Relatively Complete Verification of Probabilistic Programs

An Expressive Language for Expectation-Based Reasoning

KEVIN BATZ*, RWTH Aachen University, Germany
BENJAMIN LUCIEN KAMINSKI, University College London, United Kingdom
JOOST-PIETER KATOEN*, RWTH Aachen University, Germany
CHRISTOPH MATHEJA, ETH Zürich, Switzerland

We study *a syntax for specifying quantitative "assertions"*—functions mapping program states to numbers—for probabilistic program verification. We prove that our syntax is expressive in the following sense: Given any probabilistic program C, if a function f is expressible in our syntax, then the function mapping each initial state σ to the expected value of f evaluated in the final states reached after termination of C on σ (also called the weakest preexpectation $\mathrm{wp}[\![C]\!](f)$) is also expressible in our syntax.

As a consequence, we obtain a *relatively complete verification system* for reasoning about expected values and probabilities in the sense of Cook: Apart from proving a single inequality between two functions given by syntactic expressions in our language, given f, g, and C, we can check whether $g \preceq \mathrm{wp}[\![C]\!](f)$.

CCS Concepts: • **Theory of computation → Probabilistic computation**; **Logic and verification**; *Hoare logic*; **Programming logic**; *Denotational semantics*; **Program verification**; *Program specifications*; *Pre- and post-conditions*; *Assertions*; Automated reasoning.

Additional Key Words and Phrases: probabilistic programs, randomized algorithms, formal verification, quantitative verification, completeness, weakest precondition, weakest preexpectation

ACM Reference Format:
Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2021. Relatively Complete Verification of Probabilistic Programs: An Expressive Language for Expectation-Based Reasoning. *Proc. ACM Program. Lang.* 5, POPL, Article 39 (January 2021), 30 pages. https://doi.org/10.1145/3434320

1 INTRODUCTION

Probabilistic programs are ordinary programs whose execution may depend on the outcome of random experiments, such as sampling from primitive probability distributions or branching on the outcome of a coin flip. Consequently, running a probabilistic program (repeatedly) on a single input generally gives not a single output but a *probability distribution* over outputs.

Introducing randomization into computations is an important tool for the design and analysis of *efficient algorithms* [Motwani and Raghavan 1999]. However, increasing efficiency by randomization often comes at the price of introducing a non-zero probability of producing incorrect outputs. Furthermore, even though a program may be efficient *in expectation*, individual executions may exhibit a long—even infinite—run time [Bournez and Garnier 2005; Kaminski et al. 2018].

*Batz and Katoen are supported by the ERC AdG 787914 FRAPPANT.

Authors' addresses: Kevin Batz, RWTH Aachen University, Germany, kevin.batz@cs.rwth-aachen.de; Benjamin Lucien Kaminski, University College London, United Kingdom, b.kaminski@ucl.ac.uk; Joost-Pieter Katoen, RWTH Aachen University, Germany, katoen@cs.rwth-aachen.de; Christoph Matheja, ETH Zürich, Switzerland, cmatheja@inf.ethz.ch.

2475-1421/2021/1-ART39
https://doi.org/10.1145/3434320

Reasoning about probabilistic phenomena is hard. For instance, deciding termination for probabilistic programs is strictly harder than for ordinary programs [Kaminski and Katoen 2015; Kaminski et al. 2019]. Nonetheless, probabilistic program verification is an active research area: After seminal work on semantics by Kozen [1979, 1981], many different techniques have been developed, see [Hart et al. 1982] for an early example. Modern approaches include martingale-based techniques [Chakarov and Sankaranarayanan 2013; Chatterjee et al. 2016a,b, 2017; Fu and Chatterjee 2019; Huang et al. 2018] and weakest-precondition-style calculi [Batz et al. 2019; Kaminski 2019; Kaminski et al. 2018; McIver and Morgan 2005; Ngo et al. 2018]. The former can be phrased in terms of the latter, and all aforementioned techniques can be understood as instances or extensions of Kozen's probabilistic propositional dynamic logic (PPDL) [Kozen 1983, 1985].

Probabilistic program verification, extensionally. There are two perspectives for reasoning about programs: the *extensional* and the *intensional*. Whereas intensional approaches provide a syntax, i.e., a formal language, for assertions, extensional approaches admit arbitrary assertions and dispense with considerations about syntax altogether—they treat assertions as purely mathematical entities.

A standard technique for probabilistic program verification that takes the extensional approach is the *weakest preexpectation* (wp) *calculus* of McIver and Morgan [2005]—itself an instance of Kozen's PPDL [Kozen 1983, 1985]. Given a probabilistic program C and *some function* f (called the *postexpectation*), mapping (final) states to numbers, the weakest preexpectation $\mathsf{wp}[\![C]\!](f)$ is a mapping from (initial) states to numbers, such that

$$\mathsf{wp}[\![C]\!](f)(\sigma) \;=\; \begin{array}{l}\text{Expected value of } f\text{, measured in final states reached}\\ \text{after termination of } C \text{ on initial state } \sigma\ .\end{array}$$

For probabilistic programs with *discrete* probabilistic choices, the wp calculus can be defined for *arbitrary* real-valued postexpectations f [Kaminski 2019; McIver and Morgan 2005].

Probabilistic program verification, intensionally. While the extensional approach often yields elegant formalisms, it is unsuitable for developing practical verification tools, which ultimately rely on some syntax for assertions. In particular, we cannot—in general—rely on the property, implicitly assumed in the extensional approach, that there is no distinction between assertions representing the same mathematical entity: a tool may not realize that $4 \cdot 0.5$ and $\sum_{i=0}^{\infty} 1/2^i$ represent the same mathematical entity (the number 2).

An example of intensional probabilistic program verification is the verifier of Ngo et al. [2018] which specifies a simple syntax which is extensible by user-specified base and rewrite functions.

Main contribution. Given a calculus for program verification and an assertion language, two fundamental questions immediately arise:

(1) *Soundness: Are only true assertions derivable* in the calculus?
(2) *Completeness*: *Can every true assertion be derived* and *is it expressible* in the assertion language?

While soundness is typically a *must* for any verification system, completeness is—as noted by Apt and Olderog [2019] in their recent survey of 50 years of Hoare logic—a "subtle matter and requires careful analysis". In fact, to the best of our knowledge, existing probabilistic program verification techniques (including all of the above references amongst many other works) either take the extensional approach or do not aim for completeness. In this paper, we take the intensional path and make the following contribution to formal reasoning about probabilistic programs:

> We provide a simple formal *language of functions* for probabilistic program verification such that:
> If f is syntactically expressible, then $\mathsf{wp}[\![C]\!](f)$ is syntactically expressible.

A language from which we can draw functions f with the above property is called *expressive*. Having an expressive language renders the wp calculus *relatively* complete [Cook 1978]: Given functions f and g in our language and a probabilistic program C, suppose we want to verify $g \preceq \mathrm{wp}[\![C]\!](f)$, where $\preceq$ denotes the point-wise order of functions mapping states to numbers. Due to expressiveness, we can effectively construct in our language a function h representing $\mathrm{wp}[\![C]\!](f)$. Hence, verification is complete *modulo* checking whether the inequality $g \preceq h$ between two functions in our language holds. Indeed, Hoare logic is also only complete *modulo* deciding an implication between two formulae in the language of first-order arithmetic [Apt and Olderog 2019].

Challenges and usefulness. Notice that providing *some* expressive language is rather easy: A singleton language that can only represent the null-function is trivially expressive since, for any program C, the expected value of 0 is 0. That is, $\mathrm{wp}[\![C]\!](0) = 0$. The challenge in a quest for an expressive language for probabilistic program verification is hence to find a language that (i) is closed under taking weakest preexpectations and (ii) can express *interesting (quantitative) properties*.

Indeed, our language can: For instance, it is capable of expressing *termination probabilities* (via $\mathrm{wp}[\![C]\!](1)$—the expected value of the constant function 1). These can be *irrational numbers* like the reciprocal of the golden ratio $1/\varphi$ [Olmedo et al. 2016]. In general, termination probabilities carry a high internal degree of complexity [Kaminski et al. 2019]. Our language can also express probabilities over program variables on termination of a program and that can be expressed in terms of π, $\sqrt{3}$ and so forth. These can e.g., be generated by Buffon machines, i.e., probabilistic programs that only use Bernoulli experiments [Flajolet et al. 2011].

Termination probabilities already hint at one of the technical challenges we face: Even starting from a constant function like 1, our language needs to express mappings from states to highly complex real numbers. Another challenge we face is that when constructing $\mathrm{wp}[\![C]\!](f)$, due to probabilistic branching in combination with loops, considering single execution traces is not enough: We have to collect all terminating traces and average over the values of f in terminal states. We attack these challenges via Gödel numbers for rational sequences and encodings of Dedekind cuts.

Aside from termination probabilities, our language is capable of expressing *a wide range of practically relevant functions*, like *polynomials* or *Harmonic numbers*. Polynomials are a common subclass of ranking functions[1] for automated probabilistic termination analysis; harmonic numbers are ubiquitous in expected runtime analysis. We present more scenarios covered by our syntax and avenues for future work in Sections 12 and 13.

Overall, we believe that an expressive *syntax* for probabilistic program verification is what really expedites a search for tractable fragments of both programs and "assertion" language in the first place. Studying such fragments may also yield additional insights: For example, Kozen [2000] and Kozen and Tiuryn [2001] studied the propositional fragment of Hoare logic and showed that it is subsumed by an extension of KAT—Kleene algebra with tests.

Further related work. Relative completeness of Hoare logic was shown by Cook [1978]. Winskel [1993] and Loeckx et al. [1984] proved expressiveness of first-order arithmetic for Dijkstra's weakest precondition calculus. For *separation logic* [Reynolds 2002]—a very successful logic for compositional reasoning about *pointer programs*—expressiveness was shown by Tatsuta et al. [2009, 2019], almost a decade later than the logic was originally developed and started to be used.

Perhaps most directly related to this paper is the work by den Hartog and de Vink [2002] on a Hoare-like logic for verifying probabilistic programs. They prove relative completeness (also in the sense of Cook [1978]) of their logic for *loop-free* probabilistic programs and *restricted postconditions*;

[1] In probabilistic program analysis terminology: ranking supermartingales.

they leave expressiveness for loops as an open problem: "It is not clear whether the probabilistic predicates are sufficiently expressive [...] for a given while loop."

Organization of the paper. We give an introduction to *syntax, extensional semantics, and verification systems* for probabilistic programs, in particular the weakest preexpectation calculus, in Section 2. We formulate the *expressiveness problem* in Section 3. We *define the syntax and semantics* of our *expressive language of expectations* in Section 4. We *prove expressiveness of our language for loop-free probabilistic programs* in Section 5. We then move to proving expressiveness of our language for loops. We *outline the expressiveness proof for loops* in Section 6 and *do the full technical proof* throughout Sections 7 – 10. In Section 11 and Section 12, we discuss extensions and a few scenarios in which our language could be useful; we conclude and discuss *open problems* in Section 13.

Detailed proofs of all theorems are found in an extended version of this paper, which is available online [Batz et al. 2020].

2 PROBABILISTIC PROGRAMS — THE EXTENSIONAL PERSPECTIVE

We briefly recap classical reasoning about probabilistic programs á la Kozen [1985], which is agnostic of any particular syntax for expressions or formulae—it takes an *extensional* approach.

2.1 The Probabilistic Guarded Command Language

We consider the imperative probabilistic programming language pGCL featuring discrete probabilistic choices—branching on outcomes of coin flips—as well as standard control-flow instructions.

2.1.1 Syntax. Formally, a program C in pGCL adheres to the grammar

$$
\begin{array}{lcll}
C & \longrightarrow & \texttt{skip} & \text{(effectless program)} \\
& & \mid\ x := a & \text{(assignment)} \\
& & \mid\ C\,;\,C & \text{(sequential composition)} \\
& & \mid\ \{\,C\,\}\,[\,p\,]\,\{\,C\,\} & \text{(probabilistic choice)} \\
& & \mid\ \texttt{if}\,(\varphi)\,\{\,C\,\}\,\texttt{else}\,\{\,C\,\} & \text{(conditional choice)} \\
& & \mid\ \texttt{while}\,(\varphi)\,\{\,C\,\}\,, & \text{(while loop)}
\end{array}
$$

where x is taken from a countably infinite set of *variables* Vars, a is an *arithmetic expression* over variables, $p \in [0,1] \cap \mathbb{Q}$ is a rational probability, and φ is a Boolean expression (also called *guard*) over variables. For an overview of metavariables C, x, a, φ, ..., used throughout this paper, see Table 1 at the end of this section.

For the moment, we assume that both arithmetic and Boolean expressions are standard expressions without bothering to provide them with a concrete syntax. However, we will require them to adhere to a concrete syntax which we provide in Sections 4.1 and 4.2.

2.1.2 Program States. A program state σ maps each variable in Vars to its value—a positive rational number in $\mathbb{Q}_{\geq 0}$.[2] To ensure that the set of program states is countable,[3] we restrict ourselves to states in which at most finitely many variables—intuitively those that appear in a given program—are assigned non-zero values; every state can thus be understood as a finite mapping that only keeps track of assignments to non-zero values. Formally, the set Σ of program states is

$$\Sigma \;=\; \left\{\, \sigma \colon \mathsf{Vars} \to \mathbb{Q}_{\geq 0} \;\middle|\; \{\, x \in \mathsf{Vars} \mid \sigma(x) \neq 0 \,\} \text{ is finite} \right\}.$$

[2]To keep the presentation simple, we consider only *unsigned* variables; we discuss this design choice and an extension to signed variables, which can also evaluate to negative rationals, in Section 11.

[3]Working with probabilistic programs over a countable set of states avoids technical issues related to measurability.

We use metavariables $\sigma, \tau, \ldots$, for program states, see also Table 1. We denote by ${}^{\sigma}[\![e]\!]$ the evaluation of (arithmetic or Boolean) expression e in σ, i.e., the value obtained from evaluating e after replacing every variable x in e by $\sigma(x)$. We define the semantics of expressions more formally in Section 4.4.

2.1.3 Forward Semantics. One of the earliest ways to give semantics to a probabilistic program C is by means of *forward-moving measure transformers* [Kozen 1979, 1981]. These transform an initial state σ into a probability distribution μ_C^σ over final states (i.e., a measure on Σ). We consider Kozen's semantics the *reference* forward semantics. More operational semantics are provided in the form of probabilistic transition systems [Gretz et al. 2014; Kaminski 2019], where programs describe potentially infinite Markov chains whose state spaces comprise of program states, or trace semantics [Cousot and Monerau 2012; Di Pierro and Wiklicky 2016; Kaminski et al. 2019], where the traces are sequences of program states and each trace is assigned a certain probability.

In any of these semantics, the probabilistic choice $\{\,C_1\,\}\ [\,p\,]\ \{\,C_2\,\}$ flips a coin with bias p towards heads. If the coin yields heads, C_1 is executed (with probability p); otherwise, C_2. Moreover, `skip` does nothing. x `:=` a assigns the value of expression a (evaluated in the current program state) to x. The sequential composition $C_1\,;\,C_2$ first executes C_1 and then C_2. The conditional choice `if` $(\,\varphi\,)\ \{\,C_1\,\}$ `else` $\{\,C_2\,\}$ executes C_1 if the guard φ is satisfied; otherwise, it executes C_2. Finally, the loop `while` $(\,\varphi\,)\,\{\,C\,\}$ keeps executing the loop body C as long as φ evaluates to true.

2.2 Weakest Preexpectations

Dually to the forward semantics, probabilistic programs can also be provided with semantics in the form of *backward-moving random variable transformers*, originally due to Kozen [1983, 1985]. This paper is set within this dual view, which is a standard setting for probabilistic program verification.

2.2.1 Expectations. *Floyd-Hoare logic* [Floyd 1967; Hoare 1969] as well as the *weakest precondition calculus* of Dijkstra [1976] employ first-order predicates for reasoning about program correctness. For probabilistic programs, Kozen [1983, 1985] was the first to generalize from predicates to measurable functions (or random variables). Later, McIver and Morgan [2005] coined the term *expectation*—not to be confused with expected value—for such functions. In reference to Dijkstra's weakest precondition calculus, their verification system is called the *weakest preexpectation calculus*.

Formally, the set $\mathbb{E}$ of *semantic expectations* is defined as

$$\mathbb{E} \;=\; \left\{\, X \;\middle|\; X\colon \Sigma \to \mathbb{R}^{\infty}_{\geq 0} \,\right\}\,,$$

i.e., functions X that associate a non-negative *quantity* (or infinity) to each program state. We use metavariables X, Y, Z for semantic expectations.

Expectations form the *assertion "language"* of the weakest preexpectation calculus. However, we note that—so far—expectations are *in no way defined syntactically*: They are just the whole set of functions from Σ to $\mathbb{R}^{\infty}_{\geq 0}$. It is hence borderline to speak of a *language*. The goal of this paper is to provide a *syntactically defined subclass of* $\mathbb{E}$—i.e., an *actual language*—such that formal reasoning about probabilistic programs can take place completely within this class.

We furthermore note that we work with more general expectations than McIver and Morgan [2005], who only allow *bounded* expectations, i.e., expectations X for which there is a bound $\alpha \in \mathbb{R}_{\geq 0}$ such that $\forall \sigma\colon X(\sigma) \leq \alpha$. In contrast to McIver and Morgan, our structure $(\mathbb{E}, \preceq)$ of *unbounded* expectations forms a *complete lattice* with least element 0 and greatest element ∞, where $\preceq$ lifts the standard ordering $\leq$ on the (extended) reals to expectations by pointwise application. That is,

$$X \;\preceq\; Y \qquad \text{iff} \qquad \forall \sigma \in \Sigma\colon \quad X(\sigma) \;\leq\; Y(\sigma)\,.$$

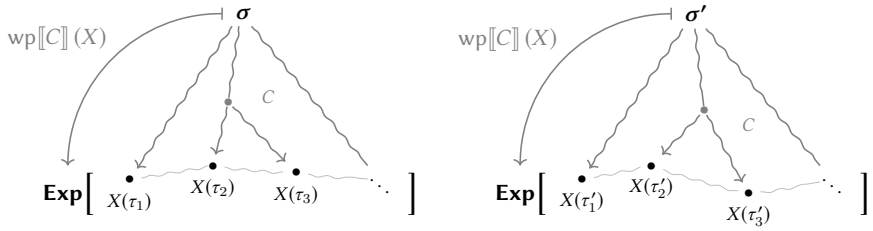

Fig. 1. The weakest preexpectation $\mathsf{wp}[\![C]\!]\,(X)$ maps every initial state σ to the expected value of X, measured with respect to the final distribution over states reached after termination of program C on input σ. $\mathsf{wp}[\![C]\!]$ is backward-moving in the sense that it transforms an $X\colon \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$, evaluated in final states after termination of C, into $\mathsf{wp}[\![C]\!]\,(X) : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$, evaluated in initial states before execution of C.

Examples of (bounded) expectations include, for instance, Iverson [1962] brackets $[\varphi]$, which associate to a Boolean expression φ its indicator function:[4]

$$[\varphi] \;=\; \lambda\sigma\mathbf{.}\; \begin{cases} 1, & \text{if } {}^{\sigma}[\![\varphi]\!] = \mathsf{true} \\ 0, & \text{if } {}^{\sigma}[\![\varphi]\!] = \mathsf{false}\,. \end{cases}$$

Iverson brackets embed Boolean predicates into the set of expectations, rendering McIver and Morgan's calculus a conservative extension of Dijkstra's calculus.

Examples of *unbounded* expectations are arithmetic expressions over variables, like

$$x + y \;=\; \lambda\sigma\mathbf{.}\; \sigma(x) + \sigma(y)\,,$$

where we point-wise lifted common operators on the reals, such as +, to operators on expectations. Strictly speaking, *McIver and Morgan's calculus cannot handle expectations like $x + y$* off-the-shelf.

We denote by $X\,[x/a]$ the "substitution" of variable x by expression a in expectation X, i.e.,

$$X\,[x/a] \;=\; \lambda\sigma\mathbf{.}\; X\Big(\sigma\left[x \mapsto {}^{\sigma}[\![a]\!]\right]\Big)\,, \qquad \text{where} \qquad \sigma\,[x \mapsto r] \;=\; \lambda y\mathbf{.}\; \begin{cases} r, & \text{if } y = x, \\ \sigma(y), & \text{else.} \end{cases}$$

2.2.2 Backward Semantics: The Weakest Preexpectation Calculus. Suppose we are interested in the expected value of the quantity (expectation) X after termination of C. In analogy to Dijkstra, X is called the *postexpectation* and the sough-after expected value is called the *weakest preexpectation* of C with respect to *postexpectation* X, denoted $\mathsf{wp}[\![C]\!]\,(X)$ [McIver and Morgan 2005]. As the expected value of X generally depends on the initial state σ on which C is executed, the *weakest preexpectation* $\mathsf{wp}[\![C]\!]\,(X)$ is itself also a map of type $\mathbb{E}$, mapping an initial program state σ to the *expected value* of X (measured in the final states) after successful termination of C on σ, see Figure 1. The weakest preexpectation calculus is a backward semantics in the sense that it transforms a postexpectation $X \in \mathbb{E}$, evaluated in final states after termination of C, into a preexpectation $\mathsf{wp}[\![C]\!]\,(X) \in \mathbb{E}$, evaluated in initial states before execution of C.

Between forward-moving measure transformers and backward-moving expectation transformers, there exists the following duality established by Kozen:

Theorem 2.1 (Kozen Duality [1983; 1985]). *If μ_C^{σ} is the distribution over final states obtained by running C on initial state σ, then for any postexpectation X,*

$$\sum_{\tau \in \Sigma} \mu_C^{\sigma}(\tau) \cdot X(\tau) \;=\; \mathsf{wp}[\![C]\!]\,(X)\,(\sigma)\,.$$

[4]We use λ-expressions to denote functions; function $\lambda x\mathbf{.}\; f$ applied to a evaluates to f in which x is replaced by a.

C	$\mathbf{wp}\,[\![C]\!]\,(X)$
`skip`	X
$x := a$	$X\,[x/a]$
$C_1\,;\,C_2$	$\mathsf{wp}[\![C_1]\!]\,(\mathsf{wp}[\![C_2]\!]\,(X))$
$\{\,C_1\,\}\,[\,p\,]\,\{\,C_2\,\}$	$p\cdot\mathsf{wp}[\![C_1]\!]\,(X)+(1-p)\cdot\mathsf{wp}[\![C_2]\!]\,(X)$
`if` $(\,\varphi\,)\,\{\,C_1\,\}$ `else` $\{\,C_2\,\}$	$[\varphi]\cdot\mathsf{wp}[\![C_1]\!]\,(X)+[\neg\varphi]\cdot\mathsf{wp}[\![C_2]\!]\,(X)$
`while` $(\,\varphi\,)\,\{\,C'\,\}$	$\mathsf{lfp}\;Y.\;[\neg\varphi]\cdot X+[\varphi]\cdot\mathsf{wp}[\![C']\!]\,(Y)$

Fig. 2. Rules defining the weakest preexpectation of program C with respect to postexpectation X.

In particular, if $X=[\varphi]$, then $\mathsf{wp}[\![C]\!]\,(X)\,(\sigma)$ is the *probability* that running C on σ terminates in a final state satisfying φ—thus generalizing Dijkstra's weakest preconditions.

As with standard weakest preconditions, weakest preexpectations are not determined monolithically for the whole program C as characterized above. Rather, they are determined *compositionally* using a backward-moving *expectation transformer*

$$\mathsf{wp}\colon \mathsf{pGCL}\to(\mathbb{E}\to\mathbb{E})$$

which is defined recursively on the structure of C according to the rules in Figure 2. Most of these rules are standard: $\mathsf{wp}[\![\mathtt{skip}]\!]$ is the identity as `skip` does not modify the program state. For the assignment $x := a$, $\mathsf{wp}[\![x := a]\!]\,(X)$ substitutes in X the assignment's left-hand side x by its right-hand side a. For sequential composition, $\mathsf{wp}[\![C_1\,;\,C_2]\!]\,(X)$ first determines the weakest preexpectation $\mathsf{wp}[\![C_2]\!]\,(X)$ which is then fed into $\mathsf{wp}[\![C_1]\!]$ as a postexpectation. For both the probabilistic choice $\{\,C_1\,\}\,[\,p\,]\,\{\,C_2\,\}$ and the conditional choice `if` $(\,\varphi\,)\,\{\,C_1\,\}$ `else` $\{\,C_2\,\}$, the weakest preexpectation with respect to X yields a convex sum $p\cdot\mathsf{wp}[\![C_1]\!]\,(X)+(1-p)\cdot\mathsf{wp}[\![C_2]\!]\,(X)$. In the former case, the weights are given by the probability p. In the latter case, they are determined by the guard φ, i.e., we have $p=[\varphi]$ and $1-[\varphi]=[\neg\varphi]$.

The weakest preexpectation of a loop is given by the least fixed point of its unrollings, i.e.,

$$\mathsf{wp}[\![\mathtt{while}\,(\,\varphi\,)\,\{\,C'\,\}]\!]\,(X)\;=\;\mathsf{lfp}\;Y.\,\Phi_X(Y)\,,$$

where the *characteristic function* Φ_X of $\mathtt{while}\,(\,\varphi\,)\,\{\,C'\,\}$ with respect to $X\in\mathbb{E}$ is defined as

$$\Phi_X\colon\quad \mathbb{E}\to\mathbb{E},\quad Y\;\mapsto\;[\neg\varphi]\cdot X+[\varphi]\cdot\mathsf{wp}[\![C']\!]\,(Y)\;.$$

Since $(\mathbb{E},\leq)$ is a complete lattice and Φ_X is monotone, fixed points exist due to the Knaster-Tarski fixed point theorem; we take the least fixed point because we reason about total correctness.

Throughout this paper, we exploit that Φ_X is, in fact, Scott-continuous (cf. [Olmedo et al. 2016]). Kleene's theorem then allows us to approximate the least fixed point iteratively:

Lemma 2.2 (Kleene et al. [1952]). *We have*

$$\mathsf{wp}[\![\mathtt{while}\,(\,\varphi\,)\,\{\,C'\,\}]\!]\,(X)\;=\;\mathsf{lfp}\;Y.\,\Phi_X(Y)\;=\;\sup_{n\in\mathbb{N}}\Phi_X^n(0)\,,$$

where $0=\lambda\sigma.\,0$ *is the constant-zero expectation and* $\Phi_X^n(Y)$ *denotes the n-fold application of* Φ_X *to* Y.

Table 1. Metavariables used throughout this paper.

Entities	Metavariables	Domain	Defined
Natural numbers	n, i, j, k	$\mathbb{N}$	
Positive rationals	r, s, t	$\mathbb{Q}_{\geq 0}$	
Positive extended reals	α, β, γ	$\mathbb{R}^{\infty}_{\geq 0}$	
Rational probabilities	p, q	$[0, 1] \cap \mathbb{Q}$	
Variables	x, y, z, v, w, u, num	Vars	Section 2.1
Arithmetic expressions	a, b	AExpr	Section 4.1
Boolean expressions	φ, ψ, ξ	Bool	Section 4.2
Syntactic expectations	f, g, h	Exp	Section 4.3
Semantic expectations	X, Y, Z	$\mathbb{E}$	Section 2.2.1
Programs	C	pGCL	Section 2.1
Program states	σ, τ	Σ	Section 2.1.2

3 TOWARDS AN EXPRESSIVE LANGUAGE FOR EXPECTATIONS

As long as we take the extensional approach to program verification, i.e., we admit all expectations in $\mathbb{E}$, reasoning about expected values of pGCL programs is *complete*: For every program C and postexpectation X, it is, in principle, possible to find an expectation $\mathsf{wp}[\![C]\!](X) \in \mathbb{E}$ which—by the above soundness property—coincides with the expected value of X after termination of C.

The main goal of this paper is to enable (relatively) complete verification of probabilistic programs by taking an *intensional* approach. That is, we use the same verification technique described in Section 2 (i.e., the weakest preexpectation calculus) but

fix a set Exp of syntactic expectations f.

We use metavariables $f, g, h, \ldots$, for syntactic expectations, as opposed to $X, Y, Z, \ldots$, for semantic expectations in $\mathbb{E}$, see also Table 1. While f itself is merely a syntactic entity to begin with, we denote by $[\![f]\!]$ the corresponding semantic expectation in $\mathbb{E}$. Having a syntactic set of expectations at hand immediately raises the question of *expressiveness*:

For f expressible in Exp, is the weakest preexpectation $\mathsf{wp}[\![C]\!]([\![f]\!])$ again expressible in Exp?

Definition 3.1 (Expressiveness of Expectations). The set Exp of syntactic expectations is *expressive* iff for all programs C and all $f \in \mathsf{Exp}$ there exists a syntactic expectation $g \in \mathsf{Exp}$, such that

$$\mathsf{wp}[\![C]\!]([\![f]\!]) = [\![g]\!] \ . \qquad \triangle$$

Notice that constructing *some* expressive set of syntactic expectations is straightforward. For example, the set $\mathsf{Exp} = \{0\}$, which consists of a single expectation 0—interpreted as the constant expectation $[\![0]\!] = \lambda\sigma.\ 0$—is expressive: $\mathsf{wp}[\![C]\!]([\![0]\!]) = [\![0]\!]$ holds for every C by strictness of wp.[5]

The main challenge is thus to find a syntactic set Exp that (i) can be proven expressive *and* (ii) covers interesting properties—at the very least, it should cover all Boolean expressions φ (to reason about probabilities) and all arithmetic expressions a (to reason about expected values).

[5] wp being strict means that $\mathsf{wp}[\![C]\!](0) = 0$ for every C, see [Kaminski 2019].

4 SYNTACTIC EXPECTATIONS

We now describe the syntax and semantics for a set Exp of syntactic expectations which we will (in the subsequent sections) prove to be expressive and which can be used to express interesting properties such as, amongst others, the expected value of a variable x, the probability to terminate, the probability to terminate in a set described by a first-order arithmetic predicate φ, etc.

4.1 Syntax of Arithmetical Expressions

We first describe a *syntax for arithmetic expressions*, which form *precisely the right-hand-sides of assignments that we allow in* pGCL *programs*. Naturally, the syntax of arithmetical expressions will reoccur in our syntax of expectations. Formally, the set AExpr of arithmetic expressions is given by

$$\begin{array}{rcll} a & \longrightarrow & r \in \mathbb{Q}_{\geq 0} & \text{(non-negative rationals)} \\ & \mid & x \in \mathsf{Vars} & (\mathbb{Q}_{\geq 0}\text{-valued variables}) \\ & \mid & a + a & \text{(addition)} \\ & \mid & a \cdot a\,, & \text{(multiplication)} \\ & \mid & a \dot{-} a\,, & \text{(subtraction truncated at 0 (“monus”))} \end{array}$$

where Vars is a *countable* set of $\mathbb{Q}_{\geq 0}$-valued variables. We use metavariables r, s, t for non-negative rationals, x, y, z, v, w, u for variables, and a, b, c for arithmetic expressions, see also Table 1.

4.2 Syntax of Boolean Expressions

We next describe a *syntax for Boolean expressions* over AExpr, which form *precisely the guards that we allow in* pGCL *programs* (for conditional choices and while loops). Again, the syntax of Boolean expressions will also naturally reoccur in our syntax of expectations. Formally, the set Bool of Boolean expressions is given by

$$\begin{array}{rcll} \varphi & \longrightarrow & a < a & \text{(strict inequality of arithmetic expressions)} \\ & \mid & \varphi \wedge \varphi & \text{(conjunction)} \\ & \mid & \neg\varphi\,. & \text{(negation)} \end{array}$$

We use metavariables φ, ψ, ξ for Boolean expressions, see also Table 1.

The following expressions are syntactic sugar with their standard interpretation and semantics:

$$\mathsf{false}\,, \qquad \mathsf{true}\,, \qquad \varphi \vee \psi\,, \qquad \varphi \longrightarrow \psi\,, \qquad a = b\,, \qquad \text{and} \qquad a \leq b\,.$$

4.3 Syntax of Expectations

We now describe the syntax of a set of *expressive expectations* which can be used as both pre- and postexpectations for the verification of probabilistic programs. Formally, the set Exp of *syntactic expectations* is given by

$$\begin{array}{rcll} f & \longrightarrow & a & \text{(arithmetic expressions)} \\ & \mid & [\varphi] \cdot f & \text{(guarding)} \\ & \mid & f + f & \text{(addition)} \\ & \mid & a \cdot f & \text{(scaling by arithmetic expressions)} \\ & \mid & \mathfrak{S} x \colon f & \text{(supremum over } x\text{)} \\ & \mid & \mathfrak{C} x \colon f\,. & \text{(infimum over } x\text{)} \end{array}$$

As mentioned before, we use metavariables f, g, h for syntactic expectations, see also Table 1. Let us go over the different possibilities of syntactic expectations according to the above grammar.

Table 2. The semantics of arithmetic expressions a and Boolean expressions φ.

a	${}^{\sigma}[\![a]\!]$	φ	${}^{\sigma}[\![\varphi]\!] = \mathsf{true}$ **iff**
$r \quad (\in \mathbb{Q}_{\geq 0})$	r	$a < b$	${}^{\sigma}[\![a]\!] < {}^{\sigma}[\![b]\!]$
$x \quad (\in \mathsf{Vars})$	$\sigma(x)$	$\psi \wedge \xi$	${}^{\sigma}[\![\psi]\!] = \mathsf{true} = {}^{\sigma}[\![\xi]\!]$
$b + c$	${}^{\sigma}[\![b]\!] + {}^{\sigma}[\![c]\!]$	$\neg\psi$	${}^{\sigma}[\![\psi]\!] = \mathsf{false}$
$b \cdot c$	${}^{\sigma}[\![b]\!] \cdot {}^{\sigma}[\![c]\!]$		
$b \dot{-} c$	$\begin{cases} {}^{\sigma}[\![b]\!] - {}^{\sigma}[\![c]\!], & \text{if } {}^{\sigma}[\![b]\!] \geq {}^{\sigma}[\![c]\!] \\ 0, & \text{else} \end{cases}$		

Arithmetic expressions. These form the base case and it is immediate that they are needed for an expressive language. Assume, for instance, that we want to know the "expected" (in fact: certain) value of variable x—itself an arithmetic expression by definition—after executing $x := a$. Then this is given by $\mathsf{wp}[\![x := a]\!](x) = a$—again an arithmetic expression. As a could have been *any* arithmetic expression, we at least need all arithmetic expressions in an expressive expectation language.

Guarding and addition. Both guarding—multiplication with a predicate—and addition are used for expressing weakest preexpectations of conditional choices and loops. As we have, for instance,

$$\mathsf{wp}[\![\texttt{if}\ (\varphi)\ \{C_1\}\ \texttt{else}\ \{C_2\}]\!](f) \;=\; [\varphi] \cdot \mathsf{wp}[\![C_1]\!](f) + [\neg\varphi] \cdot \mathsf{wp}[\![C_1]\!](f)\ ,$$

it is evident that guarding and addition is convenient, if not necessary, for being expressive.

Scaling by arithmetic expressions. One could ask why we restrict to multiplications of arithmetic expressions and expectations and do not simply allow for multiplication of two arbitrary expectations $f \cdot g$. We will defer this discussion to Section 4.6. For now, it suffices to say that we can express all multiplications we need without running into trouble with quantifiers which would happen otherwise.

Suprema and infima. The supremum and infimum constructs $\sup x\colon f$ and $\inf x\colon f$ take over the role of the $\exists$ and $\forall$ quantifiers of first-order logic. We use them to *bind* variables x. The $\sup$ and $\inf$ quantifiers are necessary to make our expectation language expressive in the same was as, for instance, at least the $\exists$ quantifier is necessary to make first-order logic expressive for weakest preconditions of non-probabilistic programs.

As is standard, we additionally admit *parentheses* for clarifying the order of precedence in syntactic expectations. To keep the amount of parentheses to a minimum, we assume that $\cdot$ has precedence over $+$ and that the quantifiers $\sup$ and $\inf$ have the *least* precedence.

The set of *free variables* $\mathsf{FV}(f) \subseteq \mathsf{Vars}$ is the set of all variables that occur syntactically in f and that are not in the scope of some $\sup$ or $\inf$ quantifier. We write $f(x_1, \ldots, x_n)$ to indicate that *at most* the variables $x_1, \ldots, x_n$ occur freely in f. Given a syntactic expectation f, a variable $x \in \mathsf{FV}(f)$, and an arithmetic expression a, we denote by $f[x/a]$ the *syntactic replacement* of every occurrence of x in f by a. Given a syntactic expectation of the form $f(\ldots, x_i, \ldots)$, we often write $f(\ldots, a, \ldots)$ instead of the more cumbersome $f(\ldots, x_i, \ldots)[x_i/a]$.

4.4 Semantics of Expressions and Expectations

The semantics of arithmetic and Boolean expressions is standard—see Table 2. For a program state σ,

we define

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

The semantics ${}^{\sigma}[\![f]\!]$ of an expectation f under state σ is an *extended positive real* (i.e., a positive real number or ∞) defined inductively as follows:

$$\begin{aligned}
{}^{\sigma}[\![a]\!] &\triangleq {}^{\sigma}[\![a]\!] \quad [6] \\
{}^{\sigma}[\![[\varphi]\cdot f]\!] &\triangleq \begin{cases} {}^{\sigma}[\![f]\!], & \text{if } {}^{\sigma}[\![\varphi]\!] = \mathsf{true} \\ 0, & \text{else} \end{cases} \\
{}^{\sigma}[\![f+g]\!] &\triangleq {}^{\sigma}[\![f]\!] \;+\; {}^{\sigma}[\![g]\!] \\
{}^{\sigma}[\![a\cdot f]\!] &\triangleq {}^{\sigma}[\![a]\!] \;\cdot\; {}^{\sigma}[\![f]\!] \\
{}^{\sigma}[\![\mathcal{E}x\colon f]\!] &\triangleq \sup\left\{ {}^{\sigma[x\mapsto r]}[\![f]\!] \;\middle|\; r \in \mathbb{Q}_{\geq 0} \right\} \\
{}^{\sigma}[\![\mathcal{A}x\colon f]\!] &\triangleq \inf\left\{ {}^{\sigma[x\mapsto r]}[\![f]\!] \;\middle|\; r \in \mathbb{Q}_{\geq 0} \right\}
\end{aligned}$$

We assume that $0 \cdot \infty = 0$. Most of the above are self-explanatory. The most involved definitions are the ones for quantifiers. The interpretation of the $\mathcal{E}x\colon f$ quantification, for example, interprets f under all possible values of the bounded variable x and then returns the supremum of all these values. Analogously, $\mathcal{A}x\colon f$ returns the infimum. Notice that—even though all variables evaluate to rationals—both the supremum and the infimum are taken over a set of reals. Hence, an expectation f involving $\mathcal{E}$ or $\mathcal{A}$ possibly evaluates to an *irrational* number. For example, the expectation

$$f \;=\; \mathcal{E}x\colon\, [x \cdot x < 2] \cdot x\,,$$

evaluates to $\sqrt{2} \notin \mathbb{Q}_{\geq 0}$ under every state σ.

The supremum of $\emptyset$ is 0. Dually, the infimum of $\emptyset$ is ∞. The supremum of an unbounded set is ∞. We also note that our semantics can generate ∞ only by using a $\mathcal{E}$ quantifier.

As a shorthand for turning syntactic expectations into semantic ones, we define

$$[\![f]\!] \;\triangleq\; \lambda\sigma.\, {}^{\sigma}[\![f]\!]\,.$$

4.5 Equivalence and Ordering of Expectations

For two expectations f and g, we write $f = g$ only if they are *syntactically equal*. On the other hand, we say that two expectations f and g are *semantically equivalent*, denoted $f \equiv g$, if their semantics under every state is equal, i.e.,

$$f \;\equiv\; g \qquad \text{iff} \qquad [\![f]\!] \;=\; [\![g]\!]\,.$$

Similarly to the partial order $\preceq$ on semantical expectations in $\mathbb{E}$, we define a (semantical) partial order $\preceq$ on syntactic expectations in Exp by

$$f \;\preceq\; g \qquad \text{iff} \qquad [\![f]\!] \;\preceq\; [\![g]\!]\,.$$

[6] Here, on the left-hand-side ${}^{\sigma}[\![\cdot]\!]$ denotes the semantics of expectations, whereas on the right-hand-side ${}^{\sigma}[\![\cdot]\!]$ denotes the semantics of arithmetic expressions.

4.6 A Note on Forbidding $f \cdot g$ in our Syntax

Analogously to classical logic, a syntactic expectation f is in *prenex normal form*, if it is of the form

$$f \;=\; \mathfrak{Q}_1 x_1 \ldots \mathfrak{Q}_k x_k \colon g\,,$$

where $\mathfrak{Q}_i \in \{\mathcal{E}, \mathcal{U}\}$ and where g is quantifier-free. Being able to transform any syntactic expectation into prenex normal form while preserving its semantics will be essential to our expressiveness proof. In particular, we require that there is an algorithm that brings arbitrary syntactic expectations into prenex normal form, without inspecting their semantics.

The problem with allowing $f \cdot g$ arises in the context of the $0 \cdot \infty = 0$ phenomenon. Suppose for the moment that we allow for $f \cdot g$ syntactically and define

$$[\![f \cdot g]\!]^{\sigma} \;\triangleq\; [\![f]\!]^{\sigma} \cdot [\![g]\!]^{\sigma}$$

semantically, where $0 \cdot \infty = \infty \cdot 0 = 0$. Because of commutativity of multiplication, the above is an absolutely natural definition. This also immediately gives us that $[\![f \cdot g]\!]^{\sigma} = [\![g \cdot f]\!]^{\sigma}$.

We now show that we encounter a problem when trying to transform expectations into prenex normal form. For that, consider the two expectations

$$f \;=\; \mathcal{U} x \colon \frac{1}{x+1} \qquad \text{and} \qquad g \;=\; \mathcal{E} y \colon y\,.$$

Notice that we slightly abuse notation since, strictly speaking, $\frac{1}{x+1}$ is not allowed by our syntax. We can however express it as $\mathcal{E} z \colon [z \cdot (x+1) = 1] \cdot z$. Clearly, we have $[\![f]\!]^{\sigma} = 0$ and $[\![g]\!]^{\sigma} = \infty$ for all σ, i.e., both f and g are constant expectations.

Let us now consider the product of f and g. For all σ, its semantics is given by

$$[\![f \cdot g]\!]^{\sigma} \;=\; [\![f]\!]^{\sigma} \cdot [\![g]\!]^{\sigma} \;=\; 0 \cdot \infty \;=\; 0 \;=\; \infty \cdot 0 \;=\; [\![g]\!]^{\sigma} \cdot [\![f]\!]^{\sigma} \;=\; [\![g \cdot f]\!]^{\sigma}\,.$$

Now consider the following:

$$\begin{aligned}
[\![f \cdot g]\!]^{\sigma} \;&=\; \left[\!\!\left[\left(\mathcal{U} x \colon \frac{1}{x+1} \right) \cdot \left(\mathcal{E} y \colon y \right) \right]\!\!\right]^{\sigma} \\
&=\; \left[\!\!\left[\mathcal{U} x \colon \mathcal{E} y \colon \frac{1}{x+1} \cdot y \right]\!\!\right]^{\sigma} && \text{(by prenexing)} \\
&=\; \inf \left\{ \sup \left\{ \frac{1}{r+1} \cdot s \;\middle|\; s \in \mathbb{Q}_{\geq 0} \right\} \;\middle|\; r \in \mathbb{Q}_{\geq 0} \right\} \\
&=\; \inf \{\, \infty \mid r \in \mathbb{Q}_{\geq 0} \,\} \\
&=\; \infty \\
&\neq\; 0 \\
&=\; \sup \{\, 0 \mid s \in \mathbb{Q}_{\geq 0} \,\} \\
&=\; \sup \left\{ \inf \left\{ \frac{1}{r+1} \cdot s \;\middle|\; r \in \mathbb{Q}_{\geq 0} \right\} \;\middle|\; s \in \mathbb{Q}_{\geq 0} \right\} \\
&=\; \sup \left\{ \inf \left\{ s \cdot \frac{1}{r+1} \;\middle|\; r \in \mathbb{Q}_{\geq 0} \right\} \;\middle|\; s \in \mathbb{Q}_{\geq 0} \right\} && \text{(by commutativity of } \cdot \text{ in } \mathbb{R}^{\infty}_{\geq 0}) \\
&=\; \left[\!\!\left[\mathcal{E} y \colon \mathcal{U} x \colon y \cdot \frac{1}{x+1} \right]\!\!\right]^{\sigma} \\
&=\; \left[\!\!\left[\left(\mathcal{E} y \colon y \right) \cdot \left(\mathcal{U} x \colon \frac{1}{x+1} \right) \right]\!\!\right]^{\sigma} && \text{(by un-prenexing)} \\
&=\; [\![g \cdot f]\!]^{\sigma}
\end{aligned}$$

We see that $\mathfrak{E}y\colon \mathfrak{L}x\colon \frac{1}{x+1}\cdot y$ is a sound prenex normal form of $g\cdot f$ whereas $\mathfrak{L}x\colon \mathfrak{E}y\colon \frac{1}{x+1}\cdot y$ apparently is not a sound prenex normal form of $f\cdot g$. A fact that seems even more off-putting is that—even though $f\equiv 0$—the above argument would not have worked for $f=0$.

To summarize, we deem the above considerations enough grounds to forbid $f\cdot g$ altogether, in particular since the rescaling $a\cdot f$ suffices in order for our syntactic expectations to be expressive. We also note that we will later provide a syntactic, but much more complicated, way to write down arbitrary products between syntactic expectations, see Theorem 9.4.

5 EXPRESSIVENESS FOR LOOP-FREE PROGRAMS

Before we deal with loops, we now show that our set Exp of syntactic expectations is *expressive for all loop-free* pGCL *programs*. Proving expressiveness for loops is *way more involved* and will be addressed separately in the remaining sections.

Lemma 5.1. Exp *is expressive (see* Definition 3.1*) for all loop-free* pGCL *programs C, i.e., for all $f\in\mathsf{Exp}$ there exists a syntactic expectation $g\in\mathsf{Exp}$, such that*

$$\mathsf{wp}[\![C]\!]\,([\![f]\!]) \;=\; [\![g]\!]\ .$$

For proving this expressiveness lemma (and also for the case of loops), we need the following technical lemma about substitution of variables by values in our semantics:

Lemma 5.2. *For all σ, f, and a,*

$$^{\sigma}[\![f\,[x/a]]\!] \;=\; {}^{\sigma[x\mapsto {}^{\sigma}[\![a]\!]]}[\![f]\!] \qquad \text{or equivalently} \qquad [\![f\,[x/a]]\!] \;=\; [\![f]\!]\,[x/a]$$

Proof. By induction on the structure of f. □

Intuitively, Lemma 5.2 states that syntactically replacing variable x by an arithmetical expression a in expectation f amounts to interpreting f in states where the variable x has been substituted by the evaluation of a under that state.

Proof of Lemma 5.1. Let $f\in\mathsf{Exp}$ be arbitrary. The proof goes by induction on the structure of loop-free programs C. It is somewhat standard, but we present it here because it demonstrates nicely that our syntactic constructs are actually needed. We start with the atomic programs:

The effectless program `skip`. We have $\mathsf{wp}[\![\mathtt{skip}]\!]\,([\![f]\!]) = [\![f]\!]$ and $f\in\mathsf{Exp}$ by assumption.

The assignment $x := a$. We have

$$\begin{aligned}\mathsf{wp}[\![x := a]\!]\,([\![f]\!]) \;&=\; [\![f]\!]\,[x/a]\\ &=\; [\![f\,[x/a]]\!] \qquad \text{(by Lemma 5.2)}\end{aligned}$$

and $f\,[x/a]\in\mathsf{Exp}$ since $f\,[x/a]$ is obtained from f by a syntactic replacement.

Induction Hypothesis. For arbitrary but fixed loop-free programs C_1 and C_2, there exist syntactic expectations $g_1, g_2\in\mathsf{Exp}$, such that

$$\mathsf{wp}[\![C_1]\!]\,([\![f]\!]) \;=\; [\![g_1]\!] \qquad \text{and} \qquad \mathsf{wp}[\![C_2]\!]\,([\![f]\!]) \;=\; [\![g_2]\!]\ .$$

We then proceed with the compound loop-free programs:

The probabilistic choice $\{\,C_1\,\}\,[\,p\,]\,\{\,C_2\,\}$. We have

$$\begin{aligned}
&\mathsf{wp}[\![\{\,C_1\,\}\,[\,p\,]\,\{\,C_2\,\}]\!]\,([\![f]\!]) \\
&= p \cdot \mathsf{wp}[\![C_1]\!]\,([\![f]\!]) + (1-p) \cdot \mathsf{wp}[\![C_2]\!]\,([\![f]\!]) && \text{(by definition of wp)} \\
&= p \cdot [\![g_1]\!] + (1-p) \cdot [\![g_2]\!] && \text{(by I.H. on } C_1 \text{ and } C_2\text{)} \\
&= [\![p \cdot g_1 + (1-p) \cdot g_2]\!] && \text{(pointwise addition and multiplication)}
\end{aligned}$$

and $p \cdot g_1 + (1-p) \cdot g_2 \in \mathsf{Exp}$, see Section 4.3.

The conditional choice $\texttt{if}\,(\,\varphi\,)\,\{\,C_1\,\}\,\texttt{else}\,\{\,C_2\,\}$. We have

$$\begin{aligned}
&\mathsf{wp}[\![\texttt{if}\,(\,\varphi\,)\,\{\,C_1\,\}\,\texttt{else}\,\{\,C_2\,\}]\!]\,([\![f]\!]) \\
&= [\varphi] \cdot \mathsf{wp}[\![C_1]\!]\,([\![f]\!]) + [\neg\varphi] \cdot \mathsf{wp}[\![C_2]\!]\,([\![f]\!]) && \text{(by definition of wp)} \\
&= [\varphi] \cdot [\![g_1]\!] + [\neg\varphi] \cdot [\![g_2]\!] && \text{(by I.H. on } C_1 \text{ and } C_2\text{)} \\
&= [\![[\varphi] \cdot g_1 + [\neg\varphi] \cdot g_2]\!] && \text{(pointwise addition and multiplication)}
\end{aligned}$$

and $[\varphi] \cdot g_1 + [\neg\varphi] \cdot g_2 \in \mathsf{Exp}$, see Section 4.3.

Hence, Exp is expressive for loop-free programs. □

6 EXPRESSIVENESS FOR LOOPY PROGRAMS — OVERVIEW

Before we get to the proof itself, we outline the main challenges—and the steps we took to address them—of proving expressiveness of our syntactic expectations Exp for pGCL programs including loops; the technical details of the involved encodings and auxiliary results are considered throughout Sections 7 – 10. This section is intended to support navigation through the individual components of the expressiveness proof; as such, we provide various references to follow-up sections.

6.1 Setup

As in the loop-free case considered in Section 5, we prove expressiveness of Exp for all pGCL programs (including loopy ones) by induction on the program structure; all cases except loops are completely analogous to the proof of Lemma 5.1. Our remaining proof obligation thus boils down to proving that, for every loop $C = \texttt{while}\,(\,\varphi\,)\,\{\,C'\,\}$,

$$\forall\, f \in \mathsf{Exp}\ \exists\, g \in \mathsf{Exp}\colon \quad \mathsf{wp}[\![\,\texttt{while}\,(\,\varphi\,)\,\{\,C'\,\}]\!]\,([\![f]\!]) \;=\; [\![g]\!]\;, \tag{†}$$

where we already know by the I.H. that the same property holds for the loop body C', i.e.,

$$\forall\, f' \in \mathsf{Exp}\ \exists\, g' \in \mathsf{Exp}\colon \quad \mathsf{wp}[\![C']\!]\,([\![f']\!]) \;=\; [\![g']\!]\;. \tag{1}$$

Remark (A Simplification for this Overview). Just for this overview section, we assume that the set Vars of all variables is *finite* instead of countable. This is a convenient simplification to avoid a few purely technical details such that we can focus on the actual ideas of the proof. We do *not* make this assumption in follow-up sections. Rather, our construction will ensure that only the finite set of "relevant" variables—those that appear in the program or the postcondition under consideration—is taken into account. △

6.2 Basic Idea: Exploiting the Kozen Duality

We first move to an alternative characterization of the weakest preexpectation of loops whose components are simpler to capture with syntactic expectations. In particular, we will be able to apply our induction hypothesis (1) to some of these components.

Recall the Kozen duality between forward moving measure transformers and backward moving expectation transformers (see Theorem 2.1 and Figure 1 in Section 2):

$$\mathsf{wp}[\![C]\!]\,(X) \;=\; \lambda\sigma_0.\ \sum_{\tau\in\Sigma} X(\tau)\cdot\mu_C^{\sigma_0}(\tau)\,,$$

where $\mu_C^{\sigma_0}$ is the probability distribution over final states obtained by running C on initial state σ_0. Adapting the above equality to our concrete case in which C is a loop and $X = [\![f]\!]$, we obtain

$$\mathsf{wp}[\![\,\texttt{while}\,(\varphi)\,\{\,C'\,\}\,]\!]\,([\![f]\!]) \;=\; \lambda\sigma_0.\ \sum_{\tau\in\Sigma} [\![[\neg\varphi]\cdot f]\!]\,(\tau)\cdot\mu^{\sigma_0}_{\texttt{while}(\varphi)\{C'\}}(\tau)\,,$$

where we strengthened the postexpectation f to $[\neg\varphi]\cdot f$ to account for the fact that the loop guard φ is violated in every final state, see [Kaminski 2019, Corollary 4.6, p. 85]. The main idea is—instead of viewing the whole distribution $\mu^{\sigma_0}_{\texttt{while}(\varphi)\{C'\}}$ in a single "big step"—to take a more operational "small-step" view: we consider the intermediate states reached after each guarded loop iteration, which corresponds to executing the program

$$C_{\text{iter}} \;=\; \texttt{if}\,(\varphi)\,\{C'\}\,\texttt{else}\,\{\texttt{skip}\}\,.$$

We then sum over all terminating *execution paths*—finite sequences of states $\sigma_0,\dots\sigma_{k-1}$ with initial state σ_0 and final state $\sigma_{k-1}=\tau$—instead of a single final state τ. The probability of an execution path is then given by the product of the probability $\mu^{\sigma_i}_{C_{\text{iter}}}(\sigma_{i+1})$ of each intermediate step, i.e., the probability of reaching the state σ_{i+1} from the previous state σ_i:

$$\mathsf{wp}[\![\,\texttt{while}\,(\varphi)\,\{\,C'\,\}\,]\!]\,([\![f]\!]) \;=\; \lambda\sigma_0.\ \sup_{k\in\mathbb{N}}\ \sum_{\sigma_0,\dots,\sigma_{k-1}\in\Sigma} [\![[\neg\varphi]\cdot f]\!]\,(\sigma_{k-1})\cdot\prod_{i=0}^{k-2}\mu^{\sigma_i}_{C_{\text{iter}}}(\sigma_{i+1})\,. \tag{2}$$

Notice that the above sum (without the sup) considers all execution paths of a fixed length k; we take the supremum over all natural numbers k to account for all terminating execution paths.

Next, we aim to apply the induction hypothesis (1) to the probability $\mu^{\sigma_i}_{C_{\text{iter}}}(\sigma_{i+1})$ of each step such that we can write it as a syntactic expectation. To this end, we need to characterize $\mu^{\sigma_i}_{C_{\text{iter}}}(\sigma_{i+1})$ in terms of weakest preexpectations. We employ a syntactic expectation $[\sigma]$—called the *characteristic assertion* [Winskel 1993] of state σ—that captures the values assigned to variables by state σ:[7]

$$[\sigma] \;=\; \left[\bigwedge_{x\in\mathsf{Vars}} x=\sigma(x)\right].$$

By Kozen duality (Theorem 2.1), the probability of reaching state σ_{i+1} from σ_i in one guarded loop iteration C_{iter} is then given by

$$\mu^{\sigma_i}_{C_{\text{iter}}}(\sigma_{i+1}) \;=\; \mathsf{wp}[\![C_{\text{iter}}]\!]\,([\![[\sigma_{i+1}]]\!])\,(\sigma_i)\,.$$

By the same reasoning as for conditional choices in Lemma 5.1 and the induction hypothesis (1), there exists a syntactic expectation $g^{\sigma_{i+1}}_{C_{\text{iter}}}\in\mathsf{Exp}$ such that

$$\mu^{\sigma_i}_{C_{\text{iter}}}(\sigma_{i+1}) \;=\; \mathsf{wp}[\![C_{\text{iter}}]\!]\,([\![[\sigma_{i+1}]]\!])\,(\sigma_i) \;=\; \left[\!\!\left[g^{\sigma_{i+1}}_{C_{\text{iter}}}\right]\!\!\right](\sigma_i)\,.$$

[7] Recall from our remark on simplification that Vars is considered finite for this section.

Plugging the above equality into our "small-step" characterization of loops (2) then yields the following characterization of $[\![g]\!]$ in (†):

$$\mathsf{wp}[\![\,\texttt{while}\,(\varphi)\,\{\,C'\,\}\,]\!]([\![f]\!]) \;=\; \lambda\sigma_0.\ \sup_{k\in\mathbb{N}} \underbrace{\sum_{\sigma_0,\ldots,\sigma_{k-1}\in\Sigma} \underbrace{\underbrace{\Big[\!\Big[\underbrace{[\neg\varphi]\cdot f}_{\in\mathsf{Exp}}\Big]\!\Big](\sigma_{k-1})\cdot \underbrace{\prod_{i=0}^{k-2} \underbrace{\Big[\!\Big[\, g_{C_{\text{iter}}}^{\sigma_{i+1}} \,\Big]\!\Big](\sigma_i)}_{\in\mathsf{Exp}}}_{\text{non-constant product expressible in }\mathsf{Exp}?}}_{\text{simple product expressible in }\mathsf{Exp}?}}_{\text{non-constant sum over paths of length } k \text{ expressible in }\mathsf{Exp}?}}_{\exists k\colon\ldots\in\mathsf{Exp}} \tag{3}$$

A formal proof of the above characterization is provided alongside Theorem 10.1.

6.3 Encoding Loops as Syntactic Expectations

Let us now revisit the individual components of the expectation (3) above and discuss how to encode them as syntactic expectations in Exp, moving through the braces from bottom to top:

6.3.1 The Supremum $\sup_{k\in\mathbb{N}}$. The supremum ensures that terminating execution paths *of arbitrary length* are accounted for; it is supported in Exp by the $\exists$ quantifier. If we already know a syntactic expectation $g_{\text{sum}}(k) \in \mathsf{Exp}$ for the entire sum that follows, we hence obtain an encoding of the whole expectation, namely

$$\exists k\colon g_{\text{sum}}(k) \;\in\; \mathsf{Exp}\,.$$

6.3.2 The Non-constant Sum $\sum_{\sigma_0,\ldots,\sigma_{k-1}\in\Sigma}$. This sum *cannot* directly be written as a syntactic expectation: First, it sums over *execution paths* whereas all variables and constants in syntactic expectations are evaluated to rational numbers. Second, its number of summands depends on the length k of execution paths whereas Exp only supports sums with a constant number of summands.

To deal with the first issue, there is a standard solution in expressiveness proofs (cf. [Loeckx and Sieber 1987; Tatsuta et al. 2009, 2019; Winskel 1993]): We employ *Gödelization* to encode both program states and finite sequences of program states as natural numbers in syntactic expectations. The details are found in Section 7. In particular:

- We show that Exp subsumes first-order arithmetic over the natural numbers.
- We adapt the approach of Gödel [1931] to encode sequences of both natural numbers and non-negative rationals as Gödel numbers in our language Exp.
- We define a predicate (in Exp) StateSequence (u, v) that is satisfied iff u is the Gödel number of a sequence of states of length $v - 1$.

To deal with the second issue (the sum having a variable number of summands), we also rely on the ability to encode sequences as Gödel numbers in Exp—the details are found in Section 9. Roughly speaking, we encode the sum as follows:

- We define a syntactic expectation $h(v_{\text{sum}})$ that serves as a map from v_{sum} to individual summands, i.e., $h\,[v_{\text{sum}}/i]$ yields the i-th summand.
- We construct a syntactic expectation $\mathsf{Sum}\,[v_{\text{sum}}, h, v]$ for partial sums, summing up the first v summands defined by the syntactic expectation h—see Theorem 9.2 for details.

P	$P_{\mathbb{Q}_{\geq 0}}$
φ	$\varphi \wedge N(x_1) \wedge \ldots \wedge N(x_n)$
$\exists x\colon P'$	$\exists x\colon P'_{\mathbb{Q}_{\geq 0}}$
$\forall x\colon P'$	$\forall x\colon N(x) \longrightarrow P'_{\mathbb{Q}_{\geq 0}}$

Fig. 3. Rules defining the formula formula $P_{\mathbb{Q}_{\geq 0}} \in \mathrm{A}_{\mathbb{Q}_{\geq 0}}$ for a Boolean expression φ and $\mathsf{FV}(P) = \{x_1, \ldots, x_n\}$.

P	$[P]$
φ	$[\varphi]$
$\exists v\colon P'$	$\mathfrak{Z} v\colon [P']$
$\forall v\colon P'$	$\mathfrak{C} v\colon [P']$

Fig. 4. Rules for transforming a formula $P \in \mathrm{A}_{\mathbb{Q}_{\geq 0}}$ into an expectation $[P] \in \mathsf{Exp}$.

6.3.3 The Product $[\![[\neg\varphi] \cdot f]\!] \cdot \ldots$. This product is not directly expressible in Exp as arbitrary products between syntactic expectations are not allowed. They are, however, expressible in our language. We define a product operation $h_1 \odot h_2$ and prove its correctness in Corollary 9.5.

6.3.4 The Non-constant Product $\prod_{i=0}^{k-2} \left[\!\left[g^{\sigma_{i+1}}_{C_{iter}} \right]\!\right](\sigma_i)$. This product consists of $k-1$ factors; its encoding requires a similar approach as for non-constant sums. That is, we define a syntactic expectation Product $\left[v_{\text{prod}}, h, v\right]$ that multiplies the first v *factors* defined by the syntactic expectation $h(v_{\text{prod}})$. Details are provided in Theorem 9.4.

6.3.5 The Expectations $[\![[\neg\varphi] \cdot f]\!]$ and $\left[\!\left[g^{\sigma_{i+1}}_{C_{iter}} \right]\!\right]$. Both are syntactic expectations by construction.

6.4 The Expressiveness Proof

It remains to glue together the constructions for the individual components of the expectation (3), which characterizes the weakest preexpectation of loops. We present the full construction, a proof of its correctness, and an example of the resulting syntactic expectation in Section 10.

7 GÖDELIZATION FOR SYNTACTIC EXPECTATIONS

We embed the (standard model of) first-order arithmetic over both the rational and the natural numbers in our language Exp—thereby addressing the first issue raised in Section 6.3.1. Consequently, Exp conservatively extends the standard assertion language of Floyd-Hoare logic (cf. [Cook 1978; Loeckx et al. 1984; Winskel 1993]), enabling us to encode finite sequences of both rationals and naturals in Exp by means of Gödelization [Gödel 1931].

Recall from Table 1 that we use, e.g., metavariables φ, ψ for Boolean expressions, σ for program states, and so on and we will omit providing the types in order to unclutter the presentation.

7.1 Embedding First-Order Arithmetic in Exp

We denote by $\mathrm{A}_{\mathbb{Q}_{\geq 0}}$ the set of formulas P in *first-order arithmetic* over $\mathbb{Q}_{\geq 0}$, i.e., the extension of Boolean expressions φ (see Section 4.2) by an existential quantifier $\exists x\colon P$ and a universal quantifier $\forall x\colon P$ with the usual semantics, e.g., ${}^{\sigma}[\![\forall x\colon P]\!] = \mathsf{true}$ iff for all $r \in \mathbb{Q}_{\geq 0}$, ${}^{\sigma[x\mapsto r]}[\![P]\!] = \mathsf{true}$. The set $\mathrm{A}_{\mathbb{N}}$ of formulas P in first-order arithmetic over $\mathbb{N}$ is defined analogously by restricting ourselves to (1) states[8] $\sigma\colon \mathsf{Vars} \to \mathbb{N}$ and (2) constants in $\mathbb{N}$ rather than $\mathbb{Q}_{\geq 0}$.

For simplicity, *we assume without loss of generality that all formulas P are in prenex normalform*, i.e., P is a Boolean expression comprising of a block of quantifiers followed by a quantifier-free formula. Recall that program states originally evaluate variables to *rationals*. Since our expressiveness proof

[8]Program states serve here the role of *interpretations* in classical first-order logic.

requires encoding sequences of *naturals*, it is crucial that we can assert that a variable evaluates to a natural. To this end, we adapt a result by Robinson [1949]:

Lemma 7.1. *$\mathbb{N}$ is definable in $\mathrm{A}_{\mathbb{Q}_{\geq 0}}$, i.e. there exists a formula $N(x) \in \mathrm{A}_{\mathbb{Q}_{\geq 0}}$, such that for all σ,*

$$ {}^{\sigma}[\![N(x)]\!] \;=\; \mathsf{true} \qquad \text{iff} \qquad \sigma(x) \;\in\; \mathbb{N}\,. $$

We use the above assertion N to first embed $\mathrm{A}_{\mathbb{N}}$ in $\mathrm{A}_{\mathbb{Q}_{\geq 0}}$. Thereafter, we embed $\mathrm{A}_{\mathbb{Q}_{\geq 0}}$ in Exp. Embedding a formula $P \in \mathrm{A}_{\mathbb{N}}$ in $\mathrm{A}_{\mathbb{Q}_{\geq 0}}$ amounts to (1) asserting $N(x)$ for every $x \in \mathsf{FV}(P)$ and (2) guarding every quantified variable x in P with $N(x)$, i.e., whenever we attempt to evaluate the embedding-formula for non-naturals, we default to false—see Figure 3 for a formal definition.

Theorem 7.2. *Let $P_{\mathbb{Q}_{\geq 0}} \in \mathrm{A}_{\mathbb{N}}$ be the embedding of $P \in \mathrm{A}_{\mathbb{N}}$ as defined in* Figure 3. *Then, for all σ,*

$$ {}^{\sigma}[\![P_{\mathbb{Q}_{\geq 0}}]\!] \;=\; \begin{cases} {}^{\sigma}[\![P]\!]\,, & \text{if } \sigma(x) \in \mathbb{N} \text{ for all } x \in \mathsf{FV}(P)\,, \\ \mathsf{false}, & \text{otherwise}\,. \end{cases} $$

Embedding a formula $P \in \mathrm{A}_{\mathbb{Q}_{\geq 0}}$ into Exp amounts to (1) taking its Iverson bracket for every Boolean expression and (2) substituting the quantifiers $\exists/\forall$ by their quantitative analogs $\mathfrak{S}/\mathfrak{C}$, see Figure 4.

Theorem 7.3. *Let $[P] \in \mathsf{Exp}$ be the embedding of $P \in \mathrm{A}_{\mathbb{Q}_{\geq 0}}$ as defined in* Figure 4. *Then, for all σ,*

$$ {}^{\sigma}[\![[P]]\!] \;=\; \begin{cases} 1, & \text{if } {}^{\sigma}[\![P]\!] = \mathsf{true} \\ 0, & \text{if } {}^{\sigma}[\![P]\!] = \mathsf{false}\,. \end{cases} $$

Given $P(v_1, \ldots, v_n) \in \mathrm{A}_{\mathbb{Q}_{\geq 0}}$, we often write $[P(v_1, \ldots, v_n)]$ instead of $[P]\,(v_1, \ldots, v_n)$.

7.2 Encoding Sequences of Natural Numbers

The embedding of $\mathrm{A}_{\mathbb{N}}$ in our language Exp of syntactic expectations gives us access to a classical result by Gödel [1931] for encoding finite sequences of naturals in a *single* natural.

Lemma 7.4 (Gödel [1931]). *There is a formula $\mathsf{Elem}(v_1, v_2, v_3) \in \mathrm{A}_{\mathbb{N}}$ (with quantifiers) satisfying: For every finite sequence of natural numbers $n_0, \ldots, n_{k-1}$, there is a (Gödel) number $gnum \in \mathbb{N}$ that encodes the sequence, i.e., for all $i \in \{0, \ldots, k-1\}$ and all $m \in \mathbb{N}$, it holds that*

$$ \mathsf{Elem}(gnum, i, m) \;\equiv\; \mathsf{true} \qquad \text{iff} \qquad m \;=\; n_i\,. $$

By Theorem 7.3, we also have an expectation $[\mathsf{Elem}(v_1, v_2, v_3)]$ expressing Elem in Exp.

Example 7.5 (Factorials via Gödel). The syntactic expectation below evaluates to the factorial $x!$:

$$ \begin{aligned} \mathsf{Fac}(x) \;=\; \mathfrak{S} v\colon \mathfrak{S} num\colon v \cdot \big[& \mathsf{Elem}(num, 0, 1) \wedge \mathsf{Elem}(num, x, v) \\ & \wedge \forall u\colon \forall w\colon \big(u < x \wedge \mathsf{Elem}(num, u, w) \longrightarrow \mathsf{Elem}(num, u+1, w \cdot (u+1))\big)\big]\,. \end{aligned} $$

For every state σ, the quantifier $\mathfrak{S} num$ selects a sequence $n_0, n_1 \ldots$ satisfying $n_{\sigma(x)} = \sigma(x)!$. The quantifier $\mathfrak{S} v$ then binds v to the value $n_{\sigma(x)} = \sigma(x)!$. Finally, by multiplying the $\{0,1\}$-valued expectation specifying the sequence by v, we get that ${}^{\sigma}[\![\mathsf{Fac}(x)]\!] = \sigma(x)!$. △

To work with *unique* Gödel numbers, we employ minimalization: The formula $\mathsf{Sequence}(num, v)$ below expresses that (a) *num* is a Gödel number of some sequence $n_0, \ldots, n_{v-1}, \ldots$ of length *at least* v and (b) *num* is the *smallest* Gödel number encoding a sequence with prefix $n_0, \ldots, n_{v-1}$.

$$ \begin{aligned} &\mathsf{Sequence}(num, v) \\ &\quad\triangleq\quad (\forall u\colon u < v \longrightarrow \exists w\colon \mathsf{Elem}(num, u, w)) && \text{(a)} \\ &\qquad\wedge \big(\forall num'\colon \big(\forall u\colon u < v \longrightarrow \exists w\colon \mathsf{Elem}(num, u, w) \wedge \mathsf{Elem}(num', u, w)\big) && \text{(b)} \\ &\qquad\quad\longrightarrow num' \geq num\big) \end{aligned} $$

For every k and every sequence $n_0, \ldots, n_{k-1}$ of length k, we then define *the* Gödel number encoding the sequence $n_0, \ldots, n_{k-1}$ of length k as the unique natural number $\langle n_0, \ldots, n_{k-1} \rangle$ satisfying

$$\mathsf{Sequence}\,(\langle n_0, \ldots, n_{k-1}\rangle, k) \;\wedge\; \bigwedge_{i=0}^{k-1} \mathsf{Elem}\,(\langle n_0, \ldots, n_{k-1}\rangle, i, n_i)\ .$$

7.3 Encoding Sequences of Non-negative Rationals

Recall that program states in pGCL map variables to values in $\mathbb{Q}_{\geq 0}$. To encode sequences of program states, we thus first lift Gödel's encoding $\mathsf{Elem}\,(num, i, n)$ to uniquely encode sequences over $\mathbb{Q}_{\geq 0}$. The main idea is to represent such a sequence by pairing two sequences over $\mathbb{N}$.

Lemma 7.6 (Pairing Functions [Cantor 1878]). *There is a formula* $\mathsf{Pair}(v_1, v_2, v_3) \in \mathrm{A}_{\mathbb{N}}$ *satisfying: For every pair of natural numbers* (n_1, n_2)*, there is* exactly one *natural number* n *such that*

$$\mathsf{Pair}(n, n_1, n_2) \;\equiv\; \mathsf{true}\ .$$

Theorem 7.7. *There is a formula* $\mathsf{RElem}\,(v_1, v_2, v_3) \in \mathrm{A}_{\mathbb{Q}_{\geq 0}}$ *satisfying: For every finite sequence* $r_0, \ldots, r_{k-1} \subset \mathbb{Q}_{\geq 0}$ *there is a Gödel number gnum, such that for all* $i \in \{0, \ldots, k-1\}$ *and* $s \in \mathbb{Q}_{\geq 0}$*,*

$$\mathsf{RElem}\,(gnum, i, s) \;\equiv\; \mathsf{true} \quad \textit{iff} \quad s \;=\; r_i\ .$$

Example 7.8 (Harmonic Numbers). For every σ with $\sigma(x) = k \in \mathbb{N}$, the expectation $\mathsf{Harmonic}\,(x) \in \mathsf{Exp}$ below evaluates to the k-th harmonic number $\mathcal{H}(k) \;=\; \sum_{i=1}^{k} \frac{1}{i}$.

$$\begin{aligned}\mathsf{Harmonic}\,(x) \;=\; \mathfrak{S} v\colon \mathfrak{S} num\colon v \cdot \big[&\mathsf{RElem}\,(num, 0, 0) \wedge \mathsf{RElem}\,(num, x, v)\\ &\wedge \forall u\colon \forall w\colon (u < x \wedge \mathsf{RElem}\,(num, u, w))\\ &\longrightarrow \exists w'\colon w' \cdot (u+1) = 1 \wedge \mathsf{RElem}\,(num, u+1, w+w')\big]\end{aligned}$$

Notice that the above Iverson bracket evaluates to 1 on state σ iff $\sigma(num)$ encodes a sequence $r_0, r_1, \ldots, r_{\sigma(x)}$ such that $\sigma(v) = r_{\sigma(x)}$ and

$$r_0 \;=\; 0\,,\; r_1 = \frac{1}{1} + r_0,\; r_2 = \frac{1}{2} + r_1\,,\; \ldots\,, r_{\sigma(x)} \;=\; \frac{1}{\sigma(x)} + r_{\sigma(x)-1}\ .$$

By Theorem 7.2, we do not need to require that $\sigma(u) \in \mathbb{N}$ as $\mathsf{RElem}\,(num, i, w)$ is false if $\sigma(u) \notin \mathbb{N}$.△

Analogously to the previous section, we define a predicate $\mathsf{RSequence}\,(num, v)$ that uses minimalization to a *unique* Gödel number *num* for every sequence $r_0, \ldots, r_{k-1}$ of length k; the only difference between $\mathsf{RSequence}\,(num, v)$ and $\mathsf{Sequence}\,(num, v)$ is that every occurrence of $\mathsf{Elem}\,(.,.,.)$ is replaced by $\mathsf{RElem}\,(.,.,.)$. Moreover, for every k and every sequence $r_0, \ldots, r_{k-1}$, we define *the* Gödel number encoding the sequence $r_0, \ldots, r_{k-1}$ as the unique natural number $\langle r_0, \ldots, r_{k-1} \rangle$ satisfying

$$\mathsf{RSequence}\,(\langle r_0, \ldots, r_{k-1}\rangle, k) \;\wedge\; \bigwedge_{i=0}^{k-1} \mathsf{RElem}\,(\langle r_0, \ldots, r_{k-1}\rangle, i, r_i)\ .$$

7.4 Encoding Sequences of Program States

To encode sequences of program states, we first fix a finite set $\mathrm{x} = \{x_0, \ldots, x_{k-1}\}$ of *relevant variables*. Intuitively, x consists of all variables that appear in a given program or a postexpectation. We define an equivalence relation $\sim_{\mathrm{x}}$ on states by

$$\sigma_1 \sim_{\mathrm{x}} \sigma_2 \qquad \text{iff} \qquad \forall x \in \mathrm{x}\colon \sigma_1(x) = \sigma_2(x)\ .$$

Every *num* satisfying $\mathsf{Sequence}(num, k)$ encodes *exactly one* state σ (modulo $\sim_{\mathrm{x}}$). The Gödel number encoding σ (w.r.t. x), which we denote by $\langle\sigma\rangle_{\mathrm{x}}$, is then the unique number satisfying

$$\mathsf{RSequence}\left(\langle\sigma\rangle_{\mathrm{x}}, k\right) \;\wedge\; \bigwedge_{i=0}^{k-1} \mathsf{RElem}\left(\langle\sigma\rangle_{\mathrm{x}}, i, \sigma(x_i)\right) \;.$$

Notice that we implictly fixed an ordering of the variables in x to identify each value stored in σ for a variable in x. The formula

$$\mathsf{EncodesState}_{\mathrm{x}}(num) \;\triangleq\; \mathsf{RSequence}(num, k) \;\wedge\; \bigwedge_{i=0}^{k-1} \mathsf{RElem}(num, i, x_i)$$

evaluates to true on state σ iff $\sigma(num)$ is the Gödel number of a state σ' with $\sigma \sim_{\mathrm{x}} \sigma'$. Now, let $\sigma_0, \ldots, \sigma_{n-1}$ be a sequence of states of length *n*. *The* Gödel number encoding $\sigma_0, \ldots, \sigma_{n-1}$ (w.r.t. x), which we denote by $\langle(\sigma_0, \ldots, \sigma_{n-1})\rangle_{\mathrm{x}}$, is then the unique number satisfying

$$\mathsf{Sequence}\left(\langle(\sigma_0, \ldots, \sigma_{n-1})\rangle_{\mathrm{x}}, n\right) \;\wedge\; \bigwedge_{i=0}^{n-1} \mathsf{Elem}\left(\langle(\sigma_0, \ldots, \sigma_{n-1})\rangle_{\mathrm{x}}, i, \langle\sigma_i\rangle_{\mathrm{x}}\right) \;.$$

We are now in a position to encode sequences of states. The formula

$$\begin{aligned} &\mathsf{StateSequence}_{\mathrm{x}}(num, v) \\ =\;& \mathsf{Sequence}(num, v) \wedge (\exists v' \colon \mathsf{Elem}(num, 0, v') \wedge \mathsf{EncodesState}_{\mathrm{x}}(v')) \\ &\wedge \forall u \colon \forall v' \colon \left((u < v \wedge \mathsf{Elem}(num, u, v')) \longrightarrow \mathsf{RSequence}(v', k)\right) \end{aligned}$$

evaluates to true on state σ iff (1) *num* is the Gödel number of some sequence $\sigma_0, \ldots, \sigma_{\sigma(v-1)} \in \Sigma$ of states of length $\sigma(v)$ and where (2) σ and σ_0 coincide on all variables in x, i.e., $\sigma \sim_{\mathrm{x}} \sigma_0$. Notice that, for every sequence $\sigma_0, \ldots, \sigma_{n-1}$ of states of length *n*, there is *exactly one num* satisfying $\mathsf{StateSequence}_{\mathrm{x}}(num, n)$. If clear from the context, we often omit the subscript x and simply write $\langle\sigma\rangle$ (resp. $\langle(\sigma_0, \ldots, \sigma_{n-1})\rangle$) instead of $\langle\sigma\rangle_{\mathrm{x}}$ (resp. $\langle(\sigma_0, \ldots, \sigma_{n-1})\rangle_{\mathrm{x}}$).

8 THE DEDEKIND NORMAL FORM

Before we encode sums and products of non-constant size in Exp—as required to deal with the challenges in Sections 6.3.2 to 6.3.4—we introduce a normal form that gives a convenient handle to encode real numbers as syntactic expectations.

As a first step, we transform syntactic expectations into prenex normal form, i.e., we rewrite every $f \in \mathsf{Exp}$ into an equivalent syntactic expectation of the form $Q_1 v_1 \ldots Q_k v_k \colon f'$, where $Q_i \in \{\mathcal{E}, \mathcal{C}\}$ and f' is "quantifier"–free, i.e., contains neither $\mathcal{E}$ nor $\mathcal{C}$. The following lemma justifies that any expectation can indeed be transformed into an equivalent one in prenex normal form by iteratively pulling out quantifiers. In case the quantified logical variable already appears in the expectation the quantifier is pulled over, we rename it by a fresh one first.

Lemma 8.1 (Prenex Transformation Rules). *For all f, f_1, $f_2 \in \mathsf{Exp}$, terms a, and Boolean expressions φ, quantifiers $Q \in \{\mathcal{E}, \mathcal{C}\}$, and fresh logical variables v', the following equivalences hold:*

(1) $(Qv \colon f_1) + f_2 \;\equiv\; Qv' \colon f_1[v/v'] + f_2$,
(2) $f_1 + (Qv \colon f_2) \;\equiv\; Qv' \colon f_1 + f_2[v/v']$,
(3) $a \cdot Qv \colon f \;\equiv\; Qv' \colon a \cdot f[v/v']$, *and*
(4) $[\varphi] \cdot Qv \colon f \;\equiv\; Qv' \colon [\varphi] \cdot f[v/v']$.

The Dedekind normal form is motivated by the notion of *Dedekind cuts* [Bertrand 1849]. We denote by $\mathsf{Cut}(\alpha)$ the Dedekind cut of a real number, i.e., the set of all rationals strictly smaller than α. In the realm of *all* reals, it is required that a Dedekind cut is neither the empty set nor the whole set

of rationals $\mathbb{Q}$. However, since we operate in the realm of non-negative reals with infinity $\mathbb{R}^{\infty}_{\geq 0}$, we *do* allow for both empty cuts and $\mathbb{Q}_{\geq 0}$. More formally, we define:

Definition 8.2. Let $\alpha \in \mathbb{R}^{\infty}_{\geq 0}$. The *Dedekind cut* $\mathsf{Cut}\,(\alpha) \subseteq \mathbb{Q}_{\geq 0}$ of α is defined as

$$\mathsf{Cut}\,(\alpha) \quad \triangleq \quad \{\, r \in \mathbb{Q}_{\geq 0} \mid r < \alpha \,\} \ .$$

Furthermore, we define $\underline{\mathsf{Cut}}\,(\alpha) \triangleq \mathsf{Cut}\,(\alpha) \cup \{0\}$. △

Dedekind cuts are relevant for our technical development as they allow to describe every real number α as a supremum over a set of rational numbers. In particular, the Dedekind cut $\mathsf{Cut}\,(0)$ of 0 is the empty set with supremum 0, and the Dedekind cut $\mathsf{Cut}\,(\infty)$ of ∞ is the set $\mathbb{Q}_{\geq 0}$ with supremum ∞. Formally:

Lemma 8.3. *For every $\alpha \in \mathbb{R}^{\infty}_{\geq 0}$, we have $\alpha = \sup \mathsf{Cut}\,(\alpha)$.*

Theorem 8.4. *For every $f \in \mathsf{Exp}$, there is a syntactic expectation in prenex normal form*

$$\mathsf{Dedekind}[v_{\mathsf{Cut}}, f] \ = \ \mathsf{Prefix}\,(f) : \ [\varphi] \ ,$$

where $\mathsf{Prefix}\,(f)$ is a quantifier prefix, φ is an effectively constructible Boolean expression, and the free variable v_{Cut} is fresh, such that: for all program states σ, we have

$${}^{\sigma}[\![\mathsf{Dedekind}[v_{\mathsf{Cut}}, f]]\!] \ = \ \begin{cases} 1, & \text{if } \sigma(v_{\mathsf{Cut}}) < {}^{\sigma}[\![f]\!] \\ 0, & \text{otherwise} \,. \end{cases}$$

We call $\mathsf{Dedekind}[v_{\mathsf{Cut}}, f]$ the Dedekind normal form *of f.*

The Dedekind normal form $\mathsf{Dedekind}[v_{\mathsf{Cut}}, f]$ defines the Dedekind cut of every ${}^{\sigma}[\![f]\!]$, i.e.,

$$\text{for all } \sigma\colon \quad \mathsf{Cut}\left({}^{\sigma}[\![f]\!]\right) \ = \ \left\{\, r \in \mathbb{Q}_{\geq 0} \;\middle|\; r = \sigma(v_{\mathsf{Cut}}),\ {}^{\sigma}[\![\mathsf{Dedekind}[v_{\mathsf{Cut}}, f]]\!] = 1 \,\right\} \ .$$

Hence, we can recover f from $\mathsf{Dedekind}[v_{\mathsf{Cut}}, f]$:

Lemma 8.5. *Let $\mathsf{Dedekind}[v_{\mathsf{Cut}}, f]$ be in Dedekind normal form. Then*

$$f \ \equiv \ \mathfrak{d} v_{\mathsf{Cut}} \colon \mathsf{Dedekind}[v_{\mathsf{Cut}}, f] \cdot v_{\mathsf{Cut}} \ .$$

9 SUMS, PRODUCTS, AND INFINITE SERIES OF SYNTACTIC EXPECTATIONS

This section deals with the syntactic Sum and Product expectations as described in Section 6.3.2. Since a syntactic expectation f evaluates to a non-negative *extended real*, we rely on a reduction from sums over reals to suprema of sums over *rationals*:

Lemma 9.1. *For all $\alpha_0, \ldots, \alpha_n \in \mathbb{R}^{\infty}_{\geq 0}$, we have*

$$\sum_{j=0}^{n} \alpha_j \ = \ \sup \left\{ \sum_{j=0}^{n} r_j \;\middle|\; \forall i \in \{0, \ldots, n\} \colon r_i \in \underline{\mathsf{Cut}}\,(\alpha_i) \right\} \ .$$

Theorem 9.2. *For every $f \in \mathsf{Exp}$ with free variable v_{sum}, there is an effectively constructible expectation $\mathsf{Sum}\,[v_{\mathrm{sum}}, f, v] \in \mathsf{Exp}$ such that for all states σ with $\sigma(v) \in \mathbb{N}$, we have*

$${}^{\sigma}[\![\mathsf{Sum}\,[v_{\mathrm{sum}}, f, v]]\!] \ = \ \sum_{j=0}^{\sigma(v)} {}^{\sigma}[\![f\,[v_{\mathrm{sum}}/j]]\!] \quad \text{and} \quad {}^{\sigma}[\![\mathfrak{d} v \colon \mathsf{Sum}\,[v_{\mathrm{sum}}, f, v]]\!] \ = \ \sum_{j=0}^{\infty} {}^{\sigma}[\![f\,[v_{\mathrm{sum}}/j]]\!] \ .$$

Proof. We sketch the construction of $\mathsf{Sum}\,[v_{\text{sum}}, f, v]$. Lemma 9.1 and the Dedekind normal form $\mathsf{Dedekind}[v_{\text{Cut}}, f]$ of f (cf. Theorem 8.4) give us

$$\begin{aligned}
&\sum_{j=0}^{\sigma(v)} {}^{\sigma}[\![f\,[v_{\text{sum}}/j]]\!] \\
&= \sup\left\{ \sum_{j=0}^{\sigma(v)} r_j \;\middle|\; \forall j \in \{0,\dots,\sigma(v)\} \colon r_j \in \underline{\mathsf{Cut}}\left({}^{\sigma}[\![f\,[v_{\text{sum}}/j]]\!]\right) \right\} \\
&= \sup\left\{ \sum_{j=0}^{\sigma(v)} r_j \;\middle|\; \forall j \in \{0,\dots,\sigma(v)\} \colon {}^{\sigma}[\![\mathsf{Dedekind}[f, r_j]]\!] = 1 \text{ or } r_j = 0 \right\} . \qquad (4)
\end{aligned}$$

Writing $\mathsf{Dedekind}[v_{\text{Cut}}, f] = \mathsf{Prefix}\,(f) : [\varphi]$ (cf. Theorem 8.4), we then construct a syntactic expectation g with free variables v and num by

$$\begin{aligned}
&\mathfrak{S} v' \colon v' \cdot \mathfrak{L} u \colon \mathfrak{L} z \colon \mathfrak{S} v_{\text{Cut}} \colon \mathsf{Prefix}\,(f) : \\
&[\mathsf{RElem}\,(num, 0, 1) \wedge \mathsf{RElem}\,(num, v+1, v) \\
&\quad \wedge \big((u < v+1 \wedge \mathsf{RElem}\,(num, u, z) \wedge ([\varphi]\,[v_{\text{prod}}/u] \vee v_{\text{Cut}} = 0)) \\
&\qquad \longrightarrow \mathsf{RElem}\,(num, u+1, z + v_{\text{Cut}})\big)] .
\end{aligned}$$

For every state σ where $\sigma(num)$ is a Gödel number encoding some sequence

$$1,\ 1 \cdot r_1,\ 1 + r_1 + r_2,\ \dots\ , 1 + r_1 + \ldots + r_{\sigma(v)}$$

with $r_j \in \underline{\mathsf{Cut}}\left({}^{\sigma}[\![f\,[v_{\text{sum}}/j]]\!]\right)$ for all $0 \leq j \leq \sigma(v)$, expectation g evaluates to the last element of the above sequence, i.e., an element of the set from Equation (4). Hence, by Lemma 9.1, the supremum over these sequences, i.e, all Gödel numbers, gives us

$$\mathsf{Sum}\,[v_{\text{sum}}, f, v] \;=\; \mathfrak{S} num \colon g\ .$$

A detailed proof is found in [Batz et al. 2020]. □

For an arithmetic expression a, we write $\mathsf{Sum}\,[v_{\text{sum}}, f, a]$ instead of $\mathsf{Sum}\,[v_{\text{sum}}, f, v]\,[v/a]$.

Example 9.3. Sum provides us with a much more convenient way to construct $\mathsf{Harmonic}\,(x)$ from Example 7.8. Let $f = {}^{1}/_{v_{\text{sum}}}$ where ${}^{1}/_{v_{\text{sum}}}$ is a shorthand for $\mathfrak{S} w \colon w \cdot [w \cdot v_{\text{sum}} = 1]$. Then, by Theorem 9.2, we have for every $\sigma \in \Sigma$

$${}^{\sigma}[\![\mathsf{Sum}\,[v_{\text{sum}}, f, x]]\!] \;=\; \sum_{j=0}^{\sigma(x)} {}^{\sigma}[\![f\,[v_{\text{sum}}/j]]\!] \;=\; \sum_{j=1}^{\sigma(x)} \frac{1}{j} \;=\; \mathcal{H}(\sigma(x))\ .$$

The construction of the syntactic Product expectation is completely analogous:

Theorem 9.4. *For every $f \in \mathsf{Exp}$ with free variable v_{prod}, there is an effectively constructible expectation $\mathsf{Product}\left[v_{\text{prod}}, f, v\right] \in \mathsf{Exp}$ such that for every state σ with $\sigma(v) \in \mathbb{N}$, we have*

$${}^{\sigma}[\![\mathsf{Product}\left[v_{\text{prod}}, f, v\right]]\!] \;=\; \prod_{j=0}^{\sigma(v)} {}^{\sigma}[\![f\left[v_{\text{prod}}/j\right]]\!]\ .$$

For an arithmetic expression a, we write $\mathsf{Product}\left[v_{\text{prod}}, f, a\right]$ instead of $\mathsf{Product}\left[v_{\text{prod}}, f, v\right][v/a]$.

An immediate, yet important, consequence of Theorem 9.4 is that, even though syntactically forbidden, *arbitrary products* of syntactic expectations are expressible in Exp. Let $f, g \in \mathsf{Exp}$, and let v_{prod} be a fresh variable. We define the *(unrestricted) product* $f \odot g$ of f and g by

$$f \odot g \quad \triangleq \quad \mathsf{Product}\left[v_{\text{prod}}, \left[v_{\text{prod}} = 0\right] \cdot f + \left[v_{\text{prod}} = 1\right] \cdot g, 1\right] \ .$$

Corollary 9.5. *Let $f, g \in \mathsf{Exp}$. For all states σ, we have*

$$^{\sigma}[\![f \odot g]\!] \quad = \quad ^{\sigma}[\![f]\!] \cdot \ ^{\sigma}[\![g]\!] \ .$$

10 EXPRESSIVENESS OF OUR LANGUAGE

With the results from the preceding sections at hand, we give a constructive expressiveness proof for our language Exp. Fix a set of variables $\mathrm{x} = \{x_0, \ldots, x_{n-1}\}$. We assume a fixed set $\Sigma_{\mathrm{x}} \subseteq \Sigma$ that contains *exactly one* state from each equivalence class of $\sim_{\mathrm{x}}$ (cf. Section 7.4). Given a state $\sigma \in \Sigma$, we define the *characteristic expectation* $[\sigma]_{\mathrm{x}}$ of σ (w.r.t. x) as

$$[\sigma]_{\mathrm{x}} \quad \triangleq \quad [x_0 = \sigma(x_0) \wedge \ldots \wedge x_{n-1} = \sigma(x_{n-1})] \ .$$

The expectation $[\sigma]_{\mathrm{x}}$ evaluates to 1 on state σ' if $\sigma \sim_{\mathrm{x}} \sigma'$, and to 0 otherwise. Finally, we denote by $\mathsf{Vars}(C)$ the set of all variables that appear in the pGCL program C.

Let us now formalize the characterization of $\mathsf{wp}[\![\texttt{while}\,(\varphi)\,\{\,C'\,\}]\!]\,([\![f]\!])$ from Section 6.2:

Theorem 10.1. *Let $C = \texttt{while}\,(\varphi)\,\{\,C'\,\}$ be a loop and let $f \in \mathsf{Exp}$. Furthermore, let x be a finite set of variables with $\mathsf{Vars}(C) \cup \mathsf{FV}(f) \subseteq \mathrm{x}$. We have*

$$\begin{aligned} & \mathsf{wp}[\![\texttt{while}\,(\varphi)\,\{\,C'\,\}]\!]\,([\![f]\!]) \\ = \ & \lambda\sigma.\ \sup_{k \in \mathbb{N}} \sum_{\sigma_0, \ldots, \sigma_{k-1} \in \Sigma_{\mathrm{x}}} [\sigma_0]_{\mathrm{x}}(\sigma) \cdot ([\neg\varphi] \cdot [\![f]\!])(\sigma_{k-1}) \\ & \qquad \cdot \prod_{i=0}^{k-2} \mathsf{wp}[\![\texttt{if}\,(\varphi)\,\{\,C'\,\}\,\texttt{else}\,\{\,\texttt{skip}\,\}]\!]\,([\sigma_{i+1}]_{\mathrm{x}})(\sigma_i) \ . \end{aligned}$$

Proof. See [Batz et al. 2020]. □

We are finally in a position to prove expressiveness (cf. Definition 3.1).

Theorem 10.2. *The language* Exp *of syntactic expectations is expressive.*

Proof. By induction on the structure of C. All cases except loops are completely analogous to the proof of Lemma 5.1. Let us thus consider the case $C = \texttt{while}\,(\varphi)\,\{\,C_1\,\}$. We employ the syntactic Sum- and Product expectations from Theorems 9.2 and 9.4 to construct the series from Theorem 10.1 in Exp, thus expressing $\mathsf{wp}[\![\texttt{while}\,(\varphi)\,\{\,C_1\,\}]\!]\,([\![f]\!])$.

The products ocurring in Theorem 10.1 are expressed by an effectively constructible syntactic expectation $\mathsf{Path}\,[f]\,(v_1, v_2)$ (where v_1 and v_2 are fresh variables) satisfying:

(1) If $\sigma(v_1) \in \mathbb{N}$ with $\sigma(v_1) > 0$ and $\sigma(v_2) = \langle(\sigma_0, \ldots, \sigma_{\sigma(v_1)-1})\rangle_{\mathrm{x}}$, then

$$\begin{aligned} & ^{\sigma}[\![\mathsf{Path}\,[f]\,(v_1, v_2)]\!] \\ = \ & ([\neg\varphi] \cdot [\![f]\!])(\sigma_{\sigma(v_1)-1}) \cdot \prod_{i=0}^{\sigma(v_1)-2} \mathsf{wp}[\![\texttt{if}\,(\varphi)\,\{\,C_1\,\}\,\texttt{else}\,\{\,\texttt{skip}\,\}]\!]\,([\sigma_{i+1}]_{\mathrm{x}})(\sigma_i) \end{aligned} \tag{5}$$

(2) If $\sigma(v_1) \notin \mathbb{N}$ or $\sigma(v_1) = 0$, then $^{\sigma}[\![\mathsf{Path}\,[f]\,(v_1, v_2)]\!] = 0$.

Then, for the syntactic expectation

$$h \;=\; [\![\mathcal{Q} \mathit{length} \colon \mathcal{Q} \mathit{nums} \colon \mathsf{Sum}\big[v_{\text{sum}}, \big[\mathsf{StateSequence}_{\mathrm{x}}\,(v_{\text{sum}}, \mathit{length})\big] \odot \mathsf{Path}\,[f]\,(\mathit{length}, v_{\text{sum}})\,, \mathit{nums}\big]\,]\!]\,,$$

we have $\mathsf{wp}[\![\texttt{while}\,(\,\varphi\,)\,\{\,C_1\,\}]\!]\,(\,[\![f]\!]) = [\![h]\!]$.

Here, the quantifier $\mathcal{Q}\mathit{length}$ in h corresponds to the $\sup k$ from Theorem 10.1. The subsequent Sum expectation expresses the sum from Theorem 10.1: Summing over sequences of states of length length is realized by summing over all Gödel numbers num satisfying $\mathsf{StateSequence}\,(\mathit{num}, \mathit{length})$. See [Batz et al. 2020] for a detailed correctness proof. □

10.1 Example

We conclude this section by sketching the construction of a syntactic expectation for a concrete loop. Consider the program C given by

$$\begin{aligned}&\texttt{while}\,(\,c = 1\,)\,\{\\ &\quad \{\,c := 0\,\}\;[\,1/2\,]\;\{\,c := 1\,\}\,;\\ &\quad x := x + 1\;\}\end{aligned}$$

where we denote the loop body by C'. Morever, let $f \;\triangleq\; x \in \mathsf{Exp}$. Then the syntactic expectation h expressing $\mathsf{wp}[\![\texttt{while}\,(\,c = 1\,)\,\{\,C'\,\}]\!]\,(\,[\![x]\!])$ as sketched in the proof of Theorem 10.2 is

$$h \;=\; [\![\mathcal{Q} \mathit{length} \colon \mathcal{Q} \mathit{nums} \colon \mathsf{Sum}\big[v_{\text{sum}}, \big[\mathsf{StateSequence}_{\mathrm{x}}\,(v_{\text{sum}}, \mathit{length})\big] \odot \mathsf{Path}\,[f]\,(\mathit{length}, v_{\text{sum}})\,, \mathit{nums}\big]\,]\!]\,,$$

where the syntactic expectation $\mathsf{Path}\,[f]\,(\mathit{length}, v_2)$ is defined as follows:

$$\begin{aligned}&[\mathit{length} < 2] \cdot (\mathcal{Q}\mathit{num} \colon\; [\mathsf{Elem}\,(v_{\text{sum}}, \mathit{length} - 1, \mathit{num})] \odot \mathsf{Subst}_{\mathrm{x}}\,[([\neg(c = 1)] \cdot x), \mathit{num}])\\ &+ [\mathit{length} \geq 2] \cdot (\mathcal{Q}\mathit{num} \colon\; [\mathsf{Elem}\,(v_{\text{sum}}, \mathit{length} - 1, \mathit{num})] \odot \mathsf{Subst}_{\mathrm{x}}\,[([\neg(c = 1)] \cdot x), \mathit{num}])\\ &\quad \odot \mathsf{Product}\big(\mathcal{Q}\mathit{num}_1 \colon \mathcal{Q}\mathit{num}_2 \colon\; \big[\mathsf{Elem}\,\big(v_{\text{sum}}, v_{\text{prod}}, \mathit{num}_1\big) \wedge \mathsf{Elem}\,\big(v_{\text{sum}}, v_{\text{prod}} + 1, \mathit{num}_2\big)\big]\\ &\qquad \odot \mathsf{Subst}_{\mathrm{x}}\,[\mathsf{Subst}_{\mathrm{x}'}\,[g, \mathit{num}_2]\,, \mathit{num}_1]\,, \mathit{length} - 2\big)\end{aligned}$$

and where

$$g \;=\; [c = 1] \cdot \frac{1}{2} \cdot ([0 = c' \wedge x + 1 = x'] + [1 = c' \wedge x + 1 = x']) + [\neg(c = 1)] \cdot [c = c' \wedge x = x']\;.$$

We omit unfolding h further. Although our general construction yields rather complex syntactic preexpectations, notice we can express $\mathsf{wp}[\![\texttt{while}\,(\,c = 1\,)\,\{\,C'\,\}]\!]\,(\,[\![x]\!])$ much more concisely as

$$x \;+\; [c = 1] \cdot 2 \quad\in\quad \mathsf{Exp}\,.$$

11 ON NEGATIVE NUMBERS

Throughout the paper, we have evaded supporting negative numbers in two aspects:

(1) In our *verification system*—the weakest preexpectation calculus—we allow expectations, both syntactic and semantic, to map program states to *non-negative* values in $\mathbb{R}^{\infty}_{\geq 0}$ only.
(2) In our *programming language*, we allow variables to assume *non-negative* values in $\mathbb{Q}_{\geq 0}$ only.

While the former restriction is fairly standard in the literature on probabilistic programs (cf. [McIver and Morgan 2005]), considering only unsigned program variables is less common. An attentive reader may thus ask whether our completeness results rely on the above restrictions. In this section, we briefly comment on our reasons for considering only non-negative numbers. Moreover, we discuss how one *could* incorporate support for negative numbers in both of the above aspects.

11.1 Signed Expectations

There exist approaches that support signed expectations, which allow arbitrary reals in their codomain. However, as working with signed expectations may lead to integrability issues, these approaches require a significant technical overhead (cf. [Kaminski and Katoen 2017] for details). Moreover, proof rules for loops become much more involved. Calculi like Kozen's PPDL *in principle* allow signed expectations off-the-shelf, but PPDL's induction rule for loops is restricted to non-negative expectations as well [Kozen 1983]. We thus opted for the more common approach of considering only unsigned expectations. An alternative is to perform a *Jordan decomposition* on the expectation (i.e., decomposing it into positive and negative parts) and then reason individually about the positive and the negative part. As outlined below, such a decomposition can already be performed on program level *without* changing the verification system.

11.2 Signed Program Variables

Omitting negative numbers does *not* affect our results because they can easily be encoded in our (Turing complete) programming language: we can emulate signed variables, for instance, by splitting each variable x into two variables $|x|$ and x_{sgn}, representing the absolute value of x and its sign ($x_{sgn} = 1$ if x negative, and $x_{sgn} = 0$ otherwise), respectively. With this convention, the program below emulates the subtraction assignment $z := x - y$ using only addition and monus:

```
if (x_sgn = y_sgn) {                          // calculuate magnitude of z
    |z| := (|x| ∸ |y|) + (|y| ∸ |x|)
} else {
    |z| := |x| + |y|
};
if (|x| > |y|) {                              // calculuate sign of z
    z_sgn := x_sgn
} else {
    if (|x| = |y|) {
        z_sgn := 0
    } else {
        z_sgn := 1 ∸ y_sgn
    }
}
```

Similar emulations can be performed for addition, multiplication, etc. For the purpose of proving relative completeness, signed variables are thus syntactic sugar; we omit them for simplicity.

Our main reason for disallowing negative numbers as values of program variables is that we want x to be a valid (unsigned) expectation. If x was signed, it would not be a valid expectation as it does not map only to non-negative values. In order to fix this problem to some extent, one would have to "make x non-negative", e.g., by instead using the expectation $[x \geq 0] \cdot x$ (x truncated at 0) or the expectation $|x|$ (absolute value of x; not supported (but can be encoded) in our current syntax). However, neither of the above expectations actually represents "the value of x".

12 DISCUSSION

We now discuss a few aspects in which our expressive language Exp of expectations could be useful.

12.1 Relative Completeness of Probabilistic Program Verification

An immediate consequence of Theorem 10.2 is that, for all pGCL programs C and all syntactic expectations $f, g \in \mathsf{Exp}$, verifying the bounds

$$[\![g]\!] \preceq \mathrm{wp}[\![C]\!]([\![f]\!]) \qquad \text{or} \qquad \mathrm{wp}[\![C]\!]([\![f]\!]) \preceq [\![g]\!]$$

reduces to *checking a single inequality* between two syntactic expectations in Exp, namely g and the *effectively constructible expectation* for $\mathrm{wp}[\![C]\!]([\![f]\!])$. In that sense, the wp calculus together with Exp form a *relatively complete* (cf. [Cook 1978]) system for probabilistic program verification. Given an oracle for discharging inequalities between syntactic expectations, every correct inequality of the above form can be derived.

12.2 Termination Probabilities

For each probabilistic program C, the weakest preexpectation

$$\mathrm{wp}[\![C]\!](1)$$

is a mapping from initial state σ to the *probability that C terminates on σ*. Since $1 \in \mathsf{Exp}$, *termination probabilities of any* pGCL *program on any input are expressible in our syntax.*

This demonstrates that our syntax is capable of capturing mappings from states to numbers that are *far from trivial* as termination probabilities in general carry a *high degree of internal complexity* [Kaminski and Katoen 2015; Kaminski et al. 2019]. More concretely, given C, σ, and α, deciding whether C terminates on σ *at least* with probability α is Σ^0_1–complete in the arithmetical hierarchy. Deciding whether C terminates on σ *at most* with probability α is even Π^0_2–complete, thus strictly harder than, e.g., the universal termination problem for non-probabilistic programs.

12.3 Probability to Terminate in Some Postcondition

For a probabilistic program C and a first-order predicate $[\varphi]$, the weakest preexpectation

$$\mathrm{wp}[\![C]\!]([\varphi])$$

is a mapping from initial state σ to the *probability that C terminates on σ in a state $\tau \models \varphi$*. Since $[\varphi]$ is expressible in Exp, we have that $\mathrm{wp}[\![C]\!]([\varphi])$ is also expressible in Exp by expressivity of Exp. We can thus embed *and generalize Dijkstra's weakest preconditions completely in our system.*

12.4 Distribution over Final States

Let C be a probabilistic program in which only the variables $x_1, \ldots, x_k$ occur. Moreover, let μ^σ_C be the final distribution obtained by executing C on input σ, cf. Section 2.1.3. Then, by the Kozen duality (cf. Theorem 2.1), we can express the probability $\mu^\sigma_C(\tau)$ of C terminating in final state τ on initial state σ, where $\tau(x_i) = x_i'$, by

$$\mu^\sigma_C(\tau) \;=\; \mathrm{wp}[\![C]\!]\left(\left[x_1 = x_1' \wedge \cdots \wedge x_k = x_k'\right]\right)(\sigma)\,.$$

Intuitively, we can write the initial values of $x_1, \ldots, x_k$ into $\sigma(x_1), \ldots, \sigma(x_k)$ and the final values into $\sigma(x_1'), \ldots, \sigma(x_k')$.

Since $\left[x_1 = x_1' \wedge \cdots \wedge x_k = x_k'\right] \in \mathsf{Exp}$, we have that $\mathrm{wp}[\![C]\!]\left(\left[x_1 = x_1' \wedge \cdots \wedge x_k = x_k'\right]\right)$ is expressible in Exp as well. Hence, *we can express Kozen's measure transformers in our syntax.*

12.5 Ranking Functions / Supermartingales

There is a plethora of methods for proving termination of probabilistic programs based on ranking supermartingales [Chakarov and Sankaranarayanan 2013; Chatterjee et al. 2016b, 2017; Fioriti and Hermanns 2015; Fu and Chatterjee 2019; Huang et al. 2018, 2019]. Ranking supermartingales are

similar to ranking functions, but one requires that the value decreases *in expectation*. Weakest preexpectations are the natural formalism to reason about this.

For algorithmic solutions, ranking supermartingales are often assumed to be, for instance, linear [Chatterjee et al. 2018] or polynomial [Chatterjee et al. 2016a; Ngo et al. 2018; Schreuder and Ong 2019]. This also applies to the allowed shape of templates for loop invariants in works [Feng et al. 2017; Katoen et al. 2010] on the automated synthesis of probabilistic loop invariants. *Functions linear or polynomial in the program variables are obviously subsumed by our syntax.* However, our syntax now enables searching for *wider* tractable classes.

12.6 Harmonic Numbers

Harmonic numbers are ubiquitous in reasoning about expected values or expected runtimes of randomized algorithms. They appear, for instance, as the expected runtime of Hoare's randomized quicksort or the coupon collector problem, or as ranking functions for proving almost-sure termination [Kaminski 2019; Kaminski et al. 2018; McIver et al. 2018; Olmedo et al. 2016]. Harmonic numbers are syntactically expressible in our language as in Example 7.8, or more conveniently as

$$H_x \;=\; \left[\!\left[\mathsf{Sum}\left[v_{\mathrm{sum}}, \tfrac{1}{v_{\mathrm{sum}}}, x\right]\right]\!\right], \qquad \text{where } \tfrac{1}{v_{\mathrm{sum}}} \;=\; \partial z\colon\ [z \cdot v_{\mathrm{sum}} = 1] \cdot z\,.$$

We note that, in termination proofs, the Harmonic numbers do *not* occur as termination probabilities, but rather *in ranking functions* whose expected values after one loop iteration need to be determined. Our syntax is capable of handling such ranking functions and we could safely add H_x to our syntax.

13 CONCLUSION AND FUTURE WORK

We have presented a *language of syntactic expectations* that is *expressive for weakest preexpectations* of probabilistic programs á la Kozen [1985] and McIver and Morgan [2005]. As a consequence, verification of bounds on expected values of functions (expressible in our language) after probabilistic program execution is *relatively complete* in the sense of Cook [1978].

We have discussed various scenarios covered by our language, such as reasoning about termination probabilities, thus demonstrating the language's usefulness.

Open Problems. We currently do not support probabilistic programs with (binary) *non-deterministic* choices, as do McIver and Morgan [2005], and it is not obvious how to incorporate it, given our current encoding. What seems even more out of reach is handling *unbounded non-determinism*, which would be needed, for instance, to come up with an expressive expectation language for *quantitative separation logic* (QSL)—an (extensional) verification system for compositional reasoning about probabilistic pointer programs with access to a heap [Batz et al. 2019; Matheja 2020].

For non-probabilistic heap-manipulating programs, a topic considered by Tatsuta et al. [2019] are inductive definitions of predicates in classical separation logic (SL) and proving that SL is expressive in this context. QSL also features inductive definitions and it would be an interesting endeavor to consider expressiveness in this setting.

Despite its similarity to the wp calculus, we did not consider the *expected runtime calculus* (ert) by Kaminski et al. [2018]. We strongly conjecture that Exp is expressive for expected runtimes as well.

Finally, the *conditional weakest preexpectation* calculus (cwp) [Kaminski 2019; Olmedo et al. 2018] for probabilistic programs with *conditioning* needs weakest *liberal* preexpectations, which generalize Dijkstra's weakest liberal preconditions. It currently remains open, whether $\mathsf{wlp}[\![C]\!]\,(f)$ is expressible in Exp. There is the duality $\mathsf{wlp}[\![C]\!]\,(f) = 1 - \mathsf{wp}[\![C]\!]\,(1-f)$, originally due to Kozen [1983], but it is not immediate how to express $1-f$ in Exp, if f is not a plain arithmetic expression.

REFERENCES

Krzysztof R. Apt and Ernst-Rüdiger Olderog. 2019. Fifty years of Hoare's logic. *Formal Asp. Comput.* 31, 6 (2019), 751–807. https://doi.org/10.1007/s00165-019-00501-3

Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2020. Relatively Complete Verification of Probabilistic Programs - Extended Version. *CoRR* (2020). to appear.

Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. 2019. Quantitative separation logic: a logic for reasoning about probabilistic pointer programs. *Proc. ACM Program. Lang.* 3, POPL (2019), 34:1–34:29.

Joseph Bertrand. 1849. *Traité d'Arithmétique.* Libraire de L. Hachette et Cie.

Olivier Bournez and Florent Garnier. 2005. Proving Positive Almost-Sure Termination. In *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3467)*, Jürgen Giesl (Ed.). Springer, 323–337. https://doi.org/10.1007/978-3-540-32033-3_24

Georg Cantor. 1878. Ein Beitrag zur Mannigfaltigkeitslehre. *Journal für die reine und angewandte Mathematik* 1878, 84 (1878), 242–258.

Aleksandar Chakarov and Sriram Sankaranarayanan. 2013. Probabilistic Program Analysis with Martingales. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 511–526.

Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2016a. Termination Analysis of Probabilistic Programs Through Positivstellensatz's. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9779)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 3–22.

Krishnendu Chatterjee, Hongfei Fu, Petr Novotný, and Rouzbeh Hasheminezhad. 2016b. Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 327–342.

Krishnendu Chatterjee, Hongfei Fu, Petr Novotný, and Rouzbeh Hasheminezhad. 2018. Algorithmic Analysis of Qualitative and Quantitative Termination Problems for Affine Probabilistic Programs. *ACM Trans. Program. Lang. Syst.* 40, 2 (2018), 7:1–7:45.

Krishnendu Chatterjee, Petr Novotný, and Dorde Zikelic. 2017. Stochastic invariants for probabilistic termination. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 145–160.

Stephen A. Cook. 1978. Soundness and Completeness of an Axiom System for Program Verification. *SIAM J. Comput.* 7 (1978), 70–90.

Patrick Cousot and Michael Monerau. 2012. Probabilistic Abstract Interpretation. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7211)*, Helmut Seidl (Ed.). Springer, 169–193.

Jerry den Hartog and Erik P. de Vink. 2002. Verifying Probabilistic Programs Using a Hoare Like Logic. *Int. J. Found. Comput. Sci.* 13, 3 (2002), 315–340.

Alessandra Di Pierro and Herbert Wiklicky. 2016. Probabilistic Abstract Interpretation: From Trace Semantics to DTMC's and Linear Regression. In *Semantics, Logics, and Calculi - Essays Dedicated to Hanne Riis Nielson and Flemming Nielson on the Occasion of Their 60th Birthdays (Lecture Notes in Computer Science, Vol. 9560)*, Christian W. Probst, Chris Hankin, and René Rydhof Hansen (Eds.). Springer, 111–139.

Edsger W. Dijkstra. 1976. *A Discipline of Programming.* Prentice-Hall. http://www.worldcat.org/oclc/01958445

Yijun Feng, Lijun Zhang, David N. Jansen, Naijun Zhan, and Bican Xia. 2017. Finding Polynomial Loop Invariants for Probabilistic Programs. In *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10482)*, Deepak D'Souza and K. Narayan Kumar (Eds.). Springer, 400–416. https://doi.org/10.1007/978-3-319-68167-2_26

Luis María Ferrer Fioriti and Holger Hermanns. 2015. Probabilistic Termination: Soundness, Completeness, and Compositionality. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 489–501.

Philippe Flajolet, Maryse Pelletier, and Michèle Soria. 2011. On Buffon Machines and Numbers. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011, San Francisco, California, USA, January 23-25, 2011*, Dana Randall (Ed.). SIAM, 172–183. https://doi.org/10.1137/1.9781611973082.15

Robert W Floyd. 1967. Assigning Meanings to Programs. In *Mathematical Aspects of Computer Science*, J.T. Schwarz (Ed.), Vol. 19. American Mathematical Society, 19–32.

Hongfei Fu and Krishnendu Chatterjee. 2019. Termination of Nondeterministic Probabilistic Programs. In *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11388)*, Constantin Enea and Ruzica Piskac (Eds.). Springer, 468–490.

Kurt Gödel. 1931. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik* 38, 1 (1931), 173–198.

Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. 2014. Operational versus weakest pre-expectation semantics for the probabilistic guarded command language. *Perform. Evaluation* 73 (2014), 110–132. https://doi.org/10.1016/j.peva.2013.11.004

Sergiu Hart, Micha Sharir, and Amir Pnueli. 1982. Termination of Probabilistic Concurrent Programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982*, Richard A. DeMillo (Ed.). ACM Press, 1–6.

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580.

Mingzhang Huang, Hongfei Fu, and Krishnendu Chatterjee. 2018. New Approaches for Almost-Sure Termination of Probabilistic Programs. In *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11275)*, Sukyoung Ryu (Ed.). Springer, 181–201.

Mingzhang Huang, Hongfei Fu, Krishnendu Chatterjee, and Amir Kafshdar Goharshady. 2019. Modular verification for almost-sure termination of probabilistic programs. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 129:1–129:29.

Kenneth E. Iverson. 1962. *A Programming Language*. John Wiley & Sons, Inc., USA.

Benjamin Lucien Kaminski. 2019. *Advanced weakest precondition calculi for probabilistic programs*. Ph.D. Dissertation. RWTH Aachen University, Germany. http://publications.rwth-aachen.de/record/755408

Benjamin Lucien Kaminski and Joost-Pieter Katoen. 2015. On the Hardness of Almost-Sure Termination. In *Mathematical Foundations of Computer Science 2015 - 40th International Symposium, MFCS 2015, Milan, Italy, August 24-28, 2015, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9234)*, Giuseppe F. Italiano, Giovanni Pighizzini, and Donald Sannella (Eds.). Springer, 307–318.

Benjamin Lucien Kaminski and Joost-Pieter Katoen. 2017. A weakest pre-expectation semantics for mixed-sign expectations. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. IEEE Computer Society, 1–12.

Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2019. On the hardness of analyzing probabilistic programs. *Acta Informatica* 56, 3 (2019), 255–285.

Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. 2018. Weakest Precondition Reasoning for Expected Runtimes of Randomized Algorithms. *J. ACM* 65, 5 (2018), 30:1–30:68.

Joost-Pieter Katoen, Annabelle McIver, Larissa Meinicke, and Carroll C. Morgan. 2010. Linear-Invariant Generation for Probabilistic Programs: - Automated Support for Proof-Based Methods. In *SAS (Lecture Notes in Computer Science, Vol. 6337)*. Springer, 390–406.

Stephen Cole Kleene, NG De Bruijn, J de Groot, and Adriaan Cornelis Zaanen. 1952. *Introduction to Metamathematics*. Vol. 483. van Nostrand New York.

Dexter Kozen. 1979. Semantics of Probabilistic Programs. In *20th Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 29-31 October 1979*. IEEE Computer Society, 101–114.

Dexter Kozen. 1981. Semantics of Probabilistic Programs. *J. Comput. Syst. Sci.* 22, 3 (1981), 328–350. https://doi.org/10.1016/0022-0000(81)90036-2

Dexter Kozen. 1983. A Probabilistic PDL. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA*. ACM, 291–297. https://doi.org/10.1145/800061.808758

Dexter Kozen. 1985. A Probabilistic PDL. *J. Comput. Syst. Sci.* 30, 2 (1985), 162–178. https://doi.org/10.1016/0022-0000(85)90012-1

Dexter Kozen. 2000. On Hoare logic and Kleene algebra with tests. *ACM Trans. Comput. Log.* 1, 1 (2000), 60–76.

Dexter Kozen and Jerzy Tiuryn. 2001. On the completeness of propositional Hoare logic. *Inf. Sci.* 139, 3-4 (2001), 187–195.

Jacques Loeckx and Kurt Sieber. 1987. *The Foundations of Program Verification, 2nd ed.* Wiley-Teubner.

J. Loeckx, K. Sieber, and R.D. Stansifer. 1984. *The Foundations of Program Verification*. John Wiley. https://books.google.de/books?id=wagmAAAAMAAJ

Christoph Matheja. 2020. *Automated Reasoning and Randomization in Separation Logic*. Ph.D. Dissertation. RWTH Aachen University, Aachen. https://doi.org/10.18154/RWTH-2020-00940

Annabelle McIver and Carroll Morgan. 2005. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer. https://doi.org/10.1007/b138392

Annabelle McIver, Carroll Morgan, Benjamin Lucien Kaminski, and Joost-Pieter Katoen. 2018. A new proof rule for almost-sure termination. *Proc. ACM Program. Lang.* 2, POPL (2018), 33:1–33:28.

Rajeev Motwani and Prabhakar Raghavan. 1999. Randomized Algorithms. In *Algorithms and Theory of Computation Handbook*, Mikhail J. Atallah (Ed.). CRC Press. https://doi.org/10.1201/9781420049503-c16

Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2018. Bounded expectations: resource analysis for probabilistic programs. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 496–512.

Federico Olmedo, Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Annabelle McIver. 2018. Conditioning in Probabilistic Programming. *ACM Trans. Program. Lang. Syst.* 40, 1 (2018), 4:1–4:50.

Federico Olmedo, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2016. Reasoning about Recursive Probabilistic Programs. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, Martin Grohe, Eric Koskinen, and Natarajan Shankar (Eds.). ACM, 672–681.

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 55–74.

Julia Robinson. 1949. Definability and Decision Problems in Arithmetic. *J. Symb. Log.* 14, 2 (1949), 98–114.

Anne Schreuder and C.-H. Luke Ong. 2019. Polynomial Probabilistic Invariants and the Optional Stopping Theorem. *CoRR* abs/1910.12634 (2019).

Makoto Tatsuta, Wei-Ngan Chin, and Mahmudul Faisal Al Ameen. 2009. Completeness of Pointer Program Verification by Separation Logic. In *Software Engineering and Formal Methods*. IEEE Computer Society, 179–188.

Makoto Tatsuta, Wei-Ngan Chin, and Mahmudul Faisal Al Ameen. 2019. Completeness and expressiveness of pointer program verification by separation logic. *Inf. Comput.* 267 (2019), 1–27.

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