Linear-invariant generation for probabilistic programs:

automated support for proof-based methods

JP Katoen[1], AK McIver[2]*, LA Meinicke[2]*, and CC Morgan[3]*

[1] Software Modeling and Verification Group, RWTH Aachen University, Germany
[2] Dept. Computer Science, Macquarie University, NSW 2109 Australia
[3] School of Comp. Sci. and Eng., Univ. New South Wales, NSW 2052 Australia

Abstract. We present a constraint-based method for automatically generating *quantitative invariants* for *linear probabilistic programs*, and we show how it can be used, in combination with proof-based methods, to verify properties of probabilistic programs that cannot be analysed using existing automated methods. To our knowledge, this is the first automated method proposed for quantitative-invariant generation.

Key words: probabilistic programs, quantitative program logic, verification, invariant generation

1 Introduction

Verification of sequential programs rests typically on the pioneering work of Floyd, Hoare and Dijkstra [12, 17, 10] in which annotations are associated with control points in the program. For probabilistic programs, quantitative annotations are needed to reason about probabilistic program correctness [23, 7, 25]. We generalise the method of Floyd, Hoare and Dijkstra to probabilistic programs by making the annotations real- rather than Boolean-valued expressions in the program variables [23, 25]. As is well known, the crucial annotations are those used for loops, the loop *invariants*. Thus in particular we focus on real-valued, *quantitative* invariants: they are random variables whose expected value is not decreased by iterations of the loop [27].

One way of finding annotations is to place them speculatively on the program, as parametrised formula containing only first-order unknowns, and then to use a constraint-solver to solve for parameter-instantiations that would make the associated "verification conditions" true [4, 28, 5, 26, 13]. Such approaches are referred to as being *constraint-based.*

Our main contribution in this paper is to generalise the constraint-based method of Colón et al. [4] to probabilistic programs. We demonstrate our generalisation on a number of small-but-intricate probabilistic programs, ones whose analyses appear to be beyond other automated techniques for probabilistic programs at this stage. We discuss this in Sec. 7.

We begin in the next section with an overview of our approach.

* We acknowledge the support of the Australian Research Council Grant DP0879529.

2 Overall summary of the approach

For qualitative (non-probabilistic) programs, Boolean annotations are called *assertions*; and the associated *verification condition* for assertions P and Q separated by a program path *path_prog* (that does not pass through other annotations) is that they must satisfy the Hoare triple [17]

$$\{P\}\ \mathit{path\_prog}\ \{Q\} \quad \textit{or equivalently} \quad P \Rightarrow \mathit{wp.path\_prog.}Q\ ,$$

where *wp* refers to Dijkstra's *weakest-precondition* semantics of programs [10]. In either formulation, this condition requires that whenever the precondition P holds before the execution of *path_prog*, the postcondition Q holds after.

In the constraint-based method of Colón et al. [4], assertions for *linear programs* –programs with real-valued program variables in which expressions occurring in both conditionals and assignment expressions must be *linear* in the program variables– are found by speculatively annotating a program with Boolean expressions of the particular linear form $a_1x_1 + \ldots + a_nx_n + a_{n+1} \leq 0$, where $a_1, \ldots, a_{n+1}$ are parameters and $x_1, \ldots, x_n$ are program variables. The verification conditions associated with these annotations are then expressed as a set of polynomial constraints over the annotation-parameters and solved (for those unknown parameters) using off-the-shelf SAT solvers. This process yields Boolean annotations, that is assertions, from which program correctness can subsequently be inferred.

For probabilistic programs our real-valued (not Boolean) annotations are called *expectations* (rather than assertions), and the verification condition $\{P\}$ *path_prog* $\{Q\}$ is now interpreted as follows: if *path_prog* takes some initial state σ to a final distribution δ' on states, then the expected value of *post-expectation* Q over δ' is at least the (actual) value of *pre-expectation* P over σ. Using the quantitative *wp* semantics whose definition appears at Fig. 1, this condition is equivalently written as $P \leq \mathit{wp.path\_prog.}Q$. When there is no probability, quantitative *wp* is in fact isomorphic to ordinary (qualitative) *wp* [25].

Example 1. Consider a slot machine with three dials and two symbols, hearts ($\heartsuit$) and diamonds ($\diamondsuit$), on each one. The state of the machine is the configuration of the dials: a mapping from dials d_1, d_2 and d_3 to suits. The semantics of program *flip* that spins the dials independently so that they come to rest on each of the suits with equal probability,

$$\mathit{flip} \quad := \quad (d_1 := \heartsuit\ {}_{\frac{1}{2}}\oplus\ d_1 := \diamondsuit); (d_2 := \heartsuit\ {}_{\frac{1}{2}}\oplus\ d_2 := \diamondsuit); (d_3 := \heartsuit\ {}_{\frac{1}{2}}\oplus\ d_3 := \diamondsuit)\ ,$$

is then a function that maps each initial state to a single distribution δ' in which the probability of being in a slot-machine state is $\frac{1}{8}$ for each. If $\mathsf{all}.x$ is the expression $x{=}d_1{=}d_2{=}d_3$ and $[\cdot]$ is the function that takes false to 0 and true to 1, then we have for example that the expected value of $[\mathsf{all}.\heartsuit]$ over δ', $\mathit{wp.flip.}[\mathsf{all}.\heartsuit] = \frac{1}{8}$, is the probability of reaching final state $\mathsf{all}.\heartsuit$. This means that the probabilistic Hoare triple $\{1/8\}$ *flip* $\{[\mathsf{all}.\heartsuit]\}$ holds.

In general, a post-expectation may be any real-valued expression in the program variables, as the following example shows.

Example 2. Again with *flip*, a post-expectation Q may be used to represent the winnings assigned to each final configuration of suits. For instance, we could have $Q := 1\times[\mathsf{all}.\heartsuit] + \frac{1}{2}\times[\mathsf{all}.\diamondsuit]$ to represent that a gamer wins the whole jackpot if there are three hearts, a half if there are three diamonds, and nothing otherwise. Pre-expectation *wp.flip.Q* then represents a mapping from initial configurations of the slot machine to the least fraction of the jackpot the gamer can expect to win from that configuration. For the above Q, we have that *wp.flip.Q* is a mapping from each state to the value $\frac{3}{16}$, that is $6\times\frac{1}{8}\times 0 + \frac{1}{8}\times 1 + \frac{1}{8}\times\frac{1}{2}$.

Our first main technical contribution is to show how to determine the verification conditions for a probabilistic program annotation. To do this we must identify the appropriate notion of execution paths between annotations: this is because it doesn't make sense to speak of some annotation P's "being true here" when P is a real-valued expression over the program variables (rather than a Boolean predicate). The principal problem is paths through decision points, e.g. conditionals, where the truth (or falsity) of the Boolean condition cannot determine a "dead path" in the way that Colón does: we are not able to formulate a notion of "probably dead." Thus we explain in Sec. 4 below how, by imposing an extra condition on the program annotation, we can avoid this problem. For now we concentrate on the special case of annotating a single loop.

A single loop, *loop* := $\mathsf{while}\ G\ \mathsf{do}\ \mathit{body}\ \mathsf{od}$, is annotated as follows

$$\{I\};\mathsf{while}\ G\ \mathsf{do}\ \{[G]\times I\};\ \mathit{body}\ \mathsf{od};\{[\neg G]\times I\}\ ,$$

where I is some expectation. Such annotations are verifiable (i.e. *valid*) just when the expected value of I does not decrease after an iteration of the loop body, that is

$$[G]\times I \quad\leq\quad \mathit{wp.body.I}\ . \tag{1}$$

In this situation we refer to I as a *quantitative invariant* (or invariant) of the loop [27, 25]. In the case that the loop terminates (i.e. it terminates with probability 1), and indeed all of our examples in this paper are terminating, we may reason that if (1) holds so does the probabilistic Hoare triple $\{I\}\ \mathit{loop}\ \{[\neg G]\times I\}$.[4]

Example 3. The behaviour of a gamer that plays the slot-machine described earlier (at least once) until the dials show all hearts or all diamonds is represented by program

$$\begin{array}{ll} \mathit{init}: & \mathit{flip};\\ \mathit{loop}: & \mathsf{while}\ \ \neg(\mathsf{all}.\heartsuit \vee \mathsf{all}.\diamondsuit)\ \mathsf{do}\ \mathit{flip}\ \mathsf{od}\ . \end{array}$$

If the potential winnings are again described by Q (from Ex. 2), we can use the invariant $I := \frac{3}{4}\times[\neg(\mathsf{all}.\heartsuit \vee \mathsf{all}.\diamondsuit)] + 1\times[\mathsf{all}.\heartsuit] + \frac{1}{2}\times[\mathsf{all}.\diamondsuit]$ to calculate the gamer's expected winnings. (Playing the machine costs nothing in this simplistic example.) Since I is an invariant of *loop*, which terminates, and Q equals $[\mathsf{all}.\heartsuit \vee \mathsf{all}.\diamondsuit]\times I$, we have that $\{I\}\ \mathit{loop}\ \{Q\}$ holds. Thus the gamer can expect to win at least $\mathit{wp.init.I} = 6\times\frac{1}{8}\times\frac{3}{4} + \frac{1}{8}\times 1 + \frac{1}{8}\times\frac{1}{2} = \frac{3}{4}$ of the jackpot. (Half the time the loop will terminate showing all hearts and the gamer will win the whole jackpot, and half the time it will terminate with all diamonds and he will win half.)

[4] Quantitative invariants may also be used to reason about loops that terminate with some probability between 0 and 1 (see [25]).

Our second main technical contribution is to identify in Sec. 5.1 a class of probabilistic programs and parametrised expectations for which machine-solvable verification conditions can readily be extracted. As for Colón et al. [4] the class of probabilistic programs that our method works for is the set of *linear probabilistic programs*: the set of linear qualitative programs that may also contain discrete probabilistic choices made with a constant probability. Using our parametrised expectations it is possible to express invariants like I from Ex. 3.

Our third main technical contribution is to show in Sec. 5.2 how to convert our verification conditions on parametrised annotations to the same form as those generated by Colón et al. [4], so that they can be machine-solved in much the same way. Since this verification-condition translation is an equivalence, our method is both correct and fully general. That is, it can be used to find all parameter solutions that make an annotation valid, and no others.

3 Probabilistic programs

Probabilistic programs with nondeterministic and discrete probabilistic choices can be written using the probabilistic guarded command language (pGCL); in Fig. 1 we set out its syntax and *wp* semantics. Non-negative real-valued functions that are bounded above by some constant are referred to as *expectations*, and written as expressions in the program variables. For a probabilistic program *prog* and expectation Q, $wp.prog.Q$ represents the *least expected value* of Q in the final-state of *prog* (as an expression on the initial value of the program variables). This semantics is dual to an operational-style interpretation of program execution, where from an initial state σ the result of a computation is a set of probability distributions over final states; it is dual because $wp.prog.Q$ evaluated at σ is exactly the minimal expected value of Q over any of the result distributions. When Q is of the form $[R]$ for some Boolean expression R, then *wp* is in fact just the least probability that the final state will satisfy R, as in Example 1 above; but it can be more generally applied, as in Example 2.

Probabilistic guarded commands are *scaling*, $c{*}wp.prog.Q{=}wp.prog.(c{*}Q)$, and *monotonic*, $Q_1{\leq}Q_2{\Rightarrow}wp.prog.Q_1{\leq}wp.prog.Q_2$, for all expectations Q, Q_1, Q_2 and constants c [25]. Scaling, for example, is essentially linearity of expected values.

4 Probabilistic program annotations

If we imagine a program as a flowchart, a program annotation associates predicates with arcs and conventionally has the interpretation that a predicate is true of the program state whenever its associated arc is traversed during execution. Our generalisation of qualitative program annotations is to replace predicates (Boolean-valued expressions over the program variables) with expectations.

In order to specify verification conditions on these annotations, we impose restrictions on the program annotations that we allow: first, as for the qualitative case, there must be at least one annotation along any cyclic program path. This is so that verification conditions only involve cycle-free program fragments. Second,

	prog	$wp.prog.Q$
Identity	skip	Q
Assignment	$x := E$	$Q[x\backslash E]$
Composition	$prog_1; prog_2$	$wp.prog_1.(wp.prog_2.Q)$
Cond. choice	if G then $prog_1$ else $prog_2$ fi	$[G]\times wp.prog_1.Q + [\neg G]\times wp.prog_2.Q$
Nondet. choice	$prog_1 \sqcap prog_2$	$wp.prog_1.Q \sqcap wp.prog_2.Q$
Prob. choice	$prog_1\ {}_p\oplus\ prog_2$	$p{*}wp.prog_1.Q + (1{-}p){*}wp.prog_2.Q$
While-loop	while G do $body$ od	$(\mu X \cdot [G]\times wp.body.X + [\neg G]\times Q)$

x is a program variable; E is an expression in the program variables; $prog_{\{1,2\}}$ and *body* are probabilistic programs; G is a Boolean-valued expression in the program variables; p is a constant probability in $[0,1]$; and Q is an expectation (represented as a real-valued expression in the program variables). We write $Q[x\backslash E]$ to mean expression Q in which free occurrences of x have been replaced by expression E.

For expectations (interpreted as real-valued functions), scalar multiplication $*$, multiplication, $\times$, addition, $+$, subtraction, $-$, minimum, $\sqcap$, and the comparison (such as $\leq$ and $<$) between expectations are defined by the usual pointwise extension of these operators (as they apply to the real numbers). Multiplication and scalar multiplication have the highest precedence, followed by addition, subtraction, minimum and finally the comparison operators. Operators of equal precedence are evaluated from the left. μ is the least fixed point operator w.r.t. the ordering $\leq$ between expectations.

Function $[\cdot]$ takes Boolean expression *false* to 0 and *true* to 1. For $\{0,1\}$-valued functions, operation $\leq$ has the same meaning as implication over predicates, and $\times$ and $\sqcap$ represent conjunction, and addition over disjoint predicates is equivalent to disjunction.

Fig. 1. Probabilistic program notation and weakest-precondition semantics.

we assume that there is an annotation at the beginning and end of the program so that we can reason about the correctness of the whole. The third restriction is made so that we can reason about the branching behaviour of probabilistic programs. We require that if there is any "interior" annotation on a while-loop, conditional, nondeterministic or probabilistic choice, i.e. one following its choice point but occurring before the two choices rejoin, then the choice point itself must have three "immediate" annotations as well: one at its entry, and one at each of its (two) exits. Thus if we consider the flowchart generated by the annotated program fragment

$$\{P\}; prog_1; \textsf{if } G \textsf{ then } prog_2; \{Q\}; prog_3 \textsf{ else } prog_4 \textsf{ fi}; \{R\}\ , \tag{2}$$

we see that the conditional "if G" has an interior annotation Q — and so we must augment (2) with further annotations S, T, U as follows:

$$\{P\}; prog_1; \{S\}; \textsf{if } G \textsf{ then } \{T\}; prog_2; \{Q\}; prog_3 \textsf{ else } \{U\}; prog_4 \textsf{ fi}; \{R\}\ , \tag{3}$$

The S annotation is just before the choice point "if G"; the T annotation is just after its *true* exit; and the U annotation is just after its (implied) *false* exit. Loops and the other kinds of choice statements are similarly annotated.

A program annotation is *valid* when it satisfies the following verification conditions:

- For every pair (P, Q) of annotations separated by a program *path_prog* that does not contain annotations, if P does not appear just before a choice-point with an interior annotation then $\{P\}$ *path_prog* $\{Q\}$ holds. For example, for (3) we must have that $\{P\}\ prog_1\ \{S\}$, $\{T\}\ prog_2\ \{Q\}$, $\{Q\}\ prog_3\ \{R\}$ and $\{U\}\ prog_4\ \{R\}$ hold.
- Annotations appearing just before choice-points with interior annotations must be treated differently. In the case of program (3) for instance, it makes no sense to give a meaning to the Hoare triple $\{S\}$ "G is *true*" $\{T\}$. For annotation S in (3) we require that the "special" constraint $S \leq [G]{\times}T + [\neg G]{\times}U$ –that does not involve program execution at all– holds. Choice-point annotations on nondeterministic and probabilistic choices and while-loops must satisfy similar constraints. For example, annotation P in fragment $\{P\}; (\{Q\}; prog_1 \sqcap \{R\}; prog_2)$ must satisfy $P \leq Q \sqcap R$. Likewise, for $\{P\}; (\{Q\}; prog_1\ {}_p{\oplus}\ \{R\}; prog_2)$ we must have $P \leq p{*}Q + (1{-}p){*}R$.

Theorem 1. *Given a valid annotation of a terminating probabilistic program prog such that the first annotation is P and the last is Q, we have that prog satisfies the probabilistic Hoare triple $\{P\}$ prog $\{Q\}$.*

Proof. By structural induction over program texts.

For example, if (3) terminates and the annotation is valid then the probabilistic Hoare triple $\{P\}\ prog_1; \mathsf{if}\ G\ \mathsf{then}\ prog_2; prog_3\ \mathsf{else}\ prog_4\ \mathsf{fi}\ \{R\}$ holds.

4.1 The special case of loops

In this paper we deal only with single-loop programs (mostly) and the conditions above require that a loop be annotated (at least) with an expectation just before the loop, one just before the loop body (the *true* branch of the loop conditional) and one just after the loop (the *false* branch). For $loop := \mathsf{while}\ G\ \mathsf{do}\ body\ \mathsf{od}$, this amounts to the following annotation, $\{I\}; \mathsf{while}\ G\ \mathsf{do}\ \{J\}; body\ \mathsf{od}; \{K\}$, which is valid if

$$\{J\}\ body\ \{I\} \quad \textit{and special constraint} \quad I \leq [G]{\times}J + [\neg G]{\times}K$$

holds. We simplify this further by taking J to be $[G]{\times}I$ and K to be $[\neg G]{\times}I$ so that the special constraint is satisfied by construction — we need only find an I so that $\{[G]{\times}I\}\ body\ \{I\}$. Such an I is referred to as a *quantitative invariant.*

5 Constraint-solving for quantitative annotations

Given a "linear probabilistic program" annotated with parametrised real-valued expressions that are "propositionally linear" (the definitions for which appear in the following section), we show how to extract a set of polynomial constraints that are sufficient and necessary to show that the annotation is valid. Once we have the constraints we are able to apply constraint solvers to solve for the annotation parameters.

5.1 Linear probabilistic programs and parametrised annotations

An expression E on a given state space is *linear* if it is a linear combination of the program variables. A predicate P is a *linear constraint* if it is an inequality *or* a strict inequality between linear expressions. A *linear assertion* is then a finite conjunction of linear constraints. Finally, for any natural-valued constants M and N, and linear constraints P_{mn}, Boolean expression $(\bigwedge_{m:\,[1..M]} \bigvee_{n:\,[1..N]} P_{mn})$ is said to be a *propositionally linear predicate* with *conjunctive-degree* M and *disjunctive-degree* N.

A quantitative expression of the form $\sum_{m:\,[1..M]}[\bigwedge_{n:\,[1..N]} P_{mn}] \times Q_m$, where M and N are naturals, each P_{mn} is a linear constraint and Q_m is a linear expression, is referred to as a *propositionally linear expression* with *additive-degree* M and *conjunctive-degree* N. Such an expression is written in *disjoint normal form* if for all $i, j\colon [1..M]$ where $i \neq j$, we have $(\bigwedge_{n:\,[1..N]} P_{in}) \wedge (\bigwedge_{n:\,[1..N]} P_{jn}) = \mathit{false}$.

Lemma 1. *Any propositionally linear expression is semantically equivalent to another propositionally linear expression in disjoint normal form. (See App. C for proof.)*

A probabilistic program is said to be *linear* if the variables are real-valued, all of the guards are linear constraints, and updates are linear expressions.

To find valid quantitative annotations for a program with variables $x_1, \ldots, x_X$, we parametrise each annotation with a propositionally linear expression

$$\sum_{m:\,[1..M]}[\bigwedge_{n:\,[1..N]} \alpha_{(j,mn,1)}x_1 + \ldots + \alpha_{(j,mn,X)}x_X + \beta_{(j,mn)} \ll 0] \\ \times (\gamma_{(j,m,1)}x_1 + \ldots + \gamma_{(j,m,X)}x_X + \delta_{(j,m)})$$

containing free real-valued variables $\alpha_{(j,mn,x)}$, $\beta_{(j,mn)}$, $\gamma_{(j,m,x)}$ and $\delta_{(j,m)}$, in which each occurrence of $\ll$ is instantiated to either $<$ or $\leq$.

To ensure that each annotation P is an expectation (a real-valued expression bounded below by 0 and above by some real number) we require $0 \leq P \leq 1$. Restricting each annotation to be bounded above by 1 instead of an arbitrary upper bound does not limit our method because programs satisfy scaling (Sec. 3) so that if *prog* is correctly annotated with expectations P, Q, etc., then the modified annotation in which P is replaced by $c{*}P$, Q is replaced by $c{*}Q$ etc. is also valid for any non-negative constant c.

5.2 Constructing machine-solvable constraints

For qualitative programs the verification conditions for a linear program annotated with linear constraints can be formulated as Boolean expressions on linear constraints. After rewriting these expressions in conjunctive normal form (as propositionally linear predicates), Colón et al. [4] showed how it was possible to use Motzkin's Transposition theorem [14] to reduce each (universally quantified) finite disjunction of linear constraints to an existentially quantified polynomial formula over the annotation parameters.[5]

[5] To be precise, Colón et al. [4] used a specialisation of this theorem, Farkas' lemma, since they did not consider parametrised forms of invariants that could include strict inequalities.

For probabilistic programs, the verification conditions for a program annotation are of one of the five possible forms (Sec. 4):

$$0 \leq P \leq 1 \tag{4}$$

$$P \leq \mathit{wp.path\_prog.Q} \tag{5}$$

$$R \leq [G]\times S + [\neg G]\times T \quad \mathit{or} \quad R \leq S \sqcap T \quad \mathit{or} \quad R \leq p{*}S + (1{-}p){*}T \tag{6}$$

where P, Q, R, S and T are annotations, G is a Boolean expression in the program variables, p is a constant in $[0, 1]$, and *path_prog* is a loop- and annotation-free program fragment occurring in the program. For linear probabilistic programs, G must be a linear constraint, and sub-program *path_prog* must also be linear. By parametrisation the annotations are propositionally linear.

To convert each constraint of the form (4–6) to machine-solvable form we must first formulate each of them as a finite Boolean expression on linear constraints. We can then use Colón et al.'s method to convert these to polynomial formulae over the annotation parameters. We start by showing how this novel first step is performed and then we recount Motzkin's transposition theorem.

Equivalence translation to a finite Boolean expression. This translation occurs in two stages. First we convert each expression of the form (4–6) to inequalities between propositionally linear expressions. This first step is made possible by the following observations:

Lemma 2. *Let S and T be any propositionally linear expressions, G a linear constraint, p a constant expression, x a program variable, and E a linear expression. We have that $[G] \times S$, $S \sqcap T$ and $p{*}S$ are semantically equivalent to propositionally linear expressions; $\neg G$ may be expressed as a linear constraint; $S + T$ and $S[x\backslash E]$ are propositionally linear. (See App. C for proof.)*

Using these observations we immediately have that constraints of the form (4) and (6) can be translated to inequalities between propositionally linear expressions. For (5), we may use them to show by structural induction that *wp.path_prog.Q* may be evaluated to a propositionally linear expression.

In the second step we convert each of these inequalities between propositionally linear expressions to finite Boolean expressions over linear constraints (which may then be translated to conjunctive normal form):

Lemma 3. *Any inequality $Q_a \leq Q_b$ between non-negative propositionally linear expressions can be equivalently formulated as a finite Boolean expression over linear constraints.*

Proof. *First rewrite Q_a and Q_b in disjoint normal form (Lem. 1) as propositionally linear expressions $[P_{a1}]\times Q_{a1} + \cdots + [P_{aM}]\times Q_{aM}$, and $[P_{b1}]\times Q_{b1} + \cdots + [P_{bK}]\times Q_{bK}$, where each P_{am}, P_{bk} are linear assertions and Q_{am}, Q_{bk} are linear expressions. Then $Q_a \leq Q_b$ if and only if for all $m\colon [1..M]$ and $k\colon [1..K]$ we have $P_{am} \wedge P_{bk} \Rightarrow (Q_{am} - Q_{bk} \leq 0)$ and $P_{am} \wedge (\bigwedge_{k\colon [1..K]} \neg P_{bk}) \Rightarrow (Q_{am} \leq 0)$.*

Equivalence translation using Motzkin's transposition theorem. Motzkin's Transposition theorem can be used to equivalently represent any (universally quantified) propositionally linear predicate as a conjunction of existentially quantified constraints. Since the linear constraints in our propositionally linear predicate contain unknown coefficients, the constraints derived from Motzkin's Transposition theorem are polynomial, and not linear.

Theorem 2. *Motzkin's Transposition Theorem* Given the set of linear, and strict linear, inequalities over real-valued variables $x_1, ..., x_n$

$$S := \begin{bmatrix} \alpha_{(1,1)}x_1 & + \ldots + \alpha_{(1,n)}x_n & + \beta_1 & \leq 0 \\ \vdots & \vdots & \vdots & \\ \alpha_{(m,1)}x_1 & + \ldots + \alpha_{(m,n)}x_n & + \beta_m & \leq 0 \end{bmatrix}$$

$$T := \begin{bmatrix} \alpha_{(m+1,1)}x_1 & + \ldots + \alpha_{(m+1,n)}x_n & + \beta_{m+1} & < 0 \\ \vdots & \vdots & \vdots & \\ \alpha_{(m+k,1)}x_1 & + \ldots + \alpha_{(m+k,n)}x_n & + \beta_{m+k} & < 0 \end{bmatrix} \quad ,$$

in which $\alpha_{(1,1)}, ..., \alpha_{(m+k,n)}$ and $\beta_1, ..., \beta_{m+k}$ are real-valued, we have that S and T simultaneously are *not* satisfiable (i.e. they have no solution in x) if and only if there exist non-negative real numbers $\lambda_0, \lambda_1, \ldots, \lambda_{m+k}$ such that either

$$0 = \sum_{i=1}^{m+k} \lambda_i \alpha_{(i,1)}, \quad \ldots \quad , 0 = \sum_{i=1}^{m+k} \lambda_i \alpha_{(i,n)}, \ \ 1 = (\sum_{i=1}^{m+k} \lambda_i \beta_i) - \lambda_0 \ ,$$

or at least one coefficient λ_i for i in the range $[m+1 \ldots m+k]$ is non-zero and

$$0 = \sum_{i=1}^{m+k} \lambda_i \alpha_{(i,1)}, \quad \ldots \quad , 0 = \sum_{i=1}^{m+k} \lambda_i \alpha_{(i,n)}, \ \ 0 = (\sum_{i=1}^{m+k} \lambda_i \beta_i) - \lambda_0 \ .$$

Proof. This is a geometric rephrasing of the theorem as it appears in a standard reference [14, p.268].

5.3 Solving constraints and heuristics

Constraint solving. Our generated constraints are of the same form as those generated by Colón et al. [4] for qualitative programs, and may therefore be solved using exactly the same tools and techniques applicable there.

A survey of techniques for solving constraints is given by Bockmayr and Weispfenning [1]. Colón et al. [4] used, for example, REDLOG's[6] [11] implementation of quantifier-elimination algorithms for polynomial constraints. In addition to quantifier-elimination techniques, other methods such as factorisation and root finding were employed.

Quantifier-elimination implementations are exponential in complexity, which limits the size of annotation-generation problems that may be addressed using this approach. In our examples from Sec. 6 –which are of a small size– we solved our constraints using REDLOG.

[6] Available from `http://redlog.dolzmann.de/`.

Heuristics. In practice it may not be possible automatically to solve constraints when the program size is large or the parametrised invariants have either additive- or conjunctive- degree greater than say two or three or, even if we can, still the output of quantifier-elimination procedures might be unreadable. Colón et al. [4] encountered similar problems for trying to generate "k-linear inductive assertions" for values of k greater than one. As in [4] we recommend, where possible, (i) reducing the size of a problem by guessing values of certain parameters, and (ii) decomposing the task into finding structurally smaller invariants and (iii) finding invariants for sub-programs separately. Other suggestions (such as polynomial factorisation) may be found in [4].

To illustrate (ii), we have for instance that for linear assertion P and propositionally linear expression J the expression $I := [P]\times J$ is an invariant of the loop $\mathsf{while}\ G\ \mathsf{do}\ \mathit{body}\ \mathsf{od}$ if $[P]$ is invariant and $0 \le I \le 1$ and $[G]\times I \le \mathit{wp.body}.J$ holds. This method of decomposing the problem –although often applicable– is not complete. That is, there exist loop invariants of the form $[P]\times J$, where P is a linear assertion and J is a linear expression, such that $[P]$ on its own is not invariant.

Example 4. Consider program $x, y := 1, 1; \mathsf{while}\ y < N\ \mathsf{do}\ (y := 2y\ {}_{1/2}\oplus\ x := 0)\ \mathsf{od}$, in which N is a positive constant. Although $[x = 1 \wedge 0 \le y \le 2N]$ isn't an invariant of the loop since x is not guaranteed to remain at the value 1, $[x = 1 \wedge 0 \le y \le 2N]\times y$ is, since transitions that set x to 0 are balanced by y's doubling in value.

5.4 Soundness and completeness

Theorem 3. *For any linear probabilistic program annotated with propositionally linear expressions, our method is correct and fully general. That is, it can be used to find all parameter solutions that make the annotation valid, and no others.*

Proof. This follows from the fact that our translation of the annotation verification conditions to machine-solvable form (as defined in Sec. 4) is an equivalence.

This means, for instance, that our method can be used to find all propositionally linear invariants of a chosen degree for a single (i.e. un-nested) loop.

6 Three examples

We will now use the invariant-generation method set out on the preceding sections together with proof-based techniques to analyse three simple, terminating[7], probabilistic programs. The Boolean and natural-valued variables are interpreted more generally as reals (with Boolean value *true* represented by real-value 1 and *false* by 0), so that we can apply our approach.

[7] In each case a separate (and very simple) argument can be used to show that the programs terminate with probability 1.

```
init :  $x, n := 0, 0$;
loop :  while $n < N$ do
body :      $(x := x + 1 \; {}_p\!\oplus$ skip$); n := n + 1$
        od
```

Variables x and n are of type $\mathbb{N}$, constant $N\colon \mathbb{N}$, and constant $p\colon [0,1]$. To verify that the expected final value of x is at least $p{\times}N$ we must show that $wp.(init; loop).x \geq pN$, which is implied by the discovered loop invariant $[0{\leq}x{\leq}\, n \wedge n{\leq}N]{\times}(\alpha x - p\alpha n + p\alpha N)$, where $0 \leq \alpha \leq 1/N$.

Fig. 2. Binomial update.

6.1 Example one: binomial update

The program in Fig. 2 sets variable x to a value between 0 and constant N according to the binomial distribution with parameter p. We use our invariant-generation method to find invariants of *loop* for calculating lower-bounds on the final expected value of x.

We first search for invariants for *loop* of the form $I := [\alpha x + \beta n + \gamma \leq 0]$, that we can use to describe upper and lower bounds on the values of program variables x and n. In other words, we search for parameters α, β and γ that make the following program annotation valid:

$$\{I\}; \textsf{while } n{<}N \textsf{ do } \{[n{<}N] \times I\}; (x := x{+}1\,{}_p{\oplus} \textsf{skip}); n := n{+}1 \textsf{ od}; \{[n{\geq}N] \times I\}\ .$$

Solving for the constraints on the parameters, we find that $[0 \leq x]$, $[x \leq n]$ and $[n \leq N+1]$ are invariants.[8] Next, we search for quantitative invariants for *loop* of the form $I := J{\times}(\alpha x + \beta n + \gamma)$. where $J := [0 \leq x \wedge x \leq n \wedge n \leq N]$. Since J is invariant it suffices to show that for all values of x and n we have

$$0 \leq I \quad and \quad I \leq 1 \quad and \quad [n < N]{\times}(\alpha x{+}\beta n{+}\gamma) \leq wp.body.(\alpha x{+}\beta n{+}\gamma) \quad (7)$$

where $wp.body.(\alpha x{+}\beta n{+}\gamma)$ can be evaluated to $\alpha x{+}\beta n{+}p\alpha{+}\beta{+}\gamma$. Using the result of this *wp*-calculation, constraints (7) may then be equivalently formulated as the following finite Boolean expressions on linear constraints:

$$0 \leq x \wedge x \leq n \wedge n \leq N \quad \Rightarrow \quad (0 \leq \alpha x + \beta n + \gamma) \quad (8)$$

$$0 \leq x \wedge x \leq n \wedge n \leq N \quad \Rightarrow \quad (\alpha x + \beta n + \gamma \leq 1) \quad (9)$$

$$n < N \quad \Rightarrow \quad (0 \leq p\alpha + \beta)\ . \quad (10)$$

To translate (8), (9) and (10) into a set of existentially quantified constraints that can be used as inputs to a SAT-solver, we use Motzkin's Theorem. Condition (10) for instance, which holds if the strict linear inequalities

$$\begin{bmatrix} 0x + \ \ n + \qquad -N < 0 \\ 0x + 0n + p\alpha + \beta < 0 \end{bmatrix}$$

[8] Invariant $[n \leq N]$ *cannot* be generated since –although n only takes natural values in the context of the program– we are solving constraints over the reals, and not the natural numbers.

```
init :  $x := p$;  $b := true$;
loop :  while $b$ do
            $b := false\ {}_{1/2}\oplus\ true$;
            if $b$ then
                $x := 2x$; if $x \geq 1$ then $x := x-1$ else skip fi
            elseif $x \geq 1/2$ then $x := 1$
            else  $x := 0$  fi
        od
```

Variable x is of type $\mathbb{R}$ and b of type $\mathbb{B}$. This program sets x to 1 with probability at least p, a fact verified by establishing $wp.(init; loop).[x = 1] \geq p$, which follows from the discovery of the loop invariant $[0{\leq}x{\leq}1]{\times}x$.

Fig. 3. Generating a biased coin from a fair one.

are *not* satisfiable, is equivalent (by Motzkin's Theorem) to the following polynomial constraints:

$$\exists\lambda_0, \lambda_1, \lambda_2 \cdot$$

$$\begin{pmatrix} \lambda_0 \geq 0 \wedge \lambda_1 \geq 0 \wedge \lambda_2 \geq 0 \wedge \\ 0 = \lambda_1 0 + \lambda_2 0 \wedge \\ 0 = \lambda_1 + \lambda_2 0 \wedge \\ 1 = -\lambda_1 N + \lambda_2(p\alpha + \beta) - \lambda_0 \end{pmatrix} \vee \begin{pmatrix} \lambda_0 \geq 0 \wedge \lambda_1 \geq 0 \wedge \lambda_2 \geq 0 \wedge \\ (\lambda_1 \neq 0 \vee \lambda_2 \neq 0) \wedge \\ 0 = \lambda_1 0 + \lambda_2 0 \wedge \\ 0 = \lambda_1 + \lambda_2 0 \wedge \\ 0 = -\lambda_1 N + \lambda_2(p\alpha + \beta) - \lambda_0 \end{pmatrix}$$

Simplifying this constraint reveals that parameters α, β and γ must satisfy $p\alpha + \beta \geq 0$. This condition is satisfied, for example, if $\beta = -p\alpha$ and $\gamma = p\alpha$. Assuming that $\beta = -p\alpha$ and $\gamma = p\alpha$ holds, we have that (8) and (9) hold if N is positive and $0 < \alpha \leq 1/N$. Consequently, for positive N and $\alpha\colon (0, 1/N]$ we have that

$$J{\times}(\alpha x - p\alpha n + p\alpha N) \tag{11}$$

is invariant. Assuming N is positive, (11) can be used to calculate a lower bound on the expected value of x produced by the binomial program. We have

$$\begin{array}{lll} & wp.(init; loop).(\alpha x) & \\ \geq & wp.(init; loop).([n \geq N]{\times}J{\times}\alpha x) & \text{“}\alpha x \geq [n \geq N]{\times}J{\times}\alpha x\text{; monotonicity”} \\ = & wp.init.(wp.loop.([n \geq N] \times I)) & \text{“simplify; sequential composition”} \\ \geq & wp.init.I & \text{“}loop\text{ terminates and } I \text{ is invariant; monotonicity”} \\ = & [0 \leq N] \times p\alpha N & \text{“calculate”} \\ = & p\alpha N\ . & \text{“we have assumed that } N \text{ is positive”} \end{array}$$

From scaling (Sec. 3), a lower bound of the least expected value of x (i.e. $(1/\alpha) \times \alpha x$) that may be produced by the Binomial program is pN (i.e. $(1/\alpha) \times p\alpha N$).

6.2 Example two: generating a biased coin from a fair one

The program in Fig. 3 (which appears in [18, Ch4]) uses a stream of fair coin flips to generate a (single) biased coin. To verify that on termination it correctly sets x

```
$init_1$ : $n, g := 1, N$;
$loop_1$ : while $g \geq N$ do
$init_2$ :     $n, g := 1, 0$;
$loop_2$ :     while $n < N$ do $n := 2n$; $(g := 2g\ {}_{1/2}\oplus\ g := 2g + 1)$ od
         od
```

n and g are both variables of type $\mathbb{N}$ and N is a constant positive natural number. This program uniformly sets g to a value in $[0..N)$, verified by establishing $wp.init_1; loop_1.[g = k] \geq 1/N$, implied by the invariant (for the outer loop) $[g = k] + [g \geq N] \times (1/N)$.

Fig. 4. Uniform distribution.

to 1 with probability (at least) p, we need to determine that $p \leq wp.loop.[x = 1]$. We used our techniques to discover that $[0 \leq x \leq 1] \times x$ is an invariant of *loop*, and then additional reasoning to show that it implies correctness. First, it simplifies to the post-expectation on termination (because on exiting the loop x takes only the values 0 or 1); next substituting values for the initialisation $x := p$ yields the required lower bound.[9] Details of the generating constraints are set out in App. A.

6.3 Example three: uniform distribution; nested loops

Cryptographic applications often require a variable to be chosen uniformly from some interval $[0 \ldots N]$; in practice this must be achieved using a fair coin as above, and the program in Fig. 4 is an example. Intuitively its inner loop sets g uniformly to some interval $[0 \ldots c]$ where c is the smallest power of 2 exceeding N (i.e. $2^{\lceil \log_2 N \rceil}$); the function of the outer loop is to repeat the process from scratch until g lies in the required interval $[0 \ldots N]$.

To verify this program we use the above technique to generate automatically a linear invariant for the inner loop. We then use that invariant to reason *manually* about the effect of the outer loop. In the conclusion we suggest ways in which we might be able to extend our method so that this (manual) reasoning could be automated as well.

Verifying this program thus requires the combination of automated invariant generation and interactive proof, and in this section we sketch how it was done.

Interactive proof: First, we make an assumption that the inner loop correctly sets g uniformly within $[0 \ldots c]$; this is formalised by the set of (parametrised) Hoare triples

$$\{1/c\} \quad init_2; loop_2 \quad \{[g = k]\}\ , \quad 0 \leq k \leq c\ . \tag{12}$$

With this assumption we are able to use the *wp*-calculus directly to verify that $[g = k] + [g \geq N] \times d$ (for $d \leq 1/N$) is an invariant of the outer loop, $loop_1$, and that is sufficient to verify the whole program; the details are set out at App. B.

[9] It can be in fact be shown that this bound on the expected value of x is tight.

That leaves us with the problem of establishing (12) ; we use our automated invariant generation technique to find invariants to do so.

Automatic invariant generation for the inner-loop analysis: First we considered the special case of (12) where $k = 0$ and searched for $loop_2$ invariants of the form $I := [g = 0 \wedge J] \times (\alpha n + \gamma)$, where $J := 1 \leq n \leq c$, and we needed that $[J]$ is an invariant of $loop_2$. This gave us $[g = 0 \wedge 1 \leq n \leq c] \times n/c$ as an invariant, which is sufficient for the special case. To generalise this, we then searched for $loop_2$ invariants of the form $[\alpha n + \beta < g \leq \gamma n + \delta \wedge J] \times dn$, and we found that for any α and $dc = 1$, $[\alpha n - 1 < g \leq \alpha n \wedge J] \times dn$ is also invariant, from which we can derive our result. Details are set out in App. B.

7 Alternative automated methods

Markov decision processes (MDP's) are a natural candidate for an operational model for probabilistic programs. Analysis of quantitative properties relative to Markov decision processes are available via probabilistic model checking. Examples include PRISM [16] (supporting PCTL model checking) and LiQuor [3] (supporting LTL). Whilst the invariant technique produces general statements about program behaviour, model checking is restricted to the verification of particular instances: for the generation of a biased coin from a fair coin, it can be checked whether eventually the probability that x equals 1 is p, for a given p. One cannot check that for any p this property holds.

Recent developments using abstraction refinement increase the potential for generality. In particular PASS [15] and a SAT-based extension of PRISM [22] both compute sound approximations of the underlying MDP, with the former yielding over approximations, and the latter computing both upper- and lower bounds. In neither case does it appear that the methods could feasibly be used to treat the examples in this paper, however. In particular the analysis of loops by the extension of PRISM tends to be extremely costly, and do not perform well when the variables can take real values [21].

Testing for language equivalence between probabilistic programs over finite integer datatypes has been exploited by the tool APEX [24], but again this would not be able to treat the examples that use real-valued variables straightforwardly.

Abstract interpretation methods [6] have also been applied to probabilistic programs [8, 9]. As for non-probabilistic abstract interpretation methods, these might –in contrast to constraint-based methods– only produce "approximate answers".

Finally, none of PASS, SAT-based PRISM, APEX nor the probabilistic abstract interpretation methods generate quantitative loop invariants.

8 Aims and conclusions

We have defined a constraint-based method for generating propositionally linear annotations for linear probabilistic programs, and demonstrated it using a number of realistic (but small) probabilistic programs. We have primarily focused on generating invariants for loops. As for other constraint-based methods, the program-size and the size of the parametrised invariants is constrained to small-to-medium sized problem instances by the capabilities of current constraint-solving tools.

Once found, quantitative invariants can be used to prove very general properties of probabilistic programs. Practical experience in automating proofs in HOL [19, 2] has shown that some of the quantitative invariants crucial to proof are not at all obvious; the development of an automated assistant for invariant discovery to augment interactive proofs is one of the main motivations for this work. Our third example in Sec. 6.3 is typical of how invariant generation can enhance an interactive proof session. It also suggests that propositionally linear annotations are unlikely to be sufficient *in themselves* for proving all properties of interest: recall that we used a *set* of discovered linear annotations to approximate the inner loop behaviour. On the other hand, this suggests a method in which sets of annotation pairs could be used more generally to abstract from program behaviour. For us that implies the following hierarchical method: first linear invariants are discovered for inner loops, and then used to abstract the loops' behaviour as sets of annotations. The analysis of all the enclosing loop(s) can then proceed as outlined in this paper, but with the inner loops summarised by their sets of annotations. That is our next step.

Beyond that, we would also like to build tool support for our approach. This would involve, among other tasks, the mechanisation of weakest-precondition calculations involving propositionally linear expressions over probabilistic programs. Earlier mechanisations of the quantitative logic for pGCL (e.g. [19]) suggest that this task is feasible.

Finally, it would be interesting to consider whether other advances in constraint-based invariant generation methods, such as [29, 5, 20] could be adapted to generate polynomial forms of quantitative invariants.

Bibliography

[1] A. Bockmayr and V. Weispfenning. Solving numerical constraints. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 12, pages 751–842. Elsevier Science, 2001.

[2] O. Celiku. *Mechanized Reasoning for Dually-Nondeterministic and Probabilistic Programs.* PhD thesis, TUCS, 2006.

[3] F. Ciesinski and C. Baier. LiQuor: A tool for qualitative and quantitative linear time analysis of reactive systems. In *Quantitative Evaluation of Systems (QEST)*, pages 131–132. IEEE Computer Society Press, 2006.

[4] M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In W. A. Hunt Jr. and F. Somenzi, editors, *Computer Aided Verification (CAV)*, volume 2725 of *LNCS*, pages 420–432. Springer Verlag, 2003.

[5] P. Cousot. Proving program invariance and termination by parametric abstraction, Lagrangian relaxation and semidefinite programming. In R. Cousot, editor, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3385 of *LNCS*, pages 1–24. Springer, 2005.

[6] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (PoPL)*, pages 238–252. ACM, 1977.

[7] J. den Hartog and E.P. de Vink. Verifying probabilistic programs using a Hoare like logic. *Int. J. Found. Comput. Sci.*, 13(3):315–340, 2002.

[8] A. Di Pierro and H. Wiklicky. Concurrent constraint programming: towards probabilistic abstract interpretation. In M. Gabbrielli and F. Pfenning, editors, *Principles and Practice of Declarative Programming (PPDP)*, pages 127–138. ACM, 2000.

[9] A. Di Pierro and H. Wiklicky. Measuring the precision of abstract interpretations. In K. Lau, editor, *Logic Based Program Synthesis and Transformation (LOPSTR)*, volume 2042 of *LNCS*, pages 147–164. Springer Verlag, 2001.

[10] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[11] A. Dolzmann and T. Sturm. REDLOG: computer algebra meets computer logic. *SIGSAM Bull.*, 31(2):2–9, 1997.

[12] R.W. Floyd. Assigning meanings to programs. In J.T. Schwartz, editor, *Mathematical Aspects of Computer Science*, number 19 in Proc. Symp. Appl. Math., pages 19–32. American Mathematical Society, 1967.

[13] S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. *Programming Language Design and Implementation (PLDI)*, 43(6):281–292, 2008.

[14] M. Hazewinkel. *Encyclopedia of Mathematics.* Springer, 2002.

[15] H. Hermanns, B. Wachter, and L. Zhang. Probabilistic CEGAR. In A. Gupta and S. Malik, editors, *Computer-Aided Verification (CAV)*, volume 5123 of *LNCS*, pages 162–175. Springer Verlag, 2008.

[16] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In H. Hermanns and J. Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3920 of *LNCS*, pages 441–444. Springer, 2006.

[17] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–80, 1969.

[18] J. Hurd. *Formal Verification of Probabilistic Algorithms.* PhD thesis, University of Cambridge, 2002.

[19] J. Hurd, A.K. McIver, and C.C. Morgan. Probabilistic guarded commands mechanised in HOL. *Theoretical Computer Science*, 346(1):96–112, 2005.

[20] D. Kapur. Automatically generating loop invariants using quantifier elimination. In *Deduction and Applications*, 2005.

[21] M. Kattenbelt. Private communication. 2010.

[22] M. Kattenbelt, M. Kwiatkowska, G. Norman, and D. Parker. Abstraction refinement for probabilistic software. In N.D. Jones and M. Müller-Olm, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 5403 of *LNCS*, pages 182–197. Springer Verlag, 2009.

[23] D. Kozen. Semantics of probabilistic programs. *Jnl. Comp. Sys. Sciences*, 22:328–50, 1981.

[24] A. Legay, A.S. Murawski, J. Ouaknine, and J. Worrell. On automated verification of probabilistic programs. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS, pages 173–187. Springer, 2008.

[25] A.K. McIver and C.C. Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems.* Monographs in Computer Science. Springer Verlag, 2004.

[26] D. Monniaux. Abstract interpretation of probabilistic semantics. In *International Static Analysis Symposium (SAS)*, volume 1824 of *LNCS*, pages 322–339. Springer Verlag, 2000.

[27] C.C. Morgan. Proof rules for probabilistic loops. In He Jifeng, J.Cooke, and P. Wallis, editors, *BCS-FACS 7th Refinement Workshop*, Workshops in Computing. Springer Verlag, 1996.

[28] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In B. Steffen and G. Levi, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, LNCS, pages 239–251. Springer Verlag, 2004.

[29] S. Sankaranarayanan, H.B. Sipma, and Z. Manna. Non-linear loop invariant generation using Gröbner bases. In *Principles of Programming Languages (PoPL)*, pages 318–329. ACM, 2004.

A Constraints for Fig. 3

First, it is possible to generate *loop* invariants $[\gamma \leq x]$ and $[x \leq \gamma]$ where $\gamma = 0$ or $\gamma = 1$, which describe lower- and upper bounds on the value of x. Having identified bounds on x we fix $J := [0 \leq x \leq 1]$ and search for parametrised invariants for *loop* of the form $I := J \times (\alpha x + \beta n + \gamma)$. I is invariant of *loop* if

$$0 \leq x \leq 1 \quad \Rightarrow \quad (0 \leq \alpha x + \beta n + \gamma) \ , \tag{13}$$

$$0 \leq x \leq 1 \quad \Rightarrow \quad (\alpha x + \beta n + \gamma \leq 1) \ \text{ and} \tag{14}$$

$$b = 1 \quad \Rightarrow \quad (\alpha x + \beta n + \gamma) \leq \mathit{wp.body}.(\alpha x + \beta n + \gamma) \ . \tag{15}$$

We calculate that (15) holds if $\beta \leq 0$. Setting β and γ to zero and α to one trivially satisfies the bounding constraints (13) and (14). This gives us that $J \times x$ is invariant, as desired.

B Calculations for Fig. 4

Inner-loop analysis. The inner loop, loop_2 in Fig. 4, sets value g uniformly between values in the range $[0...2^{\lceil \log_2 N \rceil})$. To prove this we would like to find an invariant of loop_2 we could use to show that, for each $k\colon [0..2^{\lceil \log_2 N \rceil})$, we have (a lower bound on) the expected value of $g = k$ is $1/2^{\lceil \log_2 N \rceil}$.

By focusing on the special case where $k = 0$ we learn something about the programs behaviour that can guide our search for a more general invariant. Assuming that there exists a constant c such that $[J]$, where $J := 1 \leq n \leq c$, is an invariant of loop_2, we calculate that $I := [g = 0 \wedge J] \times (\alpha n + \gamma)$ is invariant provided $\gamma \leq 0 \vee N = 1$ and boundedness condition $0 \leq I \leq 1$ holds. For γ=0, boundedness is satisfied if $0 \leq \alpha \wedge \alpha c \leq 1$. Hence

$$[g = 0 \wedge 1 \leq n \leq c] \times n/c \tag{16}$$

is invariant. Using our constraint-solving method, we find that J is invariant if $c \geq 2N$, but this does not provide us on a tight lower bound for c. Creative user input must be used to find the exact bound, $1/2^{\lceil \log_2 N \rceil}$, which is non-linear in the program variables. Taking c=$1/2^{\lceil \log_2 N \rceil}$, and using the additional knowledge that $n = 2^{\lceil \log_2 N \rceil}$ when the loop terminates, invariant (16) can be used to show that the probability of reaching the state g=0 is at least $1/2^{\lceil \log_2 N \rceil}$.

For the general case we want to calculate the expected value of $g = k$, for some $k \neq 0$. By generating invariants of the form $[\alpha n{+}\beta < g \leq \gamma n{+}\delta {\wedge} J] {\times} dn$ –which is reminiscent of (16)– we find that for any α and $dc = 1$, $[\alpha n{-}1 < g \leq \alpha n {\wedge} J] \times dn$ is invariant. Since g may only take integer values this implies invariance of

$$[g = \lfloor \alpha n \rfloor \wedge J] \times dn \ . \tag{17}$$

For any $k\colon [0..2^{\lceil \log_2 N \rceil})$, taking $c = 2^{\lceil \log_2 N \rceil}$ and $\alpha = k/c$, invariant (17) can be used to calculate that $g = k$ is reached with probability $1/2^{\lceil \log_2 N \rceil}$, i.e. $\{1/2^{\lceil \log_2 N \rceil}\}\ \mathit{init}_2; \mathit{loop}_2\ \{[g = k]\}$.

Outer-loop analysis. The outer loop $loop_1$ sets g uniformly in the range $[0..N)$ by repeatedly initialising and executing $loop_2$ until g is assigned a value in that range. This algorithm works since –as we know from our earlier analysis– the probability that $loop_2$ assigns g to be a value in the range $[0..N)$ is uniform. This behaviour is captured by the set of linear invariants

$$[g = k] + [g \geq N] \times d \tag{18}$$

for $k\colon [0..N)$ and $dN = 1$, which says that once g has been assigned a value k, it stays there, and when g has not yet reached k and it has not been assigned another value in the range $[0..N)$, it retains a $d = \frac{1}{N}$ chance of reaching k. To verify invariance of (18) using our inner-loop analysis we reason (by hand):

$$
\begin{array}{ll}
 & wp.(init_2; loop_2).([g = k] + [g \geq N] \times d) \\
\geq & wp.(init_2; loop_2).([g = k] + d * \sum_{h\colon [N..2^{\lceil \log_2 N \rceil}]} [g = h]) \qquad \text{"monotonicity"} \\
= & \qquad \text{"additivity of } wp \text{ [25] for deterministic programs and scaling "} \\
 & wp.(init_2; loop_2).[g = k] + d * \sum_{h\colon [N..2^{\lceil \log_2 N \rceil}]} wp.(init_2; loop_2).[g = h] \\
\geq & \qquad \text{" for each } k,\ \{1/2^{\lceil \log_2 N \rceil}\}\ init_2; loop_2\ \{[g = k]\} \text{ "} \\
 & 1/2^{\lceil \log_2 N \rceil} + d * (2^{\lceil \log_2 N \rceil} - N) * 1/2^{\lceil \log_2 N \rceil} \\
= & d \qquad \text{"simplify using } dN = 1\text{"} \\
\geq & [g \geq N] \times ([g = k] + [g \geq N] \times d) \qquad \text{"} k < N \text{"}
\end{array}
$$

and so (18) is invariant.

C Proofs

C.1 Proof of Lem. 1:

Proof. We have that

$$
\begin{array}{ll}
 & Q \\
= & \qquad \text{"propositionally linear parametrisation"} \\
 & (\sum m\colon [1..M] \cdot [\bigwedge_{n\colon [1..N]} P_{mn}] \times Q_m) \\
= & (\sum X\colon \mathbb{P}[1..M] \cdot \qquad \text{"rewrite as a summation with disjoint predicates"} \\
 & \quad [(\bigwedge_{x\colon X; n\colon [1..N]} P_{xn}) \wedge (\bigwedge_{x\colon [1..M]-X} \neg(\bigwedge_{n\colon [1..N]} P_{xn}))] \times (\sum_{x\colon X} Q_x)) \\
= & (\sum X\colon \mathbb{P}[1..M] \cdot \qquad \text{"negate conjunction"} \\
 & \quad [(\bigwedge_{x\colon X; n\colon [1..N]} P_{xn}) \wedge (\bigwedge_{x\colon [1..M]-X} (\bigvee_{n\colon [1..N]} \neg P_{xn}))] \times (\sum_{x\colon X} Q_x)) \\
= & \text{" for each permutation } X\text{, convert the predicate to the disjunctive normal form} \\
 & \qquad \text{where each } Q_{Xij} \text{ is a linear constraint } P_{mn} \text{ or } \neg P_{mn} \text{ for some } m,\ n. \text{ "}
\end{array}
$$

$$(\textstyle\sum X\colon \mathbb{P}[1..M] \cdot [(\bigvee_{i\colon I_X}(\bigwedge_{j\colon J_X} Q_{Xij}))] \times (\sum_{x\colon X} Q_x))$$

$=$ "rewrite disjunction as summation"

$$(\textstyle\sum X\colon \mathbb{P}[1..M]; i\colon I_X \cdot [(\bigwedge_{j\colon J_X} Q_{Xij})] \times (\sum_{x\colon X} Q_x))$$

which is a propositionally linear expression in disjoint normal form, since each Q_{Xij} is a linear constraint (or representable as one since the negation of a linear constraint is also linear) and the summation of linear expressions $(\sum_{x\colon X} Q_x)$ is also a linear expression. Any differences betwen the number of conjunctions in each predicate can be overcome by padding with extra conjunctions of the predicate *true*.

C.2 Proof of Lem. 2

Proof. Let S and T be the propositionally linear expressions

$$[P_{S1}]\times Q_{S1} + \cdots + [P_{SM}]\times Q_{SM} \quad \textit{and} \quad [P_{T1}]\times Q_{T1} + \cdots + [P_{TK}]\times Q_{TK} \ ,$$

where each P_{Sm}, P_{Tk} are linear assertions and Q_{Sm}, Q_{Tk} are linear expressions such that $Q_{Sm} = \gamma_{(Sm,1)}x_1 + \ldots + \gamma_{(Sm,X)}x_X + \gamma_{(Sm)}$. If G is a linear constraint and p is a constant we then have that:

1. $[G]\times S$ is semantically equivalent to the propositionally linear expression
$$[P_{S1} \wedge G]\times Q_{S1} + \cdots + [P_{SM} \wedge G]\times Q_{SM} \ .$$
2. Without loss of generality (Lem. 1) assume that S and T are in disjoint normal form. Expression $S \sqcap T$ is then semantically equivalent to
$$\textstyle\sum_{m\colon [1..M]; k\colon [1..K]} [P_{Sm} \wedge P_{Tk} \wedge (Q_{Sm} - Q_{Tk} \leq 0)] \times Q_{Sm} \\ + [P_{Sm} \wedge P_{Tk} \wedge (Q_{Tk} - Q_{Sm} < 0)] \times Q_{Tk} \ ,$$
which is propositionally linear.
3. $p * S$ is semantically equivalent to the propositionally linear expression
$$\sum_{m\colon [1..M]} [P_{Sm}]\times(p * \gamma_{(Sm,1)}x_1 + \ldots + p * \gamma_{(Sm,X)}x_X + p * \gamma_{(Sm)}) \ .$$

It is trivially true that $\neg G$ may be expressed as a linear constraint; and that $S + T$ and $S[x\backslash E]$ are propositionally linear if E is a linear expression.