSE] rules. The rule [WHILE-S] corresponds to the pRHL two-sided rule for while loops. This rule assumes that the two loops are synchronized, i.e., the guards of the two loops are equal assuming the loop invariant. This rule can be derived from the general [WHILE] rule by setting $p_0 = \top$.
p_1 = p_2 = \bot, and k_1 = k_2 = 1, and e = e_1, and using the [FALSE] rule.

3.3 Soundness and (relative) completeness

We have formally verified the soundness theorem below in the Coq proof assistant, with Sreflect.

Theorem 7 (Soundness). Every derivable judgment is valid.

Proof sketch. By induction on the proof derivation. The most interesting case is the loop rule, which we briefly sketch here. We will use the same notations from Fig. 3. Proving that the product program satisfies the post-condition is immediate; by construction, the condition e does not hold after the execution of the loop. Moreover, the premises ensure that the loop body of the product preserves the invariant \( \Psi \). So, we just need to prove that the semantics of the projections of the product program correspond to respective semantics of the original programs. We detail the first projection, i.e. that for any initial memory \( m \) s.t. \( m \models \Psi \), we have

\[
[w]_{\Psi} = \pi_1 ([w]_m)
\]

where \( w \triangleq \text{while } e \text{ do } c \) and \( w_i \triangleq \text{while } e_i \text{ do } c_i \); the other projection is similar. By definition, we have for any \( e \) and \( c \)

\[
[w]_m = \lim_{n \to \infty} [c_e.n]_m
\]

Since \([c_e.n]_m\) is monotonic in \( n \), proving Eq. (1) amounts to showing

\[
[e_1.e_2.n]_{x_1} \leq \pi_1 ([w]_m)
\]

\[
\pi_1 ([c_e.n]_m) \leq [w]_{\Psi} \quad (3)
\]

for all \( n \). We proceed by induction on \( n \). Let \( m_1 \triangleq \pi_i (m) \) for \( i \in \{1, 2\} \). The base case is immediate and we consider the \( n + 1 \) case. We will prove (2) by case analysis on \([e_1]_m\) and \([e_2]_m\). If \([e_1]_m\) does not hold, then \([e_1.e_2.n]_{x_1} = 1.m_1 \). Now if \([e_2]_m\) is false, then \([e]_m\) is also false and (2) holds. Otherwise if \([e_2]_m\) is true, \([p_0]_m\) and \([p_1]_m\) cannot hold, since that would imply that \([e_1.n]_{x_1}\) holds. So in this case, we can conclude (2) since \([w]_m = 1.m_1 \times [w_2]_{m_2}\) and \( w_2 = \Psi\text{-lossless}.

Otherwise, \([e_1]_m\) holds. We proceed by case analysis on \([p_0]_m\), \([p_1]_m\) and \([p_1]_m\), as they are mutually exclusive by the premise.
\[
\begin{align*}
\text{ASSG-L} & \quad \{ \Psi[e_1/x_1] \} \quad x_1 \leftarrow e_1 \quad \{ \Psi \} \quad x_1 \leftarrow e_1 \\
\text{RAND-L} & \quad \{ \forall v_1 \in \text{supp}(\mu_1), \Psi[v_1/x_1] \} \quad x_1 \leftarrow \text{\textcolor{red}{\mu_1}} \quad \{ \Psi \} \quad x_1 \leftarrow \text{\textcolor{red}{\mu_1}} \\
\text{COND-L} & \quad \{ \Phi \land e_1 \} \quad c_1 \quad c_2 \quad \{ \Psi \} \quad \rightarrow \quad \{ \Phi \land \lnot e_1 \} \quad c' \quad c_1 \quad c_2 \quad \{ \Psi \} \quad \rightarrow \quad \text{if } e_1 \text{ then } c_1 \text{ else } c' \\
\text{COND-S} & \quad \{ \Phi \} \quad \text{if } e_1 \text{ then } c_1 \text{ else } c' \quad \{ \Psi \} \quad \rightarrow \quad \text{if } e_1 \text{ then } c_1 \text{ else } c' \\
\text{WHILE-L} & \quad \{ \Psi \} \quad \text{while } e_1 \text{ do } c_1 \quad \{ \Psi \land \lnot e_1 \} \quad \rightarrow \quad \text{while } e_1 \text{ do } c_1 \\
\text{WHILE-S} & \quad \{ \Psi \} \quad \text{while } e_1 \text{ do } c_1 \quad \{ \Psi \land \lnot e_1 \} \quad \rightarrow \quad \text{while } e_1 \text{ do } c \\
\end{align*}
\]

Figure 4. Derived rules

- If \([p_0]_m \) holds, \(k_1 > 0 \) implies

\[
[c_{e_1,n+1}]_{m_1} \leq c_{e_1} \cdot c_{e_1,n} \leq c_{e_1,k_1} \cdot c_{e_1,n} \]

which allows us to conclude using the induction hypothesis, since \([w]_m = [c_0; w]_m \) in this case and \([c_{e_1,k_1}]_{m_1} = \pi_1([c_0]_m) \).

- If \([p_1]_m \) holds, we have

\[
[c_{e_1,n+1}]_{m_1} = c_{e_1} \cdot c_{e_1,n} \]

which allows to conclude using the induction hypothesis.

- If \([p_2]_m \) holds, we have

\[
[c_{e_1,n+1}]_{m_1} = E_{m_1 \sim \pi_1} [\text{while } e_2 \land p_2 \text{ do } c_2; w]_m \cdot \pi_1(w)_{m'}
\]

Note that after the executing while \(e_2 \land p_2 \text{ do } c_2 \)—which is lossless—\( \Psi \) and \(e_1\) still hold while \(e_2\) does not. By the premises of the rule, \(p_2\) must be false so \(p_2 \) or \(p_1 \) holds on every memory \(m_1 \)'s in the support. By the first two cases with initial memory \(m_1 \)'s, we have:

\[
[c_{e_1,n+1}]_{m'} \leq \pi_1[w]_{m'}.
\]

We can then conclude:

\[
[c_{e_1,n+1}]_{m} \leq E_{m' \sim \pi_1} [\text{while } e_2 \land p_2 \text{ do } c_2; w]_m \cdot \pi_1[w]_{m'} \text{ (linearity)} = \pi_1(E_{m' \sim [\text{while } e_2 \land p_2 \text{ do } c_2; w]_m} [w]_{m'}) \text{ (separable)} = \pi_1[w]_{m'} \text{ (definition)}
\]

Where the second to last step is because while \(e_2 \land p_2 \text{ do } c_2 \) does not modify \(\pi_1 m \) since \(e_1, e_2 \) are separable.

The proof of Eq. \([3] \) follows the same pattern. \(\square\)

Although it is not a primary objective of our work, we briefly comment on completeness of the logic. First, the coupling method is not complete for proving rapid mixing of Markov chains\(^7\). Second, it is not clear that our proof system is complete with respect to hoisting random assignments out of loops.

However, we note that the \textit{deterministic} fragment of our logic achieves completeness for programs that satisfy a sufficiently strong termination property; the key is that the new rule for while subsumes self-composition for while loops, provided they terminate on all initial memories satisfying the invariant. More precisely, we can prove the following completeness theorem.

\begin{theorem}
Let \(c_1\) and \(c_2\) be separable deterministic programs. If \(\{ \Phi \} c_1; c_2 \{ \Psi \} \) is derivable using the rules from Figure \([2] \) then \(\{ \Phi \} c_1 \{ \Psi \} \) \(\rightsquigarrow \) \(\lnot \) is derivable. Therefore, \(\times\)RHL is relatively complete for deterministic programs.
\end{theorem}

\begin{proof}
It suffices to prove that if \(\{ \Phi \} c_1 \{ \Psi \} \) is derivable using Hoare logic, then

\[
\{ \Phi \} \quad \text{skip} \quad \{ \Psi \} \quad \rightarrow \quad \lnot
\]

is derivable. The proof proceeds by induction on the derivation. \(\square\)

\(^7\) Kumar and Ramesh \([9] \) show that the class of \textit{causal couplings}—which contains all couplings in our logic—are unable to prove rapid mixing for some rapidly-mixing Markov chains \([29] [34] \).
3.4 Convergence from couplings

The fundamental theorem of couplings (Theorem 3) gives a powerful method to prove convergence of random processes. First, we recast it in terms of \( \times \text{pRHL} \).

Proposition 9. Let \( c_1 \) and \( c_2 \) be separable programs and assume that the following judgment is valid:

\[
\{ \Phi \} \quad c_1 \quad \{ \Psi \Rightarrow x_1 = x_2 \} \quad \leadsto \quad c
\]

Then for every memory \( m \) such that \( m \models \Phi \), we have

\[
\text{TV}(\mu_1, \mu_2) \leq \text{Pr}_{x \sim [c_1]_m} [-\Psi]
\]

where \( \mu_1, \mu_2 \) are the distributions obtained by sampling \( x_1 \) from \([c_1]_m \) and \( x_2 \) from \([c_2]_m \), respectively.

This result is a direct consequence of the soundness of the logic, and allows proving convergence in two parts. First, we use \( \times \text{pRHL} \) to establish a valid post-condition of the form \( \Psi \Rightarrow x_1 = x_2 \). Second, we prove that for every memory \( m \) satisfying some precondition, the product program \( c \) built from the derivation satisfies:

\[
\text{Pr}_{x \sim [c]_m} [-\Psi] \leq \beta
\]

There are multiple approaches for proving properties of this form—reasoning directly about the semantics of programs; existing formalisms for bounding probabilities and reasoning about expectations (e.g., [17, 24, 25]); program logics for probabilistic programs (e.g., [10, 24, 25]). We will check the property on pen and paper; mechanizing the proofs is left for future work.

4. Application: convergence of Markov chains

We now turn to our first group of examples: proving convergence of probabilistic processes. Suppose we have a probabilistic process on a set \( \Omega \) of possible states. At each time step, the process selects the next state probabilistically. Consider two runs of the same probabilistic process started from two different states in \( \Omega \). We would like to know how many steps we need to run before the two distributions on states converges to a common distribution. We consider several classic examples.

Notation. Throughout this section, we consider two copies of the same program. To ensure that the two copies are separable, we tag all the variables of the first copy with 1, and all the variables of the second copy with 2.

4.1 Simple, symmetric random walk

Our first example is a simple random walk on the integers. Let the state space \( \Omega \) be \( \mathbb{Z} \). At each step, the process chooses uniformly to move left (decreasing the position by 1) or right (increasing the position by 1). The program \( \text{rwalk} \) in Figure 5 implements the process, executed for \( t \) steps. The variable \( l \) is a ghost variable. While it does not influence the process, it keeps track of the list of sampled values, and will be used to state assertions when we construct the coupling.

Now, consider the random walk started from starting positions \( s_1, s_2 \). We want to show that the TV-distance between the two distributions on positions decreases as we run for more steps \( T \); roughly, the random walk forgets its initial position. It is not hard to see that if \( s_1 - s_2 \) is an odd integer, then we will not have convergence: at any timestep \( t \), the support of one distribution will be on even integers while the support of the other distribution will be on odd integers, so the TV-distance remains 1.

When \( s_1 - s_2 = 2k \) is even, we can construct a coupling to show convergence. Barthe et al. [12] used pRHL to couple these random walks by mirroring; informally, the coupled walks make mirror moves until they meet, when they make identical moves to stay equal. Specifically, they show that

\[
\{ \Phi \} \quad \text{rwalk}_1 \quad \{ k \in \text{psum}(\text{rev}(l_1)) \Rightarrow x_1 = x_2 \}
\]

where \( \Phi \equiv s_1 + s_2 = 2k, \text{rev}(l) \) reverses the list, and \( \text{psum}(l) \) is the list of partial sums of \( l \) (sums over its initial segments). We can lift the judgment to \( \times \text{pRHL} \):

\[
\{ \Phi \} \quad \text{rwalk}_1 \quad \{ k \in \text{psum}(\text{rev}(l_1)) \Rightarrow x_1 = x_2 \} \quad \leadsto \quad \text{rwalk}_0
\]

where \( \text{rwalk}_0 \) is the following product program:

\[
x_1 \leftarrow s_1; \quad x_2 \leftarrow s_2; \quad t_1 \leftarrow 0; \quad t_2 \leftarrow 0; \quad l_1 \leftarrow []; \quad l_2 \leftarrow [];
\]

while \( s_1 < T \) do

\[
\text{if } x_1 = x_2 \text{ then } \quad (r_1, r_2) \leftarrow D^-\left([-1, 1]\right);
\]

\[
\text{else } \quad (r_1, r_2) \leftarrow D_{\text{opp}}\left([-1, 1]\right);
\]

\[
x_1 \leftarrow x_1 + r_1; \quad x_2 \leftarrow x_2 + r_2; \quad l_1 \leftarrow x_1 :: l_1; \quad l_2 \leftarrow x_2 :: l_2;
\]

\[
t_1 \leftarrow t_1 + 1; \quad t_2 \leftarrow t_2 + 1;
\]

return \( (x_1, x_2) \)

Figure 5. Random walk

We briefly sketch the derivation of the product. We start by an application of the [WHILE-S] rule with the invariant

\[
\Psi \equiv k \in \text{psum}(\text{rev}(l_1)) \Rightarrow x_1 = x_2
\]

\[
\land k \notin \text{psum}(\text{rev}(l_1)) \Rightarrow x_1 - x_2 = 2k - \text{sum}(l_1),
\]

where \( \text{sum}(l) \) is the sum of the list. Then, we apply a [SEQ] rule (consuming the first random sampling on each side) with intermediate assertion

\[
\Xi \equiv \Psi \land (x_1 = x_2 \Rightarrow r_1 = r_2) \land (x_1 \neq x_2 \Rightarrow r_1 = -r_2)
\]

The sub-proof obligation on tails is straightforward, the interesting one is for the random sampling. We start by using the [CASE] rule with \( e \equiv x_1 = x_2 \), which introduce the conditional in the product program. If the equality holds, then the two random values are synchronized using the \( D^-\left([-1, 1]\right); \) if not, they are mirrored using \( D_{\text{opp}}\left([-1, 1]\right). \) In our mirror coupling, \( k \in \text{psum}(l_1) \) implies that both the have already met, and continue to have the same position.

However, the derivation by itself does not tell us how far the two distributions are, as a function of \( T \). To get this information, we will use the probabilistic product construction and the following classical result from the theory of random walks.

Theorem 10 (e.g., [3]). Let \( X_0, X_1, \ldots \) be a symmetric random walk on the integers with initial position \( X_0 = 0 \). Then, for any position \( k \in \mathbb{Z} \), the probability that the walk does not reach \( k \) within \( t \) steps is at most

\[
\text{Pr}[X_0, \ldots, X_t \neq k] \leq \frac{k \sqrt{2}}{\pi t/4}
\]

Now, we can analyze how quickly the two walks mix.

Theorem 11. If we perform a simple random walk for \( T \) steps from two positions that are \( 2k \) apart, then the resulting distributions \( \mu_1 \) and \( \mu_2 \) on final positions satisfy

\[
\text{TV}(\mu_1, \mu_2) \leq \frac{k \sqrt{2}}{\pi T/4}
\]

Formally, for every two memories \( m_1 \) and \( m_2 \) such that \( m_1(x) - m_2(x) = 2k \),

\[
\text{TV}(\mu_1, \mu_2) \leq \frac{k \sqrt{2}}{\pi T/4}
\]
we have
\[ \text{TV}([\text{rwalk}]_{m_1}, [\text{rwalk}]_{m_2}) \leq \frac{ke\sqrt{2}}{\pi \sqrt{T}}, \]

**Proof.** Conceptually, we can think of the difference \( x_1 - x_2 \) as a random walk which increases by \( 2 \) with half probability and decreases by \( 2 \) with half probability. By applying Theorem 10 to this random walk, we find that in the product program \( c_0 \),
\[ \Pr[k \notin \text{psum}(l)] \leq \frac{ke\sqrt{2}}{\pi \sqrt{T}}. \]

Then, we can conclude by Proposition 9.

\[ \square \]

### 4.2 The Dynkin process

Our second example models a process called the Dynkin process. There is a sequence of \( N \) concealed cards, each with a number drawn uniformly at random from \( \{1, \ldots, 10\} \). A player starts at some position in \( \{1, \ldots, N\} \). Randomly, the player looks at the number at their current position, and moves forward for many steps. For instance, if the player reveals 2 at their current location, then she moves forward two spaces. The player stops when she passes the last card of the sequence. We want to show fast convergence of this process if we start from any two initial positions.

In code, the Dynkin process is captured by the program defined in Figure 6. Here, \( l \) stores the history of positions of the player; this ghost variable will be useful both for writing assertions about the coupling, and for assertions in the product program. Just like for random walks, we can consider the mixing rate of this process, starting from two positions \( s_1, s_2 \).

We will couple two runs of the Dynkin process from two starting positions asynchronously: we will move whichever process is behind, holding the other process temporarily fixed. If both processes are at the same position, then they move together.

**Figure 6. Dynkin process**

Formally, we prove the following \( \times \text{pRHL} \) judgment:

\[
\{ \top \} \quad \text{dynkin}_1, \text{dynkin}_2 \quad \{ (l_1 \cap l_2 \neq \emptyset) \Rightarrow x_1 = x_2 \} \quad \leadsto \quad \text{dynkin}_0
\]

where \( \text{dynkin}_0 \) is the following program:

\[
x_1 \leftarrow s_1; \quad x_2 \leftarrow s_2;
\]

\[
l_1 \leftarrow [x_1]; \quad l_2 \leftarrow [x_2];
\]

\[\text{while } x_1 < N \text{ and } x_2 < N \text{ do}\]

\[
\text{if } x_1 = x_2 \text{ then}
\]

\[
(r_1, r_2) \in \mathcal{D}^N([1, 10]);
\]

\[
x_1 \leftarrow x_1 + r_1; \quad x_2 \leftarrow x_2 + r_2;
\]

\[
l_1 \leftarrow x_1 :: l_1; \quad l_2 \leftarrow x_2 :: l_2;
\]

\[\text{else}\]

\[
r_2 \in \mathcal{D}([1, 10]);
\]

\[
x_2 \leftarrow x_2 + r_2;
\]

\[
l_2 \leftarrow x_2 :: l_2;
\]

\[\text{return } (x_1, x_2)\]

To couple the two programs, we use the rule [WHILE] with \( k_1 = k_2 = 1 \). To control which process will advance, we define:

\[ p_0 \triangleq (x_1 = x_2) \quad p_1 \triangleq (x_1 < x_2) \quad p_2 \triangleq (x_1 > x_2). \]

The lossless conditions are satisfied since the distance between \( x_1 \) and \( x_2 \) strictly decreases at each iteration of the loops. For the samplings, when \( p_1 \) or \( p_2 \) hold we use the one-sided rules for random sampling ([RAND-L] and corresponding [RAND-R]); when \( p_0 \) holds, we use the identity coupling.

**Theorem 12.** Let \( m_1 \) and \( m_2 \) two memories such that \( m_1(s), m_2(s) \in [1, 10] \) and suppose \( N > 10 \). Then:

\[ \text{TV}([\text{dynkin}]_{m_1}, [\text{dynkin}]_{m_2}) \leq (\frac{9}{10})^{N/5-2}. \]

The two distributions converge exponentially fast as \( N \) grows.

**Proof.** In the product program \( \text{dynkin}_0 \), we want to bound the probability that \( l_1 \) and \( l_2 \) are disjoint; i.e., the probability that the two processes never meet. We proceed in two steps. First, it is not hard to show that \( |x_1 - x_2| < 10 \) is an invariant. Thus, at each iteration, there is a \( 1/10 \) chance that the lagging process hits the leading process. Second, each process moves at most 10 positions each iteration and we finish when both processes reach the end, so there are at least \( 2(\frac{N}{10} - 1) = \frac{N}{5} - 2 \) chances to hit. Therefore, in the product we can show

\[ \Pr[(l_1 \cap l_2) = \emptyset] \leq (\frac{9}{10})^{N/5-2}. \]

(4)

We can then conclude by Proposition 9.

\[ \square \]

To highlight the quantitative information verified by our approach, we note that the corresponding theorem for random walks (Theorem 11) shows that the total variation distance between two random walks decreases as \( O(1/\sqrt{T}) \). In contrast, Theorem 12 shows that total variation distance between two Dynkin processes converges as \( O(0.9^N) \), giving much faster mixing (exponentially fast instead of polynomially fast).

**Remark.** The Dynkin process is inspired by the following two-player game. Both players pick a starting position. There is one sequence of random cards that is shared by both players, and players look at the card at their current position and move forward that many spaces. The random cards are shared, so a player samples the card only if the other player has not yet visited the position. If the other player has already landed on the position, the later player looks at the revealed card and moves forward.

While the Dynkin process samples every card that the player lands on, it is not hard to see that the product program exactly models the two-player game. More specifically, the product interleaves the players so that at each turn, the player that is lagging behind makes the next move. By this scheduling, as long as one player is lagging behind, the players have not landed on the same position and so each player lands on unseen cards and draws random samples to reveal. Once the players meet, the product program makes the same move for both players.

For this game, Eq. (4) bounds the probability that the players do not land on the same final position from any two initial positions. This result is the basis of the magic trick called *Dynkin’s card trick* or the *Kruskal count*. If one player is the magician and the other player is a spectator, if the spectator starts at a secret position and runs the process mentally, the magician can guess the correct final position with high probability by starting at any position and counting along.

### 4.3 Path coupling

So far, we have seen how to prove convergence of probabilistic processes by constructing a coupling, reasoning about the probability of the processes not meeting, and applying the coupling theorem. While this reasoning is quite powerful, for more complicated processes it may be difficult to directly construct a coupling that shows fast mixing. For example, it can be difficult to find and reason
about a coupling on the distributions from two states \( s, s' \) if there are many transitions apart in the Markov chain.

To address this problem, Bubley and Dyer \cite{bubley2014efficient} proposed the path coupling technique, which allows us to consider just pairs of adjacent states, that is states where \( s \) can transition in one step to \( s' \). Roughly speaking, if we can give a good coupling on the distributions from two adjacent states for every pair of adjacent states, then path coupling shows that the state distributions started from two arbitrary states converge quickly.

As the name suggests, path coupling considers paths of states in a probabilistic process. For this to make sense, we need to equip the state space with additional structure. For the basic setup, let \( \Omega \) be a finite set of states and suppose that we have a metric \( d : \Omega \times \Omega \rightarrow \mathbb{N} \).

We require that \( d \) is a path metric: if \( d(s, s') > 1 \), then there exists \( s'' \neq s, s' \) such that \( d(s, s') = d(s, s'') + d(s'', s') \). Two states are said to be adjacent if \( d(s, s') = 1 \). We will assume that the diameter of the state space, i.e., the distance between any two states, is finite: \( \Delta < \infty \). The Markov chain is then defined by iterating a transition function \( f : \Omega \rightarrow \text{Dist}(\Omega) \).

The main idea behind path coupleings is that if we can couple the distributions from any two adjacent states, then there exists a coupling for the distributions from two states at an arbitrary distance, constructed by piecing together the coupleings along the path. Furthermore, if the expected distance between states contracts under the coupling on adjacent states, i.e., the resulting expected distance is strictly less than 1, then the same holds for the coupling on two states at any distance. More formally, we have the following main theorem of path coupling:

**Lemma 13** (Bubley and Dyer \cite{bubley2014efficient}). Consider a Markov chain with transition function \( f \) over a set \( \Omega \) with diameter at most \( \Delta \). Suppose that for any two states \( s \) and \( s' \) such that \( d(s, s') = 1 \), there exists a coupling \( \mu \) of \( f(s), f(s') \) such that

\[
\mathbb{E}_{(r, r') \sim \mu}[d(r, r')] \leq \beta.
\]

Then, starting from any two states \( s \) and \( s' \) and running \( t \) steps of the chain, there is a coupling \( \mu' \) of \( f^t(s), f^t(s') \) such that

\[
\text{TV}(f^t(s), f^t(s')) \leq \mathbb{E}_{(r, r') \sim \mu'}[r \neq r'] \leq \beta^t \Delta.
\]

This lemma applies for all \( \beta \), but is most interesting for \( \beta < 1 \) when it implies that the Markov chain mixes quickly. With the main theorem in hand, we will show how to verify the conditions for path coupling on two examples from Bubley and Dyer \cite{bubley2014efficient}.

### 4.3.1 Graph coloring: the Glauber dynamics.

Our first example is a Markov chain to provide approximately uniform samples from the set of colorings of a finite graph; it was first analyzed by Jerrum \cite{jerrum1993polynomial}, our analysis follows Bubley and Dyer \cite{bubley2014efficient}.

Recall that a finite graph \( G \) is defined by a finite set of vertices \( V \), and a symmetric relation \( E \) relating pairs vertices that are connected by an edge; we will let \( N_G(v) \subseteq V \) denote the set of neighbors of \( v \); i.e., the set vertices that have an edge to \( v \) in \( G \). Let \( C \) be the set of colors; throughout, we assume that \( C \) is finite. A coloring \( \omega \) of \( G \) is a map from \( V \) to \( C \). A coloring is valid (sometimes called proper) if all neighboring vertices have different colors; for all \( v' \in N_G(v) \) we have \( \omega(v') \neq \omega(v) \). We write \( \mathcal{V}_G(\omega) \) if \( \omega \) is a valid coloring.

The following program \texttt{glauber}(\( T \)) models \( T \) steps of the Glauber dynamics in statistical physics:

\[
t \leftarrow 0;
\]

\[
\text{while } t < T \text{ do:}
\]

\[
v \leftarrow V; \quad c \leftarrow C;
\]

\[
\text{if } \mathcal{V}_G(\omega[v \rightarrow c]) \text{ then } \omega[v \rightarrow c]
\]

\[
t \leftarrow t + 1
\]

Informally, the algorithm starts from a valid coloring \( \omega \) and iteratively modifies it by sampling uniformly a vertex \( v \) and a color \( c \), recoloring the \( v \) with \( c \) if this continues to be a valid coloring.

We want to measure the statistical distance between two executions of the process starting from two initial colorings \( w_1 \) and \( w_2 \). There are two natural approaches. The first option is to couple the two copies of glauber directly, analyze the product program and apply Proposition \ref{prop:coupling}. The problem is that when the two colorings are far apart, it is hard to reason about how the processes might meet under a coupling; Jerrum \cite{jerrum1993polynomial} adopted this strategy, but the resulting proof is dense and complex.

The second, far simpler option is to apply path coupling. Here, we build a product for just one iteration of the loop, and it suffices to consider cases where the two initial states are adjacent. This drastically simplifies the coupling and analysis of the product program, so we adopt this approach here. For the sake of clarity, we adapt the transition function so that its output is stored in a fresh variable \( w' \), and call the resulting program \texttt{glauber'}:

\[
v \leftarrow \emptyset; \quad V; \quad c \leftarrow C;
\]

\[
\text{if } \mathcal{V}_G(\omega[v \rightarrow c]) \text{ then } w' \leftarrow \omega[v \rightarrow c] \quad \text{else } w' \leftarrow w
\]

Note that in \texttt{glauber'}, \( w \leftarrow w' \); \( t \leftarrow t + 1 \) is semantically equivalent to the loop body of \texttt{glauber}.

To apply the path coupling theorem Lemma \ref{lem:path_coupling}, we need to define a path metric on \( \Omega \) and construct a coupling for the process started from two adjacent states. For the path metric, we define the distance \( d(w_1, w_2) \) to be the Hamming distance: the number of vertices where \( w_1 \) and \( w_2 \) provide different colors. We say two states are adjacent if \( d(w_1, w_2) = 1 \); these states differ in the color of exactly one vertex. In order to apply path coupling, we need to find a coupling of the transition function on adjacent states such that the expected distance shrinks. We first build the coupling using \( \times \text{pRHL} \). Specifically, we prove

\[
\left\{ d(w_1, w_2) = 1 \right\} \overset{\text{glauber'}}{\sim} \left\{ d(w'_1, w'_2) \leq 2 \right\} \sim c_0 \tag{5}
\]

where \( c_0 \) is the following program:

\[
v_1, v_2 \leftarrow D^\omega(V);
\]

\[
\text{if } v_1 \in \mathcal{N}_G(v_0) \text{ then } c_1, c_2 \leftarrow D_\omega(C) \quad \text{else } c_1, c_2 \leftarrow D^\omega(C)
\]

\[
\text{if } \mathcal{V}_G(w_1 [v_1 \rightarrow c_1]) \text{ then } w'_1 \leftarrow w_1 [v_1 \rightarrow c_1] \quad \text{else } w'_1 \leftarrow w_1
\]

\[
\text{if } \mathcal{V}_G(w_2 [v_2 \rightarrow c_2]) \text{ then } w'_2 \leftarrow w_2 [v_2 \rightarrow c_2] \quad \text{else } w'_2 \leftarrow w_2
\]

We briefly sketch how to prove the judgment. Note that the two states must agree at all vertices, except at a single vertex \( v_0 \). Let \( w_1(v_0) = a \) and \( w_2(v_0) = b \). First, we couple the vertex sampling with the rule \[ \text{RAND} \] using the identity coupling, ensuring \( v_1 = v_2 \). Then, we use the rule \[ \text{CASE} \] to perform a case analysis on the sampled vertex, call it \( v \). If \( v \) is a neighbor of the differing vertex \( v_0 \), we use the rule \[ \text{RAND} \] and the transposition bijection \( \pi_{ab} : C \rightarrow C \) defined by the clauses:

\[
\pi_{ab}(a) = b \quad \pi_{ab}(b) = a \quad \pi_{ab}(x) = x \quad \text{otherwise}
\]

ensure that \( c_1 = \pi_{ab}(c_2) \). Otherwise, we use the rule \[ \text{RAND} \] and the identity coupling to ensure \( c_1 = c_2 \). By applying the one-sided rules for conditionals to the left and the right sides (\[ \text{COND-}L \] and \[ \text{COND-R} \]), we can conclude the derivation.

Next, we must reason about the expected value of the distance between \( w'_1 \) and \( w'_2 \) after executing the product program.

**Lemma 14.** Let \( n = |V| \) and \( k = |C| \), and suppose that the graph \( G \) has degree bounded by \( D \). That is, for any \( v \in V \), there are at most \( D \) vertices \( v' \) such that \( E(v, v') \). If \( k \geq D \), then there is a coupling \( \mu \) of the distributions after running \texttt{glauber} on adjacent states such that

\[
\mathbb{E}_{(w_1, w_2) \sim \mu}[d(w'_1, w'_2)] \leq 1 - 1/n + 2D/kn.
\]
Proof. Let \( w_1 \) and \( w_2 \) be adjacent states. We must bound the expected distance between the states \( w_1', w_2' \) in the product program. Let \( d' = d(w_1, w_2) \), we have:

\[
\mathbb{E}_{(w_1, w_2) \sim \mu}[d'] = 0 \cdot \Pr[d' = 0] + 1 \cdot \Pr[d' = 1] + 2 \cdot \Pr[d' = 2] \\
= 1 - \Pr[d' = 0] + \Pr[d' = 2] \\
\leq 1 - \Pr[v_1 = v_0 \land \mathcal{V}_G(w')] + \Pr[v_1 \in \mathcal{N}_G(v_0) \land c_1 = b] \\
\leq 1 - \frac{1}{n} \left(1 - \frac{D}{k}\right) + \frac{D}{nk} = 1 - \frac{1}{n} + \frac{2D}{nk},
\]

where \( w' = w_1[v_0 \mapsto c_1] \). The first equality holds because the distance between the two resulting colorings will be at most 2 by judgment Eq. (11). The second equality holds since \( 1 = \Pr[d' = 0] + \Pr[d' = 1] + \Pr[d' = 2] \). The second to last step follows since each vertex has at most \( D \) neighbors, so there are at least \( k - D \) valid colors at any vertex.

Applying the path coupling lemma (Lemma 13), noting that the diameter is \( n \) since there are \( n \) vertices, proves that the Glauber dynamics mixes quickly if there are sufficiently many colors \( k \).

Theorem 15. Consider the Glauber dynamics on \( k \) colors with a graph \( G \) with \( n \) vertices and degree at most \( D \), and suppose \( k \geq 2D + 1 \). Then, for some constant \( \beta < 1 \),

\[
\text{TV}(\mu^{\text{glauber}}_1, \mu^{\text{glauber}}_2) \leq \beta^T n
\]

for any two initial distributions \( \mu^{\text{glauber}}_1 \) and \( \mu^{\text{glauber}}_2 \) containing valid colorings.

This theorem recovers the result by Bubley and Dyer (23): this is the key step to showing that running Glauber dynamics for a small number of steps and taking a sample is almost equivalent to drawing a uniformly random sample from all proper colorings of the graph.

4.3.2 Independent sets: the conserved hard-core model

Our second example is from graph theory and statistical physics, modeling the evolution of a physical system in the conserved hard-core lattice gas (CHLG) model (23). Suppose that we have a set \( P \) of particles, and we have a graph \( G \). A placement is a map \( \omega : P \rightarrow V \), placing each particle at a vertex of the graph. We wish to place the particles so that each vertex has at most one particle, and no two particles are located at adjacent vertices; we call such a placement safe and denote it by \( \mathcal{S}_G(\omega) \). For a specific graph, there could be multiple safe placements.

If we want to sample a uniformly random safe placement, we can use a simple Markov chain. We take the state space \( \Omega = P \rightarrow V \) to be the set of placements. Again, we take \( G \) and \( P \) to be finite. We start using a safe initial placement. Each step, we sample a particle \( p \) from \( P \) and a vertex \( v \) from \( V \) uniformly at random and try to move \( p \) to \( v \). If \( \omega(p \rightarrow v) \) is safe, then we make this the new placement; otherwise, we keep the same placement. We can model \( T \) steps of this dynamics with the following program \( \text{chlg}(T) \):

\[
\begin{align*}
t &\leftarrow 0; \\
\text{while } t < T \text{ do:} & \\
\quad p &\notin P; v \notin V; \\
\quad \text{if } \mathcal{S}_G(\omega(p \rightarrow v)) &\text{ then } \omega \leftarrow \omega(p \rightarrow v) \\
\quad t &\leftarrow t + 1
\end{align*}
\]

As in the previous example, we adapt the loop body to form \( \text{chlg}^1 \):

\[
\begin{align*}
p &\notin P; v \notin V; \\
\quad \text{if } \mathcal{S}_G(\omega(p \rightarrow v)) &\text{ then } w' \leftarrow \omega(p \rightarrow v) \text{ else } w' \leftarrow w
\end{align*}
\]

Like the graph coloring sampler, we take the path metric on placements and try to find a coupling on the distributions from adjacent initial placements.

Lemma 16. Let \( s = |P| \) and \( n = |V| \), and suppose that the graph \( G \) has degree bounded by \( D \). Starting from any two adjacent safe placements \( w_1, w_2 \), there is a coupling \( \mu \) on the distributions after one step such that

\[
\mathbb{E}(w_1, w_2)_{\sim \mu}[d(w_1', w_2')] \leq \left(1 - \frac{1}{s}\right) \left(\frac{3(D + 1)}{n}\right).
\]

Proof. Let \( \text{chlg}_1, \text{chlg}_2 \) be two copies of the transition function, with variables tagged. Consider two adjacent placements \( w_1, w_2 \). We will sketch how to couple the transitions.

We use rule \( \text{RAND} \) twice to couple the particle and vertex samplings with the identity coupling, ensuring \( p_1 = p_2 \) and \( v_1 = v_2 \). Then, we can apply the one-sided rules for conditionals to the left and the right sides (\( \text{COND-L} \) and \( \text{COND-R} \)) to conclude the following judgment:

\[
\left\{ d(w_1, w_2) = 1 \right\} \text{chlg}_1 \uparrow \text{chlg}_2 \left\{ d(w_1, w_2) \leq 2 \right\} \sim c_0,
\]

where \( c_0 \) is the following product program:

\[
\begin{align*}
(p_1, p_2) &\notin D^-(V); (v_1, v_2) \notin D^+(V); \\
\text{if } \mathcal{S}_G(w_1[p \rightarrow v_1]) &\text{ then } \omega' \leftarrow \omega(p \rightarrow v_1) \text{ else } \omega' \leftarrow \omega \\
\text{if } \mathcal{S}_G(w_2[p \rightarrow v_2]) &\text{ then } \omega'' \leftarrow \omega(p \rightarrow v_2) \text{ else } \omega'' \leftarrow \omega
\end{align*}
\]

Then, we can bound the expected distance between \( w_1', w_2' \) in the product program. Let \( d' = d(w_1', w_2') \), we have:

\[
\begin{align*}
\Pr\left[ d'(w_1', w_2') \leq 1 \right] &\leq 1 - \Pr[d' = 0] + \Pr[d' = 2] \\
= 1 - \Pr[p \neq p_0 \land \mathcal{S}_G(w_1[p \rightarrow v])] \\
+ \Pr[p \neq p_0 \land \mathcal{S}_G(w_2[p \rightarrow v])] &\leq 1 - \Pr[p \neq p_0 \land \mathcal{S}_G(w_1[p \rightarrow v])] \\
+ \Pr[p \neq p_0 \land \mathcal{S}_G(w_2[p \rightarrow v])] &\leq 1 - \frac{1}{s} \left(\frac{3(D + 1)}{n}\right).
\end{align*}
\]

Applying the path coupling lemma (Lemma 13), shows that if we iterate the transition function on two initial placements, the resulting distributions on placements converge quickly.

Theorem 17. Consider the conserved hard-core lattice gas model with \( s = |P| \) particles on a graph \( G \) with \( n = |V| \) vertices and degree at most \( D \), where \( s \leq n/3(D + 1) + 1 \). Then, for a constant \( \beta < 1 \),

\[
\text{TV}(\text{chlg}_1, \text{chlg}_2) \leq \beta^T s
\]
for any two initial memories \(m_1, m_2\) containing safe placements.

Remark. This theorem is slightly weaker than the corresponding result by Bubley and Dyer [23], who prove rapid mixing under the weaker condition \(s \leq n^2/(D + 1) + 1\). Roughly, they use the maximal coupling on the two transition distributions, giving a tighter analysis and better bound. It is also possible to use the maximal coupling in \(\times pRHL\), but the corresponding specification of the coupling distribution would be proved as part of soundness of the logic, rather than as a property of a probabilistic program.

5. Application: loop optimizations

Program equivalence is one of the original motivations for relational program logics [17]. In this section, we demonstrate the effectiveness of our logic using several examples of exact and approximate program equivalence. Our first example is a loop transformation which originates from the literature on parallelizing compilers but also has applications in computer-aided cryptography. Our second example is drawn from the recent literature on approximate computing, and is an instance of loop perforation.

5.1 Loop strip-mining

Loop strip-mining (or loop sectioning) is a transformation that turns a loop into a nested loop. While the transformation originates from literature on parallelizing compilers and is primarily used to take advantage of vectorized instructions, it is also useful for formally proving the computational security of certain cryptographic constructions. The following example, Fig. 7 is inspired from a proof of indistinguishability of the SHA3 hash function [24].

Using the rule for while loops, we can prove the following:

\[
\begin{align*}
x_1 &= x_2 \\
\frac{c_1}{c_2} \quad \frac{x_1 = x_2}{\leadsto c}
\end{align*}
\]

The crux of the proof is applying the [WHILE] rule with \(k_1 = 1\) and \(k_2 = M\), and \(e = i < N\), and \(p_0 = \top\), and \(p_1 = p_2 = \bot\) and an invariant \(\Psi\) which strengthens the assertion \(x_1 = x_2\) mainly by adding \(l_2 = l_1 \times M\). Side conditions using \(p_1\) and \(p_2\) are trivial to prove (using the FALSE rule) since they have \(\bot\) in hypothesis. It remains to check the premise for \(p_0\), but we now have two synchronized loops; we can use the [STRUCT] rule to remove the conditional on \(p_0\) which is always true in this case.

5.2 Loop perforation

Loop perforation [44, 51] is a program transformation that delivers good trade-offs between performance and accuracy, and is practical in many applications, including image and audio processing, simulations and machine-learning. Informally, loop perforation transforms a loop that performs \(n\) iterations of its body into a loop that performs \(m < n\) iterations of its body, followed by a simple post-processing statement. Figure 8 shows an example of loop perforation inspired from a financial analysis application, called swaptions. In this example, every other loop iteration is skipped, and the post-processing statement simply multiplies by 2 the value \(s\) computed by the optimized loop. As for the previous example, we can prove the following judgment:

\[
\begin{align*}
\top \quad \frac{c_1}{c_2} \quad \frac{s_1 = s_2}{\leadsto c}
\end{align*}
\]

The product program can be built using the [WHILE] rule. We use \(e = i < 2 \times n, p_0 = \top\), \(p_1 = p_2 = \bot\) and \(k_1 = 2\) and \(k_2 = 1\), and the invariant is \(s_1 = 2 \times s_2\). The invariant allows to show, using the

\[\text{for simplicity, the programs use an operator } f\text{ which takes randomness as an argument (note that the value } r\text{ is sampled immediately before the assignment using } f),\] although in the proof of the SHA3 hash function \(f\) is a procedure call whose body performs random samplings.

<table>
<thead>
<tr>
<th>Nested loop:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i_1 \leftarrow 0); while (i_1 &lt; N) do</td>
</tr>
<tr>
<td>(j_1 \leftarrow 0); while (j_1 &lt; M) do</td>
</tr>
<tr>
<td>(l_1 \leftarrow l_1 \times M + j_1);</td>
</tr>
<tr>
<td>(r_1 \leftarrow \mu);</td>
</tr>
<tr>
<td>(x_1 \leftarrow f(l_1, x_1, r_1);)</td>
</tr>
<tr>
<td>(j_1 \leftarrow j_1 + 1);</td>
</tr>
<tr>
<td>(i_1 \leftarrow i_1 + 1);</td>
</tr>
<tr>
<td>(l_1 \leftarrow M \times N);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Single loop:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(l_2 \leftarrow 0); while (l_2 &lt; N \times M) do</td>
</tr>
<tr>
<td>(i_2 \leftarrow l_2); while (i_2 &lt; n) do</td>
</tr>
<tr>
<td>(x_2 \leftarrow f(l_2, x_2, r_2);)</td>
</tr>
<tr>
<td>(l_2 \leftarrow l_2 + 1);</td>
</tr>
<tr>
<td>(i_2 \leftarrow n);</td>
</tr>
<tr>
<td>(j_2 \leftarrow M);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product program:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i_1 \leftarrow 0; i_2 \leftarrow 0;)</td>
</tr>
<tr>
<td>while (i_1 &lt; N) do</td>
</tr>
<tr>
<td>(j_1 \leftarrow 0;)</td>
</tr>
<tr>
<td>while (j_1 &lt; M) do</td>
</tr>
<tr>
<td>(l_1 \leftarrow l_1 \times M + j_1;)</td>
</tr>
<tr>
<td>(r_2 \leftarrow \mu);</td>
</tr>
<tr>
<td>(x_2 \leftarrow f(l_2, x_2, r_2);)</td>
</tr>
<tr>
<td>(j_1 \leftarrow j_1 + 1; \quad l_2 \leftarrow l_2 + 1;)</td>
</tr>
<tr>
<td>(i_1 \leftarrow i_1 + 1;)</td>
</tr>
<tr>
<td>(l_1 \leftarrow N \times M; \quad i_2 \leftarrow N; \quad j_2 \leftarrow M;)</td>
</tr>
</tbody>
</table>

**Figure 7. Loop strip-mining**

**Figure 8. Loop perforation**

[STRUCT] rule, the loop on \(k_1\) (denoted by \(c^{k_1, k_2}_{i, i}\) in [WHILE]) perform exactly 2 iterations. Using the product, one can also analyze the (probabilistic) accuracy rate of the transformed program, using concentration bounds to achieve a more precise bound.

Finally, in some applications the number of iterations performed by the perforated loop is probabilistic: for instance, the program

\[k \neq \text{factors}(n); \quad s \leftarrow 0;\]

for \((i \leftarrow 0, i < n, i \leftarrow i + k)\) do

\[x \leftarrow \mu; \quad s \leftarrow s + x;\]

\[s \leftarrow k \times s;\]

selects uniformly at random a factor \(k\) of the original number \(n\) of iterations, and performs \(n/k\) iterations. It is possible to relate the original and the perforated loop, using the [WHILE] rule as before.
5.3 Other optimizations and program transformations

Barthe et al. [6] define an inductive method for building valid product programs, and use their method for validating instances of loop optimizations. Their method combines a rule for each program construction and a rule akin to our [STRUCT] rule. Despite this similarity, the two methods are fundamentally different: their treatment of while loops is restricted to synchronized executions. As a consequence, their structural rule is based on a more advanced refinement relation between programs. Nevertheless, we can reproduce all their examples in our formalism, taking advantage of our more powerful rule for loops.

6. Related work

Relational logics can be seen as a proof-theoretical counterpart of semantics-based relational methods such as logical relations. Under this view, our logic bears strong similarities with proof-relevant logical relations [18]. As for proof-relevant logical relations, we expect that manipulating explicit witnesses rather than existentially quantified can help developing meta-theoretical studies of our logic.

Much of the recent work on product programs and relational logics has been motivated by applications to security and compiler correctness. For instance, Barthe et al. [4] explore self-composition for a variety of programming languages and show that it induces a sound and complete reduction of an information flow policy to a safety property. Independently, Darvas et al. [27] consider self-composition and deductive verification based on dynamic logic, also for verifying information flow policies. Later, Terauchi and Aiken [53] introduce the class of 2-safety properties and show a reduction from 2-safety to safety of the self-composed program. Their reduction is more efficient than self-composition as it selectively applies self-composition or a synchronous product construction akin to cross-products (described below). Further improvements appear in Kovács et al. [35], Müller et al. [46]. In a related work, Beringer and Hofmann [20] observe that one can encode 2-safety properties in standard Hoare logic, provided that assertions are sufficiently expressive to model the denotational semantics of programs. Beringer [19] further refines this approach, by introducing the notion of relational decomposition.

Zaks and Pnueli [57] define a cross-product construction, which is well-suited for reasoning about programs with identical control-flow. Barthe et al. [6] generalize the notion of cross-product by proposing a more general notion of product program which subsumes self-composition and cross-products, and show how it enables validation of common loop optimizations. Specifically, they define an inductive relation for proving that c is a valid product for c1 and c2; informally, their rules closely follow those of our system (for deterministic constructs), except for the general rule for while loops; instead, they use a rule that is closer to the pRHL rule, and a rule akin to our [STRUCT] rule, with a much stronger relationship between programs in order to compensate for the lack of generality of their rule. Barthe et al. [14] carry a more precise study of the relative expressiveness of product program constructions and relational program logics. In a different thread of work, Barthe et al. [6] generalize the notion of product program so that it supports verification of refinement properties (modeled by universal quantification over runs of the first program and existential quantification over runs of the second program), as well as the 2-safety properties (modeled by universal quantification over runs of the first and the second programs). These constructions are focused on non-probabilistic programs. Motivated by applications to differential privacy, Barthe et al. [11] define a specialized product construction from probabilistic programs to deterministic programs, so that the original program is differentially private, provided its deterministic product program satisfies some Hoare specification. To the best of our knowledge, this is the sole product construction that goes from a probabilistic language to a deterministic one.

Benton [17] and Yang [56] were among the first to consider relational program logics that support direct reasoning about two programs. Benton [17] introduces Relational Hoare Logic, proves correctness of several program transformations, and soundly embeds a type system for information flow security into his logic. Yang [56] defines Relational Separation Logic and proves the equivalence between Depth-First Search and the Schorr-Waite algorithm. Barthe et al. [5] develop probabilistic Relational Hoare Logic, and use it for proving computational security of cryptographic constructions. In a follow-up work, Barthe et al. [7] develop an approximate variant of probabilistic Relational Hoare Logic, and verify differential privacy of several algorithms. More recently, Sousa and Dillig [52] propose Cartesian Hoare Logic, an extension of relational Hoare logic to an arbitrary finite number of executions.

Several authors have considered relational logics for higher-order programs. Nanoski et al. [43] develop a relational logic to reason about information flow properties of a higher-order language with mutable state. Ghani et al. [31] introduce a relational type theory, and a supporting categorical model, for reasoning about parametricity. Barthe et al. [10] propose a relational extension of a subset of the F∗; in a follow-up work, Barthe et al. [13] combine a relational refinement type system with a graded monad which they use for modeling differentially private computations.

There are several works that develop more specialized program logics for analyzing relational properties of programs. For instance, Amtott al. [4] introduce independence assertions and a supporting program logic for proving information flow security. In a similar way, Chaudhuri et al. [27] propose a logical approach for proving continuity properties of programs, and Carbin et al. [28] develop a logical approach for reasoning about the reliability of approximate computations.

Further afield, there has been a significant amount of work on semantical methods for probabilistic programs and processes initiated by Kozen [36], see e.g. [21, 38, 40, 49] for some recent developments. In conjunction with these semantics, research in deductive verification methods for non-relational properties of probabilistic programs is an active area of research; examples include [28, 37, 45, 48] to cite only a few systems.

7. Conclusion and future directions

We have introduced $\times$ pRHL, a new program logic that deepens the connection between probabilistic couplings and relational verification of probabilistic programs in two different ways. First, $\times$ pRHL broadens the class of couplings supported by relational verification. Second, $\times$ pRHL derivations explicitly build a probabilistic product program, which can be used to analyze mixing times. We have shown the flexibility of our approach on several examples.

There is ample room for future work. On the theoretical side, it would be interesting to extend $\times$ pRHL to handle continuous distributions as was recently done by Sato [50] for pRHL. Also, we believe that we are just scratching the surface of probabilistic product programs; there should be many further applications, notably in relationship with path couplings, in domains such as Brownian motion [22], molecular evolution [30], and anonymity [12]. On the more practical side, it would be natural to integrate $\times$ pRHL in EasyCrypt [9], a proof-assistant used for reasoning about computational security of cryptographic constructions, and currently the main implementation of pRHL. We expect that several proofs of cryptographic constructions can be simplified using the new loop rule, and we are also planning to use the loop rule in an ongoing formalization of indifferentiality of the SHA3 standard for hash functions.
References


We only consider deterministic programs in this section, i.e. programs without random samplings. In that case, the semantic $[s]_m$ of a deterministic command is either 0 if the program does not terminate, or of the form $1_{m'}$. If it does from that, we define the operator $[\cdot]_m$ as follows:

$[s]_m = \begin{cases} m' & \text{if } [s]_m = 1_{m'} \\ \bot & \text{otherwise} \end{cases}$

A while loop while $e$ do $s$ is said to terminate certainly under condition $\Phi$, written $\Phi \Downarrow$ while $e$ do $s$, iff for any $m$ s.t. $\Phi(m)$, we have $[\text{while } e \text{ do } s]_m \neq \bot$.

A Hoare triple is a triple of the form $\{ \Phi \} s \{ \Psi \}$ where $\Phi$ and $\Psi$ are assertions and $s$ is a deterministic command. A Hoare triple is said to be valid, written $\vdash \{ \Phi \} s \{ \Psi \}$, iff for any memory $m$ s.t. $\Phi(m)$, either $[s]_m = 1$ or $\Psi([s]_m)$. Figure 9 gives a sound inference system for Hoare triples: If $\vdash \{ \Phi \} s \{ \Psi \}$ is derivable from the rules of Figure 9 then $\vdash \{ \Phi \} s \{ \Psi \}$. Moreover, this system is complete for programs that only contain certainly terminating loops.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Precondition</th>
<th>Postcondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\vdash { \Phi } \text{skip} { \Psi }$</td>
<td>$\vdash { \top } \text{abort} { \bot }$ [ABORT]</td>
<td></td>
</tr>
<tr>
<td>$\vdash { \Phi } \text{assign} { \Psi }$</td>
<td>$\vdash { \Phi[\check{x} \leftarrow e] } x \leftarrow e { \Phi }$ [ASSIGN]</td>
<td></td>
</tr>
<tr>
<td>$\vdash { \Phi \land e } c_1 { \Psi }$</td>
<td>$\vdash { \Phi \land \neg e } c_2 { \Psi }$ [COND]</td>
<td></td>
</tr>
<tr>
<td>$\vdash { \Phi } \text{if } e \text{ then } c_1 \text{ else } c_2 { \Psi }$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\vdash { \Phi \land e } c { \Psi }$</td>
<td>$\vdash { \Phi } \text{while } e \text{ do } c { \Psi }$ [WHILE]</td>
<td></td>
</tr>
<tr>
<td>$\vdash { \Phi } \text{seq} { \Psi }$</td>
<td>$\vdash { \Psi } c_1 { \Xi }$ $\vdash { \Xi } c_2 { \Psi }$ [SEQ]</td>
<td></td>
</tr>
<tr>
<td>$\vdash { \Phi } \text{seq} { \Psi }$</td>
<td>$\vdash { \Phi } \text{seq} { \Psi }$</td>
<td></td>
</tr>
</tbody>
</table>

Figure 9. Hoare Logic Rules for Deterministic Programs