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 $\text{wp}[[C]](f)$) 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 $g \preceq \text{wp}[[C]](f)$.

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:

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.

This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2021 Copyright held by the owner/author(s).
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 wp[C](f) is a mapping from (initial) states to numbers, such that

\[ \text{wp}[C](f)(\sigma) = \text{Expected value of } f, \text{ measured in final states reached} \]
\[ \text{after termination of } C \text{ on initial state } \sigma. \]

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 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 $\text{wp}$ calculus relatively complete [Cook 1978]: Given functions $f$ and $g$ in our language and a probabilistic program $C$, suppose we want to verify $g \leq \text{wp}[C](f)$, where $\leq$ 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 $\text{wp}[C](f)$. Hence, verification is complete modulo checking whether the inequality $g \leq h$ 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, $\text{wp}[C](0) = 0$. 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 $\text{wp}[C](1)$—the expected value of the constant function 1). These can be irrational numbers like the reciprocal of the golden ratio $\frac{1}{\phi}$ [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 $\pi$, $\sqrt{3}$ 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 $\text{wp}[C](f)$, 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 à 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

$C \rightarrow \text{skip} \quad \text{(effectless program)}$

$\mid x := a \quad \text{(assignment)}$

$\mid C; C \quad \text{(sequential composition)}$

$\mid \{ C \} [p] \{ C \} \quad \text{(probabilistic choice)}$

$\mid \text{if} (\varphi) \{ C \} \text{else} \{ C \} \quad \text{(conditional choice)}$

$\mid \text{while}(\varphi)\{ C \} \quad \text{(while loop)}$

where $x$ is taken from a countably infinite set of variables $\text{Vars}$, $a$ is an arithmetic expression over variables, $p \in [0, 1] \cap \mathbb{Q}$ is a rational probability, and $\varphi$ is a Boolean expression (also called guard) over variables. For an overview of metavariables $C, x, a, \varphi, \ldots$, 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 $\sigma$ maps each variable in $\text{Vars}$ to its value—a positive rational number in $\mathbb{Q}_{\geq 0}$.

To ensure that the set of program states is countable, 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 $\Sigma$ of program states is

$$\Sigma = \{ \sigma : \text{Vars} \rightarrow \mathbb{Q}_{\geq 0} \mid \{ x \in \text{Vars} \mid \sigma(x) \neq 0 \} \text{ is finite} \} .$$

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.

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 $\sigma$, 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 $\sigma$ into a probability distribution $\mu^C_\sigma$ over final states (i.e., a measure on $\Sigma$). 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 $\varphi$ is satisfied; otherwise, it executes $C_2$. Finally, the loop $while (\varphi) \{ C \}$ keeps executing the loop body $C$ as long as $\varphi$ 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 $E$ of semantic expectations is defined as

$$E = \{ X \mid X : \Sigma \rightarrow R_{\geq 0}^\infty \},$$

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 $\Sigma$ to $R_{\geq 0}^\infty$. It is hence borderline to speak of a language. The goal of this paper is to provide a syntactically defined subclass of $E$—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 $\alpha \in R_{\geq 0}$ such that $\forall \sigma : X(\sigma) \leq \alpha$. In contrast to McIver and Morgan, our structure $(E, \preceq)$ of unbounded expectations forms a complete lattice with least element 0 and greatest element $\infty$, where $\preceq$ lifts the standard ordering $\leq$ on the (extended) reals to expectations by pointwise application. That is,

$$X \preceq Y \quad \text{iff} \quad \forall \sigma \in \Sigma : X(\sigma) \leq Y(\sigma).$$
As the [1985] where we point-wise lifted common operators on the reals, such as \( wp \)
\( \sigma \) with respect to the final distribution over states reached after termination of program \( C \) on input \( \sigma \). \( wp[C] \) is backward-moving in the sense that it transforms an \( X: \Sigma \to \mathbb{R}^{\geq 0} \), evaluated in final states after termination of \( C \), into \( wp[C](X): \Sigma \to \mathbb{R}^{\geq 0} \), evaluated in initial states before execution of \( C \).

The weakest preexpectation calculus is a backward semantics in the sense that it transforms a

\[ \expct{\sigma}{X} \]

of \( C \) with respect to the final distribution over states reached after termination of program \( C \) on input \( \sigma \). \( wp[C] \) is backward-moving in the sense that it transforms an \( X: \Sigma \to \mathbb{R}^{\geq 0} \), evaluated in final states after termination of \( C \), into \( wp[C](X): \Sigma \to \mathbb{R}^{\geq 0} \), 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 \( \varphi \) its indicator function:

\[ [\varphi] = \lambda \sigma. \begin{cases} 1, & \text{if } \sigma[\varphi] = \text{true} \\ 0, & \text{if } \sigma[\varphi] = \text{false} \end{cases} \]

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

\[ x + y = \lambda \sigma. \sigma(x) + \sigma(y) \]

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.,

\[ X[x/a] = \lambda \sigma. X(\sigma[x \mapsto \sigma[a]]) \]

where \( \sigma[x \mapsto r] = \begin{cases} r, & \text{if } y = x \\ \sigma(y), & \text{else} \end{cases} \).

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 \( wp[C](X) \) [McIver and Morgan 2005]. As the expected value of \( X \) generally depends on the initial state \( \sigma \) on which \( C \) is executed, the weakest preexpectation \( wp[C](X) \) is itself also a map of type \( \mathbb{E} \), mapping an initial program state \( \sigma \) to the expected value of \( X \) (measured in the final states) after successful termination of \( C \) on \( \sigma \), 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 \( wp[C](X) \in \mathbb{E} \), 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 \( \mu^\sigma_C \) is the distribution over final states obtained by running \( C \) on initial state \( \sigma \), then for any postexpectation \( X \),

\[ \sum_{r \in \Sigma} \mu^\sigma_C(r) \cdot X(r) = wp[C](X)(\sigma) \]

---

We use \( \lambda \)-expressions to denote functions; function \( \lambda x. f \) applied to \( a \) evaluates to \( f \) in which \( x \) is replaced by \( a \).
where the characteristic function $X$ weakest preexpectation with respect to $C$ which is defined recursively on the structure of preexpectation $wp$

These rules are standard: $wp$ is a complete lattice and $0$ is the constant-zero expectation and $wp$·.

Kleene’s theorem then allows us to approximate the least fixed point iteratively:

$$ \text{lfp } Y. \lnot \phi \cdot X + [\phi] \cdot wp[C'](Y) $$

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

$$ wp : \text{pGCL} \rightarrow (\mathbb{E} \rightarrow \mathbb{E}) $$

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

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

$$ wp[\text{while } (\phi) \{C'\}](X) = \text{lfp } Y. \Phi_X(Y), $$

where the characteristic function $\Phi_X$ of $\text{while } (\phi) \{C'\}$ with respect to $X \in \mathbb{E}$ is defined as

$$ \Phi_X : \mathbb{E} \rightarrow \mathbb{E}, \ Y \mapsto [\lnot \phi] \cdot X + [\phi] \cdot wp[C'](Y) . $$

Since $(\mathbb{E}, \leq)$ is a complete lattice and $\Phi_X$ 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 $\Phi_X$ 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

$$ wp[\text{while } (\phi) \{C'\}](X) = \text{lfp } Y. \Phi_X(Y) = \sup_{n \in \mathbb{N}} \Phi_X^n(0), $$

where $0 = \lambda \sigma. 0$ is the constant-zero expectation and $\Phi_X^n(Y)$ denotes the $n$-fold application of $\Phi_X$ to $Y$.
Table 1. Metavariables used throughout this paper.

<table>
<thead>
<tr>
<th>Entities</th>
<th>Metavariables</th>
<th>Domain</th>
<th>Defined</th>
</tr>
</thead>
<tbody>
<tr>
<td>Natural numbers</td>
<td>n, i, j, k</td>
<td>(\mathbb{N})</td>
<td></td>
</tr>
<tr>
<td>Positive rationals</td>
<td>r, s, t</td>
<td>(\mathbb{Q}_{\geq 0})</td>
<td></td>
</tr>
<tr>
<td>Positive extended reals</td>
<td>(\alpha, \beta, \gamma)</td>
<td>(\mathbb{R}_{\geq 0})</td>
<td></td>
</tr>
<tr>
<td>Rational probabilities</td>
<td>p, q</td>
<td>([0, 1] \cap \mathbb{Q})</td>
<td></td>
</tr>
<tr>
<td>Variables</td>
<td>x, y, z, v, w, u, num</td>
<td>Vars</td>
<td>Section 2.1</td>
</tr>
<tr>
<td>Arithmetic expressions</td>
<td>a, b</td>
<td>AExpr</td>
<td>Section 4.1</td>
</tr>
<tr>
<td>Boolean expressions</td>
<td>(\varphi, \psi, \xi)</td>
<td>Bool</td>
<td>Section 4.2</td>
</tr>
<tr>
<td>Syntactic expectations</td>
<td>f, g, h</td>
<td>Exp</td>
<td>Section 4.3</td>
</tr>
<tr>
<td>Semantic expectations</td>
<td>X, Y, Z</td>
<td>E</td>
<td>Section 2.2.1</td>
</tr>
<tr>
<td>Programs</td>
<td>C</td>
<td>pGCL</td>
<td>Section 2.1</td>
</tr>
<tr>
<td>Program states</td>
<td>(\sigma, \tau)</td>
<td>(\Sigma)</td>
<td>Section 2.1.2</td>
</tr>
</tbody>
</table>

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 \(\mathrm{wp}[C] (X) \in \mathbb{E}\) 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 \(\mathrm{Exp}\) 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 \(\llbracket f \rrbracket\) 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 \(\mathrm{Exp}\), is the weakest preexpectation \(\mathrm{wp}[C] (\llbracket f \rrbracket)\) again expressible in \(\mathrm{Exp}\)?

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

\[
\mathrm{wp}[C] (\llbracket f \rrbracket) = \llbracket g \rrbracket.
\]

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

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

\(\triangle\)

---

5\(\mathrm{wp}\) being strict means that \(\mathrm{wp}[C] (0) = 0\) for every \(C\), see [Kaminski 2019].
4 SYNTACTIC EXPECTATIONS

We now describe the syntax and semantics for a set $\text{Exp}$ 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 $\varphi$, 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 $\text{AExpr}$ of arithmetic expressions is given by

$$a \rightarrow r \in \mathbb{Q}_{\geq 0} \quad \text{(non-negative rationals)}$$
$$\mid x \in \text{Vars} \quad \text{($\mathbb{Q}_{\geq 0}$-valued variables)}$$
$$\mid a + a \quad \text{(addition)}$$
$$\mid a \cdot a \quad \text{(multiplication)}$$
$$\mid a \div a \quad \text{(subtraction truncated at 0 (“monus”))}$$

where $\text{Vars}$ 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 $\text{AExpr}$, 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 $\text{Bool}$ of Boolean expressions is given by

$$\varphi \rightarrow a < a \quad \text{(strict inequality of arithmetic expressions)}$$
$$\mid \varphi \land \varphi \quad \text{(conjunction)}$$
$$\mid \neg \varphi \quad \text{(negation)}$$

We use metavariables $\varphi$, $\psi$, $\xi$ for Boolean expressions, see also Table 1.

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

false, true, $\varphi \lor \psi$, $\varphi \rightarrow \psi$, $a = b$, and $a \leq b$.

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 $\text{Exp}$ of syntactic expectations is given by

$$f \rightarrow a \quad \text{(arithmetic expressions)}$$
$$\mid [\varphi] \cdot f \quad \text{(guarding)}$$
$$\mid f + f \quad \text{(addition)}$$
$$\mid a \cdot f \quad \text{(scaling by arithmetic expressions)}$$
$$\mid \Sigma x : f \quad \text{(supremum over } x \text{)}$$
$$\mid \xi x : f \quad \text{(infimum over } x \text{)}$$

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.
4.4 Semantics of Expressions and Expectations

The semantics of arithmetic and Boolean expressions is standard—see Table 2. For a program state $\sigma$, the semantics of arithmetic expressions $a$ and Boolean expressions $\varphi$.

<table>
<thead>
<tr>
<th>$a$</th>
<th>$\sigma[a]$</th>
<th>$\varphi$</th>
<th>$\sigma[\varphi] = \text{true}$ iff</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r$ ($\in \mathbb{Q}_{\geq 0}$)</td>
<td>$r$</td>
<td>$a &lt; b$</td>
<td>$\sigma[a] &lt; \sigma[b]$</td>
</tr>
<tr>
<td>$x$ ($\in \text{Vars}$)</td>
<td>$\sigma(x)$</td>
<td>$\psi \land \xi$</td>
<td>$\sigma[\psi] = \text{true} = \sigma[\xi]$</td>
</tr>
<tr>
<td>$b + c$</td>
<td>$\sigma[b] + \sigma[c]$</td>
<td>$\neg \psi$</td>
<td>$\sigma[\psi] = \text{false}$</td>
</tr>
<tr>
<td>$b \cdot c$</td>
<td>$\sigma[b] \cdot \sigma[c]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$b \div c$</td>
<td>$\begin{cases} \sigma[b] - \sigma[c], &amp; \text{if } \sigma[b] \geq \sigma[c] \ 0, &amp; \text{else} \end{cases}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 $\wp[x := a](x) = a$—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,

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

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 $\exists x \colon f$ and $\forall x \colon f$ take over the role of the $\exists$ and $\forall$ quantifiers of first-order logic. We use them to bind variables $x$. The $\exists$ and $\forall$ 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 $\exists$ and $\forall$ have the least precedence.

The set of free variables $\text{FV}(f) \subseteq \text{Vars}$ is the set of all variables that occur syntactically in $f$ and that are not in the scope of some $\exists$ or $\forall$ 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 $x \in \text{FV}(f)$, 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 $f(\ldots, x_i/a)$.
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 \(\sigma\) is an extended positive real (i.e., a positive real number or \(\infty\)) defined inductively as follows:

\[
\sigma[a] \triangleq \sigma[a] \tag{6}
\]

\[
\sigma[\varphi \cdot f] \triangleq \begin{cases} \sigma[f], & \text{if } \sigma[\varphi] = \text{true} \\ 0, & \text{else} \end{cases}
\]

\[
\sigma[f + g] \triangleq \sigma[f] + \sigma[g]
\]

\[
\sigma[a \cdot f] \triangleq \sigma[a] \cdot \sigma[f]
\]

\[
\sigma[\exists x: f] \triangleq \sup \{ \sigma[x \mapsto r][f] \mid r \in \mathbb{Q}_{\geq 0} \}
\]

\[
\sigma[\forall x: f] \triangleq \inf \{ \sigma[x \mapsto r][f] \mid r \in \mathbb{Q}_{\geq 0} \}
\]

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 \(\exists x:\ f\) quantification, for example, interprets \(f\) under all possible values of the bounded variable \(x\) and then returns the supremum of all these values. Analogously, \(\forall x:\ f\) 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 \(\exists\) or \(\forall\) possibly evaluates to an irrational number. For example, the expectation

\[
f = \exists x:\ [x \cdot x < 2] \cdot x,
\]

evaluates to \(\sqrt{2} \notin \mathbb{Q}_{\geq 0}\) under every state \(\sigma\).

The supremum of \(\emptyset\) is 0. Dually, the infimum of \(\emptyset\) is \(\infty\). The supremum of an unbounded set is \(\infty\).

We also note that our semantics can generate \(\infty\) only by using a \(\exists\) 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.,

\[
f \equiv g \quad \text{iff} \quad [f] = [g].
\]

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

\[
f \preceq g \quad \text{iff} \quad [f] \leq [g].
\]

\(^6\)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

\[
f = \bigodot_{i=1}^{k} x_i \cdot g,
\]

where \( \bigodot_i \in \{ \exists, \forall \} \) 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

\[
\sigma[f \cdot g] = \sigma[f] \cdot \sigma[g]
\]

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 \( \sigma[f \cdot g] = \sigma[g \cdot f] \).

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

\[
f = \mathcal{L} x : \frac{1}{x + 1} \quad \text{and} \quad g = \exists y : y.
\]

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 \( \exists z : [z \cdot (x + 1) = 1] \cdot z \). Clearly, we have \( \sigma[f] = 0 \) and \( \sigma[g] = \infty \) for all \( \sigma \), i.e., both \( f \) and \( g \) are constant expectations.

Let us now consider the product of \( f \) and \( g \). For all \( \sigma \), its semantics is given by

\[
\sigma[f \cdot g] = \sigma[f] \cdot \sigma[g] = 0 \cdot \infty = 0 = \infty \cdot 0 = \sigma[g] \cdot \sigma[f] = \sigma[g \cdot f].
\]

Now consider the following:

\[
\sigma[f \cdot g] = \sigma \left[ \left( \mathcal{L} x : \frac{1}{x + 1} \right) \cdot \left( \exists y : y \right) \right] \\
= \sigma \left[ \mathcal{L} x : \exists y : \frac{1}{x + 1} \cdot y \right] \quad \text{(by prenexing)} \\
= \inf \left\{ \sup \left\{ \frac{1}{r + 1} \cdot s \mid s \in \mathcal{Q}_{\geq 0} \right\} \mid r \in \mathcal{Q}_{\geq 0} \right\} \\
= \inf \{ \infty \mid r \in \mathcal{Q}_{\geq 0} \} \\
= \infty \neq 0 \\
= \sup \{ 0 \mid s \in \mathcal{Q}_{\geq 0} \} \\
= \sup \left\{ \inf \left\{ \frac{1}{r + 1} \cdot s \mid r \in \mathcal{Q}_{\geq 0} \right\} \mid s \in \mathcal{Q}_{\geq 0} \right\} \\
= \sup \left\{ \inf \left\{ s \cdot \frac{1}{r + 1} \mid r \in \mathcal{Q}_{\geq 0} \right\} \mid s \in \mathcal{Q}_{\geq 0} \right\} \quad \text{(by commutativity of \cdot in } \mathbb{R}_{\geq 0}^\infty \text{)} \\
= \sigma \left[ \exists y : \mathcal{L} x : y \cdot \frac{1}{x + 1} \right] \\
= \sigma \left[ \left( \exists y : y \right) \cdot \left( \mathcal{L} x : \frac{1}{x + 1} \right) \right] \quad \text{(by un-prenexing)} \\
= \sigma[\exists y : y \cdot \mathcal{L} x : \frac{1}{x + 1}].
\]
We see that $\exists y: \exists x: \frac{1}{x+1} \cdot y$ is a sound prenex normal form of $g \cdot f$ whereas $\exists x: \exists y: \frac{1}{x+1} \cdot y$ 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 $\text{Exp}$ 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.** $\text{Exp}$ is expressive (see Definition 3.1) for all loop-free pGCL programs $C$, i.e., for all $f \in \text{Exp}$ there exists a syntactic expectation $g \in \text{Exp}$, such that

$$\text{wp}[C](\llbracket f \rrbracket) = \llbracket g \rrbracket.$$ 

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 $\sigma, f$, and $a$,

$$\sigma[f[x/a]] = \sigma[x \mapsto \sigma(a)][f]$$

or equivalently

$$[f[x/a]] = [f][x/a]$$

**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 $f \in \text{Exp}$ 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 $\text{wp}[\text{skip}](\llbracket f \rrbracket) = [f]$ and $f \in \text{Exp}$ by assumption.

The assignment $x := a$. We have

$$\text{wp}[x := a](\llbracket f \rrbracket) = [f][x/a]$$

(by Lemma 5.2)

and $f[x/a] \in \text{Exp}$ 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 $g_1, g_2 \in \text{Exp}$, such that

$$\text{wp}[C_1](\llbracket f \rrbracket) = \llbracket g_1 \rrbracket \quad \text{and} \quad \text{wp}[C_2](\llbracket f \rrbracket) = \llbracket g_2 \rrbracket.$$ 

We then proceed with the compound loop-free programs:
The probabilistic choice \( \{ C_1 \} \{ p \} \{ C_2 \} \). We have
\[
\begin{align*}
\wp[\{ C_1 \} \{ p \} \{ C_2 \}](f) & = p \cdot \wp[C_1](f) + (1-p) \cdot \wp[C_2](f) \\
& = p \cdot [\varphi_1] + (1-p) \cdot [\varphi_2] \\
& = [p \cdot \varphi_1 + (1-p) \cdot \varphi_2] \\
\end{align*}
\]
(by definition of \( \wp \))

and \( p \cdot \varphi_1 + (1-p) \cdot \varphi_2 \in \text{Exp} \), see Section 4.3.

The conditional choice \( \text{if } (\varphi) \{ C_1 \} \text{ else } \{ C_2 \} \). We have
\[
\begin{align*}
\wp[\text{if } (\varphi) \{ C_1 \} \text{ else } \{ C_2 \}](f) & = [\varphi] \cdot \wp[C_1](f) + [\neg \varphi] \cdot \wp[C_2](f) \\
& = [\varphi] \cdot \varphi_1 + [\neg \varphi] \cdot \varphi_2 \\
& = \llbracket \varphi \cdot \varphi_1 + [\neg \varphi] \cdot \varphi_2 \rrbracket \\
& \quad \text{(pointwise addition and multiplication)}
\end{align*}
\]
(by definition of \( \wp \))

and \( [\varphi] \cdot \varphi_1 + [\neg \varphi] \cdot \varphi_2 \in \text{Exp} \), see Section 4.3.

Hence, \( \text{Exp} \) 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 \( \text{Exp} \) 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 \( \text{Exp} \) 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 \( C = \text{while}(\varphi)\{ C' \}, \)
\[
\forall f \in \text{Exp} \exists g \in \text{Exp} \quad \wp[\text{while}(\varphi)\{ C' \}](f) = \llbracket g \rrbracket, \quad (\dagger)
\]
where we already know by the I.H. that the same property holds for the loop body \( C' \), i.e.,
\[
\forall f' \in \text{Exp} \exists g' \in \text{Exp} \quad \wp[C'](f') = \llbracket g' \rrbracket. \quad (1)
\]

Remark (A Simplification for this Overview). Just for this overview section, we assume that the set \( \text{Vars} \) 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):

\[
\text{wp}[\sigma](X) = \lambda \sigma_0 \cdot \sum_{\tau \in \Sigma} X(\tau) \cdot \mu^\sigma_C(\tau),
\]

where \(\mu^\sigma_C\) is the probability distribution over final states obtained by running \(C\) on initial state \(\sigma_0\).

Adapting the above equality to our concrete case in which \(C\) is a loop and \(X = [f]\), we obtain

\[
\text{wp}[\text{while}(\phi)\{C'\}](\llbracket f \rrbracket) = \lambda \sigma_0 \cdot \sum_{\tau \in \Sigma} \llbracket \neg \phi \rrbracket \cdot f(\tau) \cdot \mu^\sigma_{\text{while}(\phi)\{C'\}}(\tau),
\]

where we strengthened the postexpectation \(f\) to \([\neg \phi] \cdot f\) to account for the fact that the loop guard \(\phi\) is violated in every final state, see [Kaminski 2019, Corollary 4.6, p. 85]. The main idea is—instead of viewing the whole distribution \(\mu^\sigma_{\text{while}(\phi)\{C'\}}\) 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

\[
C_{\text{iter}} = \text{if } (\phi) \{C'\} \text{ else } \text{skip}.
\]

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

\[
\text{wp}[\text{while}(\phi)\{C'\}](\llbracket f \rrbracket) = \lambda \sigma_0 \cdot \sup_{k \in \mathbb{N}} \sum_{\sigma_0, \ldots, \sigma_{k-1} \in \Sigma} \llbracket \neg \phi \rrbracket \cdot f(\sigma_{k-1}) \cdot \prod_{i=0}^{k-2} \mu^\sigma_{C_{\text{iter}}} (\sigma_{i+1}).
\]

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

\[
[\sigma] = \left\{ x \in \text{Vars} \mid : x = \sigma(x) \right\}.
\]

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

\[
\mu^\sigma_{C_{\text{iter}}} (\sigma_{i+1}) = \text{wp}[C_{\text{iter}}](\llbracket [\sigma_{i+1}] \rrbracket)(\sigma_i).
\]

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

\[
\mu^\sigma_{C_{\text{iter}}} (\sigma_{i+1}) = \text{wp}[C_{\text{iter}}](\llbracket [\sigma_{i+1}] \rrbracket)(\sigma_i) = \llbracket g^\sigma_{C_{\text{iter}}} \rrbracket(\sigma_i).
\]

---

\(^7\)Recall from our remark on simplification that \(\text{Vars}\) 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 (\dagger):

\[
wp\left[\textbf{while} (\phi) \{ C' \}\right](\lceil f \rceil) = \lambda \sigma_0 . \sum_{k \in \mathbb{N}} \sum_{\sigma_0, \ldots, \sigma_{k-1} \in \Sigma} \left[ \lceil \lnot \phi \cdot f \rceil \sigma_{k-1} \right] \cdot \prod_{i=0}^{k-2} \left[ g_{C_{\text{term}}}^{\sigma_{i+1}} \right] \sigma_i \]

\[
\text{non-constant product expressible in } \text{Exp}^? \]

\[
\text{simple product expressible in } \text{Exp}^? \]

\[
\text{non-constant sum over paths of length } k \text{ expressible in } \text{Exp}^? \]

\[
\exists k : \ldots \in \text{Exp} \]

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 \(g_{\text{sum}}(k) \in \text{Exp}\) for the entire sum that follows, we hence obtain an encoding of the whole expectation, namely

\[
\exists k : g_{\text{sum}}(k) \in \text{Exp}^? \]

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 \(h(\nu_{\text{sum}})\) that serves as a map from \(\nu_{\text{sum}}\) to individual summands, i.e., \(h[\nu_{\text{sum}}/i]\) yields the \(i\)-th summand.
- We construct a syntactic expectation \(\text{Sum} [\nu_{\text{sum}}, h, \nu]\) for partial sums, summing up the first \(\nu\) summands defined by the syntactic expectation \(h\)—see Theorem 9.2 for details.
### 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 $\prod_{i=1}^{k-1} g_{\text{iter}}^{\sigma_i}(x_i)$, 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 $\prod \left[ v_{\text{prod}}, h, v \right]$ 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 $g_{\text{iter}}^{\sigma_i}(x_i)$. 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; Loekx 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 $\varphi, \psi$ for Boolean expressions, $\sigma$ 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 $\mathcal{A}_{\geq 0}$ the set of formulas $P$ in first-order arithmetic over $\mathbb{Q}_{\geq 0}$, i.e., the extension of Boolean expressions $\varphi$ (see Section 4.2) by an existential quantifier $\exists x: P$ and a universal quantifier $\forall x: P$ with the usual semantics, e.g., $\sigma[\forall x: P] = \text{true}$ iff for all $r \in \mathbb{Q}_{\geq 0}$, $\sigma[x \mapsto r][P] = \text{true}$. The set $\mathcal{A}_{\mathbb{N}}$ of formulas $P$ in first-order arithmetic over $\mathbb{N}$ is defined analogously by restricting ourselves to (1) states $\sigma: \text{Vars} \to \mathbb{N}$ 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 normal form, 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

---

5Program 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.** \( N \) is definable in \( A_{\mathbb{Q}_{\geq 0}} \), i.e. there exists a formula \( N(x) \in A_{\mathbb{Q}_{\geq 0}} \), such that for all \( \sigma \),

\[ \sigma[N(x)] = \text{true} \quad \text{iff} \quad \sigma(x) \in N. \]

We use the above assertion \( N \) to first embed \( A_N \) in \( A_{\mathbb{Q}_{\geq 0}} \). Thereafter, we embed \( A_{\mathbb{Q}_{\geq 0}} \) in \( \text{Exp} \). Embedding a formula \( P \in A_N \) in \( A_{\mathbb{Q}_{\geq 0}} \) amounts to (1) asserting \( N(x) \) for every \( x \in \text{FV}(P) \) 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 \( P_{\mathbb{Q}_{\geq 0}} \in A_N \) be the embedding of \( P \in A_N \) as defined in Figure 3. Then, for all \( \sigma \),

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

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

**Theorem 7.3.** Let \([P] \in \text{Exp} \) be the embedding of \( P \in A_{\mathbb{Q}_{\geq 0}} \) as defined in Figure 4. Then, for all \( \sigma \),

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

Given \( P(v_1, \ldots, v_n) \in A_{\mathbb{Q}_{\geq 0}} \), 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 \( A_N \) in our language \( \text{Exp} \) 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 \( \text{Elem}(v_1, v_2, v_3) \in A_N \) (with quantifiers) satisfying: For every finite sequence of natural numbers \( n_0, \ldots, n_{k-1} \), there is a (Gödel) number \( \text{gnum} \in \mathbb{N} \) that encodes the sequence, i.e., for all \( i \in \{0, \ldots, k-1\} \) and all \( m \in \mathbb{N} \), it holds that

\[ \text{Elem}(\text{gnum}, i, m) \equiv \text{true} \quad \text{iff} \quad m = n_i. \]

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

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

\[ \text{Fac}(x) = \mathcal{E} v : \mathcal{E} \text{num} : v \cdot [\text{Elem}(\text{num}, 0, 1) \land \text{Elem}(\text{num}, x, v) \land \forall u \land \forall w : (u < x \land \text{Elem}(\text{num}, u, w) \rightarrow \text{Elem}(\text{num}, u + 1, w \cdot (u + 1)))]. \]

For every state \( \sigma \), the quantifier \( \mathcal{E} \text{num} \) selects a sequence \( n_0, n_1 \ldots \) satisfying \( n_{\sigma(x)} = \sigma(x)! \). The quantifier \( \mathcal{E} v \) 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 \( \sigma[\text{Fac}(x)] = \sigma(x)! \).

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

\[ \text{Sequence}(\text{num}, v) \triangleq (\forall u : u < v \rightarrow \exists w : \text{Elem}(\text{num}, u, w)) \land (\forall v' : (u < v \rightarrow \exists w : \text{Elem}(\text{num}, u, w) \land \text{Elem}(v', u, w)) \rightarrow \text{num}' \geq \text{num}) \]

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

\[
\text{Sequence} \left( \langle n_0, \ldots, n_{k-1} \rangle, k \right) \land \bigwedge_{i=0}^{k-1} \text{Elem} \left( \langle n_0, \ldots, n_{k-1} \rangle, i, n_i \right).
\]

### 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 $\text{Elem} \left( \text{num}, i, n \right)$ 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 $\text{Pair} \left( \psi_1, \psi_2, \psi_3 \right) \in \mathbb{A}_N$ satisfying: For every pair of natural numbers $(n_1, n_2)$, there is exactly one natural number $n$ such that

\[
\text{Pair} (n, n_1, n_2) \equiv \text{true}.
\]

**Theorem 7.7.** There is a formula $\text{RElem} \left( \psi_1, \psi_2, \psi_3 \right) \in \mathbb{A}_{\mathbb{Q}_{\geq 0}}$ satisfying: For every finite sequence $r_0, \ldots, r_{k-1} \in \mathbb{Q}_{\geq 0}$ there is a Gödel number $\text{num}$, such that for all $i \in \{0, \ldots, k-1\}$ and $s \in \mathbb{Q}_{\geq 0}$,

\[
\text{RElem} \left( \text{num}, i, s \right) \equiv \text{true} \iff s = r_i.
\]

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

\[
\text{Harmonic} \left( x \right) = \exists v : \exists \text{num} : v \cdot \left[ \text{RElem} \left( \text{num}, 0, 0 \right) \land \text{RElem} \left( \text{num}, x, v \right) \land \forall u : \forall w : (u < x \land \text{RElem} \left( \text{num}, u, w \right)) \rightarrow \exists w' : w' \cdot (u + 1) = 1 \land \text{RElem} \left( \text{num}, u + 1, w + w' \right) \right]
\]

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

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

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

### 7.4 Encoding Sequences of Program States

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

\[
\sigma_1 \sim_x \sigma_2 \quad \text{iff} \quad \forall x : \sigma_1(x) = \sigma_2(x).
\]
Every num satisfying Sequence (num, k) encodes exactly one state σ (modulo ∼x). The Gödel number encoding σ (w.r.t. x), which we denote by ⟨σ⟩x, is then the unique number satisfying
\[
\text{RSequence}((σ)_x, k) \wedge \bigwedge_{i=0}^{k-1} \text{RElem}(⟨σ⟩_x, i, σ(x_i)) .
\]
Notice that we implicitly fixed an ordering of the variables in x to identify each value stored in σ for a variable in x. The formula
\[
\text{EncodesState}_x (\text{num}) \triangleq \text{RSequence} (\text{num}, k) \wedge \bigwedge_{i=0}^{k-1} \text{RElem} (\text{num}, i, x_i)
\]
evaluates to true on state σ iff σ(num) is the Gödel number of a state σ′ with σ ∼x σ′. Now, let σ0, . . . , σn−1 be a sequence of states of length n. The Gödel number encoding σ0, . . . , σn−1 (w.r.t. x), which we denote by ⟨(σ0, . . . , σn−1)x⟩, is then the unique number satisfying
\[
\text{Sequence} ((⟨(σ0, . . . , σn−1)⟩_x, n) \wedge \bigwedge_{i=0}^{n-1} \text{Elem} ((⟨σ0, . . . , σn−1⟩)_x, i, ⟨σ⟩_x) .
\]
We are now in a position to encode sequences of states. The formula
\[
\text{StateSequence}_x (\text{num}, ν)
\]
\[
= \text{Sequence} (\text{num}, ν) \wedge (∃ν′: \text{Elem} (\text{num}, 0, ν′) \wedge \text{EncodesState}_x (ν′))
\]
\[
\wedge ∀u: ∀ν′: ((u < ν ∧ \text{Elem} (\text{num}, u, ν′)) \rightarrow \text{RSequence} (ν′, k))
\]
evaluates to true on state σ iff (1) num is the Gödel number of some sequence σ0, . . . , σ(σ′−1) ∈ Σ of states of length σ(ν) and where (2) σ and σ0 coincide on all variables in x, i.e., σ ∼x σ0. Notice that, for every sequence σ0, . . . , σn−1 of states of length n, there is exactly one num satisfying StateSequence_x (num, n). If clear from the context, we often omit the subscript x and simply write ⟨σ⟩ (resp. ⟨(σ0, . . . , σn−1)⟩) instead of ⟨σ⟩x (resp. ⟨(σ0, . . . , σn−1)⟩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 ∈ Exp into an equivalent syntactic expectation of the form O1ν1 . . . Okνk: f′, where Oi ∈ {Σ, L} and f′ is “quantifier”-free, i.e., contains neither Σ nor L. 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, f1, f2 ∈ Exp, terms a, and Boolean expressions ϕ, quantifiers O ∈ {Σ, L}, and fresh logical variables ν′, the following equivalences hold:

1. (Oν: f1) + f2 ≡ Oν′: f1 [ν/ν′] + f2,
2. f1 + (Oν: f2) ≡ Oν′: f1 + f2 [ν/ν′],
3. a · Oν: f ≡ Oν′: a · f [ν/ν′], and
4. [ϕ] · Oν: f ≡ Oν′: [ϕ] · f [ν/ν′].

The Dedekind normal form is motivated by the notion of Dedekind cuts [Bertrand 1849]. We denote by Cut (α) 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 \( \mathbb{R}_{\geq 0}^\infty \), we do allow for both empty cuts and \( \mathbb{Q}_{\geq 0} \). More formally, we define:

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

\[
\text{Cut} (\alpha) \triangleq \{ r \in \mathbb{Q}_{\geq 0} \mid r < \alpha \}.
\]

Furthermore, we define \( \text{Cut} (\alpha) \triangleq \text{Cut} (\alpha) \cup \{0\} \).

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

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

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

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

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

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

We call \( \text{Dedekind}[v_{\text{Cut}}, f] \) the Dedekind normal form of \( f \).

The Dedekind normal form \( \text{Dedekind}[v_{\text{Cut}}, f] \) defines the Dedekind cut of every \( \sigma[f] \), i.e.,

\[
\text{for all } \sigma: \quad \text{Cut} (\sigma[f]) = \{ r \in \mathbb{Q}_{\geq 0} \mid r = \sigma(v_{\text{Cut}}), \sigma[\text{Dedekind}[v_{\text{Cut}}, f]] = 1 \}.
\]

Hence, we can recover \( f \) from \( \text{Dedekind}[v_{\text{Cut}}, f] \):

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

\[
f \equiv \exists v_{\text{Cut}} : \text{Dedekind}[v_{\text{Cut}}, f] \cdot v_{\text{Cut}}.
\]

### 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 \( \alpha_0, \ldots, \alpha_n \in \mathbb{R}_{\geq 0}^\infty \), we have

\[
\sum_{j=0}^{n} \alpha_j = \sup \left\{ \sum_{j=0}^{n} r_j \mid \forall i \in \{0, \ldots, n\}: r_i \in \text{Cut} (\alpha_i) \right\}.
\]

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

\[
\sigma[\text{Sum} [v_{\text{sum}}, f, v]] = \sum_{j=0}^{\sigma(v)} \sigma[f[v_{\text{sum}}/j]] \quad \text{and} \quad \sigma[\exists v : \text{Sum} [v_{\text{sum}}, f, v]] = \sum_{j=0}^{\infty} \sigma[f[v_{\text{sum}}/j]].
\]
PROOF. We sketch the construction of $\text{Sum} [v_{\text{sum}}, f, v]$. Lemma 9.1 and the Dedekind normal form Dedekind $[v_{\text{Cut}}, f]$ of $f$ (cf. Theorem 8.4) give us

$$\sum_{j=0}^{\sigma(v)} \sigma[f[v_{\text{sum}}/j]]$$

$$= \sup \left\{ \sum_{j=0}^{\sigma(v)} r_j : \forall j \in \{0, \ldots, \sigma(v)\} : r_j \in \text{Cut} \left( \sigma[f[v_{\text{sum}}/j]] \right) \right\}$$

$$= \sup \left\{ \sum_{j=0}^{\sigma(v)} r_j : \forall j \in \{0, \ldots, \sigma(v)\} : \sigma[\text{Dedekind}[f, r_j]] = 1 \text{ or } r_j = 0 \right\}. \quad (4)$$

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

$$\exists v' : v' \cdot \land u : \land z : \exists v_{\text{Cut}} : \text{Prefix}(f) :$$

$$[\text{RElem}(\sigma, 0, 1) \land \text{RElem}(\sigma, v + 1, v) \land (\exists \phi \left[ v_{\text{prod}}/u \right] \lor v_{\text{Cut}} = 0))$$

$$\land (u < v + 1) \land \text{RElem}(\sigma, v, z) \land \left[ \varphi \right]$$

$$\longrightarrow \text{RElem}(\sigma, u + 1, z + v_{\text{Cut}})) \right\}.$$

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

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

with $r_j \in \text{Cut} \left( \sigma[f[v_{\text{sum}}/j]] \right)$ 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

$$\text{Sum} [v_{\text{sum}}, f, v] = \exists \sigma : \text{num} : g.$$

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

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

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

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

The construction of the syntactic Product expectation is completely analogous:

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

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

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

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 \( f, g \in \Exp \), and let \( v_{\text{prod}} \) be a fresh variable. We define the (unrestricted) product \( f \odot g \) of \( f \) and \( g \) by

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

**Corollary 9.5.** Let \( f, g \in \Exp \). For all states \( \sigma \), we have

\[
\sigma[f \odot g] = \sigma[f] \cdot \sigma[g].
\]

**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 \( x = \{x_0, \ldots, x_{n-1}\} \). We assume a fixed set \( \Sigma_x \subseteq \Sigma \) that contains exactly one state from each equivalence class of \( \sim_x \) (cf. Section 7.4). Given a state \( \sigma \in \Sigma \), we define the characteristic expectation \( [\sigma]_x \) of \( \sigma \) (w.r.t. \( x \)) as

\[
[\sigma]_x \triangleq [x_0 = \sigma(x_0) \land \ldots \land x_{n-1} = \sigma(x_{n-1})].
\]

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

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

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

\[
\wp[\text{while} (\varphi) \{ C' \}]([f]) = \lambda \sigma. \sup_{\sigma \in \Sigma_x} \sum_{\sigma_0, \ldots, \sigma_{k-1} \in \Sigma_x} [\sigma_0]_x(\sigma) \cdot ([\neg \varphi] \cdot [f])(\sigma_{k-1}) \\
\quad \cdot \prod_{i=0}^{k-2} \wp[\text{if} (\varphi) \{ C' \} \text{ else } \{ \text{skip} \}][([\sigma_{i+1}]_x)(\sigma_i)]
\]

**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 \( C = \text{while} (\varphi) \{ C_1 \} \). 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 \( \wp[\text{while} (\varphi) \{ C_1 \}]([f]) \).

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

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

\[
\sigma[\text{Path}[f](v_1, v_2)] = ([\neg \varphi] \cdot [f])(\sigma_{\sigma(v_1)-1}) \cdot \prod_{i=0}^{\sigma(v_1)-2} \wp[\text{if} (\varphi) \{ C_1 \} \text{ else } \{ \text{skip} \}][([\sigma_{i+1}]_x)(\sigma_i)]
\]

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

Then, for the syntactic expectation
\[ h = \llbracket \text{length} : \mathcal{E} \text{nums} : \text{Sum}[v_{\text{sum}}, \text{StateSequence}_x (v_{\text{sum}}, \text{length})] \circ \text{Path}[f](\text{length}, v_{\text{sum}}, \text{nums}) \rrbracket, \]
we have \( \text{wp}[\text{while}(\phi)\{C_1\}](\llbracket f \rrbracket) = [h] \).

Here, the quantifier \( \mathcal{E} \text{length} \) 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 \( \text{length} \) is realized by summing over all Gödel numbers \( \text{num} \) satisfying \( \text{StateSequence}(\text{num}, \text{length}) \). 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
\[
\text{while}(c = 1)\{
\{c := 0\} \parallel \{c := 1\};
x := x + 1
\}
\]
where we denote the loop body by \( C' \). Moreover, let \( f \triangleq x \in \text{Exp} \). Then the syntactic expectation \( h \) expressing \( \text{wp}[\text{while}(c = 1)\{C'\}](\llbracket x \rrbracket) \) as sketched in the proof of Theorem 10.2 is
\[ h = \llbracket \mathcal{E} \text{length} : \mathcal{E} \text{nums} : \text{Sum}[v_{\text{sum}}, \text{StateSequence}_x (v_{\text{sum}}, \text{length})] \circ \text{Path}[f](\text{length}, v_{\text{sum}}, \text{nums}) \rrbracket, \]
where the syntactic expectation \( \text{Path}[f](\text{length}, v_{\text{sum}}) \) is defined as follows:
\[
[\text{length} < 2] \cdot (\mathcal{E} \text{num} : [\text{Elem}(v_{\text{sum}}, \text{length} - 1, \text{num})] \circ \text{Subst}_x [(\lnot(c = 1) \land x), \text{num}])
+ [\text{length} \geq 2] \cdot (\mathcal{E} \text{num} : [\text{Elem}(v_{\text{sum}}, \text{length} - 1, \text{num})] \circ \text{Subst}_x [(\lnot(c = 1) \land x), \text{num}])
\circ \text{Product}(\mathcal{E} \text{num}_1 : \mathcal{E} \text{num}_2 : [\text{Elem}(v_{\text{sum}}, v_{\text{prod}}, \text{num}_1) \land (\text{Elem}(v_{\text{sum}}, v_{\text{prod}} + 1, \text{num}_2))] \circ \text{Subst}_x [\text{Subst}_{\text{num}}[g, \text{num}_2], \text{num}_1], \text{length} - 2)
\]
and where
\[ g = [c = 1] \cdot \frac{1}{2} \cdot ([0 = c' \land x + 1 = x'] + [1 = c' \land x + 1 = x']) + [\lnot(c = 1)] \cdot [c = c' \land x = x']. \]
We omit unfolding \( h \) further. Although our general construction yields rather complex syntactic preexpectations, notice we can express \( \text{wp}[\text{while}(c = 1)\{C'\}](\llbracket x \rrbracket) \) much more concisely as
\[ x + [c = 1] \cdot 2 \in \text{Exp}. \]

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 \( \mathbb{R}_{\geq 0} \) 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 \( x_{\text{sgn}} \), representing the absolute value of \( x \) and its sign (\( x_{\text{sgn}} = 1 \) if \( x \) negative, and \( x_{\text{sgn}} = 0 \) otherwise), respectively. With this convention, the program below emulates the subtraction assignment \( z := x - y \) using only addition and monus:

```plaintext
if (x_{\text{sgn}} = y_{\text{sgn}}) {
    |z| := (|x| + |y|) + (|y| - |x|)
} else {
    |z| := |x| + |y|
}
```

```plaintext
if (|x| > |y|) {
    z_{\text{sgn}} := x_{\text{sgn}}
} else {
    if (|x| = |y|) {
        z_{\text{sgn}} := 0
    } else {
        z_{\text{sgn}} := 1 - y_{\text{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 \( \text{Exp} \) 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 $f, g \in \text{Exp}$, verifying the bounds

$$[g] \leq \text{wp}[C]\left(\llbracket f \rrbracket\right) \quad \text{or} \quad \text{wp}[C]\left(\llbracket f \rrbracket\right) \leq [g]$$

reduces to checking a single inequality between two syntactic expectations in Exp, namely $g$ and the effectively constructible expectation for $\text{wp}[C]\left(\llbracket f \rrbracket\right)$. 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

$$\text{wp}[C]\left(\llbracket \sigma \rrbracket\right)$$

is a mapping from initial state $\sigma$ to the probability that $C$ terminates on $\sigma$. Since $1 \in \text{Exp}$, 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 $C$, $\sigma$, and $\alpha$, deciding whether $C$ terminates on $\sigma$ at least with probability $\alpha$ is $\Sigma^0_1$-complete in the arithmetical hierarchy. Deciding whether $C$ terminates on $\sigma$ at most with probability $\alpha$ is even $\Pi^0_2$-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 $[\phi]$, the weakest preexpectation

$$\text{wp}[C]\left(\llbracket [\phi] \rrbracket\right)$$

is a mapping from an initial state $\sigma$ to the probability that $C$ terminates on $\sigma$ in a state $\rho \models \phi$. Since $[\phi]$ is expressible in Exp, we have that $\text{wp}[C]\left(\llbracket [\phi] \rrbracket\right)$ 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 $\mu^\sigma_C$ be the final distribution obtained by executing $C$ on input $\sigma$, 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 $\tau$ on initial state $\sigma$, where $\tau(x_i) = x'_i$, by

$$\mu^\sigma_C(\tau) = \text{wp}[C]\left(\llbracket \{x_1 = x'_1 \land \cdots \land x_k = x'_k\} \rrbracket\right)(\sigma).$$

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 $\{x_1 = x'_1 \land \cdots \land x_k = x'_k\} \in \text{Exp}$, we have that $\text{wp}[C]\left(\llbracket \{x_1 = x'_1 \land \cdots \land x_k = x'_k\} \rrbracket\right)$ 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

\[
H_x = \left[ \text{Sum} \left[ \frac{1}{v_{\text{sum}}}, x \right] \right], \quad \text{where} \quad \frac{1}{v_{\text{sum}}} = \mathcal{E} z : [z \cdot v_{\text{sum}} = 1] \cdot z.
\]

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 à 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 \(\text{wp}[C](f)\) is expressible in Exp. There is the duality \(\text{wp}[C](f) = 1 - \text{wp}[\neg C](1-f)\), 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


