A Deductive Verification Infrastructure for Probabilistic Programs[*]

PHILIPP SCHRÖER, RWTH Aachen University, Germany
KEVIN BATZ, RWTH Aachen University, Germany
BENJAMIN LUCIEN KAMINSKI, Saarland University, Germany and University College London, United Kingdom
JOOST-PIETER KATOEN, RWTH Aachen University, Germany
CHRISTOPH MATHEJA, Technical University of Denmark, Denmark

This paper presents a quantitative program verification infrastructure for discrete probabilistic programs. Our infrastructure can be viewed as the probabilistic analogue of Boogie: its central components are an intermediate verification language (IVL) together with a real-valued logic. Our IVL provides a programming-language-style for expressing verification conditions whose validity implies the correctness of a program under investigation. As our focus is on verifying quantitative properties such as bounds on expected outcomes, expected run-times, or termination probabilities, off-the-shelf IVLs based on Boolean first-order logic do not suffice. Instead, a paradigm shift from the standard Boolean to a *real-valued* domain is required.

Our IVL features quantitative generalizations of standard verification constructs such as `assume`- and `assert`-statements. Verification conditions are generated by a weakest-precondition-style semantics, based on our real-valued logic. We show that our verification infrastructure supports natural encodings of numerous verification techniques from the literature. With our SMT-based implementation, we automatically verify a variety of benchmarks. To the best of our knowledge, this establishes the first deductive verification infrastructure for expectation-based reasoning about probabilistic programs.

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

Additional Key Words and Phrases: deductive verification, quantitative verification, probabilistic programs, weakest preexpectations, real-valued logics, automated reasoning

1 INTRODUCTION AND OVERVIEW

Probabilistic programs differ from ordinary programs by the ability to base decision on samples from probability distributions. They are found in randomized algorithms, communication protocols, models of physical and biological processes, and – more recently – statistical models used in machine learning and artificial intelligence (cf. [Barthe et al. 2020; Gordon et al. 2014]). Typical questions in the design and analysis of probabilistic programs are concerned with *quantifying* aspects of their *expected* – or average – *behavior*, e.g. the *expected runtime* of a randomized algorithm, the *expected number of retransmissions* in a protocol, or the *probability* that a particle reaches its destination.

Writing correct probabilistic programs is notoriously hard. They may contain subtle bugs occurring with low probability or undesirably favor certain results in the long run. In fact, reasoning about the expected behavior of probabilistic programs is known to be strictly harder than for ordinary programs [Kaminski et al. 2019].

[*] Accepted for publication at OOPSLA 2023 (https://doi.org/10.1145/3622870).

Authors' addresses: Philipp Schröer, phisch@cs.rwth-aachen.de, RWTH Aachen University, Germany; Kevin Batz, kevin.batz@cs.rwth-aachen.de, RWTH Aachen University, Germany; Benjamin Lucien Kaminski, kaminski@cs.uni-saarland.de, Saarland University, Germany and University College London, United Kingdom; Joost-Pieter Katoen, katoen@cs.rwth-aachen.de, RWTH Aachen University, Germany; Christoph Matheja, chmat@dtu.dk, Technical University of Denmark, Denmark.

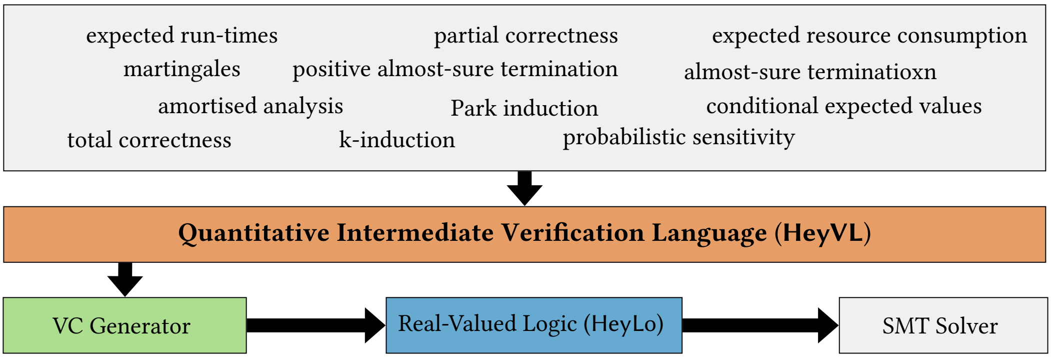
expected run-times
partial correctness
expected resource consumption
martingales
positive almost-sure termination
almost-sure terminatioxn
amortised analysis
Park induction
conditional expected values
total correctness
k-induction
probabilistic sensitivity
Quantitative Intermediate Verification Language (HeyVL)
VC Generator
Real-Valued Logic (HeyLo)
SMT Solver

Fig. 1. Architecture of our verification infrastructure.

There exists a plethora of research on verification techniques for probabilistic programs, ranging from program logics (cf. [Kaminski et al. 2018; McIver and Morgan 2005]) to highly specialized proof rules [Hark et al. 2019; McIver et al. 2018], often with little (if any) automation. These techniques are based on different branches of mathematics – e.g. domain theory or martingale analysis – and their relationships are non-trivial (cf. Takisaka et al. [2021]). This poses major challenges for comparing – let alone *combining* – such different approaches.

In this paper, we build a *verification infrastructure* for reasoning about the expected behavior of (discrete) probabilistic programs; Figure 1 gives an overview. Modern program verifiers for non-probabilistic programs often have a front-end that translates a given program and its specification into an intermediate language, such as Boogie [Leino 2008], Why3 [Filliâtre and Paskevich 2013], or Viper [Müller et al. 2016b]. Such intermediate languages enable the encoding of complex verification techniques, while allowing for the separate development of efficient back-ends, e.g. verification condition generators. In this very spirit, we introduce a novel *quantitative intermediate verification language* that enables researchers to (i) prototype and automate new verification techniques, (ii) combine proof rules, and (iii) benefit from back-end improvements. Before we dive into details, we discuss five examples of probabilistic programs from the literature that have been verified with five different techniques – all of them have been encoded in our language and verified with our tool.

Example 1.1 (Rabin's Mutual Exclusion Protocol [Kushilevitz and Rabin 1992]*).* This protocol controls processes competing for access to a critical section. To determine which process gets access, every process will repeatedly toss a fair coin until it sees heads; the process that needed the largest number of tosses is then granted access. Figure 2 shows a probabilistic program modeling Rabin's protocol: i is the number of remaining processes competing for access. While more than 1 competitor remains, each competitor tosses one coin (inner loop). If the coin shows heads (i.e. if `flip`(0.5) samples a 1), that competitor is removed from the pool of remaining competitors (by subtracting $d = 1$ from i). One can verify with the *weakest liberal preexpectation calculus* by McIver and Morgan [2005] that *the probability to select exactly one process (plus the probability of nontermination) is at least* $^2/_3$ if there are initially at least 2 processes.

Example 1.2 (The Coupon Collector [Wikipedia 2023a]*).* Figure 3 models the coupon collector problem – a well-known problem in probability theory: Suppose any box of cereals contains one of N different coupons. What is the average number of boxes one needs to buy to collect at least one of all N different coupons, assuming that each coupon type occurs with the same probability? Our formulation is taken from [Kaminski et al. 2018]; the authors develop an *expected runtime*

```
while (1 < i) {
  n := i;
  while (0 < n) {
    d := flip(0.5);
    i := i − d;
    n := n − 1
  }
}
```

Fig. 2. Model of Rabin's Protocol

```
while (0 < x) {
  i := N + 1;
  while (0 < x < i) {
    i :≈ unif(1, N)
  }
  x := x − 1
}
```

Fig. 3. The Coupon Collector's Problem

```
fn lossy(l: List) {
  if (len(l) > 0) {
    { lossy(tail(l)) } [0.5] { diverge }
  }
}
```

Fig. 4. Lossy list traversal

```
while (x > 0) {
  q := x/(2 · x + 1);
  {x := x − 1} [q] {x := x + 1}
}
```

Fig. 5. Variant of a random walk

```
while (x ≠ 0) {
  {x := 0} [0.5] {y := y + 1}
  n := n + 1
}
```

Fig. 6. Counterexample from [Hark et al. 2019]

calculus and use *invariant-based arguments* to show that the *expected number of loop iterations*, which coincides with the average number of boxes one needs to buy, *is bounded from above by* $N \cdot H_N$, where H_N is the N-th harmonic number.

Example 1.3 (Lossy List Traversal [Batz et al. 2019]*).* Figure 4 depicts a recursive function implementing a lossy list traversal; it flips a fair coin (using the probabilistic choice $\{\dots\}$ [0.5] $\{\dots\}$) and, depending on the outcome, either calls itself with the list's tail or diverges, i.e. enters an infinite loop. Using the *weakest preexpectation calculus* [Kozen 1983; McIver and Morgan 2005], one can prove that this program terminates with probability *at most* $0.5^{len(l)}$. Analyzing the lossy list traversal is intuitive – for every non-empty list, there is exactly one execution that does not diverge; its probability is $0.5^{len(l)}$. What is noteworthy, however, is that even for such a simple program, we need to reason about an exponential function. This is common when verifying probabilistic programs: proving non-trivial bounds often requires non-linear arithmetic.

Example 1.4 (Fair Random Walk [Wikipedia 2023b]*).* Figure 5 depicts a variant of a one-dimensional random walk of a particle with position x – a well-studied model in physics. Analyzing the program's termination behavior is hard because the probability q of moving to the left or right changes in every loop iteration depending on the previous position x. McIver et al. [2018] propose a proof rule based on *quasi-variants* that allows proving that *this program terminates almost-surely*, i.e. with probability one. Fair random walks, i.e. if $q = 1/2$, are well-known to terminate almost-surely but still have infinite expected runtime.

Example 1.5 (Lower Bounds on Expected Values [Hark et al. 2019]*).* Figure 6 shows an another loop whose control flow depends on the outcome of coin flips. Hark et al. [2019] studied this example to demonstrate that induction-based proof rules for *lower bounds*[1], which are sound for classical verification, may become unsound when reasoning about probabilistic programs. The authors used martingale analysis and the optional stopping theorem to develop a sound proof rule capable of proving that, whenever $x \neq 0$ initially holds, then the expected value of y after the program's termination is *at least* $1 + y$.

Challenges. We summarize the challenges of developing an infrastructure for automated verification of probabilistic programs unvealed by the examples in Figures 2 to 6:

First, there are many different verification techniques for probabilistic programs that are based on different concepts, e.g. quantitative invariants, quasi-variants, different notions of martingales, or stopping times of stochastic processes. Developing a language that is sufficiently expressive to encode these techniques while keeping it amenable to automation is a major challenge.

Second, verification of probabilistic programs involves *reasoning about both lower- and upper bounds* on expected values. This is different from classical program verification, which can be understood as proving that a given precondition implies a program's weakest precondition, i.e. $\texttt{pre} \Rightarrow \texttt{wp}[\![C]\!](\texttt{post})$. In other words, pre is a *lower bound* (in the Boolean lattice) on $\texttt{wp}[\![C]\!](\texttt{post})$. Proving *upper bounds*, i.e. $\texttt{wp}[\![C]\!](\texttt{post}) \Rightarrow \texttt{pre}$, has received scarce attention.[2]

Third, in Figures 3 to 5, we noticed that verification of probabilistic programs often involves reasoning about *unbounded* random variables and non-linear arithmetic involving exponentials, harmonic numbers, limits, and possibly infinite sums.

Our approach. We address the first challenge by developing a quantitative IVL and a real-valued logic tailored to verification of probabilistic programs. The IVL features quantitative generalizations of standard verification constructs such as assume- and assert-statements. Our quantitative constructs are inspired by Gödel logics [Baaz 1996; Preining 2010]. In particular, they have *dual* co-*constructs* for verifying upper- instead of lower bounds, thereby addressing the second challenge. These dual constructs are not only interesting for quantitative reasoning, but indeed also for Boolean reasoning à la $\texttt{wp}[\![C]\!](\texttt{post}) \Rightarrow \texttt{pre}$. To address the third challenge, we rely on modern SMT solvers' abilities to deal with custom theories, standard techniques for limiting the number of user-defined function applications, and custom optimizations.

Figure 7 shows a program written in our quantitative IVL; it encodes the verification of Example 1.3. We use a *co*procedure to prove that the quantitative precondition $\mathit{exp}(0.5, \mathit{len}(l)) = 0.5^{\mathit{len}(l)}$ is an *upper* bound on the procedure's termination probability[3] given by the quantitative postcondition 1. We establish the above bound for the procedure body while assuming that it holds for recursive calls (cf. [Olmedo et al. 2016]). Our dual quantitative assert- and assume-statements encode the call in the usual way: we assert the procedure's pre and assume its post.

Contributions. The main contributions of our work are:

(1) A *novel intermediate verification language* (⟶ Section 3) for automating probabilistic program verification techniques featuring *quantitative generalizations of standard verification constructs*, e.g. assert and assume, and a *formalization* of its semantics based on a real-valued logic (⟶ Section 2) with constructs inspired by Gödel logics.

(2) *Encodings of verification techniques and proof rules* with different theoretical underpinnings (e.g. domain theory, martingales, and the optional stopping theorem) taken from the probabilistic program verification literature into our intermediate language (⟶ Section 4).

[1] Specifically: *lower bound on partial correctness* plus *proof of termination* gives *lower bound on total correctness.*

[2] Notable exceptions are Cousot's necessary preconditions [Cousot et al. 2013] and recent works on (partial) incorrectness logic [O'Hearn 2020; Zhang and Kaminski 2022].

[3] Technically, $\mathit{exp}(0.5, \mathit{len}(l))$ upper-bounds the expected value of the random variable 1 after the procedure's termination.

```
coproc lossy (l: List) -> ()
  pre exp(0.5, len(l))
  post 1
{
    if (len(l) > 0) {
      var coin: 𝔹 :≈ flip(0.5)  // coin flip
      if (coin) {
        coassert exp(0.5, len(tail(l))); covalidate; coassume 1 // call of lossy(tail(l))
      } else {assert ?(false) } // diverge
    }
}
```

Fig. 7. Encoding of the lossy list traversal (see Figure 4) in our intermediate language.

(3) An SMT-backed *verification infrastructure* that enables researchers to prototype and automate verification techniques for probabilistic programs by encoding to our intermediate language, an *experimental evaluation* of its feasibility, and a prototypical *frontend* for verifying programs written in the probabilistic guarded command language (⟶ Section 5).

2 HEYLO: A QUANTITATIVE ASSERTION LANGUAGE

When analyzing quantitative program properties such as runtimes, failure probabilities, or space usage, it is often more direct, more intuitive, and more practical to reason directly about *values* like the runtime n^2, the probability $^1/_{2^x}$, or a list's length, instead of *predicates* like $rt = n^2$, $prob \leq {}^1/_{2^x}$, or length$(ls) > 0$ (cf., [Kaminski et al. 2018; Ngo et al. 2018]).

This section introduces HeyLo – a real-valued logic for quantitative verification of probabilistic programs, which aims to take the role that predicate logic has for classical verification. By syntactifying real-valued functions, HeyLo serves as (1) a language for specifying quantitative properties – in particular those that McIver and Morgan [2005] (and many other authors) call *expectations*[4] –, and (2) a foundation for automation by reducing many verification problems to a decision problem for HeyLo, e.g. validity or entailment checking. To ensure that HeyLo is expressive enough for (1), we design it reminiscently of the language by Batz et al. [2021b], which is relatively complete for the verification of probabilistic programs.

To ensure that HeyLo is suitable for (2), HeyLo is *first-order*, so as to simplify automation. Moreover, verification problems can often be stated as inequalities between to functions. To ensure that such inequalities can, in principle, be encoded into a *single* decision problem for HeyLo, we introduce *quantitative (co)implications* – which provide a syntax for comparing HeyLo formulae – and prove an analogue to the classical deduction theorem for predicate logic [Kleene 1952]. Supporting comparisons between expectations via (co)implications is essential for encoding proof rules for probabilistic programs. The (co)implications are inspired by intuitionistic Gödel logics [Baaz 1996; Preining 2010] and form Heyting algebras (cf. Theorem 2.1), hence the name HeyLo.

[4] For historical reasons, the term *expectations* refers to random variables on a program's state space.

2.1 Program States and Expectations

Let $\mathsf{Vars} = \{x, y, \ldots\}$ be a countably infinite set of typed variables. We write $x\colon \tau$ to indicate that x is of type τ, i.e. τ is the set of values x can take. We assume the built-in types $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$, $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{Q}_{\geq 0}$, $\mathbb{R}$, $\mathbb{R}_{\geq 0}$, and $\mathbb{R}_{\geq 0}^{\infty} = \mathbb{R}_{\geq 0} \cup \{\infty\}$; our verification infrastructure also supports user-defined mathematical types (cf. Section 5.1). We collect all types in Types and all values in $\mathsf{Vals} = \bigcup_{\tau \in \mathsf{Types}} \tau$. A *(program) state* σ maps every variable $x\colon \tau$ to a value in τ. The set of states is thus

$$\mathsf{States} \quad = \quad \{\sigma\colon \mathsf{Vars} \to \mathsf{Vals} \quad \mid \quad \text{for all } x \in \mathsf{Vars}\colon \quad x\colon \tau \quad \text{implies} \quad \sigma(x) \in \tau\ \} ~.$$

Expectations are the quantitative analogue to logical predicates: they map program states to $\mathbb{R}_{\geq 0}^{\infty}$ instead of truth values. The complete lattice $(\mathbb{E}, \preceq)$ of expectations is given by

$$\mathbb{E} \;=\; \left\{X \;\middle|\; X\colon \mathsf{States} \to \mathbb{R}_{\geq 0}^{\infty}\right\} \qquad \text{with} \qquad X \;\preceq\; Y \quad \text{iff} \quad \text{for all } \sigma \in \mathsf{States}\colon\; X(\sigma) \;\leq\; Y(\sigma)~.$$

2.2 Syntax of HeyLo

We start with the construction of HeyLo's atoms. The set $\mathcal{T}$ of *terms* is given by the grammar

$$t \quad ::= \quad c \mid x \mid f(t, \ldots, t)~,$$

where c is a *constant* in $\mathbb{Q} \cup \mathbb{B}$, x is a *variable* in Vars, and f is either one of the *built-in function* symbols $+, \cdot, -, \dot{-}, <, =, \wedge, \vee, \neg$ ($\dot{-}$ is subtraction truncated at 0) or a typed *user-defined function* symbol $f\colon \tau_1 \times \ldots \times \tau_n \to \tau$ for some $n \geq 0$ and types $\tau_1, \ldots, \tau_n, \tau$ (cf. Section 5.1). Function symbols include, for example, the length of lists $\mathsf{len}\colon \mathsf{Lists} \to \mathbb{N}$ and the exponential function $exp\colon \mathbb{R} \times \mathbb{Z} \to \mathbb{R}$ mapping (r, n) to r^n.

We write $t\colon \tau$ to indicate that term t is of type τ. Typing and subtyping of terms is standard. In particular, if $t\colon \tau_1$ and $\tau_1 \subseteq \tau_2$, then $t\colon \tau_2$. We only consider well-typed terms.

We denote terms of type $\mathbb{Q}_{\geq 0}$ (resp. $\mathbb{B}$) by a (resp. b) and call them *arithmetic* expressions (resp. *Boolean expressions*). The set of HeyLo *formulae* is given by the following grammar:

$$\begin{array}{lllrlr}
\varphi & ::= & a & \text{(arithmetic expressions)} & \mid\ ?(b) & \text{(Boolean embedding)} \\
& \mid & \varphi + \varphi & \text{(addition)} & \mid\ \varphi \cdot \varphi & \text{(multiplication)} \\
& \mid & \varphi \sqcap \varphi & \text{(minimum)} & \mid\ \varphi \sqcup \varphi & \text{(maximum)} \\
& \mid & \reflectbox{J} x\colon \tau.\ \varphi & \text{(infimum over } x\colon \tau) & \mid\ \reflectbox{S} x\colon \tau.\ \varphi & \text{(supremum over } x\colon \tau) \\
& \mid & \varphi \to \varphi & \text{(implication)} & \mid\ \varphi \leftsquigarrow \varphi & \text{(coimplication)}
\end{array}$$

We explain the meaning of HeyLo formulae in the next subsection. Free- and bound (by $\reflectbox{$\mathsf{S}$}$ or $\reflectbox{$\mathsf{J}$}$ quantifiers) variables of a HeyLo formula φ are defined as usual. The order of precedence for arithmetic- and Boolean expressions is standard. For HeyLo formulae, the order of precedence is,

$$\reflectbox{J}, \reflectbox{S} \qquad < \qquad \to, \leftsquigarrow \qquad < \qquad \sqcup \qquad < \qquad \sqcap \qquad < \qquad + \qquad < \qquad \cdot \qquad ,$$

i.e. $\reflectbox{$\mathsf{J}$}$ and $\reflectbox{$\mathsf{S}$}$ are least binding and $\cdot$ is most binding. We use parentheses to resolve ambiguities.

2.3 Semantics and Properties of HeyLo

A term $t\colon \tau$ evaluates to value $[\![t]\!](\sigma) \in \tau$ on state σ. We assume the standard semantics for constants and built-in functions and that $[\![f]\!]$ is given for all user-defined functions.

The *semantics* of a HeyLo formula φ is an expectation $[\![\varphi]\!]\colon \mathsf{States} \to \mathbb{R}_{\geq 0}^{\infty}$ defined by induction on the structure of φ in Figure 8, where we define $0 \cdot \infty = \infty \cdot 0 = 0$ as is common in measure theory. Two HeyLo formulae φ and ψ are *equivalent*, denoted $\varphi \equiv \psi$, iff $[\![\varphi]\!] = [\![\psi]\!]$. A HeyLo formula

$$\varphi \text{ is } \textit{valid} \quad \text{iff} \quad \varphi \;\equiv\; \infty \qquad \text{and} \qquad \varphi \text{ is } \textit{covalid} \quad \text{iff} \quad \varphi \;\equiv\; 0~.$$

ρ	$\llbracket\rho\rrbracket(\sigma)$	ρ	$\llbracket\rho\rrbracket(\sigma)$
a	$\llbracket a\rrbracket(\sigma)$	$?(b)$	$\begin{cases}\infty, & \text{if } \llbracket b\rrbracket(\sigma) = \mathsf{true}\\ 0, & \text{otherwise}\end{cases}$
$\varphi + \psi$	$\llbracket\varphi\rrbracket(\sigma) + \llbracket\psi\rrbracket(\sigma)$	$\varphi \cdot \psi$	$\llbracket\varphi\rrbracket(\sigma) \cdot \llbracket\psi\rrbracket(\sigma)$
$\varphi \sqcap \psi$	$\min\left\{\, \llbracket\varphi\rrbracket(\sigma),\ \llbracket\psi\rrbracket(\sigma) \,\right\}$	$\varphi \sqcup \psi$	$\max\left\{\, \llbracket\varphi\rrbracket(\sigma),\ \llbracket\psi\rrbracket(\sigma) \,\right\}$
$\reflectbox{$\mathsf{J}$}\, x\colon \tau.\, \varphi$	$\inf\left\{\, \llbracket\varphi\rrbracket(\sigma[x \mapsto v]) \;\middle\vert\; v \in \tau \,\right\}$	$\reflectbox{$\mathsf{S}$}\, x\colon \tau.\, \varphi$	$\sup\left\{\, \llbracket\varphi\rrbracket(\sigma[x \mapsto v]) \;\middle\vert\; v \in \tau \,\right\}$
$\varphi \rightarrow \psi$	$\begin{cases}\infty, & \text{if } \llbracket\varphi\rrbracket(\sigma) \leq \llbracket\psi\rrbracket(\sigma)\\ \llbracket\psi\rrbracket(\sigma), & \text{otherwise}\end{cases}$	$\varphi \leftsquigarrow \psi$	$\begin{cases}0, & \text{if } \llbracket\varphi\rrbracket(\sigma) \geq \llbracket\psi\rrbracket(\sigma)\\ \llbracket\psi\rrbracket(\sigma), & \text{otherwise}\end{cases}$

Fig. 8. Semantics of HeyLo. inf and sup are taken over $\mathbb{R}^{\infty}_{\geq 0}$. Here $\sigma[x \mapsto v](y) = \begin{cases} v, & \text{if } x = y\\ \sigma(y), & \text{otherwise}\,. \end{cases}$

For $\varphi, \psi \in$ HeyLo, we define

$$\underbrace{\varphi \sqsubseteq \psi}_{\text{read: } \varphi \text{ lower-bounds } \psi} \quad \text{iff} \quad \underbrace{\llbracket\varphi\rrbracket \leq \llbracket\psi\rrbracket}_{\text{pointwise inequality}} \qquad \text{and} \qquad \underbrace{\varphi \sqsupseteq \psi}_{\text{read: } \varphi \text{ upper-bounds } \psi} \quad \text{iff} \quad \llbracket\varphi\rrbracket \geq \llbracket\psi\rrbracket.$$

These notions are central since we will encode verification problems as inequalities between HeyLo formulae. In contrast to classical IVLs, HeyLo contains constructs for *both* reasoning about lower-bounds and for reasoning about upper bounds. We briefly go over each construct in Figure 8.

Arithmetic- and Boolean Expressions. These expressions form the atoms of HeyLo. Consider, e.g. the arithmetic expressions $x + 1$ for some numeric variable x and $2 \cdot \mathsf{len}(y)$ for a variable $y\colon$ Lists. On state σ, $x + 1$ evaluates to $\sigma(x) + 1$, and $2 \cdot \mathsf{len}(y)$ evaluates to 2 times the length of list $\sigma(y)$.

Boolean expressions b are embedded in HeyLo using the *embedding operator* $?(\cdot)$: On state σ, $?(b)$ evaluates to ∞ (think: true, since ∞ is the top element in the lattice of expectations) if σ satisfies b, and to 0 otherwise. For instance, $?(x + 1 = 2 \cdot \mathsf{len}(y))$ evaluates to ∞ if $\sigma(x) + 1$ is equal to two times the length of the list $\sigma(y)$, and to 0 otherwise.

Addition, Multiplication, Minimum, and Maximum. HeyLo formulae can be composed by standard binary arithmetic operations for sums (+), products ($\cdot$), minimum ($\sqcap$), and maximum ($\sqcup$). Each of these operations are understood pointwise (with the assumption that $\infty \cdot 0 = 0$). For instance, $\llbracket\mathsf{len}(y_1) \sqcap \mathsf{len}(y_2)\rrbracket(\sigma)$ is the minimum length of lists $\sigma(y_1)$ and $\sigma(y_2)$.

Quantifiers. The *infimum quantifier* $\reflectbox{$\mathsf{J}$}$ and the *supremum quantifier* $\reflectbox{$\mathsf{S}$}$ from [Batz et al. 2021b] are the quantitative analogues of the universal $\forall$ and the existential $\exists$ quantifier from predicate logic. Intuitively, the $\reflectbox{$\mathsf{J}$}$ quantifier minimizes a quantity, just like the $\forall$ quantifier minimizes a predicate's truth value. Dually, the $\reflectbox{$\mathsf{S}$}$ quantifier maximizes a quantity just like $\exists$ maximizes a predicate's truth value. The quantitative quantifiers embed $\forall$ and $\exists$ in HeyLo, i.e. for $b\colon \mathbb{B}$ and $\sigma \in$ States,

$$\llbracket \reflectbox{J}\, x\colon \tau.\, ?(b)\rrbracket(\sigma) = \begin{cases}\infty, & \text{if } \sigma \models \forall x\colon \tau.\, b\\ 0, & \text{otherwise}\end{cases} \qquad \text{and} \qquad \llbracket \reflectbox{S}\, x\colon \tau.\, ?(b)\rrbracket(\sigma) = \begin{cases}\infty, & \text{if } \sigma \models \exists x\colon \tau.\, b\\ 0, & \text{otherwise}\end{cases}$$

Here, $\models$ denotes the standard satisfaction relation of first-order logic. The above construction extends canonically to nested quantifiers, e.g. $\exists x\colon \tau.\, \forall y\colon \tau'.\, b$ corresponds to $\reflectbox{$\mathsf{S}$}\, x\colon \tau.\ \reflectbox{J}\, y\colon \tau'.\, ?(b)$.

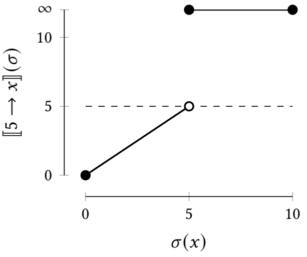

Fig. 9. $[\![5 \longrightarrow x]\!](\sigma)$ for $\sigma(x) \in [0, 10]$.

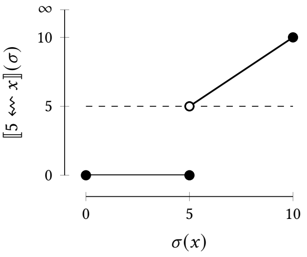

Fig. 10. $[\![5 \leftsquigarrow x]\!](\sigma)$ for $\sigma(x) \in [0, 10]$.

For a quantitative example, consider the formula $\varphi = \mathcal{S} x \colon \mathbb{Q}_{\geq 0}.\, ?(x \cdot x < 2) \sqcap x$. On state σ, the subformula $?(x \cdot x < 2) \sqcap x$ evaluates to $\sigma(x)$ if $\sigma(x) \cdot \sigma(x) < 2$, and to 0 otherwise. Consequently,

$$[\![\varphi]\!](\sigma) \;=\; \sup\{\, r \in \mathbb{Q}_{\geq 0} \;\mid\; r \cdot r < 2 \,\} \;=\; \sqrt{2}\,.$$

Notice that $[\![\varphi]\!](\sigma)$ is *irrational* even though all constituents of φ are rational-valued. It has been shown in [Batz et al. 2021b] that — similar to our above construction of $\sqrt{2}$ — the quantitative quantifiers combined with arithmetic- and (embedded) Boolean expressions over $\mathbb{Q}_{\geq 0}$ enable the construction of *all* expected values emerging from discrete probabilistic programs.

(Co)implication. $\longrightarrow$ and $\leftsquigarrow$ generalize Boolean implication and converse nonimplication.[5] For state σ, the *implication* $\varphi \longrightarrow \psi$ evaluates to ∞ if $[\![\varphi]\!](\sigma) \leq [\![\psi]\!](\sigma)$, and to $[\![\psi]\!](\sigma)$ otherwise. Dually, the *coimplication* $\varphi \leftsquigarrow \psi$ evaluates to 0 if $[\![\varphi]\!](\sigma) \geq [\![\psi]\!](\sigma)$, and to $[\![\psi]\!](\sigma)$ otherwise.

To gain some intuition, we first note that the top element ∞ of our quantitative domain $\mathbb{R}_{\geq 0}^{\infty}$ can be viewed as "entirely true" (i.e. as true as it can possibly get) and 0 can be viewed as "entirely false" (i.e. as false as it can possibly get). The implication $\varphi \longrightarrow \psi$ makes ψ *more true* by *lowering the threshold above which ψ is considered entirely true* – and thus ∞ – to φ. In other words: Anything that is at least as true as φ is considered entirely true. Anything less true than φ remains as true as ψ. Figure 9 illustrates this for the formula $5 \longrightarrow x$.

As another example, $x^2 \longrightarrow x$ evaluates to ∞ for states σ with $\sigma(x) \in [0, 1]$; otherwise, x is below the threshold x^2 at which x is considered entirely true and thus the implication evaluates to x.

The intuition underlying the coimplication is dual: $\varphi \leftsquigarrow \psi$ makes ψ *less true* by *raising the threshold below which ψ is considered entirely false* – and thus 0 – to φ. In other words: Anything that is not more true than φ is considered entirely false. Anything that is more true than φ remains as true as ψ. Figure 10 illustrates this for the formula $5 \leftsquigarrow x$.

Chained implications can also be understood in terms of lowering thresholds: $\varphi \longrightarrow (\psi \longrightarrow \rho)$ lowers the threshold at which ρ is considered entirely true to φ *and* ψ, whichever is lower. Formally, $\varphi \longrightarrow (\psi \longrightarrow \rho)$ is equivalent to $(\varphi \sqcap \psi) \longrightarrow \rho$. More generally, (co)implications are the adjoints of the minimum $\sqcap$ and maximum $\sqcup$:

THEOREM 2.1 (ADJOINTNESS PROPERTIES). *For all* HeyLo *formulae φ, ψ, and ρ, we have*

$$\varphi \sqcap \psi \;\sqsubseteq\; \rho \quad \text{iff} \quad \varphi \;\sqsubseteq\; \psi \longrightarrow \rho \qquad \textit{and} \qquad \psi \sqcup \rho \;\sqsupseteq\; \varphi \quad \text{iff} \quad \rho \;\sqsupseteq\; \psi \leftsquigarrow \varphi\,.$$

Both $\longrightarrow$ and $\leftsquigarrow$ are backward compatible to Boolean implication and converse nonimplication:

$$[\![?(b_1) \longrightarrow\, ?(b_2)]\!](\sigma) = \begin{cases} \infty, & \text{if } \sigma \models b_1 \longrightarrow b_2 \\ 0, & \text{otherwise} \end{cases} \qquad [\![?(b_1) \leftsquigarrow\, ?(b_2)]\!](\sigma) = \begin{cases} \infty, & \text{if } \sigma \models \neg(b_1 \longleftarrow b_2) \\ 0, & \text{otherwise} \end{cases}$$

[5] The converse nonimplication of propositions P and Q is defined as $\neg(P \longleftarrow Q)$ and is to be read as "Q does *not imply* P".

We will primarily use (co)implications to (1) incorporate the capability of *comparing* expectations syntactically in HeyLo and to (2) express *assumptions*. Application (1) is justified by the following quantitative version of the well-known deduction theorem[6] from first-order logic [Kleene 1952]:

Theorem 2.2 (HeyLo Deduction Theorem). *For all* HeyLo *formulae φ and ψ, we have*

$$\varphi \sqsubseteq \psi \quad \text{iff} \quad \varphi \rightarrow \psi \text{ is valid} \qquad \text{and} \qquad \varphi \sqsupseteq \psi \quad \text{iff} \quad \varphi \leftsquigarrow \psi \text{ is covalid}\,.$$

The proof is in Appendix A. For application (2), consider the implication $?(b) \rightarrow \psi$; it evaluates to ψ whenever b holds, and to ∞ otherwise. As in predicate logic, the implication can be read as *assuming* b holds before evaluating ψ. Formally,

$$[\![?(b) \rightarrow \psi]\!](\sigma) \;=\; \begin{cases} [\![\psi]\!](\sigma), & \text{if } \sigma \models b \\ \infty, & \text{otherwise}\,. \end{cases}$$

Now, consider the inequality $\varphi \sqsubseteq ?(b) \rightarrow \psi$. For all states σ *not* satisfying b (i.e. the set of states that we do *not* assume), the inequality vacuously holds. For all other states (i.e. those states that we actually assume), φ must lower-bound ψ in order for the inequality to hold.

Example 2.3. Let $\varphi,\ \psi \in$ HeyLo and $b\colon \mathbb{B}$. We construct a HeyLo formula ρ that, on state σ, evaluates to φ if $\sigma \models b$, and to ψ otherwise. For that, we use the Boolean embedding and the implication:

$$\rho \quad = \quad \underbrace{(?(b) \rightarrow \varphi)}_{\text{if } b \text{ holds, evaluate to } \varphi} \quad \underbrace{\sqcap}_{\text{and}} \quad \underbrace{(?(\neg b) \rightarrow \psi)}_{\text{if } \neg b \text{ holds, evaluate to } \psi}$$

To encode assumptions using the coimplication $\leftsquigarrow$, we first introduce Boolean *co*-embeddings

$$[\![\mathrm{co}?(b)]\!] \;=\; [\![?(\neg b)]\!] \;=\; \lambda\sigma.\begin{cases} 0, & \text{if } [\![b]\!](\sigma) = \mathsf{true} \\ \infty, & \text{otherwise}\,. \end{cases}$$

We then obtain a dual construction using $\leftsquigarrow$ for encoding assumptions: By Theorem 2.1, we have

$$\varphi \sqsupseteq \mathrm{co}?(b) \leftsquigarrow \psi \qquad \text{iff} \qquad \text{for all } \sigma \in \mathsf{States}\colon [\![b]\!](\sigma) = \mathsf{true} \text{ implies } [\![\varphi]\!](\sigma) \geq [\![\psi]\!](\sigma)\,,$$

i.e. the coimplication $\mathrm{co}?(b) \leftsquigarrow \psi$ ensures that it suffices to reason about states satisfying b.

2.4 Qualitative Reasoning in HeyLo

The verification of probabilistic programs comprises both quantitative *and* qualitative reasoning. Whereas questions like “what is the expected value of program variable x upon termination” are inherently quantitative, questions like “does x increase in expectation after one loop iteration?” are qualitative. HeyLo marries quantitative and qualitative reasoning. To shift to a qualitative statement, we first consider the *negation* $\neg\varphi$ and *conegation* $\sim\varphi$ of φ obtained from our (co)implications:

$$[\![\neg\varphi]\!] \;=\; [\![\varphi \rightarrow 0]\!] \;=\; \lambda\sigma.\begin{cases} \infty, & \text{if } [\![\varphi]\!](\sigma) = 0 \\ 0, & \text{otherwise} \end{cases} \qquad [\![\sim\varphi]\!] \;=\; [\![\varphi \leftsquigarrow \infty]\!] \;=\; \lambda\sigma.\begin{cases} 0, & \text{if } [\![\varphi]\!](\sigma) = \infty \\ \infty, & \text{otherwise}\,. \end{cases}$$

The (co)negation always evaluates to either ∞, the top element of $\mathbb{R}^{\infty}_{\geq 0}$ (entirely true), or 0, the bottom element of $\mathbb{R}^{\infty}_{\geq 0}$ (entirely false). By applying a (co)negation twice, we turn an arbitrary

[6] We mean the deduction theorem that relates semantical entailment $\models$ with the material conditional $\rightarrow$. Another theorem also known as *deduction theorem* relates syntactical entailment (i.e. provability) $\vdash$ with the material conditional $\rightarrow$.

expectation into a qualitative statement. Formally, we define the *(pointwise) validation* $\triangle(\varphi)$ and *(pointwise) covalidation* $\triangledown(\varphi)$ by[7]

$$[\![\triangle(\varphi)]\!] ~=~ [\![\sim\sim\varphi]\!] ~=~ \lambda\sigma.\begin{cases}\infty, & \text{if } [\![\varphi]\!](\sigma)=\infty\\ 0, & \text{otherwise}\end{cases} \quad\text{and}\quad [\![\triangledown(\varphi)]\!] ~=~ [\![\neg\neg\varphi]\!] ~=~ \lambda\sigma.\begin{cases}0, & \text{if } [\![\varphi]\!](\sigma)=0\\ \infty, & \text{otherwise}\,.\end{cases}$$

In words, the validation $\triangle(\varphi)$ is (pointwise) entirely true whenever φ is entirely true, and entirely false otherwise. Dually, $\triangledown(\varphi)$ is entirely false whenever φ is entirely false, and entirely true otherwise. Thus, both validations and covalidations "boolify" HeyLo formulae. The difference is that validations pull intermediate truth values down to entire falsehood whereas covalidations lift intermediate truth values up to entire truth.

Turning expectations into qualitative statements has an important application, which often arises when encoding verification problems: Suppose we are given two formulae φ, ψ with free variables $y_1, \ldots, y_n$. Moreover, our goal is to construct a HeyLo formula ρ that evaluates to x of type $\mathbb{Q}_{\geq 0}$ if $\varphi \sqsubseteq \psi$, and to 0 otherwise. For that, we first construct the formula $\ell\, y_1, \ldots, y_n.\, \triangle(\varphi \rightarrow \psi)$. Due to the infimum quantifier over all free variables, this formula is *equivalent* to ∞ if $\varphi \sqsubseteq \psi$, and *equivalent* to 0 otherwise. Hence, we construct ρ as

$$\underbrace{\big(\ell\, y_1\colon \tau_1, \ldots, y_n\colon \tau_n.\; \triangle(\varphi \rightarrow \psi)\,\big)}_{\text{evaluate to 0 if } \varphi \not\sqsubseteq \psi} \qquad \underbrace{\sqcap}_{\text{and}} \qquad \underbrace{x}_{\text{evaluate to } x \text{ otherwise}} \qquad .$$

Moreover, we obtain a dual construction using $\leftsquigarrow$ and the supremum quantifier:

$$\underbrace{\big(\reflectbox{$\mathcal{S}$}\, y_1\colon \tau_1, \ldots, y_n\colon \tau_n.\; \triangledown(\varphi \leftsquigarrow \psi)\,\big)}_{\text{evaluate to } \infty \text{ if } \varphi \not\sqsupseteq \psi} \qquad \underbrace{\sqcup}_{\text{and}} \qquad \underbrace{x}_{\text{evaluate to } x \text{ otherwise}}$$

3 HeyVL: A QUANTITATIVE INTERMEDIATE VERIFICATION LANGUAGE

Many verification problems for probabilistic programs reduce naturally to checking inequalities between HeyLo formulae.[8] Consider, for instance, the program

$$y :\approx {}^1\!/_2 \cdot \langle x \rangle + {}^1\!/_2 \cdot \langle x+1 \rangle\,,$$

which sets y either to x or to $x+1$, depending on the outcome of a fair coin flip. Suppose we want to verify that $x + \frac{1}{2}$ is a *lower* bound on the expected value of y after executing above program. According to McIver and Morgan [2005], verifying this bound amounts to proving the inequality

$$\underbrace{x + \tfrac{1}{2}}_{\text{proposed lower bound}} \quad\sqsubseteq\quad \tfrac{1}{2}\cdot x + \tfrac{1}{2}\cdot(x+1) \quad\triangleq\quad \underbrace{\mathrm{wp}[\![y :\approx {}^1\!/_2 \cdot \langle x \rangle + {}^1\!/_2 \cdot \langle x+1 \rangle]\!](y)}_{\text{expected outcome of } x \,+\, \text{fair coin flip stored in } y}\,, \qquad \text{(ex)}$$

where the weakest preexpectation $\mathrm{wp}[\![C]\!](f)$ is a function (which we can represent as a HeyLo formula) that maps every initial state σ to the expected value of f after executing the program C on input σ. Our goal is to simplify writing, composing, and reasoning *modularly* about such expected values and similar quantities. To this end, we propose HeyVL, a novel intermediate verification language for modeling quantitative verification problems.

HeyVL *programs* are organized as a *collection of procedures*. Each procedure P is equipped with a body S and a specification. The body S is a HeyVL *statement* and can for now be thought of as a more or less ordinary probabilistic program.[9] The specification of a procedure comprises a *pre* φ and a *post* ψ, both HeyLo formulae. Intuitively, a procedure P *verifies* if its body S adheres

[7] In Gödel logics, these are also called *projection modalities* [Baaz 1996].

[8] Or equivalently by Theorem 2.2: Checking (co)validity, i.e. whether a HeyLo formula is equivalent to ∞ (resp. 0).

[9] There are verification-specific statements which can be part of the procedure body which we will describe later.

```
proc $ex$ ($x$: UInt) -> ($y$: UInt) // procedure that takes $x$ as input and returns the value of $y$
    pre $x + 1/2$ // lower bound on the expected value of $y$ after termination of the body
    post $y$ // quantity of interest evaluated in final states
{
    $y :\approx 1/2 \cdot \langle x \rangle + 1/2 \cdot \langle x + 1 \rangle$ // returns the sum of $x$ plus outcome of a fair coin flip
}
```

Fig. 11. A HeyVL procedure whose verification condition is equation (ex).

to P's specification, meaning essentially that the inequality $\varphi \sqsubseteq \mathrm{wp}[\![S]\!](\psi)$ holds, i.e. the expected value of ψ after executing S is lower-bounded by φ. This inequality will be called the *verification condition* of P. An entire HeyVL program *verifies* if all of its procedures verify.

How do we describe the verification problem (ex) in HeyVL? As shown in Figure 11, we write a single procedure P with body $y :\approx 1/2 \cdot \langle x \rangle + 1/2 \cdot \langle x + 1 \rangle$, pre $x + \frac{1}{2}$, and post y. This gives rise to the verification condition $x + \frac{1}{2} \sqsubseteq \mathrm{wp}[\![y :\approx 1/2 \cdot \langle x \rangle + 1/2 \cdot \langle x + 1 \rangle]\!](y)$, which is precisely the inequality (ex) we aim to verify. The HeyLo program (i.e. the single procedure P) verifies if and only if we have positively answered the verification problem (ex).

To encode more complex verification problems or proof rules, one may need to write more than one HeyVL procedure. For example, in Section 4.1, we will encode a proof rule for conditional expected values that requires establishing a lower *and* a different upper bound. The latter can be described using a second HeyVL procedure, see Section 3.1. Furthermore, it is natural to break down large programs and/or complex proof rules into smaller (possibly mutually recursive) procedures, which can be verified modularly based on the truth of their verification conditions.

3.1 HeyVL Procedures

A HeyVL procedure consists of a name, a list of (typed) input and output variables, a body, and a quantitative specification. Syntactically, a HeyVL procedure is of the form

```
proc $P$ ($\overline{in\colon \tau}$) -> ($\overline{out\colon \tau}$)   // procedure name $P$ with read-only inputs $\overline{in}$ and outputs $\overline{out}$
    pre $\varphi$                      // pre: HeyLo formula over inputs
    post $\psi$                     // post: HeyLo formula over inputs or outputs
{ $S$ }                          // procedure body
```

where P is the procedure's name, $\overline{in}$ and $\overline{out}$ are (possibly empty and pairwise distinct) lists of typed program variables called the *inputs* and *outputs* of P. The specification is given by a *pre* φ which is a HeyLo formula over variables in $\overline{in}$ and a *post* ψ which is also a HeyLo formula but ranging over variables in $\overline{in}$ or $\overline{out}$. The *procedure body* S is a HeyVL statement, whose syntax and semantics will be formalized in Sections 3.2 and 3.3.

As mentioned above, the procedure P gives rise to a verification condition, namely $\varphi \sqsubseteq \mathrm{wp}[\![S]\!](\psi)$. However, this is only accurate if S is an ordinary probabilistic program. As our statements S may also contain non-executable[10] verification-specific `assume` and `assert` commands, the *verification condition generated by* P is actually

$$\varphi \quad \sqsubseteq \quad \mathrm{vp}[\![S]\!](\psi)\,,$$

[10] But expected value changing.

```
proc n_dice (n: UInt) -> (r: UReal)
    pre 3.5 · n
    post r
{ S }
```

Fig. 12. Expected sum of rolling n fair dice.

```
proc rabin (i: UInt) -> (ok: Bool)
    pre 2/3 ⊓ ?(1 < i)
    post 1 ⊓ ?(ok)
{ S }
```

Fig. 13. Rabin's mututal exclusion property.

where vp is the *verification preexpectation transformer* that extends the aforementioned weakest preexpectation wp by semantics for the verification-specific statements, see Section 3.3. For procedure calls, we approximate the weakest preexpectation based on the callee's specification to enable modular verification, see Section 3.5.

Readers familiar with classical Boolean deductive verification may think of the verification condition $\varphi \sqsubseteq \mathsf{vp}[\![S]\!](\psi)$ as a *quantitative Hoare triple* $\langle\varphi\rangle\; S\; \langle\psi\rangle$, where $\sqsubseteq$ takes the quantitative role of the Boolean $\Longrightarrow$, i.e. we have

$$\langle\varphi\rangle\; S\; \langle\psi\rangle \text{ is valid} \qquad \text{iff} \qquad \varphi \quad \sqsubseteq \quad \mathsf{vp}[\![S]\!](\psi).$$

Indeed, if φ and ψ are ordinary Boolean predicates and S is a non-recursive non-probabilistic program, then $\langle\varphi\rangle\; S\; \langle\psi\rangle$ is a standard Hoare triple: whenever state σ satisfies precondition φ, then procedure body S must successfully terminate on σ in a state satisfying postcondition ψ.

Phrased differently: for every initial state σ, the truth value $\varphi(\sigma)$ lower-bounds the *anticipated* truth value (evaluated in σ) of postcondition ψ after termination of S on σ. For arbitrary HeyLo formulae φ, ψ and probabilistic procedure bodies S, the second view generalizes to quantitative reasoning à la McIver and Morgan [2005]: The quantitative triple $\langle\varphi\rangle\; S\; \langle\psi\rangle$ is valid iff the pre φ lower-bounds the *expected value* (evaluated in initial states) of the post ψ after termination of S. In Section 3.5, we will describe how *calling* a (verified) procedure P can be thought of as "invoking" the validity of the quantitative Hoare triple that is given by P's specification.

Notice that the above inequality is our definition of validity of a quantitative Hoare triple and we do not provide an operational definition of validity. This is due to a lack of an intuitive operational semantics for quantitative `assume` and `assert` statements (cf. also Section 7).

Examples. Besides Figure 11, Figures 12 and 13 further illustrate how HeyVL procedures specify quantitative program properties; we omit concrete procedure bodies S to focus on the specification. The procedure in Figure 12 specifies that the expected value of output r must be at least $3.5 \cdot n$ – a property satisfied by any statement S that rolls n fair dice. The procedure in Figure 13 specifies that the expected value of output ok being true after termination of S, i.e. the probability that the returned value ok will be true, is at least $2/3$ whenever input i is greater than one – a key property of Rabin's randomized mutual exclusion algorithm [Kushilevitz and Rabin 1992] from Figure 2 and discussed in the introduction. Since we aim to reason about probabilities, we ensure that the post is one-bounded by considering $1 \sqcap {?}(ok)$ instead of ${?}(ok)$.

Coprocedures – Duals to Procedures. Proving *upper* bounds is often relevant for quantitative verification, e.g. when analyzing expected runtimes of randomized algorithms (cf. [Kaminski et al. 2018]). HeyVL also supports *co*procedures which give rise to the dual verification condition $\varphi \sqsupseteq \mathsf{vp}[\![S]\!](\psi)$.[11] The syntax of coprocedures is analogous to HeyVL procedures; the only difference is the keyword `coproc` instead of `proc`. For example, a coprocedure which was defined as in Figure 12 (except for replacing `proc` by `coproc`) would specify that the expected value of output

[11] Notice $\sqsupseteq$ for coprocedures as opposed to $\sqsubseteq$ for procedures.

r must be *at most* $3.5 \cdot n$. We demonstrate in Section 4 that intricate verification techniques for probabilistic programs may require lower *and* upper bound reasoning, i.e. HeyVL programs that are collections of both procedures and coprocedures.

HeyVL *Programs.* To summarize, a HeyVL *program* is a list of procedures and coprocedures that each give rise to a verification condition, i.e. a HeyLo inequality. We say that a HeyVL program *verifies* iff all verification conditions of its (co)procedures hold.

Design Decisions. Since HeyVL is an intermediate language, we favor simplicity over convenience. In particular, we require procedure inputs to be read-only, i.e. evaluate to the same values in initial and final states. Moreover, HeyVL has no loops and no global variables. All variables that can possibly be modified by a procedure call are given by its outputs. All of the above restrictions can be lifted by high-level languages that encode to HeyVL.

3.2 Syntax of HeyVL Statements

HeyVL statements, which appear in procedure bodies, provide a programming-language-style to express and approximate *expected values* arising in the verification of probabilistic programs, including expected outcomes of program variables, reachability probabilities such as the probability of termination, and expected rewards. HeyVL statements consist of (a) *standard constructs* such as assignments, sampling from discrete probability distributions, sequencing, and nondeterministic branching, and (b) *verification-specific constructs* for modeling rewards such as runtime, quantitative assertions and assumptions, and for forgetting values of program variables in the current state.

The syntax of HeyVL statements S is given by the grammar

$$\begin{array}{lll}
S \;::=\; \texttt{var}\; x\colon \tau :\approx \mu & \mid \texttt{if}\ (\sqcap)\ \{S\}\ \texttt{else}\ \{S\} & \mid \texttt{if}\ (\sqcup)\ \{S\}\ \texttt{else}\ \{S\} \\
\quad\;\; \mid x_1,\ldots,x_n \;:=\; P(e_1,\ldots,e_m) & \mid \texttt{assert}\ \psi & \mid \texttt{coassert}\ \psi \\
\quad\;\; \mid \texttt{reward}\ a & \mid \texttt{assume}\ \psi & \mid \texttt{coassume}\ \psi \\
\quad\;\; \mid S\,;\ S & \mid \texttt{havoc}\ x & \mid \texttt{cohavoc}\ x \\
 & \mid \texttt{validate} & \mid \texttt{covalidate}\,,
\end{array}$$

where $x \in \mathsf{Vars}$ is of type τ, a is an arithmetic expression, and ψ is a HeyLo formula. Moreover, μ is a *distribution expression* of type τ[12]

$$\mu \;=\; p_1 \cdot \langle t_1 \rangle + \ldots + p_n \cdot \langle t_n \rangle$$

with $n \geq 1$, where each p_i is a term of type $[0, 1]$, each t_i is a term of type τ, and $\sum_{i=1}^{n} [\![p_i]\!](\sigma) = 1$ for every state σ. A distribution expression μ represents finite-support probability distributions, which assign probability p_i to each t_i. We often write $\texttt{flip}(p)$ instead of $p \cdot \langle \mathsf{true} \rangle + (1-p) \cdot \langle \mathsf{false} \rangle$.

We briefly go over the above constructs. $\texttt{var}\; x\colon \tau :\approx \mu$ is a *probabilistic assignment* which assigns to variable x a value *sampled* from the probability distribution described by μ. The statement $x_1,\ldots,x_n \;:=\; P(e_1,\ldots,e_m)$ is a (co)procedure call. We can think of it as passing the parameters $e_1,\ldots,e_m$ to (co)procedure P, executing P's body, and assigning the return values to variables $x_1,\ldots,x_n$. The statement $\texttt{reward}\ a$ collects/accumulates/adds a reward of a, modeling e.g. progression in (run)time or resource consumption. $S_1\,;\ S_2$ puts HeyVL statements in sequence. $\texttt{if}\ (\cdot)\ \{S_1\}\ \texttt{else}\ \{S_2\}$ is a *nondeterministic* choice between S_1 and S_2, where $\cdot$ determines whether the nondeterminisim is resolved in a minimizing ($\sqcap$) or maximizing ($\sqcup$) manner. $\texttt{assert}\ \psi$ and $\texttt{assume}\ \psi$ are quantitative generalizations of assertions and assumptions from classical IVLs. $\texttt{coassert}\ \psi$

[12] μ can be instantiated with more general distribution expressions as long as the vp semantics (cf. Section 3.3) is computable.

S	$\mathsf{vp}[\![S]\!](\varphi)$	S	$\mathsf{vp}[\![S]\!](\varphi)$
$\mathtt{var}\ x\colon \tau \mathrel{:\approx} \mu$	$p_1 \cdot \varphi[x \mapsto t_1]$ $+ \ldots + p_n \cdot \varphi[x \mapsto t_n]$	$\mathtt{reward}\ a$	$\varphi + a$
		$S_1;\ S_2$	$\mathsf{vp}[\![S_1]\!](\mathsf{vp}[\![S_2]\!](\varphi))$
$\mathtt{if}\ (\sqcap)\ \{S_1\}\ \mathtt{else}\ \{S_2\}$	$\mathsf{vp}[\![S_1]\!](\varphi)\ \sqcap\ \mathsf{vp}[\![S_2]\!](\varphi)$	$\mathtt{if}\ (\sqcup)\ \{S_1\}\ \mathtt{else}\ \{S_2\}$	$\mathsf{vp}[\![S_1]\!](\varphi)\ \sqcup\ \mathsf{vp}[\![S_2]\!](\varphi)$
$\mathtt{assert}\ \psi$	$\psi \sqcap \varphi$	$\mathtt{coassert}\ \psi$	$\psi \sqcup \varphi$
$\mathtt{assume}\ \psi$	$\psi \rightarrow \varphi$	$\mathtt{coassume}\ \psi$	$\psi \leftsquigarrow \varphi$
$\mathtt{havoc}\ x$	$ℓ x.\ \varphi$	$\mathtt{cohavoc}\ x$	$Ƨ x.\ \varphi$
$\mathtt{validate}$	$\triangle(\varphi)$	$\mathtt{covalidate}$	$\triangledown(\varphi)$

Fig. 14. Semantics of HeyVL statements. Here $\mu = p_1 \cdot \langle t_1 \rangle + \ldots + p_n \cdot \langle t_n \rangle$ and $\varphi[x \mapsto t_i]$ is the formula obtained from substituting every occurrence of x in φ by t_i in a capture-avoiding manner. For procedure calls, see Section 3.5.

and $\mathtt{coassume}\ \psi$ are novel statements that enable reasoning about upper bounds; there is yet no analogue in classical verification infrastructures.

$\mathtt{havoc}\ x$ and $\mathtt{cohavoc}\ x$ forget the current value of x by branching nondeterministically over all possible values of x either in a minimizing ($\mathtt{havoc}\ x$) or maximizing ($\mathtt{cohavoc}\ x$) manner. Finally, $\mathtt{validate}$ and $\mathtt{covalidate}$ turn *quantitative* expectations into *qualitative* expressions, much in the flavor of validation and covalidation described earlier (see Section 2.4).

Declarations and Types. We assume that all local variables (those that are neither inputs nor outputs) are initialized by an assignment before they are used; those assignments also declare the variables' types. If we assign to an already initialized variable, we often write $x \mathrel{:\approx} \mu$ instead of $\mathtt{var}\ x\colon \tau \mathrel{:\approx} \mu$. Moreover, if μ is a *Dirac* distribution, i.e. if $p_1 = 1$, we often write $x\ :=\ t_1$ instead of $x \mathrel{:\approx} \mu$. Finally, we assume that all programs and associated HeyLo formulae are well-typed.

3.3 Semantics of HeyVL Statements

Inspired by weakest preexpectations [Kaminski 2019; McIver and Morgan 2005], we give semantics to HeyVL statements as a backward-moving continuation-passing style HeyLo transformer

$$\mathsf{vp}[\![S]\!]\colon \mathsf{HeyLo} \rightarrow \mathsf{HeyLo}$$

by induction on S in Figure 14. (Co)procedure calls are treated separately in Section 3.5. We call $\mathsf{vp}[\![S]\!](\varphi)$ the *verification preexpectation* of S with respect to post φ. Intuitively, $[\![\mathsf{vp}[\![S]\!](\varphi)]\!](\sigma)$ is the expected value of φ w.r.t. the distribution of final states obtained from "executing"[13] S on σ. The post φ is either given by the surrounding procedure declaration or can be thought of as the verification preexpectation described by the *remaining* HeyVL statement: for $S = S_1;\ S_2$, we first obtain the intermediate verification preexpectation $\mathsf{vp}[\![S_2]\!](\varphi)$ — the expected value of what remains after executing S_1 — and pass this into $\mathsf{vp}[\![S_1]\!]$.

Random Assignments. The expected value of φ after executing $\mathtt{var}\ x\colon \tau \mathrel{:\approx} \mu$ is the weighted sum $p_1 \cdot \varphi[x \mapsto t_1] + \ldots + p_n \cdot \varphi[x \mapsto t_n]$, where each p_i is the probability that x is assigned t_i.

Rewards. Suppose that the post φ captures the expected reward collected in an execution that follows *after* executing $\mathtt{reward}\ a$. Then the entire expected reward is given by $\varphi + a$.

[13] Some verification-specific statements are not really *executable* but serve the purpose of manipulating expected values.

Nondeterministic Choices. $\mathsf{vp}[\![\texttt{if}\ (\cdot)\ \{S_1\}\ \texttt{else}\ \{S_2\}]\!](\varphi)$ is the pointwise minimum ($\cdot\ =\ \sqcap$) or maximum ($\cdot = \sqcup$) of the expected values obtained from S_1 and S_2, respectively.

(Co)assertions. In *classical* intermediate verification languages, the statement `assert` A for some predicate A models a proof obligation: All states reaching `assert` A on some execution must satisfy A. In terms of classical weakest preconditions, `assert` A transforms a postcondition B to

$$\mathsf{wp}[\![\texttt{assert}\,A]\!](B) \;=\; A \land B\,.$$

In words, `assert` A *caps* the truth of postcondition B at A: all lower-bounds on the above weakest precondition (in terms of the Boolean lattice $(\mathsf{States} \longrightarrow \mathbb{B},\ \Rightarrow)$) must not exceed A.

This perspective generalizes well to our quantitative assertions: Given a HeyLo formula ψ, the statement `assert` ψ *caps* the post at ψ. Thus, analogously to classical assertions, all *lower* bounds on the verification preexpectation $\mathsf{vp}[\![\texttt{assert}\ \psi]\!](\varphi)$ (in terms of $\sqsubseteq$) must not exceed ψ.

Coassertions are dual to assertions: `coassert` ψ *raises* the post φ to at least ψ. Hence, all *upper* bounds on $\mathsf{vp}[\![\texttt{coassert}\ \psi]\!](\varphi)$ must not *sub*ceed ψ.

(Co)assumptions. In the classical setting, the statement `assume` A for some predicate A *weakens* the verification condition: verification succeeds vacuously for all states not satisfying A. In terms of classical weakest preconditions, `assume` A transforms a postcondition B to

$$\mathsf{wp}[\![\texttt{assume}\,A]\!](B) \;=\; A \longrightarrow B$$

i.e. `assume` A *lowers* the threshold at which the post B is considered true (the top element of the Boolean lattice) to A. Indeed, if we identify $\mathsf{true} = 1$ and $\mathsf{false} = 0$, then

$$[\![\mathsf{wp}[\![\texttt{assume}\,A]\!](B)]\!](\sigma) \;=\; \begin{cases} 1, & \text{if } [\![A]\!](\sigma) \leq [\![B]\!](\sigma) \\ [\![B]\!](\sigma), & \text{otherwise}\,. \end{cases}$$

The above perspective on classical assumptions generalizes to our quantitative assumptions. Given a HeyLo formula ψ, `assume` ψ lowers the threshold above which the post φ is considered entirely true (i.e. ∞ – the top element of the lattice of expectations) to ψ. Formally,

$$[\![\mathsf{vp}[\![\texttt{assume}\ \psi]\!](\varphi)]\!](\sigma) \;=\; \begin{cases} \infty, & \text{if } [\![\psi]\!](\sigma) \leq [\![\varphi]\!](\sigma) \\ [\![\varphi]\!](\sigma), & \text{otherwise}\,. \end{cases}$$

Reconsider Figure 9 on page 8, which illustrates $\mathsf{vp}[\![\texttt{assume}\ 5]\!](x)$: `assume` 5 lowers the threshold at which the post x is considered entirely true to 5, i.e. whenever the post-expectation x evaluates at least to 5, then $\mathsf{vp}[\![\texttt{assume}\ 5]\!](x)$ evaluates to ∞. Notice furthermore that our quantitative `assume` is backward compatible to the classical one in the sense that $\mathsf{vp}[\![\texttt{assume}\ ?(b)]\!](\varphi)$ evaluates to φ for every state satisfying b, and to ∞ otherwise.

Coassumptions are dual to assumptions. `coassume` ψ raises the threshold at which the post φ is considered entirely false (i.e. 0 – the bottom element of the lattice of expectations) to ψ. Reconsider Figure 10 on page 8 illustrating $\mathsf{vp}[\![\texttt{coassume}\ 5]\!](x)$: `coassume` 5 raises the threshold below which the post x is considered entirely false to 5, i.e. if the post x evaluates at most to 5, then $\mathsf{vp}[\![\texttt{coassume}\ 5]\!](x)$ evaluates to 0.

Example 3.1 (Modeling Conditionals). We did not include `if` (b) $\{S_1\}$ `else` $\{S_2\}$ for conditional branching in HeyVL's grammar. We can encode it as follows (and will use it from now on):

$$\texttt{if}\ (\sqcap)\ \{\texttt{assume}\ ?(b);\ S_1\}\ \texttt{else}\ \{\texttt{assume}\ ?(\neg b);\ S_2\}$$

The vp semantics of this statement is analogous to the formula described in Example 2.3 and complies with our above description of assumptions: Depending on the satisfaction of b by the current state σ, the vp of S either evaluates to the vp of S_1 or S_2, respectively.

$$
\begin{array}{l}
\texttt{proc}\ P\,(x_1\colon \tau_1, \ldots, x_n\colon \tau_\mathrm{n})\ \texttt{->}\ (y_1\colon \tau_1', \ldots, y_m'\colon \tau_\mathrm{m}) \\
\qquad \texttt{pre}\ \rho \\
\qquad \texttt{post}\ \psi \\
\{\ S\ \}
\end{array}
$$

Fig. 15. A procedure P. We encode calls $z_1, \ldots, z_n \;\coloneqq\; P(t_1, \ldots, t_n)$ for arbitrary probabilistic statements S.

(Co)havocs. In the classical setting, havoc x forgets the current value of x by universally quantifying over all possible initial values of x. In terms of classical weakest preconditions, we have

$$\mathsf{wp}[\![\mathsf{havoc}\ x]\!](B) \;=\; \forall x\colon \tau.\, B\,,$$

i.e. havoc x *minimizes* the post B under all possible values for x, thus requiring B to hold for all x. This perspective generalizes to our quantitative setting: In terms of vp, havoc x forgets the current value of x by minimizing the post-expectation under all possible values of x. Dually, cohavoc x forgets the value of x but this time *maximizes* the post-expectation under all possible values for x.

(Co)validations. These statements convert quantitative statements into qualitative ones by casting expectations into the $\{0, \infty\}$-valued realm, thus eradicating intermediate truth values strictly between 0 and ∞. Their classical analogues would be effectless, as the Boolean setting features no intermediate truth values. We briefly explained in Section 2.4 how such a conversion to a qualitative statement works in HeyLo. An example will be discussed in Section 4.2.

3.4 Properties of HeyVL Statements

We study two properties of HeyVL. First, our vp semantics is *monotonic* — a crucial property for encoding proof rules (cf. Section 3.5).

Theorem 3.2 (Monotonicity of vp). *For all* HeyVL *statements S and* HeyLo *formulae φ, φ',*

$$\varphi \;\sqsubseteq\; \varphi' \quad \textit{implies} \quad \mathsf{vp}[\![S]\!](\varphi) \;\sqsubseteq\; \mathsf{vp}[\![S]\!](\varphi')\,.$$

Furthermore, HeyVL conservatively extends an existing IVL for non-probabilistic programs due to Müller [2019] in the following sense:

Theorem 3.3 (Conservativity of HeyVL). *Let C be a program in the programming language of* Müller [2019] *and let B be a postcondition. Moreover, let $\overline{C}$ be obtained by replacing every* assert A *and every* assume A *occurring in C by* assert $?(A)$ *and* assume $?(A)$*, respectively (cf. Boolean embeddings,* Section 2.3*). Then*

$$?(\underbrace{\,\mathsf{wp}[\![C]\!](B)\,}_{\textit{verification condition obtained from }[\text{Müller 2019}]}) \qquad\equiv\qquad \overbrace{\mathsf{vp}[\![\overline{C}]\!](?(B))}^{\mathsf{HeyVL}}\;.$$

3.5 Procedure Calls

We conclude this section with a treatment of (co)procedure calls. Consider a callee *procedure P* as shown in Figure 15. Intuitively, the effect of a call $z_1, \ldots, z_m \;\coloneqq\; P(t_1, \ldots, t_n)$ corresponds to (1) initializing P's formal input parameters $x_1, \ldots, x_n$ with the arguments $t_1, \ldots, t_n$, (2) inlining P's body S, and (3) assigning to $z_1, \ldots, z_m$ the values of outputs $y_1, \ldots, y_m$. The semantics of

$z_1, \ldots, z_m \ \coloneqq\ P(t_1, \ldots, t_n)$ can be thought of as the statement[14]

$$\underbrace{x_1 \ \coloneqq\ t_1\,;\ \ldots\,;\ x_n \ \coloneqq\ t_n\,;}_{=:\ \mathit{init}\quad \text{(initialize procedure inputs)}}\quad \overbrace{S\,;}^{\text{inlining of the procedure body}}\quad \underbrace{z_1 \ \coloneqq\ y_1\,;\ \ldots\,;\ z_m \ \coloneqq\ y_m}_{=:\ \mathit{return}\quad \text{(assign procedure outputs)}}\ .$$

There are two main issues that arise when we would actually inline S at every call-site: (1) For recursive procedure calls [Olmedo et al. 2016], we would need to define a (non-computable) fixed point semantics for the vp transformer. Our goal, however, is to render verification feasible in practice, so we would like to avoid fixed point computations. (2) Even without recursive calls, we would have to re-verify S at every call-site, which would not scale.

We thus do not inline the procedure body but use an *encoding* $S_{encoding}$ which *underapproximates* the effect of S in the sense that $\mathsf{vp}[\![S_{encoding}]\!](\varphi) \sqsubseteq \mathsf{vp}[\![S]\!](\varphi)$ for all HeyLo formulae φ. By monotonicity of vp, we can then verify lower bounds for calls: for all $\varphi, \gamma \in \mathsf{HeyLo}$,

$$\gamma \ \sqsubseteq\ \mathsf{vp}[\![\underbrace{\mathit{init}\,;\ S_{encoding}\,;\ \mathit{return}}_{\text{modular encoding of calls}}]\!](\varphi) \qquad \text{implies} \qquad \gamma \ \sqsubseteq\ \mathsf{vp}[\![\underbrace{\mathit{init}\,;\ S\,;\ \mathit{return}}_{\text{actual inlining of calls}}]\!](\varphi)\ ,$$

so whenever we can verify a HeyVL program using the modular encoding, we could have also verified it using inlining. The advantage of the modular encoding is that $S_{encoding}$ does not contain the procedure body – it could be changed without requiring re-verification of call sites, so long as the updated procedure body still adheres to the procedure's specification. To construct $S_{encoding}$, we leverage only P's specification pre ρ and post ψ, cf. Figure 15: Assuming that P verifies, we can safely assume that P's verification condition – namely $\rho \sqsubseteq \mathsf{vp}[\![S]\!](\psi)$ – holds.[15] By monotonicity of vp, we have $\rho \sqsubseteq \mathsf{vp}[\![S]\!](\psi) \sqsubseteq \mathsf{vp}[\![S]\!](\varphi)$ whenever $\psi \sqsubseteq \varphi$ holds. To underapproximate $\mathsf{vp}[\![S]\!](\varphi)$, we construct $S_{encoding}$ such that $\mathsf{vp}[\![S_{encoding}]\!](\varphi)$ is the known lower bound ρ if $\psi \sqsubseteq \varphi$; otherwise, it is the trivial lower bound 0. So how do we construct $S_{encoding}$ concretely?

In classical verification infrastructures (cf. [Müller 2019]), $S_{encoding}$ corresponds to the statement

$$\mathtt{assert}\ \rho;\ \mathtt{havoc}\ z_1;\ \ \ldots;\ \mathtt{havoc}\ z_m;\ \mathtt{assume}\ \psi.$$

That is, we assert the procedure's pre ρ before the call, forget the values of all outputs, i.e. variables that are potentially modified by the call, and assume the procedure's post ψ after the call. Phrased in terms of underapproximations: We assert that we have at most ρ before the call and, while minimising over all possible outputs (using the havoc statements), lower the threshold at which the post is considered entirely true (i.e. ∞) to ψ, i.e. whenever ψ lower-bounds the post.

The intuition underlying the above HeyVL statement works for encoding procedure calls of non-probabilistic programs. However, there is a subtle *unsoundness* that arises when reasoning about *expected* behaviors. Figure 16 shows two procedures, *foo* and *bar*. Intuitively, *foo* flips a fair coin and aborts execution if the result is heads (false). Read backwards, the expected value of the post will be at most x after executing *foo* – exactly as stated in *foo*'s specification. Procedure *bar* encodes the call $\mathit{foo}(x)$ in its body[16] and requires in its specification that the expected value of x does not decrease, i.e. is at least x. Both procedures verify. However, when inlining *foo*, i.e. using its body instead of the encoding $\mathtt{assert}\ x;\ \mathtt{assume}\ 2 \cdot x$, *bar* does *not* verify. Hence, the above encoding does, in general, not model a sound underapproximation of a procedure's inlining.

[14] For the sake of simplicity, we ignore potential scoping issues arising if S uses variables that are declared in the calling context; these issues can be resolved by a straightforward yet tedious variable renaming.

[15] Otherwise, procedure P in Figure 15 does not verify and verification of the whole HeyVL program fails anyway.

[16] There are no havoc statements because *foo* has no outputs; we also omitted *init* and *return* for simplicity.

```
proc $foo$ ($x$: $\mathbb{N}$) -> ()
    pre $x$
    post $2 \cdot x$
{ // verifies: $x \sqsubseteq 2 \cdot x \sqcap 0.5 \cdot \infty$
    var $b$: $\mathbb{B}$ :≈ $0.5 \cdot \langle\mathsf{true}\rangle + 0.5 \cdot \langle\mathsf{false}\rangle$;
    assert $?(b)$ }
```

```
proc $bar$ ($x$: $\mathbb{N}$) -> ()
    pre $x$
    post $x$
{ // verifies: $x \sqsubseteq x \sqcap (2 \cdot x \to x)$
    // encoding of $foo(x)$
    assert $x$; assume $2 \cdot x$ }
```

Fig. 16. Unsound encoding of a procedure call $foo(x)$ in bar. Both procedures verify but inlining the body of foo in bar does not as it produces the (wrong) inequality $x \sqsubseteq x \sqcap (0.5 \cdot \infty)$.

Taking a closer look, recall from above that `assume` $2 \cdot x$ is used to encode a monotonicity check,[17] which is an inherently *quali*tative property. However, verifying *bar* involves proving $x \sqsubseteq x \sqcap (2 \cdot x \to x)$, where the quantitative implication $2 \cdot x \to x$ evaluates to x for $x > 0$; the expectation x does not reflect the inherently qualitative nature of the monotonicity check. To fix this issue, we add a `validate` statement that turns *quanti*tative results into *quali*tative ones: it reduces any value less than ∞, which indicates a failed monotonicity check, to 0. An encoding underapproximating the inlining of $foo(x)$ – and thus correctly failing verification of *bar* – is `assert` x; `validate`; `assume` $2 \cdot x$. Similarly to Section 2.4, verifying *bar* for the fixed encoding involves proving $x \sqsubseteq x \sqcap \triangle(2 \cdot x \to x)$, which does not hold for $x > 0$.

More generally, a sound construction of $S_{encoding}$ (wrt. underapproximating procedure body S) is

$$S_{encoding}: \qquad \texttt{assert}\ \rho;\ \texttt{havoc}\ z_1;\ \ \ldots;\ \texttt{havoc}\ z_m;\ \texttt{validate};\ \texttt{assume}\ \psi.$$

Formally, we obtain an underapproximating HeyVL encoding of procedure calls of the form $z_1, \ldots, z_m \ := \ P(t_1, \ldots, t_n)$ for arbitrary probabilistic procedures as in Figure 15:

Theorem 3.4. *Let S be the body of the procedure P in Figure 15. Then, for every* HeyLo *formula φ,*

$$\mathsf{vp}[\![S_{encoding}]\!](\varphi) \ \sqsubseteq\ \mathsf{vp}[\![S]\!](\varphi) \quad \text{and} \quad \mathsf{vp}[\![init;\ S_{encoding};\ return]\!](\varphi) \ \sqsubseteq\ \mathsf{vp}[\![init;\ S;\ return]\!](\varphi).$$

A proof is found in Appendix B. A HeyVL encoding that *over*approximates calls of *co*procedures is analogous – it suffices to use the dual *co*statements in $S_{encoding}$. The presented under- and overapproximations are useful when encoding proof rules in HeyVL. Whether they are meaningful does, however, depend on the verification technique at hand that should be encoded.

4 ENCODING CASE STUDIES

To evaluate the expressiveness of our verification language, we encoded various existing calculi and proof rules targeting verification problems for probabilistic programs in HeyVL. We will first focus on programs without `while` loops (Section 4.1) and then consider loops (Section 4.2). The practicality of our automated verification infrastructure will be evaluated separately in Section 5. A summary of all encodings is given at the end of this section. Further details are found in Appendix C.

4.1 Reasoning about While-Loop-Free pGCL Dialects

Pioneered by Kozen [1983], expectation-based techniques have been successfully applied to analyze various probabilistic program properties. McIver and Morgan [2005] incorporated nondeterminism

[17] More precisely: a check whether monotonicity of vp can be applied, namely whether $\psi \sqsubseteq \varphi$ holds where ψ is the callee's *specified* post and φ is the *actual* post at the call-site.

C	$enc_{\mathsf{wp}}\lfloor C \rfloor$
`skip`	`reward` 0
`diverge`	`assert` 0
$x := t$	$x :\approx t$
$C_1; C_2$	$enc_{\mathsf{wp}}\lfloor C_1 \rfloor; enc_{\mathsf{wp}}\lfloor C_2 \rfloor$
`if` (b) $\{ C_1 \}$	`if` $(\sqcap)$ $\{$ `assume` $?(b); enc_{\mathsf{wp}}\lfloor C_1 \rfloor$ $\}$
`else` $\{ C_2 \}$	`else` $\{$ `assume` $?(\neg b); enc_{\mathsf{wp}}\lfloor C_2 \rfloor \}$
$\{ C_1 \}$ $[p]$ $\{ C_2 \}$	`var` tmp: $\mathbb{B} :\approx$ `flip`(p);
	$enc_{\mathsf{wp}}\lfloor$`if` (tmp) $\{C_1\}$ `else` $\{C_2\}\rfloor$
$\{ C_1 \}$ $[]$ $\{ C_2 \}$	`if` $(\sqcap)$ $\{C_1\}$ `else` $\{C_2\}$

Fig. 17. Encoding of weakest preexpectation for pGCL, where *tmp* is a fresh variable.

```
proc $lower$ ($\overline{in}$) -> ($\overline{out}$)
  pre $\psi$ // $\overline{in}$: variables in $\psi$
  post $\varphi$ { // $\overline{out}$: var. in $\varphi$ but not $\psi$
    // declare local variables, i.e.
    // those not in $\varphi$ or $\psi$, using
    // var $x$: $\tau :\approx$ $default$; havoc $x$
    $enc_{\mathsf{wp}}\lfloor C \rfloor$
}
```

Fig. 18. Encoding of $\psi \sqsubseteq wp(C, \varphi)$.

and introduced the probabilistic Guarded Command Language (pGCL), which is convenient for modelling probabilistic systems. The syntax of `while`-loop-free pGCL programs C is[18]

$$C \;::=\; \texttt{skip} \mid \texttt{diverge} \mid x := t \mid C_1; C_2 \mid \texttt{if}\ (b)\ \{C_1\}\ \texttt{else}\ \{C_2\} \mid \{\, C_1 \,\}\ [p]\ \{\, C_2 \,\} \mid \{\, C_1 \,\}\ []\ \{\, C_2 \,\},$$

where `skip` has no effect, `diverge` never terminates, $x := t$ assigns the value of term t to x, $C_1; C_2$ executes C_2 after C_1, `if` (b) $\{C_1\}$ `else` $\{C_2\}$ executes C_1 if Boolean expression b holds and C_2 otherwise, $\{ C_1 \}$ $[p]$ $\{ C_2 \}$ executes C_1 with probability $p \in [0, 1]$ and C_2 with probability $(1 - p)$, and $\{ C_1 \}$ $[]$ $\{ C_2 \}$ nondeterministically executes either C_1 or C_2.

We now outline encodings of several reasoning techniques targeting pGCL and extensions thereof. We will only consider expectations that can be expressed as HeyLo formulae. To improve readability, we identify every HeyLo formula φ with its expectation $[\![\varphi]\!] \in \mathbb{E}$.

Weakest Preexpectations (wp). The *weakest preexpectation calculus* of McIver and Morgan [2005] maps every pGCL command C and postexpectation φ to the *minimal* (to resolve nondeterminism) *expected value* $wp(C, \varphi)$ *of* φ *after termination of* C – the same intuition underlying HeyVL's vp transformer. Figure 17 shows a sound and complete HeyVL encoding $enc_{\mathsf{wp}}\lfloor C \rfloor$ of the weakest preexpectation calculus, i.e. $\mathsf{vp}[\![enc_{\mathsf{wp}}\lfloor C \rfloor]\!](\varphi) = wp(C, \varphi)$. Most pGCL commands have HeyVL equivalents; conditionals are encoded as in Example 3.1. `diverge` is encoded as `assert` 0 as it never terminates, i.e. $wp(\texttt{diverge}, \varphi) = 0$. The program in Figure 18 then verifies iff ψ lower bounds $wp(C, \varphi)$, i.e. $\psi \sqsubseteq wp(C, \varphi)$. To reason about *upper* bounds, it suffices to use a *co*procedure instead.

Weakest Liberal Preexpectations (wlp). McIver and Morgan [2005] also proposed a *liberal* weakest preexpectation calculus, a partial correctness variant of weakest preexpectations. More precisely, if $\varphi \sqsubseteq 1$, then the weakest liberal preexpectation $wlp(C, \varphi)$ is the expected value of φ after termination of C *plus* the probability of non-termination of C (on a given initial state). We denote by $enc_{\mathsf{wlp}}\lfloor C \rfloor$ the HeyVL encoding of the weakest liberal preexpectation calculus; it is defined analogously to Figure 17 except for `diverge`. Since `diverge` never terminates, the probability of non-termination is one, i.e. $wlp(\texttt{diverge}, \ldots) = 1$. The updated encoding of `diverge` is

$$enc_{\mathsf{wlp}}\lfloor \texttt{diverge} \rfloor \quad = \quad \texttt{assert}\ 1;\ \texttt{assume}\ 0\,,$$

[18] pGCL usually supports only one type, e.g. integers, rationals, or reals. We are more liberal and admit arbitrary terms t but assume a sufficiently strong type inference system and consider only well-typed programs.

```
{ a := 0 } [0.5] { a := 1 };
{ b := 0 } [0.5] { b := 1 };
{ c := 0 } [0.5] { c := 1 };
r := 4 · a + 2 · b + c + 1;
observe r ≤ 6
```

Fig. 19. pGCL program C_{die}.

```
var a: ℕ :≈ 0.5 · ⟨1⟩ + 0.5 · ⟨0⟩;
var b: ℕ :≈ 0.5 · ⟨1⟩ + 0.5 · ⟨0⟩;
var c: ℕ :≈ 0.5 · ⟨1⟩ + 0.5 · ⟨0⟩;
r :≈ 4 · a + 2 · b + c + 1;
assert ?(r ≤ 6)
```

Fig. 20. HeyVL encoding S_{die} of C_{die}.

```
coproc die_wp () -> (r: UInt)
  pre 2.625
  post r
  { S_die }
```

```
proc die_wlp () -> (r: UInt)
  pre 6/8
  post 1
  { S_die }
```

Fig. 21. HeyVL encoding of the proof obligations $\mathsf{wp}[\![C_{die}]\!](r) \sqsubseteq 2.625$ and $0.75 \sqsubseteq \mathsf{wlp}[\![C_{die}]\!](1)$.

where `assert` 1 ensures one-boundedness and `assume` 0 lowers the threshold at which the post is considered entirely true to 0. Put together, we have $\mathsf{vp}[\![enc_{\mathsf{wlp}}\lfloor \mathtt{diverge} \rfloor]\!](\varphi) = 1 \sqcap \infty = 1 = wlp(\mathtt{diverge}, \varphi)$.

Conditional Preexpectations (cwp). Conditioning on observed events (in the sense of conditional probabilities) is a key feature of modern probabilistic programming languages [Gordon et al. 2014]. Intuitively, the statement `observe` b discards an execution whenever Boolean expression b does not hold. Moreover, it re-normalizes such that the accumulated probability of all executions violating no observation equals one. Olmedo et al. [2018] showed that reasoning about `observe` b requires a combination of *wp* and *wlp* reasoning. They extended both calculi such that violating an observation is interpreted as a failure resulting in pre-expectation zero; we can encode it with an assertion:

$$w(l)p(\mathtt{observe}\ b, \varphi) \;=\; ?(b) \sqcap \varphi \;=\; \mathsf{vp}[\![\mathtt{assert}\ ?(b)]\!](\varphi).$$

For every pGCL program C with observe statements, initial state σ and expectation φ, the *conditional* expected value $cwp(C, \varphi)(\sigma)$ of φ after termination of C is then given by the expected value $wp(C, \varphi)(\sigma)$ normalized by the probability $wlp(C, 1)(\sigma)$ of violating no observation:

$$cwp(C, \varphi)(\sigma) \quad = \quad \frac{wp(C, \varphi)(\sigma)}{wlp(C, 1)(\sigma)} \qquad \text{(undefined if } wlp(C, 1)(\sigma) \;=\; 0)$$

We can re-use our existing HeyVL encodings to reason about conditional expected values. Notice that proving bounds on *cwp* requires establishing both lower and upper bounds. For example, the pGCL program C_{die} in Figure 19 assigns to r the result of a six-sided die roll, which is simulated using three fair coin flips and an observation. To show that the expected value of r is at most 3.5 – the expected value of a six-sided die roll – we prove the upper bound $wp(C_{die}, r) \sqsubseteq 2.625$ and the lower bound $0.75 \sqsubseteq wlp(C_{die}, 1)$. Then, $cwp(C_{die}, r) \sqsubseteq \frac{2.625}{0.75} = 3.5$. Figure 20 shows the HeyVL encoding of C_{die} (cleaned up for readability). As shown in Figure 21, the proof obligations $wp(C_{die}, r) \sqsubseteq 2.625$ and $0.75 \sqsubseteq wlp(C_{die}, 1)$ are then encoded using a coprocedure for the upper bound and a procedure for the lower bound, respectively.

There exist alternative interpretations of conditioning. For instance, Nori et al. [2014] use $wp(C, 1)(\sigma)$ in the denominator in the above fraction. A benefit of HeyVL is that such alternative interpretations can be realized by a straightforward adaptation of our encoding.

```
assert $I$;
havoc $variables$;
validate;
assume $I$;
if ($b$) {
    $enc_{\mathsf{wlp}}\lfloor C \rfloor$;
    assert $I$;
    assume $?(\mathsf{false})$
} else { }    // $\varphi$
```

Fig. 22. Encoding of Park Induction rule for underapproximating $wlp(\texttt{while } (b)\ \{C\}, \varphi)$.

```
coassert $exp(0.5, len(l))$;
cohavoc $l$; cohavoc $tmp$;
covalidate;
coassume $len(l)$;
if ($len(l) > 0$) {
    var $tmp\colon \mathbb{B}$ :≈ flip(0.5)
    if ($tmp$) { $l$ := $tail(l)$ } else { assert 0 }
    coassert $exp(0.5, len(l))$; coassume $\mathrm{co?}(\mathsf{false})$
} else { }    // 1
```

Fig. 23. Exemplary HeyVL encoding overapproximating the wp of a loop.

4.2 Reasoning about Expected Values of Loops

We encoded various proof rules for loops $\texttt{while } (b)\ \{C\}$ in HeyVL. As an example, we consider the Park induction rule [Kaminski 2019; Park 1969] for lower bounds on weakest liberal preexpectations: for all $\varphi, I \sqsubseteq 1$,

$$\underbrace{I \ \sqsubseteq\ (?(b) \longrightarrow wlp(C,I)) \sqcap (?(\neg b) \longrightarrow \varphi)}_{I \text{ is an inductive invariant}} \quad \text{implies} \quad \underbrace{I \ \sqsubseteq\ wlp(\texttt{while } (b)\ \{C\}, \varphi)}_{I \text{ underapproximates the loop's } wlp}.$$

The rule can be viewed as a quantitative version of the loop rule from Hoare [1969] logic, where I is an *inductive invariant* underapproximating the expected value of any loop iteration. Figure 22 depicts an encoding $enc_{\mathsf{wlp}}\lfloor \texttt{while } (b)\ \{C\} \rfloor$ that underapproximates $wlp(\texttt{while } (b)\ \{C\}, \varphi)$, i.e.

$$\mathsf{vp}[\![enc_{\mathsf{wlp}}\lfloor \texttt{while } (b)\ \{C\}\rfloor]\!](\varphi) \ =\ \begin{cases} I, & \text{if } I \sqsubseteq (?(b) \longrightarrow wlp(C,I)) \sqcap (?(\neg b) \longrightarrow \varphi) \\ 0, & \text{otherwise} \end{cases} \ \sqsubseteq\ wlp(\ldots, \varphi).$$

Before we go into details, we remark for readers familiar with classical deductive verification that our encoding is almost identical to standard loop encodings (cf. [Müller 2019]). Apart from the quantitative interpretation of statements, the only exception is the `validate` in line 3.

It is instructive to go over the encoding in Figure 22 step by step for a given initial state σ. The following expanded version of the above equation's right-hand side serves as a roadmap:

$$I(\sigma) \sqcap \inf_{\sigma' \in \mathsf{States}} \begin{cases} \infty, & \text{if } I(\sigma') \ \leq\ (?(b)(\sigma') \longrightarrow wlp(C,I)(\sigma')) \ \sqcap\ (?(\neg b)(\sigma') \longrightarrow \varphi(\sigma')) \\ 0, & \text{otherwise,} \end{cases}$$

Reading the HeyVL code in Figure 22 top-down then corresponds to reading the equation from left to right as indicated by the colors. We first assert that our underapproximation of the loop's *wlp* is at most I. The remaining code will ensure that said underapproximation is exactly I whenever I is an inductive loop invariant; it will be 0 otherwise. Proving that I is an inductive loop invariant requires checking an inequality $\sqsubseteq$, where $\psi \sqsubseteq \rho$ holds iff $\psi(\sigma') \leq \rho(\sigma')$ for all states σ'. We havoc the values of all program variables such that the invariant check encoded afterward is performed for every evaluation of the program variables, i.e. for every state σ'.[19] Moreover, havoc picks the

[19] An optimized encoding may only havoc those variables that are modified in the loop body. However, we opted to encode the rule as it is typically presented in the literature.

minimal result of all those invariant checks. The statement "I is an inductive loop invariant" is inherently qualitative. We thus validate that the invariant check encoded next is a qualitative statement that can only have two results: ∞ if I is an inductive invariant and 0 if it is not. To check if I is an inductive invariant for a fixed state σ', we need to prove an inequality, namely that $I(\sigma')$ lower bounds $wlp(C, I)(\sigma')$ if loop guard b holds and $\varphi(\sigma')$ if b does not hold. We first use `assume` I to lower the threshold for the expected value of the remaining code to be considered ∞ to $I(\sigma')$. Hence, we obtain ∞ if the invariant check succeeds for σ'. The conditional choice is the invariant check's right-hand side. If state σ' satisfies b, we use our existing *wlp* encoding to compute $wlp(C, I)(\sigma')$, where `assert` I`;` `assume` $?(\mathsf{false})$ ensures that *wlp* is computed with respect to postexpectation I. If state σ' satisfies $\neg b$, we do nothing and just take the postexpectation φ.

Upper bounds. Consider an iterative version of the lossy list traversal from Figure 4 on page 3:

$$\texttt{while}\ (len(l) > 0)\ \{\ \{\ l\ :=\ pop(l)\ \}\ [0.5]\ \{\ foo(l)\ \}\ \}$$

The Park induction rule can also be used to *over*approximate weakest preexpectations. The encoding is dual, i.e. it suffices to use the *co*-versions of the involved statements. For example, Figure 23 encodes the above loop with $exp(0.5, len(l))$ as inductive invariant overapproximating the loop's termination probability. The list type and the exponential function $exp(0.5, len(l))$ are represented in HeyLo by custom domain declarations (cf. Section 5.1).

Recursion. We can encode verification of wlp-lower bounds for recursive procedure calls of pGCL programs as discussed in Section 3.5 and justified by Olmedo et al. [2016] and Matheja [2020] – it is another application of Park induction. For wp-upper bounds, the encoding is dual. Hence, Figure 7 on page 5 encodes that the termination probability of the program in Figure 4 is at most $0.5^{len(l)}$.

4.3 Overview of Encodings

Table 1 summarizes all verification techniques – program logics and proof rules – that have been encoded in HeyVL. While a detailed discussion is beyond the scope of this paper, we briefly go over Table 1. The main takeaway is that HeyVL enables the encoding – and thus automation – of advanced verification methods based on diverse theoretical foundations and targeting different verification problems. The practicality of our encodings will be evaluated in Section 5.

Expected Values. We encoded McIver and Morgan [2005]'s weakest (liberal) preexpectation calculus for analyzing expected values of probabilistic programs (cf. Section 4.1). To analyze *conditional* expected values, we combined the two calculi as suggested by Olmedo et al. [2018]. For loops, we encoded three proof rules based on domain theory:

First, *Park Induction* generalizes the standard loop rule from Hoare logic [Hoare 1969] to a quantitative setting; it can be applied to lower bound weakest liberal preexpectations and upper bound weakest preexpectations (cf. Section 4.2). However, it is unsound for the converse directions.

Second, *ω-Invariants* are sound and complete for proving lower and upper bounds. However, they are arguably more complex because users must provide a family of invariants and compute limits. We modeled families of invariants as HeyLo formulas with additional free variables and used `havoc` x and `cohavoc` x to represent limits.

Third, we encoded a quantitative version of *k-induction* (for proving upper bounds) – an established verification technique (cf. [Sheeran et al. 2000]). The encodings are based on latticed k-induction [Batz et al. 2021a], a generalization of k-induction to arbitrary complete lattices. After encoding k-induction for upper bounds on wp, we benefited from the duality of HeyVL statements: we obtained a dual encoding for lower bounds on wlp that has, to our knowledge, not been implemented before. Furthermore, we encoded an advanced proof rule for lower bounds on expected

Table 1. Verification techniques encoded in HeyVL sorted by verification problem: lower- and upper bounds on probability of events (LPROB and UPROB), upper- and lower bounds on expected values (UEXP and LEXP), conditional expected values (CEXP), almost-sure termination (AST), positive almost-sure termination (PAST), upper bounds on expected runtimes (UERT), and lower bounds on expected runtimes (LERT).

Problem	**Verification Technique**	**Source**	**Encoding**
LPROB	wlp + Park induction	McIver and Morgan [2005]	Section 4.2
	wlp + latticed k-induction	(new?)	Appendix C.1
UPROB	wlp + ω-invariants	Kaminski [2019]	Appendix C.3
UEXP	wp + Park induction	McIver and Morgan [2005]	Appendix C.2
	wp + latticed k-induction	Batz et al. [2021a]	Appendix C.2
LEXP	wp + ω-invariants	Kaminski [2019]	Appendix C.4
	wp + Optional Stopping Theorem	Hark et al. [2019]	Appendix C.5
CEXP	conditional wp	Olmedo et al. [2018]	Section 4.1
UERT	ert calculus + UEXP rules	Kaminski et al. [2016]	Appendix C.6
LERT	ert calculus + ω-invariants	Kaminski et al. [2016]	Appendix C.6
AST	parametric super-martingale rule	McIver et al. [2018]	Appendix C.7
PAST	program analysis with martingales	Chakarov and Sankaranarayanan [2013]	Appendix C.8

values by Hark et al. [2019]. In contrast to the above rules, this rule is based on stochastic processes, particularly the Optional Stopping Theorem. Using our encoding, we automated the main examples in [Hark et al. 2019].

Expected Runtimes. To analyze the performance of randomized algorithms, we encoded the expected runtime calculus by Kaminski et al. [2016, 2018] and its recent extension to amortized analysis [Batz et al. 2023b]. Although reasoning about expected runtimes of loops involves some subtleties, we could adapt our HeyVL encodings for expected values by inserting `reward` statements. We encoded and automated examples from [Kaminski et al. 2016, 2018] and [Ngo et al. 2018].

Almost-Sure Termination (AST). McIver et al. [2018] proposed a proof rule for almost-sure termination – does a probabilistic program terminate with probability one? The rule is based on a parametric martingale that must satisfy four conditions, which we encoded in separate HeyVL (co)procedures. We automated the verification of their examples, including the one in Figure 5.

Positive Almost-Sure Termination (PAST). PAST is a stronger notion than almost-sure termination, which requires a program's expected runtime to be finite. We can apply our HeyVL encodings for upper bounding expected runtimes to prove PAST. Moreover, we encoded a dedicated proof rule for PAST by Chakarov and Sankaranarayanan [2013] based on martingales and concentration bounds.

5 IMPLEMENTATION

We first describe user-defined types and functions by means of *domain declarations* in Section 5.1. We then describe our tool Caesar alongside with empirical results validating the feasibility of our deductive verification infrastructure for the automated verification of probabilistic programs.

5.1 Domain Declarations

Recall from Section 2 that we assume all type- and function symbols to be interpreted. In practice, we support custom first-order theories via *domain declarations* as is standard in classical deductive verification infrastructures [Müller et al. 2016b]. A domain declaration introduces a new type symbol alongside with a set of typed function symbols and first-order formulae (called *axioms*) characterizing feasible interpretations of the type- and function symbols.

Consider the harmonic numbers — often required for, e.g., expected runtime analysis — as an example. The n-th harmonic number is given by $H_n = \sum_{k=1}^{n} \frac{1}{k}$. To enable reasoning about verification problems involving the harmonic numbers, we introduce the following domain declaration:

$$\begin{array}{ll} \texttt{domain}\ \mathit{HarmonicNums}\ \{ \qquad & \texttt{func}\ H(n\colon \mathbb{N})\colon \mathbb{R}_{\geq 0} \\ & \texttt{axiom}\ h_0\ H(0) = 0 \\ & \texttt{axiom}\ h_n\ \forall n\colon \mathbb{N}.\ H(n+1) = H(n) + {}^{1}/_{n+1} \qquad \} \end{array}$$

HarmonicNums introduces a new function symbol $H\colon \mathbb{N} \longrightarrow \mathbb{R}_{\geq 0}$ and two axioms h_0 and h_n characterizing feasible interpretations of H recursively. Other non-linear functions such as exponential functions (e.g., $exp(0.5, n)$ from Section 4.2) as well as algebraic data types can be defined in a similar way (see, e.g., [Müller et al. 2016a]). In our implementation, validity of verification conditions — inequalities between HeyLo formulae — is defined *modulo* validity of all user-provided axioms.

5.2 The Verifier Caesar

We have implemented HeyVL in our tool Caesar[20] which consists of approximately 10k lines of Rust code. Caesar takes as input a HeyVL program C and a set of domain declarations (cf. Section 5.1). It then generates all verification conditions described by C, i.e, inequalities between HeyLo formulae of the form $\varphi \sqsubseteq \mathsf{vp}[\![S]\!](\psi)$ or $\varphi \sqsupseteq \mathsf{vp}[\![S]\!](\psi)$, and translates these verification conditions to a Satisfiability Modulo Theories (SMT) query. Our SMT back end is z3 [de Moura and Bjørner 2008]. Since the translation to SMT can involve undecidable theories, Caesar might return *unknown*. Otherwise, Caesar either returns *verified* or *not verified*. In the latter case, z3 often reports a counterexample state witnessing the violation of one of the verification conditions, which helps, e.g., debugging loop invariants.

Moreover, we have implemented a *prototypical front-end* that translates (numeric) pGCL programs and their specifications to HeyVL, and invokes Caesar for automated verification. Currently, it supports all techniques from Table 1 targeting loops.

SMT Encodings and Optimizations. We translate validity of inequalities between HeyLo to SMT following the semantics of formulae from Figure 8.

To encode the sort $\mathbb{R}_{\geq 0}^{\infty}$, we evaluated to two options, which are both supported by our implementation. The first option represents every number of sort $\mathbb{R}_{\geq 0}^{\infty}$ as a pair $(r, \mathit{isInfty})$, where r is a real number and *isInfty* is a Boolean flag that is true if and only if the represented number is equal to ∞. We add constraints $r \geq 0$ to ensure that r is non-negative. All operations on $\mathbb{R}_{\geq 0}^{\infty}$ are then defined over such pairs. For example, the addition $(r_1, \mathit{isInfty}_1) + (r_2, \mathit{isInfty}_2)$ is defined as $(r_1 + r_2, \mathit{isInfty}_1 \vee \mathit{isInfty}_2)$. For multiplication, we ensure that $0 \cdot \infty = \infty$ – a common assumption in probability theory. The second option leverages Z3-specific data type declarations to specify values that are either infinite or non-negative reals. We observed that the first option performs better overall and thus use it by default.

The ℒ- and Ƨ quantifiers are translated using the textbook definition of infima and suprema over $\mathbb{R}_{\geq 0}^{\infty}$, but are eliminated whenever possible using that for $A \subseteq \mathbb{R}_{\geq 0}^{\infty}$ and $r \in \mathbb{R}_{\geq 0}^{\infty}$, we have

$$\sup A \leq r \quad \text{iff} \quad \forall a \in A\colon a \leq r \qquad \text{and dually} \qquad r \leq \inf A \quad \text{iff} \quad \forall a \in A\colon r \leq a\ .$$

Finally, we simplify sub-formulae by, e.g., rewriting $?(b) \sqcap \psi$ to 0 if b is unsatisfiable.

[20] All tools and benchmarks are available as open-source software at https://github.com/moves-rwth/caesar.

Benchmarks. To validate whether our implementation is capable of verifying interesting quantitative properties of probabilistic programs, we have considered various verification problems taken from the literature. These benchmarks involve unbounded probabilistic loops or recursion and include quantitative correctness properties of communication protocols [D'Argenio et al. 1997; Helmink et al. 1993] and randomised algorithms [Hurd et al. 2005; Kushilevitz and Rabin 1992; Lumbroso 2013], bounds on expected runtimes of stochastic processes [Kaminski et al. 2020, 2018; Ngo et al. 2018], proofs of *positive* almost-sure termination [Chakarov and Sankaranarayanan 2013] and proofs of almost-sure termination for the case studies provided in [McIver et al. 2018]. For each of these benchmarks, we apply the HeyVL encodings provided in Section 4 and Appendix C, and cover all verification techniques from Table 1.

Table 2 summarizes the results of our benchmarks. For each benchmark, it provides the benchmark name, the verification problem, the encoded techniques (cf. Table 1), the lines of HeyVL code (without comments), notable features, and running time. For the running time, we also provide the shares of pruning. i.e. simplification of sub-formulae, and the final SAT check. Table 1 together with the column "Problem" provides pointers to each benchmark's source and encoding. For latticed k-induction, we indicate the value of k that was used for the encoding. Benchmarks that use exponential functions (e.g. rabin, zeroconf) or harmonic numbers (e.g. ast) are marked with F1. Benchmarks that use multiple possibly mixed (co)procedures are marked with F2. One example encodes verification of nested loops (feature F3).

The size of our benchmarks ranges from 19-224 lines of HeyVL code. 85% of our benchmarks (those shaded in gray) have been verified with our front-end; the remaining encodings are handcrafted. All benchmark files are available as part of our artifact.

Evaluation. On average, Caesar needs 0.2 seconds to verify a HeyVL program, with a maximum of 2.3 seconds. Most benchmarks verify within less than a second. The brp3 benchmark times out because of the large nested branching resulting from the exponential size of the k-induction encoding with $k = 23$.

We conclude that Caesar is capable of verifying interesting quantitative verification problems of probabilistic programs taken from the literature. Moreover, we conclude that modern SMT solvers are a suitable back-end besides the fact that our benchmarks often require reasoning about highly non-linear functions. This is due to the fact that it often suffices to (un)fold recursive definitions of, e.g., the harmonic numbers, finitely many times. Finally, our benchmarks demonstrate that our verification infrastructure provides a unifying interface for *encoding and solving* various kinds of probabilistic verification problems in an automated manner.

6 RELATED WORK

We focus on automated verification techniques for probabilistic programs and deductive verification infrastructures for non-probabilistic programs; encoded proof rules have been discussed in Section 4.

Probabilistic Program Verification. Expectation-based probabilistic program verification has been pioneered by Kozen [1983, 1985] and McIver & Morgan [McIver and Morgan 2005]. Hurd et al. [2005] formalised the w(l)p calculus in *Isabelle/HOL* [Nipkow et al. 2002]. They focus on the calculus' meta theory and provide a verification-condition generator for proving partial correctness. Hölzl [2016] implemented the meta theory of Kaminski et al. [2016]'s ert calculus in *Isabelle/HOL* and verified bounds on expected runtimes of randomised algorithms. We focus on unifying verification techniques in a single infrastructure.

Easycrypt [Barthe et al. 2013, 2011] is a theorem prover for verifying cryptographic protocols, featuring libraries for data structures and algebraic reasoning. *Ellora* [Barthe et al. 2018] is an assertion-based program logic for probabilistic programs implemented in *Easycrypt*, taking benefit

Name	Problem	Verification Technique	LOC	Features	Total (s)	Pruning	SAT
rabin	LPROB	wlp + Park induction	43	F1, F3	0.33	3%	96%
unif_gen1	LPROB	wlp + Latticed k-induction ($k = 2$)	61		0.02	52%	35%
unif_gen2	LPROB	wlp + Latticed k-induction ($k = 3$)	82		0.05	68%	25%
unif_gen3	LPROB	wlp + Latticed k-induction ($k = 3$)	82		0.05	71%	22%
unif_gen4	LPROB	wlp + Latticed k-induction ($k = 5$)	124		0.86	90%	7%
rabin1	LPROB	wlp + Park induction	36		0.01	45%	40%
rabin2	LPROB	wlp + Latticed k-induction ($k = 5$)	116		0.08	27%	67%
chain	UEXP	wp + Park induction	28	F1	0.03	24%	66%
ohfive	UEXP	wp + Park induction	34	F1, F3	0.02	33%	56%
brp1	UEXP	wp + Latticed k-induction ($k = 5$)	72		0.03	45%	42%
brp2	UEXP	wp + Latticed k-induction ($k = 11$)	138		0.46	70%	16%
brp3	UEXP	wp + Latticed k-induction ($k = 23$)	270		TO		
geo1	UEXP	wp + Latticed k-induction ($k = 2$)	32		0.02	44%	41%
geo (recursive)	UEXP	wp + Park induction	19		0.02	43%	42%
rabin1	UEXP	wp + Park induction	36		0.02	44%	73%
rabin2	UEXP	wp + Latticed k-induction ($k = 5$)	116		0.12	22%	46%
unif_gen1	UEXP	wp + Latticed k-induction ($k = 2$)	61		0.03	44%	46%
unif_gen2	UEXP	wp + Latticed k-induction ($k = 3$)	82		0.11	41%	53%
unif_gen3	UEXP	wp + Latticed k-induction ($k = 3$)	82		0.10	41%	53%
unif_gen4	UEXP	wp + Latticed k-induction ($k = 5$)	124		2.26	47%	49%
zeroconf	UEXP	wp + Park induction	43	F1, F2	0.03	36%	49%
ost	LEXP	wp + Optional Stopping Theorem	93	F2	0.07	33%	51%
die	CEXP	conditional wp	22	F2	0.02	17%	63%
2drwalk	UERT	ert + Park induction	224		0.02	41%	44%
bayesian_network	UERT	ert + Park induction	107		0.02	45%	40%
C4b_t303	UERT	ert + Latticed k-induction ($k = 3$)	73		0.03	29%	58%
condand	UERT	ert + Park induction	24		0.02	42%	42%
fcall	UERT	ert + Park induction	26		0.02	52%	44%
hyper	UERT	ert + Park induction	31		0.02	41%	44%
linear01	UERT	ert + Park induction	23		0.02	42%	43%
prdwalk	UERT	ert + Park induction	62		0.02	56%	31%
prspeed	UERT	ert + Park induction	45		0.02	41%	45%
rdspeed	UERT	ert + Park induction	48		0.02	38%	47%
rdwalk	UERT	ert + Park induction	24		0.02	42%	43%
sprdwalk	UERT	ert + Park induction	26		0.02	42%	43%
omega	LERT	ert + ω-invariants	33	F2	0.02	42%	47%
ast1	AST	parametric super-martingale rule	67	F2	0.06	33%	49%
ast2	AST	parametric super-martingale rule	79	F2	0.05	38%	50%
ast3	AST	parametric super-martingale rule	65	F1, F2	1.94	1%	99%
ast4	AST	parametric super-martingale rule	55	F2	0.05	33%	52%
past	PAST	program analysis with martingales	26	F2	0.04	40%	46%

Table 2. Benchmarks. Rows shaded in gray indicate HeyVL examples automatically generated from pGCL code with annotations using our frontend. Timeout (TO) was set to 10 seconds. Verification techniques correspond to those presented in Table 1. Lines of HeyVL code (LOC) are counted without comments. Features: user-defined uninterpreted functions (F1), multiple (co)procedures (F2), nested loops (F3).

from *Easycrypt*'s features. Their specifications are predicates over (sub)distributions instead of expectations. While *Ellora* employs *specialised* proof rules for loops and does not support non-determinism or recursion, thus being more restrictive than HeyVL in this regard, *Ellora* embeds, e.g., logics for reasoning about probabilistic independence. As stated in [Barthe et al. 2018], an in-depth comparison of assertion- and expectation-based approaches is difficult. Pardo et al. [2022] propose a propositional dynamic logic for pGCL featuring reasoning about convergence of estimators. Their logic is not automated yet.

Fully automatic analyses of probabilistic programs are limited to specific properties, e.g. bounding expected runtimes or proving (positive) almost-sure termination [Abate et al. 2021; Avanzini et al. 2020; Batz et al. 2023a, 2018; Chatterjee et al. 2016, 2017; Fioriti and Hermanns 2015; Fu and Chatterjee 2019; Leutgeb et al. 2022; Meyer et al. 2021; Moosbrugger et al. 2021a,b; Ngo et al. 2018]. We might also benefit from invariant synthesis approaches [Agrawal et al. 2018; Amrollahi et al. 2022; Bao et al. 2022; Barthe et al. 2016; Bartocci et al. 2020; Batz et al. 2023a, 2020; Chakarov and Sankaranarayanan 2013; Chen et al. 2015; Feng et al. 2017; Katoen et al. 2010; Susag et al. 2022].

Deductive Verification Infrastructures. Boogie [Leino 2008] and Why3 [Filliâtre and Paskevich 2013] are prominent examples of IVLs for non-probabilistic programs that lie at the foundation of various modern verifiers, such as Dafny [Leino 2010] and Frama-C [Kirchner et al. 2015]. Neither of these IVLs targets reasoning about expectations or upper bounds (aka necessary preconditions [Cousot et al. 2011]). For example, Boogie's statements are specific to verifying lower bounds on Boolean predicates. Evaluating whether our implementation could benefit from encoding HeyLo formulae into Why3 is interesting future work.

7 CONCLUSION AND FUTURE WORK

We have presented a verification infrastructure for probabilistic programs based on a novel quantitative intermediate verification language that aids researchers with prototyping and automating their proof rules. As future work, we plan to automate more rules and explore the relationship between our language, particularly its dual operators, and (partial) incorrectness logic [O'Hearn 2020; Zhang and Kaminski 2022]. A further promising direction is to generalize our infrastructure for the verification of probabilistic pointer programs [Batz et al. 2022a, 2019] and weighted programs [Batz et al. 2022b].

Furthermore, establishing a formal "ground truth" for our intermediate language HeyVL in terms of an operational semantics that assigns precise meaning to quantitative Hoare triples, which we admittedly introduced ad-hoc, is important future work. However, defining an operational semantics that yields a *pleasant forward-reading* intuition for all statements in our intermediate language HeyVL appears non-trivial. In particular, we are unaware of a semantics for (co)assume statements that is independent of the semantics of the remaining program. We believe that stochastic games might be an adequate formalism but the details have not been worked out yet.

DATA-AVAILABILITY STATEMENT

The tool Caesar, our prototypical front-end for pGCL programs, as well as our benchmarks that we submitted for the artifact evaluation are available [Schroer et al. 2023]. We also develop our tools as open-source software at https://github.com/moves-rwth/caesar.

ACKNOWLEDGMENTS

This work was partially supported by the Digital Research Centre Denmark (DIREC), the ERC Advanced Research Grant FRAPPANT (grant no. 787914), and the 2022 WhatsApp Privacy Aware Program Analysis Research Award.

REFERENCES

Alessandro Abate, Mirco Giacobbe, and Diptarko Roy. 2021. Learning Probabilistic Termination Proofs. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12760)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 3–26. https://doi.org/10.1007/978-3-030-81688-9_1

Sheshansh Agrawal, Krishnendu Chatterjee, and Petr Novotný. 2018. Lexicographic ranking supermartingales: an efficient approach to termination of probabilistic programs. *Proc. ACM Program. Lang.* 2, POPL (2018), 34:1–34:32. https://doi.org/10.1145/3158122

Daneshvar Amrollahi, Ezio Bartocci, George Kenison, Laura Kovács, Marcel Moosbrugger, and Miroslav Stankovic. 2022. Solving Invariant Generation for Unsolvable Loops. In *Static Analysis - 29th International Symposium, SAS 2022, Auckland, New Zealand, December 5-7, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13790)*, Gagandeep Singh and Caterina Urban (Eds.). Springer, 19–43. https://doi.org/10.1007/978-3-031-22308-2_3

Martin Avanzini, Georg Moser, and Michael Schaper. 2020. A modular cost analysis for probabilistic programs. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 172:1–172:30. https://doi.org/10.1145/3428240

M. Baaz. 1996. Infinite-Valued Gödel Logics with 0-1-Projections and Relativizations. In *Proc. Gödel'96, Logic Foundations of Mathematics, Computer Science and Physics – Kurt Gödel's Legacy (Lecture Notes in Logic 6)*, P. Hájek (Ed.). Springer, Brno, Czech Republic.

Jialu Bao, Nitesh Trivedi, Drashti Pathak, Justin Hsu, and Subhajit Roy. 2022. Data-Driven Invariant Learning for Probabilistic Programs. In *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13371)*, Sharon Shoham and Yakir Vizel (Eds.). Springer, 33–54. https://doi.org/10.1007/978-3-031-13185-1_3

Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. 2013. EasyCrypt: A Tutorial. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures (Lecture Notes in Computer Science, Vol. 8604)*, Alessandro Aldini, Javier López, and Fabio Martinelli (Eds.). Springer, 146–166. https://doi.org/10.1007/978-3-319-10082-1_6

Gilles Barthe, Thomas Espitau, Luis María Ferrer Fioriti, and Justin Hsu. 2016. Synthesizing Probabilistic Invariants via Doob's Decomposition. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9779)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 43–61. https://doi.org/10.1007/978-3-319-41528-4_3

Gilles Barthe, Thomas Espitau, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2018. An Assertion-Based Program Logic for Probabilistic Programs. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Amal Ahmed (Ed.). Springer International Publishing, Cham.

Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. 2011. Computer-Aided Security Proofs for the Working Cryptographer. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6841)*, Phillip Rogaway (Ed.). Springer, 71–90. https://doi.org/10.1007/978-3-642-22792-9_5

Gilles Barthe, Joost-Pieter Katoen, and Alexandra Silva (Eds.). 2020. *Foundations of Probabilistic Programming*. Cambridge University Press, Cambridge.

Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. 2020. Mora - Automatic Generation of Moment-Based Invariants. 12078 (2020), 492–498.

Kevin Batz, Mingshuai Chen, Sebastian Junges, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2023a. Probabilistic Program Verification via Inductive Synthesis of Inductive Invariants. In *TACAS (2) (Lecture Notes in Computer Science, Vol. 13994)*. Springer, 410–429.

Kevin Batz, Mingshuai Chen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Philipp Schröer. 2021a. Latticed k-Induction with an Application to Probabilistic Programs. In *CAV (2) (Lecture Notes in Computer Science, Vol. 12760)*. Springer, 524–549.

Kevin Batz, Ira Fesefeldt, Marvin Jansen, Joost-Pieter Katoen, Florian Keßler, Christoph Matheja, and Thomas Noll. 2022a. Foundations for Entailment Checking in Quantitative Separation Logic. 13240 (2022), 57–84.

Kevin Batz, Adrian Gallus, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Tobias Winkler. 2022b. Weighted Programming: A Programming Paradigm for Specifying Mathematical Models. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (April 2022).

Kevin Batz, Sebastian Junges, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Philipp Schröer. 2020. PrIC3: Property Directed Reachability for MDPs. *Computer Aided Verification* (2020).

Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2018. How long, O Bayesian network, will I sample thee? - A program analysis perspective on expected sampling times. 10801 (2018), 186–213.

Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2021b. Relatively complete verification of probabilistic programs: an expressive language for expectation-based reasoning. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30.

Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Lena Verscht. 2023b. A Calculus for Amortized Expected Runtimes. *Proc. ACM Program. Lang.* 7, POPL (2023), 1957–1986.

Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. 2019. Quantitative Separation Logic: A Logic for Reasoning about Probabilistic Pointer Programs. *Proceedings of the ACM on Programming*

Languages 3, POPL (Jan. 2019).

Aleksandar Chakarov and Sriram Sankaranarayanan. 2013. Probabilistic Program Analysis with Martingales. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 511–526. https://doi.org/10.1007/978-3-642-39799-8_34

Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2016. Termination Analysis of Probabilistic Programs Through Positivstellensatz's. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9779)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 3–22. https://doi.org/10.1007/978-3-319-41528-4_1

Krishnendu Chatterjee, Petr Novotný, and Dorde Zikelic. 2017. Stochastic invariants for probabilistic termination. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 145–160. https://doi.org/10.1145/3009837.3009873

Yu-Fang Chen, Chih-Duo Hong, Bow-Yaw Wang, and Lijun Zhang. 2015. Counterexample-Guided Polynomial Loop Invariant Generation by Lagrange Interpolation. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9206)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 658–674. https://doi.org/10.1007/978-3-319-21690-4_44

Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. 2013. Automatic Inference of Necessary Preconditions. In *Verification, Model Checking, and Abstract Interpretation (Lecture Notes in Computer Science)*, Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni (Eds.). Springer, Berlin, Heidelberg.

Patrick Cousot, Radhia Cousot, and Francesco Logozzo. 2011. Precondition Inference from Intermittent Assertions and Application to Contracts on Collections. In *Verification, Model Checking, and Abstract Interpretation*, Ranjit Jhala and David Schmidt (Eds.). Vol. 6538. Springer Berlin Heidelberg, Berlin, Heidelberg.

Pedro R. D'Argenio, Joost-Pieter Katoen, Theo C. Ruys, and Jan Tretmans. 1997. The Bounded Retransmission Protocol Must Be on Time!. In *Tools and Algorithms for Construction and Analysis of Systems, Third International Workshop, TACAS '97, Enschede, The Netherlands, April 2-4, 1997, Proceedings (Lecture Notes in Computer Science, Vol. 1217)*, Ed Brinksma (Ed.). Springer, 416–431. https://doi.org/10.1007/BFb0035403

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, Berlin, Heidelberg.

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

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 - Where Programs Meet Provers. In *ESOP (Lecture Notes in Computer Science, Vol. 7792).* Springer, 125–128.

Luis María Ferrer Fioriti and Holger Hermanns. 2015. Probabilistic Termination: Soundness, Completeness, and Compositionality. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 489–501. https://doi.org/10.1145/2676726.2677001

Hongfei Fu and Krishnendu Chatterjee. 2019. Termination of Nondeterministic Probabilistic Programs. In *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11388)*, Constantin Enea and Ruzica Piskac (Eds.). Springer, 468–490. https://doi.org/10.1007/978-3-030-11245-5_22

Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. 2014. Probabilistic Programming. In *Proceedings of the on Future of Software Engineering (FOSE 2014).* ACM, New York, NY, USA.

Marcel Hark, Benjamin Lucien Kaminski, Jürgen Giesl, and Joost-Pieter Katoen. 2019. Aiming Low Is Harder: Induction for Lower Bounds in Probabilistic Program Verification. *Proceedings of the ACM on Programming Languages* 4, POPL (Dec. 2019).

Leen Helmink, M. P. A. Sellink, and Frits W. Vaandrager. 1993. Proof-Checking a Data Link Protocol. In *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24-28, 1993, Selected Papers (Lecture Notes in Computer Science, Vol. 806)*, Henk Barendregt and Tobias Nipkow (Eds.). Springer, 127–165. https://doi.org/10.1007/3-540-58085-9_75

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

Johannes Hölzl. 2016. Formalising Semantics for Expected Running Time of Probabilistic Programs. In *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9807)*, Jasmin Christian Blanchette and Stephan Merz (Eds.). Springer, 475–482. https://doi.org/10.1007/978-

3-319-43144-4_30

J. Hurd, Annabelle McIver, and Carroll Morgan. 2005. Probabilistic Guarded Commands Mechanized in HOL. *Electron. Notes Theor. Comput. Sci.* (2005).

Benjamin Lucien Kaminski. 2019. *Advanced Weakest Precondition Calculi for Probabilistic Programs.* Ph.D. Dissertation. RWTH Aachen University.

Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2019. On the Hardness of Analyzing Probabilistic Programs. *Acta Informatica* 56, 3 (April 2019).

Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2020. Expected Runtime Analysis by Program Verification. In *Foundations of Probabilistic Programming*, Alexandra Silva, Gilles Barthe, and Joost-Pieter Katoen (Eds.). Cambridge University Press, Cambridge.

Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. 2016. Weakest Precondition Reasoning for Expected Run–Times of Probabilistic Programs. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Peter Thiemann (Ed.). Springer, Berlin, Heidelberg.

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

Joost-Pieter Katoen, Annabelle McIver, Larissa Meinicke, and Carroll C. Morgan. 2010. Linear-Invariant Generation for Probabilistic Programs: - Automated Support for Proof-Based Methods. In *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6337)*, Radhia Cousot and Matthieu Martel (Eds.). Springer, 390–406. https://doi.org/10.1007/978-3-642-15769-1_24

Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal Aspects Comput.* 27, 3 (2015), 573–609.

Stephen Cole Kleene. 1952. *Introduction to Metamathematics.* North Holland.

Dexter Kozen. 1983. A Probabilistic PDL. In *STOC*. ACM, 291–297.

Dexter Kozen. 1985. A Probabilistic PDL. *J. Comput. Syst. Sci.* 30, 2 (1985), 162–178.

Eyal Kushilevitz and Michael O. Rabin. 1992. Randomized Mutual Exclusion Algorithms Revisited. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, August 10-12, 1992*, Norman C. Hutchinson (Ed.). ACM, 275–283. https://doi.org/10.1145/135419.135468

K. Rustan M. Leino. 2008. *This Is Boogie 2.*

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (Lecture Notes in Computer Science)*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer, Berlin, Heidelberg.

Lorenz Leutgeb, Georg Moser, and Florian Zuleger. 2022. Automated Expected Amortised Cost Analysis of Probabilistic Data Structures. , 70–91 pages.

Jérémie O. Lumbroso. 2013. Optimal Discrete Uniform Generation from Coin Flips, and Applications. *CoRR* abs/1304.1916 (2013). arXiv:1304.1916 http://arxiv.org/abs/1304.1916

Christoph Matheja. 2020. *Automated reasoning and randomization in separation logic.* Ph.D. Dissertation. RWTH Aachen University, Germany.

Annabelle McIver, Carroll Morgan, Benjamin Lucien Kaminski, and Joost-Pieter Katoen. 2018. A New Proof Rule for Almost-Sure Termination. *Proceedings of the ACM on Programming Languages* 2, POPL (Jan. 2018).

Annabelle McIver and Charles Carroll Morgan. 2005. *Abstraction, Refinement and Proof for Probabilistic Systems.* Springer-Verlag, New York.

Fabian Meyer, Marcel Hark, and Jürgen Giesl. 2021. Inferring Expected Runtimes of Probabilistic Integer Programs Using Expected Sizes. In *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12651)*, Jan Friso Groote and Kim Guldstrand Larsen (Eds.). Springer, 250–269. https://doi.org/10.1007/978-3-030-72016-2_14

Marcel Moosbrugger, Ezio Bartocci, Joost-Pieter Katoen, and Laura Kovács. 2021a. Automated Termination Analysis of Polynomial Probabilistic Programs. In *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12648)*, Nobuko Yoshida (Ed.). Springer, 491–518. https://doi.org/10.1007/978-3-030-72019-3_18

Marcel Moosbrugger, Ezio Bartocci, Joost-Pieter Katoen, and Laura Kovács. 2021b. The Probabilistic Termination Tool Amber. In *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 13047)*, Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan (Eds.). Springer, 667–675. https://doi.org/10.1007/978-3-030-90870-6_36

Peter Müller. 2019. Building Deductive Program Verifiers - Lecture Notes. *Engineering Secure and Dependable Software Systems* (2019).

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016a. *Online appendix to Viper: A Verification Infrastructure for Permission-Based Reasoning.* http://viper.ethz.ch/examples/vmcai16/index.html

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016b. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9583)*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer, 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2018. Bounded Expectations: Resource Analysis for Probabilistic Programs. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018).* Association for Computing Machinery, New York, NY, USA.

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic.* Lecture Notes in Computer Science, Vol. 2283. Springer. https://doi.org/10.1007/3-540-45949-9

Aditya V. Nori, Chung-Kil Hur, Sriram K. Rajamani, and Selva Samuel. 2014. R2: An Efficient MCMC Sampler for Probabilistic Programs. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, Carla E. Brodley and Peter Stone (Eds.). AAAI Press, 2476–2482. http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8192

Peter W. O'Hearn. 2020. Incorrectness Logic. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2020).

Federico Olmedo, Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Annabelle Mciver. 2018. Conditioning in Probabilistic Programming. *ACM Transactions on Programming Languages and Systems* 40, 1 (Jan. 2018).

Federico Olmedo, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2016. Reasoning about Recursive Probabilistic Programs. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '16).* Association for Computing Machinery, New York, NY, USA.

Raúl Pardo, Einar Broch Johnsen, Ina Schaefer, and Andrzej Wasowski. 2022. A Specification Logic for Programs in the Probabilistic Guarded Command Language. In *ICTAC (Lecture Notes in Computer Science, Vol. 13572).* Springer, 369–387.

David Park. 1969. Fixpoint Induction and Proofs of Program Properties. *Machine Intelligence* 5 (1969).

Norbert Preining. 2010. Gödel Logics – A Survey. In *Logic for Programming, Artificial Intelligence, and Reasoning (Lecture Notes in Computer Science)*, Christian G. Fermüller and Andrei Voronkov (Eds.). Springer, Berlin, Heidelberg.

Philipp Schroer, Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2023. *A Deductive Verification Infrastructure for Probabilistic Programs - Artifact Evaluation.* https://doi.org/10.5281/zenodo.8146987

Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. 2000. Checking Safety Properties Using Induction and a SAT-Solver. In *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1954)*, Warren A. Hunt Jr. and Steven D. Johnson (Eds.). Springer, 108–125. https://doi.org/10.1007/3-540-40922-X_8

Zachary Susag, Sumit Lahiri, Justin Hsu, and Subhajit Roy. 2022. Symbolic execution for randomized programs. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 1583–1612. https://doi.org/10.1145/3563344

Toru Takisaka, Yuichiro Oyabu, Natsuki Urabe, and Ichiro Hasuo. 2021. Ranking and Repulsing Supermartingales for Reachability in Randomized Programs. *ACM Trans. Program. Lang. Syst.* 43, 2 (2021), 5:1–5:46.

Wikipedia. 2023a. Coupon Collector's Problem. https://en.wikipedia.org/wiki/Coupon_collector%27s_problem. [Online; accessed 4-September-2023].

Wikipedia. 2023b. Random Walk. https://en.wikipedia.org/wiki/Random_walk#One-dimensional_random_walk. [Online; accessed 4-September-2023].

Linpeng Zhang and Benjamin Lucien Kaminski. 2022. Quantitative strongest post: a calculus for reasoning about the flow of quantitative information. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–29.

A OMITTED PROOFS: HEYLO

Theorem 2.1 (Adjointness Properties). *For all* HeyLo *formulae φ, ψ, and ρ, we have*

$$\varphi \sqcap \psi ~\sqsubseteq~ \rho \quad\text{iff}\quad \varphi ~\sqsubseteq~ \psi \to \rho \qquad \textit{and} \qquad \psi \sqcup \rho ~\sqsupseteq~ \varphi \quad\text{iff}\quad \rho ~\sqsupseteq~ \psi \leftsquigarrow \varphi~.$$

Proof. Let $\varphi, \psi, \rho \in$ HeyLo. For the adjointness of $\to$ and $\sqcap$, consider the following:

$$\begin{aligned}
&\varphi \sqcap \psi \sqsubseteq \rho \\
&\quad\text{iff}\quad \forall \sigma \in \mathsf{States}.~ \min\left\{ [\![\varphi]\!](\sigma),~ [\![\psi]\!](\sigma) \right\} \leq [\![\rho]\!](\sigma) && \text{(Figure 8)} \\
&\quad\text{iff}\quad \forall \sigma \in \mathsf{States}.~ [\![\varphi]\!](\sigma) \leq [\![\rho]\!](\sigma) \vee [\![\psi]\!](\sigma) \leq [\![\rho]\!](\sigma) \\
&\quad\text{iff}\quad \forall \sigma \in \mathsf{States}.~ \begin{cases} \mathtt{true}, & \text{if } [\![\psi]\!](\sigma) \leq [\![\rho]\!](\sigma) \\ [\![\varphi]\!](\sigma) \leq [\![\rho]\!](\sigma), & \text{otherwise} \end{cases} \\
&\quad\text{iff}\quad \forall \sigma \in \mathsf{States}.~ [\![\varphi]\!](\sigma) \leq \begin{cases} \infty, & \text{if } [\![\psi]\!](\sigma) \leq [\![\rho]\!](\sigma) \\ [\![\rho]\!](\sigma), & \text{otherwise} \end{cases} \\
&\quad\text{iff}\quad \varphi \sqsubseteq \psi \to \rho. && \text{(Figure 8)}
\end{aligned}$$

The proof of adjointness of the coimplication $\leftsquigarrow$ and $\sqcup$ is analogous. □

Theorem 2.2 (HeyLo Deduction Theorem). *For all* HeyLo *formulae φ and ψ, we have*

$$\varphi ~\sqsubseteq~ \psi \quad\text{iff}\quad \varphi \to \psi \text{ is valid} \qquad \text{and} \qquad \varphi ~\sqsupseteq~ \psi \quad\text{iff}\quad \varphi \leftsquigarrow \psi \text{ is covalid}~.$$

Proof. Let $\varphi, \psi \in$ HeyLo. Then,

$$\begin{aligned}
&\varphi \sqsubseteq \psi \\
&\quad\text{iff}\quad \forall \sigma \in \mathsf{States}.~ [\![\varphi]\!](\sigma) \leq [\![\psi]\!](\sigma) && (\text{definition } \sqsubseteq) \\
&\quad\text{implies}\quad \forall \sigma \in \mathsf{States}.~ [\![\varphi \to \psi]\!](\sigma) = \infty && \text{(Figure 8)} \\
&\quad\text{iff}\quad \varphi \to \psi \text{ is valid} && \text{(definition of validity)}
\end{aligned}$$

and

$$\begin{aligned}
&\varphi \to \psi \text{ is valid} \\
&\quad\text{iff}\quad \forall \sigma \in \mathsf{States}.~ [\![\varphi \to \psi]\!](\sigma) = \infty && \text{(definition of validity)} \\
&\quad\text{implies}\quad \forall \sigma \in \mathsf{States}.~ \begin{cases} \infty = \infty, & \text{if } [\![\varphi]\!](\sigma) \leq [\![\psi]\!](\sigma) \\ [\![\psi]\!](\sigma) = \infty, & \text{otherwise} \end{cases} && \text{(Figure 8)} \\
&\quad\text{implies}\quad \varphi \sqsubseteq \psi.
\end{aligned}$$

The proof of the second equivalence is analogous. □

B OMITTED PROOFS: HEYVL

B.1 Properties of HeyVL

Recall Theorem 3.2 (page 16):

Theorem 3.2 (Monotonicity of vp). *For all* HeyVL *statements S and* HeyLo *formulae φ, φ',*

$$\varphi ~\sqsubseteq~ \varphi' \quad \textit{implies} \quad \mathsf{vp}[\![S]\!](\varphi) ~\sqsubseteq~ \mathsf{vp}[\![S]\!](\varphi')~.$$

Proof. Let $S \in \mathsf{HeyVL}$. We do a proof by induction over the structure of S to show

$$\forall \varphi, \varphi' \in \mathsf{HeyLo}. \quad \varphi \sqsubseteq \varphi' \quad \text{implies} \quad \mathsf{vp}\llbracket S \rrbracket(\varphi) \sqsubseteq \mathsf{vp}\llbracket S \rrbracket(\varphi')~.$$

For the base cases, let $\varphi, \varphi' \in \mathsf{HeyLo}$ such that $\varphi \sqsubseteq \varphi'$.

- Case $S = \mathtt{var}\ x \colon \tau :\approx \mu$.

$$\begin{aligned}
&\mathsf{vp}\llbracket \mathtt{var}\ x \colon \tau :\approx \mu \rrbracket(\varphi) && \\
&= p_1 \cdot \varphi[x \mapsto t_1] + \ldots + p_n \cdot \varphi[x \mapsto t_n] && \text{(definition)} \\
&\sqsubseteq p_1 \cdot \varphi'[x \mapsto t_1] + \ldots + p_n \cdot \varphi'[x \mapsto t_n] && (\varphi \sqsubseteq \varphi') \\
&= \mathsf{vp}\llbracket \mathtt{var}\ x \colon \tau :\approx \mu \rrbracket(\varphi') && \text{(definition)}
\end{aligned}$$

- Case $S = \mathtt{reward}\ a$.

$$\begin{aligned}
&\mathsf{vp}\llbracket \mathtt{reward}\ a \rrbracket(\varphi) && \\
&= \varphi + a && \text{(definition pf } \mathtt{reward}\text{)} \\
&\sqsubseteq \varphi' + a && (\varphi \sqsubseteq \varphi') \\
&= \mathsf{vp}\llbracket \mathtt{reward}\ a \rrbracket(\varphi') && \text{(definition of } \mathtt{reward}\text{)}
\end{aligned}$$

- Case $S = \mathtt{assert}\ \psi$.

$$\begin{aligned}
&\mathsf{vp}\llbracket \mathtt{assert}\ \psi \rrbracket(\varphi) && \\
&= \psi \sqcap \varphi && \text{(definition of } \mathtt{assert}\text{)} \\
&\sqsubseteq \psi \sqcap \varphi' && (\varphi \sqsubseteq \varphi') \\
&= \mathsf{vp}\llbracket \mathtt{assert}\ \psi \rrbracket(\varphi') && \text{(definition of } \mathtt{assert}\text{)}
\end{aligned}$$

- Case $S = \mathsf{assume}\ \psi$. For all $\sigma \in \mathsf{States}$,

$$\begin{aligned}
&\mathsf{vp}\llbracket \mathsf{assume}\ \psi \rrbracket(\varphi)(\sigma) && \\
&= \llbracket \psi \rightarrow \varphi \rrbracket(\sigma) && \text{(definition of } \mathsf{assume}\text{)} \\
&= \begin{cases} \infty, & \text{if } \llbracket \psi \rrbracket(\sigma) \leq \llbracket \varphi \rrbracket(\sigma) \\ \llbracket \varphi \rrbracket(\sigma), & \text{otherwise} \end{cases} && \text{(definition of } \rightarrow\text{)} \\
&\leq \begin{cases} \infty, & \text{if } \llbracket \psi \rrbracket(\sigma) \leq \llbracket \varphi \rrbracket(\sigma) \\ \llbracket \varphi' \rrbracket(\sigma), & \text{otherwise} \end{cases} && (\llbracket \varphi \rrbracket(\sigma) \leq \llbracket \varphi' \rrbracket(\sigma)) \\
&\leq \begin{cases} \infty, & \text{if } \llbracket \psi \rrbracket(\sigma) \leq \llbracket \varphi \rrbracket(\sigma) \\ \llbracket \varphi' \rrbracket(\sigma), & \text{if } \llbracket \varphi \rrbracket(\sigma) < \llbracket \psi \rrbracket(\sigma) \leq \llbracket \varphi' \rrbracket(\sigma) \\ \llbracket \varphi' \rrbracket(\sigma), & \text{otherwise} \end{cases} && \text{(case distinction)} \\
&\leq \begin{cases} \infty, & \text{if } \llbracket \psi \rrbracket(\sigma) \leq \llbracket \varphi' \rrbracket(\sigma) \\ \llbracket \varphi' \rrbracket(\sigma), & \text{otherwise} \end{cases} && (\llbracket \varphi' \rrbracket(\sigma) \leq \infty) \\
&= \llbracket \psi \rightarrow \varphi' \rrbracket(\sigma) && \text{(definition of } \rightarrow\text{)} \\
&= \mathsf{vp}\llbracket \mathsf{assume}\ \psi \rrbracket(\varphi') && \text{(definition of } \mathsf{assume}\text{)}
\end{aligned}$$

- Case $S = \mathsf{havoc}\ x$.

$$\begin{aligned}
&\mathsf{vp}\llbracket \mathsf{havoc}\ x \rrbracket(\varphi) && \\
&= \inf \{\, \varphi[x \mapsto v] \mid v \in \mathsf{Vals} \,\} && \text{(definition of } \mathsf{havoc}\text{)} \\
&\sqsubseteq \inf \{\, \varphi'[x \mapsto v] \mid v \in \mathsf{Vals} \,\} && (\varphi \sqsubseteq \varphi')
\end{aligned}$$

$$= \mathsf{vp}[\![\mathtt{havoc}\ x]\!](\varphi')$$

- Case $S = \mathtt{validate}$. For all $\sigma \in \mathsf{States}$,

$$\begin{aligned}
&\mathsf{vp}[\![\mathtt{validate}]\!](\varphi)(\sigma) \\
&= [\![\triangle(\varphi)]\!](\sigma) && \text{(definition of } \mathtt{validate}\text{)} \\
&= \begin{cases} \infty, & \text{if } [\![\varphi]\!](\sigma) = \infty \\ 0, & \text{otherwise} \end{cases} && \text{(definition of } \triangle\text{)} \\
&\leq \begin{cases} \infty, & \text{if } [\![\varphi']\!](\sigma) = \infty \\ 0, & \text{otherwise} \end{cases} && ([\![\varphi]\!](\sigma) \leq [\![\varphi']\!](\sigma)) \\
&= [\![\triangle(\varphi')]\!](\sigma) && \text{(definition of } \triangle\text{)} \\
&= \mathsf{vp}[\![\mathtt{validate}]\!](\varphi')(\sigma) && \text{(definition of } \mathtt{validate}\text{)}
\end{aligned}$$

The co cases are dual, but we show the coassume case for illustration:

- Case $S = \mathtt{coassume}\ \psi$. For all $\sigma \in \mathsf{States}$,

$$\begin{aligned}
&\mathsf{vp}[\![\mathtt{coassume}\ \psi]\!](\varphi)(\sigma) \\
&= [\![\psi \leftsquigarrow \varphi]\!](\sigma) && \text{(definition of } \mathtt{coassume}\text{)} \\
&= \begin{cases} 0, & \text{if } [\![\psi]\!](\sigma) \geq [\![\varphi]\!](\sigma) \\ [\![\varphi]\!](\sigma), & \text{otherwise} \end{cases} && \text{(definition of } \leftsquigarrow\text{)} \\
&\leq \begin{cases} 0, & \text{if } [\![\psi]\!](\sigma) \geq [\![\varphi]\!](\sigma) \\ [\![\varphi']\!](\sigma), & \text{otherwise} \end{cases} && ([\![\varphi]\!](\sigma) \leq [\![\varphi']\!](\sigma)) \\
&\leq \begin{cases} 0, & \text{if } [\![\psi]\!](\sigma) \geq [\![\varphi']\!](\sigma) \\ 0, & \text{if } [\![\varphi']\!](\sigma) > [\![\psi]\!](\sigma) \geq [\![\varphi]\!](\sigma) \\ [\![\varphi']\!](\sigma), & \text{otherwise} \end{cases} && \text{(case distinction)} \\
&\leq \begin{cases} 0, & \text{if } [\![\psi]\!](\sigma) \geq [\![\varphi']\!](\sigma) \\ [\![\varphi']\!](\sigma), & \text{otherwise} \end{cases} && ([\![\varphi']\!](\sigma) \geq 0) \\
&= [\![\psi \leftsquigarrow \varphi']\!](\sigma) && \text{(definition of } \leftsquigarrow\text{)} \\
&= \mathsf{vp}[\![\mathtt{coassume}\ \psi]\!](\varphi') && \text{(definition of } \mathtt{coassume}\text{)}
\end{aligned}$$

Now assume that the induction hypothesis holds for arbitrary but fixed $S_1, S_2 \in \mathsf{HeyVL}$.
Induction step:

- Case $S = x_1, \ldots, x_n \ := \ P(e_1, \ldots, e_m)$. According to Section 3.5, (co)procedure calls are encoded as a sequential composition of the atomic assert, havoc, validate, and assume (co)statements and are thus covered by the following case $S = S_1\,;\ S_2$.
- Case $S = S_1\,;\ S_2$.
  Let $\varphi, \psi \in \mathsf{HeyLo}$ such that $\varphi \sqsubseteq \varphi'$. We use the induction hypothesis for S_2:

  $$\mathsf{vp}[\![S_2]\!](\varphi) \sqsubseteq \mathsf{vp}[\![S_2]\!](\varphi')\ .$$

  By the induction hypothesis for S_1:

  $$\mathsf{vp}[\![S_1]\!](\mathsf{vp}[\![S_2]\!](\varphi)) \sqsubseteq \mathsf{vp}[\![S_1]\!](\mathsf{vp}[\![S_2]\!](\varphi'))\ .$$

  Applying definitions, we get:

  $$\mathsf{vp}[\![S_1\,;\ S_2]\!](\varphi) = \mathsf{vp}[\![S_1]\!](\mathsf{vp}[\![S_2]\!](\varphi)) \qquad \text{(definition of } ;\ \text{)}$$

$$
\begin{aligned}
&\sqsubseteq \mathsf{vp}\llbracket S_1 \rrbracket(\mathsf{vp}\llbracket S_2 \rrbracket(\varphi')) && \text{(I.H. on } S_1 \text{ and } S_2\text{)}\\
&= \mathsf{vp}\llbracket S_1\,;\; S_2 \rrbracket(\varphi') && \text{(definition of ;)}
\end{aligned}
$$

- Case $S = \texttt{if}\ (\sqcap)\ \{S_1\}\ \texttt{else}\ \{S_2\}$.

$$
\begin{aligned}
&\mathsf{vp}\llbracket S_1\,;\; S_2 \rrbracket(\varphi)\\
&\qquad = \mathsf{vp}\llbracket S_1 \rrbracket(\varphi) \sqcap \mathsf{vp}\llbracket S_2 \rrbracket(\varphi) && \text{(definition of ;)}\\
&\qquad \sqsubseteq \mathsf{vp}\llbracket S_1 \rrbracket(\varphi') \sqcap \mathsf{vp}\llbracket S_2 \rrbracket(\varphi) && \text{(induction hypothesis)}\\
&\qquad \sqsubseteq \mathsf{vp}\llbracket S_1 \rrbracket(\varphi') \sqcap \mathsf{vp}\llbracket S_2 \rrbracket(\varphi') && \text{(induction hypothesis)}\\
&\qquad = \mathsf{vp}\llbracket S_1\,;\; S_2 \rrbracket(\varphi') && \text{(definition of ;)}
\end{aligned}
$$

- Case $S = \texttt{if}\ (\sqcup)\ \{S_1\}\ \texttt{else}\ \{S_2\}$: Analogous to the $\texttt{if}\ (\sqcap)$ case.

By the principle of structual induction, Theorem 3.2 holds. □

Recall Theorem 3.3 (page 16):

Theorem 3.3 (Conservativity of HeyVL). *Let C be a program in the programming language of* Müller [2019] *and let B be a postcondition. Moreover, let $\overline{C}$ be obtained by replacing every* $\texttt{assert}\ A$ *and every* $\mathsf{assume}\ A$ *occurring in C by* $\texttt{assert}\ ?(A)$ *and* $\mathsf{assume}\ ?(A)$*, respectively (cf. Boolean embeddings,* Section 2.3*). Then*

$$
?(\underbrace{\mathsf{wp}\llbracket C \rrbracket(B)}_{\textit{verification condition obtained from } \text{[Müller 2019]}}) \qquad \equiv \qquad \overbrace{\mathsf{vp}\llbracket \overline{C} \rrbracket(?(B))}^{\mathsf{HeyVL}}\ .
$$

Proof. Let C be a program in the Boolean IVL of [Müller 2019]. Let $B \in \mathbb{P}$ be a predicate. We prove

$$
\mathsf{vp}\llbracket \overline{C} \rrbracket(?(B)) = ?(\mathsf{vc}\llbracket C \rrbracket(B))
$$

by induction over the structure of C.

Base cases:

- Case $C = \mathsf{var}\ x\colon \tau \mathrel{:\approx} e$.

$$
\begin{aligned}
&?(\mathsf{vc}\llbracket \mathsf{var}\ x\colon \tau \mathrel{:\approx} e \rrbracket(B))\\
&= ?(B[x \mapsto e])\\
&= 1 \cdot ?(B)[x \mapsto e]\\
&= \mathsf{vp}\llbracket \mathsf{var}\ x\colon \tau \mathrel{:\approx} \mu \rrbracket(?(B))
\end{aligned}
$$

- Case $C = \mathsf{havoc}\ x$ where $x\colon \tau$.

$$
\begin{aligned}
&?(\mathsf{vc}\llbracket \mathsf{havoc}\ x \rrbracket(B))\\
&= ?(\forall x \in \tau.\ B)\\
&= \boldsymbol{\ell}\, x\colon \tau.\ ?(B)\\
&= \mathsf{vp}\llbracket \mathsf{havoc}\ x \rrbracket(?(B))
\end{aligned}
$$

- Case $C = \texttt{assert}\ A$.

$$
\begin{aligned}
&?(\mathsf{vc}\llbracket \texttt{assert}\ A \rrbracket(B))\\
&= ?(A \land B)\\
&= ?(A) \sqcap ?(B)\\
&= \mathsf{vp}\llbracket \texttt{assert}\ ?(A) \rrbracket(?(B))
\end{aligned}
$$

- Case $C = \texttt{assume}\ A$.

$$
\begin{aligned}
&?(\mathsf{vc}\llbracket \texttt{assume}\ A \rrbracket(B)) \\
&= ?(A \Rightarrow B) \\
&= ?(A) \rightarrow ?(B) \\
&= \mathsf{vp}\llbracket \texttt{assume}\ ?(A) \rrbracket(?(B))
\end{aligned}
$$

Now assume that the induction hypothesis holds for arbitrary, but fixed C_1, C_2 in the Boolean IVL. Let $\overline{C_1}, \overline{C_2} \in \mathsf{HeyVL}$ be obtained from C_1 and C_2 by replacement of $\texttt{assert}\ A$ and $\texttt{assume}\ A$ by $\texttt{assert}\ ?(A)$ and $\texttt{assume}\ ?(A)$, respectively.

Induction step:

- Case $C = C_1\,;\ C_2$.

$$
\begin{aligned}
&?(\mathsf{vc}\llbracket C_1\,;\ C_2 \rrbracket(B)) \\
&= ?(\mathsf{vc}\llbracket C_1 \rrbracket(\mathsf{vc}\llbracket C_2 \rrbracket(B))) \\
&= \mathsf{vp}\llbracket \overline{C_1} \rrbracket(?(\mathsf{vc}\llbracket C_2 \rrbracket(B))) && \text{(induction hypothesis)} \\
&= \mathsf{vp}\llbracket \overline{C_1} \rrbracket(\mathsf{vp}\llbracket \overline{C_2} \rrbracket(?(B))) && \text{(induction hypothesis)} \\
&= \mathsf{vp}\llbracket \overline{C_1\,;\ C_2} \rrbracket(?(B))
\end{aligned}
$$

- Case $C = \texttt{if}\ (\sqcap)\ \{C_1\}\ \texttt{else}\ \{C_2\}$.

$$
\begin{aligned}
&?(\mathsf{vc}\llbracket \texttt{if}\ (\sqcap)\ \{C_1\}\ \texttt{else}\ \{C_2\} \rrbracket(B)) \\
&= ?(\mathsf{vc}\llbracket C_1 \rrbracket(B) \land \mathsf{vc}\llbracket C_2 \rrbracket(B)) \\
&= ?(\mathsf{vc}\llbracket C_1 \rrbracket(B)) \sqcap ?(\mathsf{vc}\llbracket C_2 \rrbracket(B)) \\
&= \mathsf{vp}\llbracket \overline{C_1} \rrbracket(?(B)) \sqcap \mathsf{vp}\llbracket \overline{C_2} \rrbracket(?(B)) && \text{(induction hypothesis)} \\
&= \mathsf{vp}\llbracket \texttt{if}\ (\sqcap)\ \{\overline{C_1}\}\ \texttt{else}\ \{\overline{C_2}\} \rrbracket(?(B)) \\
&= \mathsf{vp}\llbracket \overline{\texttt{if}\ (\sqcap)\ \{C_1\}\ \texttt{else}\ \{C_2\}} \rrbracket(?(B))
\end{aligned}
$$

By structural induction on C, Theorem 3.3 holds. □

B.2 Soundness and Semantics of Procedure Calls

We want to encode a procedure call $z_1, \ldots, z_n \ \coloneqq\ P(t_1, \ldots, t_n)$ for a procedure P:

```
proc $P$ ($x_1 : \tau_1, \ldots x_n : \tau_\mathrm{n}$) -> ($y_1 : \tau_1, \ldots y_m : \tau_\mathrm{m}$)
    pre $\rho$
    post $\psi$
{
    $S$
}
```

Recall the definition of $S_{encoding}$ for the above call and procedure from Section 3.5:

$$S_{encoding}: \qquad \texttt{assert}\ \rho\,;\ \texttt{havoc}\ z_1\,;\ \ \ldots\,;\ \texttt{havoc}\ z_m\,;\ \texttt{validate};\ \texttt{assume}\ \psi.$$

Theorem 3.4. *Let S be the body of the procedure P in Figure 15. Then, for every* HeyLo *formula φ,*

$$\mathsf{vp}\llbracket S_{encoding} \rrbracket(\varphi) \ \sqsubseteq\ \mathsf{vp}\llbracket S \rrbracket(\varphi) \quad \text{and} \quad \mathsf{vp}\llbracket \mathit{init};\ S_{encoding}\,;\ \mathit{return} \rrbracket(\varphi) \ \sqsubseteq\ \mathsf{vp}\llbracket \mathit{init};\ S;\ \mathit{return} \rrbracket(\varphi).$$

Proof. First, we show that for all $\varphi \in \mathsf{HeyLo}$ and $\sigma \in \mathsf{States}$, we have

$$\mathsf{vp}[\![S_{encoding}]\!](\varphi)(\sigma) = \begin{cases} [\![\rho]\!](\sigma), & \text{if } \psi \sqsubseteq \varphi \\ 0, & \text{otherwise .} \end{cases}$$

From the definition of vp (Figure 14), it follows that

$$\begin{aligned}
&\mathsf{vp}[\![S_{encoding}]\!](\varphi)(\sigma) \\
&= [\![\rho \sqcap \rotatebox[origin=c]{180}{ℓ}\, z_1.\ \ldots \rotatebox[origin=c]{180}{ℓ}\, z_n.\ \triangle(\psi \rightarrow \varphi)]\!](\sigma) \\
&= [\![\rho]\!](\sigma) \sqcap [\![\rotatebox[origin=c]{180}{ℓ}\, z_1.\ \ldots \rotatebox[origin=c]{180}{ℓ}\, z_n.\ \triangle(\psi \rightarrow \varphi)]\!](\sigma) && \text{(definition of } \sqcap\text{)} \\
&= [\![\rho]\!](\sigma) \sqcap \inf\{[\![\triangle(\psi \rightarrow \varphi)]\!](\sigma') \mid \sigma' \in \mathsf{States}\} && \text{(definition of } \rotatebox[origin=c]{180}{ℓ}\text{)}
\end{aligned}$$

Let $\mathrm{ite}(a,\ b,\ c)$ denote a conditional choice that evaluates to b if a is true and to c otherwise.

$$\begin{aligned}
&= [\![\rho]\!](\sigma) \sqcap \inf\{\ \mathrm{ite}([\![\psi \rightarrow \varphi]\!](\sigma') = \infty,\ \infty,\ 0)\ \mid \sigma' \in \mathsf{States}\} && \text{(definition of } \triangle\text{)} \\
&= [\![\rho]\!](\sigma) \sqcap \inf\{\ \mathrm{ite}([\![\psi]\!](\sigma') \leq [\![\varphi]\!](\sigma'),\ \infty,\ 0)\ \mid \sigma' \in \mathsf{States}\} && \text{(cf. Theorem 2.1)}
\end{aligned}$$

The infimum evaluates to ∞ iff $[\![\psi]\!](\sigma') \sqsubseteq [\![\varphi]\!](\sigma')$ for all $\sigma' \in \mathsf{States}$. Thus,

$$\begin{aligned}
&= [\![\rho]\!](\sigma) \sqcap \mathrm{ite}(\psi \sqsubseteq \varphi,\ \infty,\ 0) \\
&= \begin{cases} [\![\rho]\!](\sigma), & \text{if } \psi \sqsubseteq \varphi \\ 0, & \text{otherwise} \end{cases} && ([\![\rho]\!](\sigma) \sqcap \infty = [\![\rho]\!](\sigma))
\end{aligned}$$

Now we show for all $\varphi \in \mathsf{HeyLo}$ that $\mathsf{vp}[\![S_{encoding}]\!](\varphi) \sqsubseteq \mathsf{vp}[\![S]\!](\varphi)$. Let $\varphi \in \mathsf{HeyLo}$.
In case that $\psi \sqsubseteq \varphi$ holds, we have by monotonicity of vp (Theorem 3.2):

$$\mathsf{vp}[\![S]\!](\psi) \sqsubseteq \mathsf{vp}[\![S]\!](\varphi)\ .$$

From the assumption $\rho \sqsubseteq \mathsf{vp}[\![S]\!](\psi)$ it follows that

$$\rho \sqsubseteq \mathsf{vp}[\![S]\!](\varphi)\ .$$

Thus,

$$\mathsf{vp}[\![S_{encoding}]\!] = \rho \sqsubseteq \mathsf{vp}[\![S]\!](\varphi)\ .$$

If $\psi \sqsubseteq \varphi$ does not hold, we have

$$\mathsf{vp}[\![S_{encoding}]\!](\varphi) = 0 \sqsubseteq \mathsf{vp}[\![S]\!](\varphi)\ .$$

In conclusion,

$$\mathsf{vp}[\![S_{encoding}]\!](\varphi) \sqsubseteq \mathsf{vp}[\![S]\!](\varphi)\ .$$

The other claim,

$$\mathsf{vp}[\![init;\ S_{encoding};\ return]\!](\varphi)\ \sqsubseteq\ \mathsf{vp}[\![init;\ S;\ return]\!](\varphi)\ ,$$

follows by the above and the definition of vp. □

C PROOF RULE ENCODINGS INTO HEYVL

This appendix section details the HeyVL encodings mentioned in Section 4. These encodings are all implemented in our frontend that translates annotated pGCL programs to HeyVL. We follow Table 1 and present encodings for the various verification problems. For each encoding, we first state the formal *proof rule* on expectations. Then, we specify the *encoding inputs* that our frontend requires, as well as a schematic description of the *encoding output*. All encodings of loops require HeyVL encodings of their loop bodies. For loop-free programs, the encoding from Section 4.1 can be used. Furthermore, proof rule encodings from this section may be used to encode nested loops.

C.1 Loop Rule: k-Induction for wlp

The *k-induction* encoding for wlp encodes a while loop and under-approximates its wlp semantics. The proof rule is a generalization of Park induction (cf. Section 4.2). For the k-induction encoding, the user needs to provide a potential subinvariant $I \in \mathsf{HeyLo}$ and a number $k \in \mathbb{N}$ of how many times to unfold the loop. For the loop body, we assume another under-approximating encoding is given. If it contains loops, k-induction can be encoded recursively, but other encodings can be used as well.

Proof Rule: *Latticed k-induction* [Batz et al. 2021a] for wlp.[21]
Let $C = \texttt{while}\ (\ b\ )\ \{\ C_0\ \}$ be a pGCL loop and let $X \in \mathbb{E}_{\leq 1}$. The *$k$-induction operator* for wlp is given by

$$^{\mathsf{wlp}}\Psi_I \colon \mathbb{E}_{\leq 1} \longrightarrow \mathbb{E}_{\leq 1}, \quad Y \mapsto {}^{\mathsf{wlp}}\Phi_X(Y) \sqcup I\,,$$

where the *loop-characteristic functional* $^{\mathsf{wlp}}\Phi_X$ with respect to post X is defined as

$$^{\mathsf{wlp}}\Phi_X(Y) = [b] \cdot \mathsf{wlp}[\![C_0]\!](Y) + [\neg b] \cdot X\,.$$

Then, for $k \in \mathbb{N}$,

$$I \preceq {}^{\mathsf{wlp}}\Phi_X({}^{\mathsf{wlp}}\Psi_I^k(I)) \quad \text{implies} \quad I \preceq \mathsf{wlp}[\![C]\!](X)\,.$$

Encoding Input:

- pGCL loop $C = \texttt{while}\ (\ b\ )\ \{\ C_0\ \}$.
- HeyVL statement $enc_{\mathsf{wlp}}\lfloor C_0 \rfloor$ that satisfies $\mathsf{vp}[\![enc_{\mathsf{wlp}}\lfloor C_0 \rfloor]\!] \sqsubseteq \mathsf{wlp}[\![C_0]\!]$.
- $k \in \mathbb{N}$.
- Potential k-inductive wlp-subinvariant $I \in \mathsf{HeyLo}$ with $I \sqsubseteq 1$.

Encoding Output:

- HeyVL statement S that satisfies $\mathsf{vp}[\![S]\!] \preceq \mathsf{wlp}[\![C]\!]$.
  - If C_0 is encoded exactly, i.e. $\mathsf{vp}[\![enc_{\mathsf{wlp}}\lfloor C_0 \rfloor]\!] = \mathsf{wlp}[\![C_0]\!]$ holds, then

$$\mathsf{vp}[\![S]\!] = \begin{cases} I, & \text{if } I \preceq {}^{\mathsf{wlp}}\Phi_X({}^{\mathsf{wlp}}\Psi_I^k(I)) \\ 0, & \text{otherwise}\,. \end{cases}$$

The k-induction encoding is similar to Park induction, but the sequence $\texttt{assert}\ I;\ \texttt{assume}\ ?(\texttt{false})$ in the Park induction encoding is replaced by recursive encodings of the $^{\mathsf{wlp}}\Psi_I$ operator.
Formally, the HeyVL statement S is given by:

$$S = \quad spec(I,\ I);\ iter_{\mathsf{wlp}}(extend_{\mathsf{wlp}}^{(k-1)}(const(I)))$$

where

$$\begin{aligned} spec(\psi,\ \varphi) &= \quad \texttt{assert}\ \psi;\ \texttt{havoc}\ \mathit{variables};\ \texttt{validate};\ \texttt{assume}\ \varphi \\ iter_{\mathsf{wlp}}(S) &= \quad \texttt{if}\ (b)\ \{\ enc_{\mathsf{wlp}}\lfloor C_0 \rfloor;\ S\ \}\ \texttt{else}\ \{\ \texttt{skip}\ \} \\ const(I) &= \quad \texttt{assert}\ I;\ \texttt{assume}\ ?(\texttt{false}) \\ extend_{\mathsf{wlp}}(S) &= \quad \texttt{coassert}\ I;\ iter_{\mathsf{wlp}}(S) \end{aligned}$$

[21] In [Batz et al. 2021a], latticed k-induction is only defined for upper bounds on least fixed points. These occur e.g. in wp and ert semantics. However, the dual principle can be applied to the greatest fixed point that underlies the wlp semantics.

Sketches for $k = 2$ and $k = 3$:

```
assert $I$
havoc $variables$
validate
assume $I$
if ($b$) {
    $enc_{\mathsf{wlp}}\lfloor C_0 \rfloor$
    coassert $I$
    if ($b$) {
        $enc_{\mathsf{wlp}}\lfloor C_0 \rfloor$
        assert $I$
        assume ?(false)
    }
}
```

```
assert $I$
havoc $variables$
validate
assume $I$
if ($b$) {
    $enc_{\mathsf{wlp}}\lfloor C_0 \rfloor$
    coassert $I$
    if ($b$) {
        $enc_{\mathsf{wlp}}\lfloor C_0 \rfloor$
        coassert $I$
        if ($b$) {
            $enc_{\mathsf{wlp}}\lfloor C_0 \rfloor$
            assert $I$
            assume ?(false)
        }
    }
}
```

C.2 Loop Rule: k-Induction for wp

The k-induction encoding for wp is dual to the k-induction encoding of wlp (cf. Appendix C.1). It encodes a while loop and over-approximates its wp semantics. The user needs to provide a potential superinvariant $I \in$ HeyLo and a number $k \in \mathbb{N}$ of how many times to unfold the loop. For the loop body, we assume another over-approximating encoding is given. If it contains loops, k-induction can be encoded recursively, but other encodings can be used as well.

Proof Rule: *Latticed k-induction* [Batz et al. 2021a] for wp.
Let $C = \texttt{while}\ (\ b\ )\ \{\ C_0\ \}$ be a pGCL loop and let $X \in \mathbb{E}$. The *k-induction operator* for wp is given by

$$^{\mathsf{co}}\Psi_I \colon \mathbb{E} \longrightarrow \mathbb{E}, \quad Y \mapsto {}^{\mathsf{wp}}\Phi_X(Y) \sqcap I\,,$$

where the *loop-characteristic functional* $^{\mathsf{wp}}\Phi_X$ with respect to post X is defined as

$$^{\mathsf{wp}}\Phi_X(Y) = [b] \cdot \mathsf{wp}[\![C_0]\!](Y) + [\neg b] \cdot X\,.$$

Then, for $k \in \mathbb{N}$,

$$^{\mathsf{wp}}\Phi_X({}^{\mathsf{co}}\Psi_I^k(I)) \preceq I \quad \text{implies} \quad \mathsf{wp}[\![C]\!](X) \preceq I\,.$$

Encoding Input:

- pGCL loop $C = \texttt{while}\ (\ b\ )\ \{\ C_0\ \}$.
- HeyVL encoding $enc^{\mathsf{co}}_{\mathsf{wp}}\lfloor C_0 \rfloor$ that satisfies $\mathsf{wp}[\![C_0]\!] \preceq \mathsf{vp}[\![enc^{\mathsf{co}}_{\mathsf{wp}}\lfloor C_0 \rfloor]\!]$.
- $k \in \mathbb{N}$.
- Potential k-inductive wp-superinvariant $I \in$ HeyLo.

Encoding Output:

- HeyVL encoding S that satisfies $\mathsf{wp}[\![C]\!] \preceq \mathsf{vp}[\![S]\!]$.

- If C_0 is encoded exactly, i.e. $\mathsf{vp}[\![enc^{\mathsf{co}}_{\mathsf{wp}}\lfloor C_0 \rfloor]\!] = \mathsf{wp}[\![C_0]\!]$ holds, then

$$\mathsf{vp}[\![S]\!] = \begin{cases} I, & \text{if } {}^{\mathsf{wp}}\Phi_X({}^{\mathsf{co}}\Psi^k_I(I)) \preceq I \\ \infty, & \text{otherwise .} \end{cases}$$

Formally, the encoding S is given by:

$$S = \quad spec_{\mathsf{co}}(I,\ I);\ iter_{\mathsf{wp}}(extend^{(k-1)}_{\mathsf{wp}}(const(I)))$$

where

$$\begin{aligned} spec_{\mathsf{co}}(\psi,\ \varphi) = &\quad \texttt{coassert}\ \psi;\ \texttt{cohavoc}\ variables;\ \texttt{covalidate};\ \texttt{coassume}\ \varphi \\ iter_{\mathsf{wp}}(S) = &\quad \texttt{if}\ (b)\ \{\ enc^{\mathsf{co}}_{\mathsf{wp}}\lfloor C_0 \rfloor;\ S\ \}\ \texttt{else}\ \{\ \texttt{skip}\ \} \\ const(I) = &\quad \texttt{coassert}\ I;\ \texttt{coassume}\ ?(\texttt{true}) \\ extend_{\mathsf{wp}}(S) = &\quad \texttt{assert}\ I;\ iter_{\mathsf{wp}}(S) \end{aligned}$$

Sketches for $k = 2$ and $k = 3$:

```
coassert $I$
cohavoc $variables$
covalidate
coassume $I$
if ($b$) {
  $enc^{co}_{wp}\lfloor C_0 \rfloor$
  assert $I$
  if ($b$) {
    $enc^{co}_{wp}\lfloor C_0 \rfloor$
    coassert $I$
    coassume ?(true)
  }
}
```

```
coassert $I$
cohavoc $variables$
covalidate
coassume $I$
if ($b$) {
  $enc^{co}_{wp}\lfloor C_0 \rfloor$
  coassert $I$
  if ($b$) {
    $enc^{co}_{wp}\lfloor C_0 \rfloor$
    assert $I$
    if ($b$) {
      $enc^{co}_{wp}\lfloor C_0 \rfloor$
      coassert $I$
      coassume ?(true)
    }
  }
}
```

C.3 Loop Rule: ω-invariants for wlp

Proof Rule: *ω-invariants for* wlp (adapted from [?]) [22]. Let $C = \texttt{while}\ (\ b\ )\ \{\ C_0\ \}$ be a pGCL loop and let $X \in \mathbb{E}_{\leq 1}$. Let $(I_n)_{n \in \mathbb{N}} \subset \mathbb{E}_{\leq 1}$ with ${}^{\mathsf{wlp}}\Phi_X(1) \preceq I_0$. If I is a wlp-*ω-superinvariant*, then $\inf_{n \in \mathbb{N}} I_n$ upper-bounds $\mathsf{wlp}[\![C]\!](X)$, i.e.

$$(\forall n \in \mathbb{N}.\ {}^{\mathsf{wlp}}\Phi_X(I_n) \preceq I_{n+1}) \quad \text{implies} \quad \mathsf{wlp}[\![C]\!](X) \preceq \inf_{n \in \mathbb{N}} I_n\ ,$$

[22] Different versions of this proof rule exist. An overview is found in [Kaminski 2019, page 108].

where the *loop-characteristic functional* ${}^{\mathsf{wlp}}\Phi_X$ with respect to post X is defined as

$$ {}^{\mathsf{wlp}}\Phi_X(Y) = [b] \cdot \mathsf{wlp}[\![C_0]\!](Y) + [\neg b] \cdot X \,. $$

Encoding Input:

- pGCL loop $C = \mathtt{while}\,(\,b\,)\,\{\,C_0\,\}$.
- HeyVL encoding $enc^{\mathsf{co}}_{\mathsf{wlp}}\lfloor C_0 \rfloor$ that satisfies $\mathsf{wp}[\![C_0]\!] \preceq \mathsf{vp}[\![enc^{\mathsf{co}}_{\mathsf{wlp}}\lfloor C_0 \rfloor]\!]$.
- Potential wlp-ω-superinvariant $I_n \in$ HeyLo that represents $(I_n)_{n \in \mathbb{N}} \subset \mathbb{E}_{\leq 1}$ by a free variable n.
- Post $\varphi \in$ HeyLo.

Encoding Output:

- HeyVL encoding S that verifies only if $\mathsf{wlp}[\![C]\!](\varphi) \preceq I$.

We generate two procedures to check the proof rule conditions.

The first procedure checks that ${}^{\mathsf{wlp}}\Phi_{[\![\varphi]\!]}(1) \preceq I_0$ holds:

```
coproc condition_1 ($x_1^0 : \tau_1, \ldots, x_m^0 : \tau_m$) -> ($x_1 : \tau_1, \ldots, x_m : \tau_m$)
  pre $I_n[n \mapsto 0][x_1 \mapsto x_1^0] \ldots [x_n \mapsto x_n^0]$
  post $\varphi$
{
  var $x_1 : \tau_1$ :≈ $x_1^0$;  ...; var $x_m : \tau_1$ :≈ $x_m^0$
  if ($b$) {
    $enc^{\mathsf{co}}_{\mathsf{wp}}\lfloor C_0 \rfloor$
    coassert 1
    coassume ?(true)
  }
}
```

The second procedure checks that ${}^{\mathsf{wlp}}\Phi_{[\![\varphi]\!]}(I_n) \preceq I_{n+1}$ holds for all $n \in \mathbb{N}$:

```
coproc condition_2 ($n : \mathbb{N}, x_1^0 : \tau_1, \ldots, x_m^0 : \tau_m$) -> ($x_1 : \tau_1, \ldots, x_m : \tau_m$)
  pre $I_n[n \mapsto n + 1][x_1 \mapsto x_1^0] \ldots [x_n \mapsto x_n^0]$
  post $\varphi$
{
  var $x_1 : \tau_1$ :≈ $x_1^0$;  ...; var $x_m : \tau_1$ :≈ $x_m^0$
  if ($b$) {
    $enc^{\mathsf{co}}_{\mathsf{wp}}\lfloor C_0 \rfloor$
    coassert $I_n$
    coassume ?(true)
  }
}
```

C.4 Loop Rule: ω-invariants for wp

Proof Rule: *ω-invariants for* wp [Kaminski 2019]. Let $C = \texttt{while}\ (\ b\ )\ \{\ C_0\ \}$ be a pGCL loop and let $X \in \mathbb{E}$. Let $(I_n)_{n\in\mathbb{N}} \subset \mathbb{E}$ with $I_0 \preceq {}^{\mathsf{wp}}\Phi_X(0)$. If I is a wp-*ω-subinvariant*, then $\sup_{n\in\mathbb{N}} I_n$ lower-bounds $\mathsf{wp}[\![C]\!](X)$, i.e.

$$(\forall n \in \mathbb{N}.\ I_{n+1} \preceq {}^{\mathsf{wp}}\Phi_X(I_n)) \quad \text{implies} \quad \sup_{n\in\mathbb{N}} I_n \preceq \mathsf{wp}[\![C]\!](X)\ ,$$

where the *loop-characteristic functional* ${}^{\mathsf{wp}}\Phi_X$ with respect to post X is defined as

$${}^{\mathsf{wp}}\Phi_X(Y) = [b] \cdot \mathsf{wp}[\![C_0]\!](Y) + [\neg b] \cdot X\ .$$

Encoding Input:

- pGCL loop $C = \texttt{while}\ (\ b\ )\ \{\ C_0\ \}$.
- HeyVL encoding $enc_{\mathsf{wp}}\lfloor C_0 \rfloor$ that satisfies $\mathsf{vp}[\![enc_{\mathsf{wp}}\lfloor C_0\rfloor]\!] \preceq \mathsf{wp}[\![C_0]\!]$.
- Potential wp-ω-subinvariant $I_n \in$ HeyLo that represents $(I_n)_{n\in\mathbb{N}} \subset \mathbb{E}$ by a free variable n.
- Post $\varphi \in$ HeyLo.

Encoding Output:

- HeyVL encoding S that verifies only if $I \preceq \mathsf{wp}[\![C]\!](\varphi)$.

We generate two procedures to check the proof rule conditions.

The first procedure checks that $I_0 \preceq {}^{\mathsf{wp}}\Phi_{[\![\varphi]\!]}(0)$ holds:

```
proc $condition\_1$ ($x_1^0 : \tau_1, \ldots, x_m^0 : \tau_m$) -> ($x_1 : \tau_1, \ldots, x_m : \tau_m$)
  pre $I_n[n \mapsto 0][x_1 \mapsto x_1^0] \ldots [x_n \mapsto x_n^0]$
  post $\varphi$
{
  var $x_1 : \tau_1$ :≈ $x_1^0$;  ...; var $x_m : \tau_1$ :≈ $x_m^0$
  if ($b$) {
    $enc_{\mathsf{wp}}\lfloor C_0 \rfloor$
    assert 0
    assume ?(false)
  }
}
```

The second procedure checks that $I_{n+1} \preceq {}^{\mathsf{wp}}\Phi_{[\![\varphi]\!]}(I_n)$ holds for all $n \in \mathbb{N}$:

```
proc $condition\_2$ $(n\colon \mathbb{N}, x_1^0\colon \tau_1, \ldots, x_m^0\colon \tau_m)$ -> $(x_1\colon \tau_1, \ldots, x_m\colon \tau_m)$
  pre $I_n[n \mapsto n+1][x_1 \mapsto x_1^0] \ldots [x_n \mapsto x_n^0]$
  post $\varphi$
{
  var $x_1\colon \tau_1$ :≈ $x_1^0$; ...; var $x_m\colon \tau_1$ :≈ $x_m^0$
  if $(b)$ {
    $enc_{\mathsf{wp}}\lfloor C_0 \rfloor$
    assert $I_n$
    assume ?(false)
  }
}
```

C.5 Encoding of the Optional Stopping Theorem for wp

Proof Rule: *Optional Stopping Theorem for* wp *Reasoning* [Hark et al. 2019]. Let $C = \texttt{while}\ (\ b\ )\ \{\ C_0\ \}$ be a pGCL loop and let $X \in \mathbb{E}$. If all of the following conditions hold:

- I is a wp-subinvariant: $I \preceq {}^{\mathsf{wp}}\Phi_X(I)$,
- C is positively almost-surely terminating (PAST),
- I harmonizes with f: $\neg b \Rightarrow (I = f)$,
- ${}^{\mathsf{wp}}\Phi_X(I)$ is finite: ${}^{\mathsf{wp}}\Phi_X(I) < \infty$,
- I is *conditionally difference bounded* for some $c \in \mathbb{R}_{\geq 0}$:

$$\mathsf{wp}[\![C_0]\!](|I - I(\sigma)|)(\sigma) \leq c \text{ for all } \sigma \in \mathsf{States}\ .$$

Then,

$$I \preceq \mathsf{wp}[\![C]\!](X)\ .$$

Encoding Input:

- pGCL loop $C = \texttt{while}\ (\ b\ )\ \{\ C_0\ \}$.
- HeyVL encoding $enc_{\mathsf{wp}}\lfloor C_0 \rfloor$ that satisfies $\mathsf{vp}[\![enc_{\mathsf{wp}}\lfloor C_0 \rfloor]\!] \preceq \mathsf{wp}[\![C_0]\!]$.
- HeyVL encoding $enc^{\mathsf{co}}_{\mathsf{wlp}}\lfloor C_0 \rfloor$ that satisfies $\mathsf{wp}[\![C_0]\!] \preceq \mathsf{vp}[\![enc^{\mathsf{co}}_{\mathsf{wlp}}\lfloor C_0 \rfloor]\!]$.
- Potential wp-subinvariant $I \in \mathsf{HeyLo}$.
- Constant $c \in \mathbb{R}_{\geq 0}$.
- Post $\varphi \in \mathsf{HeyLo}$.

Side Conditions:

- C is positively almost-surely terminating (PAST).[23]

Encoding Output:

- HeyVL encoding S that verifies only if $I \preceq \mathsf{wp}[\![C]\!](\varphi)$.

Let $x_1\colon \tau_1, \ldots, x_n\colon \tau_n$ be the variables that are free in C with their types.
Let $I_0 = I[x_1 \mapsto x_1^0] \ldots [x_n \mapsto x_n^0]$.

Multiple procedures are generated to check the various conditions.

[23] For our "ost" example, we show $\mathsf{ert}[\![C]\!](0) < \infty$ using Park induction (cf. Appendix C.6) to show that C is PAST.

The first procedure checks that I is a wp-subinvariant with respect to post φ:

```
proc $subinvariant$ $(x_1^0 \colon \tau_1, \ldots, x_n^0 \colon \tau_n)$ -> $(x_1 \colon \tau_1, \ldots, x_n \colon \tau_n)$
   pre $I_0$
   post $\varphi$
{
   var $x_1 \colon \tau_1$ :≈ $x_1^0$; ...; var $x_n \colon \tau_1$ :≈ $x_n^0$
   if $(b)$ {
      $enc_{\mathrm{wp}} \lfloor C_0 \rfloor$
      assert $I$
      assume $?(\mathtt{false})$
   }
}
```

Next, we check that I harmonizes with φ, i.e. that $\neg b \longrightarrow (I = \varphi)$ holds. Formally, we do this using a procedure and a coprocedure.[24]

```
proc $harmonizes\_lower$ $(x_1 \colon \tau_1, \ldots, x_n \colon \tau_n)$ -> ()
   pre $?(\neg b) \longrightarrow I$
   post $\varphi$ {}

coproc $harmonizes\_upper$ $(x_1 \colon \tau_1, \ldots, x_n \colon \tau_n)$ -> ()
   pre $\mathrm{co}?(\neg b) \leftsquigarrow I$
   post $\varphi$ {}
```

The next procedure checks that ${}^{\mathrm{wp}}\Phi_{[\![\varphi]\!]}(I)$ is finite:

```
coproc $phi\_finite$ $(x_1^0 \colon \tau_1, \ldots, x_n^0 \colon \tau_n)$ -> $(x_1 \colon \tau_1, \ldots, x_n \colon \tau_n)$
   pre $0$
   post $\varphi$
{
   validate
   assume $\infty$
   var $x_1 \colon \tau_1$ :≈ $x_1^0$; ...; var $x_n \colon \tau_1$ :≈ $x_n^0$
   if $(b)$ {
      $enc_{\mathrm{wp}} \lfloor C_0 \rfloor$
      assert $I$
      assume $?(\mathtt{false})$
   }
}
```

[24] Our implementation *Caesar* supports HeyLo formulae of the more direct form $?(\neg b \longrightarrow (I = \varphi))$ as well.

The last procedure checks the conditional difference boundedness property:

```
coproc $cdb$ ($x_1^0 \colon \tau_1, \ldots, x_n^0 \colon \tau_n$) -> ($x_1 \colon \tau_1, \ldots, x_n \colon \tau_n$)
  pre $c$
  post $\mathrm{ite}(I_0 \leq I,\ I - I_0,\ I_0 - I)$
{
  var $x_1 \colon \tau_1$ :≈ $x_1^0$; ...; var $x_n \colon \tau_1$ :≈ $x_n^0$
  $enc_{\mathrm{wp}}^{\mathrm{co}}\lfloor C_0 \rfloor$
}
```

C.6 Proof Rules for ert

The ert calculus [Kaminski et al. 2016, 2018] is similar to the wp calculus. For loop-free pGCL programs, we obtain encodings similar to Figure 17. The only difference consists of the additional `reward 1` statements to track the run-times of each statement.

C	$enc_{\mathrm{ert}}\lfloor C \rfloor$
`skip`	`reward 1`
`diverge`	`assert 0`
x `:=` t	x `:≈` t`; reward 1;`
$C_1; C_2$	$enc_{\mathrm{wp}}\lfloor C_1 \rfloor; enc_{\mathrm{wp}}\lfloor C_2 \rfloor$
`if` (b) $\{ C_1 \}$	`if` $(\sqcap)$ $\{$ `assume` $?(b); enc_{\mathrm{wp}}\lfloor C_1 \rfloor$ $\}$
`else` $\{ C_2 \}$	`else` $\{$ `assume` $?(\neg b); enc_{\mathrm{wp}}\lfloor C_2 \rfloor \}$`; reward 1;`
$\{ C_1 \}$ $[p]$ $\{ C_2 \}$	`var` $tmp \colon \mathbb{B}$ `:≈ flip(`p`); reward 1;`
	$enc_{\mathrm{wp}}\lfloor$`if` (tmp) $\{C_1\}$ `else` $\{C_2\}\rfloor$
$\{ C_1 \}$ $[]$ $\{ C_2 \}$	`if` $(\sqcap)$ $\{C_1\}$ `else` $\{C_2\}$

Latticed k-induction (cf. Appendix C.2) and ω-invariants (cf. Appendix C.4), can be encoded similarly to wp and are implemented in our frontend. The k-induction proof rule for ert is a straightforward consequence of the latticed k-induction principle [Batz et al. 2021a]. ω-invariants for ert have been described in [Kaminski et al. 2016].

C.7 Encoding of "A New Proof Rule for Almost-Sure Termination"

Proof Rule: *"A New Proof Rule for Almost-Sure Termination"* [McIver et al. 2018]. Let C = `while` (b) $\{ C_0 \}$ be a pGCL loop. Let $I \colon \mathsf{States} \to \mathbb{B}$, let $V \colon \mathsf{States} \to \mathbb{R}_{\geq 0}$, let $p \colon \mathbb{R}_{\geq 0} \to (0, 1]$ and $d \colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{>0}$ where d and p are antitone on positive arguments. If the following four conditions hold, then $[I] \preceq \mathrm{wp}[\![C]\!](1)$ holds:

- $[I]$ is a wp-superinvariant: ${}^{\mathrm{wp}}\Phi_{[I]}([I]) \preceq [I]$,
- $b \wedge I \implies V > 0$,
- ${}^{\mathrm{wp}}\Phi_V(V) \preceq V$,
- $[I] \cdot [G] \cdot (p \circ V) \preceq \lambda\sigma.\ \mathrm{wp}[\![C_0]\!]([V < V(\sigma) - d(V(\sigma))])(\sigma)$.

Encoding Input:

- pGCL loop C = `while` (b) $\{ C_0 \}$.
- HeyVL encoding $enc_{\mathrm{wp}}\lfloor C_0 \rfloor$ that satisfies $\mathrm{vp}[\![enc_{\mathrm{wp}}\lfloor C_0 \rfloor]\!] \preceq \mathrm{wp}[\![C_0]\!]$.
- HeyVL encoding $enc_{\mathrm{wp}}^{\mathrm{co}}\lfloor C_0 \rfloor$ that satisfies $\mathrm{wp}[\![C_0]\!] \preceq \mathrm{vp}[\![enc_{\mathrm{wp}}^{\mathrm{co}}\lfloor C_0 \rfloor]\!]$.
- Expressions $I \colon \mathbb{B}$, $V \colon \mathbb{R}_{\geq 0}$, $p \colon \mathbb{R}_{\geq 0}$, $d \colon \mathbb{R}_{\geq 0}$ with a free variable x each for the parameter of the function that they represent.

Encoding Output:

- HeyVL encoding S that verifies only if $[I] \preceq \mathrm{wp}[\![C]\!](1)$.

Let $x_1 \colon \tau_1, \ldots, x_n \colon \tau_n$ be the variables that are free in C with their types.
Let $I_0 = I[x_1 \mapsto x_1^0] \ldots [x_n \mapsto x_n^0]$ and $V_0 = V[x_1 \mapsto x_1^0] \ldots [x_n \mapsto x_n^0]$ and $G_0 = G[x_1 \mapsto x_1^0] \ldots [x_n \mapsto x_n^0]$.
Multiple procedures are generated to check the various conditions.
The first two procedures check that p and V are antitone:

```
proc $p\_antitone$ ($a \colon \mathbb{R}_{\geq 0}, b \colon \mathbb{R}_{\geq 0}$) -> ()
  pre $?(a \leq b)$
  post $?(p[x \mapsto a] \geq p[x \mapsto b])$ {}

proc $v\_antitone$ ($a \colon \mathbb{R}_{\geq 0}, b \colon \mathbb{R}_{\geq 0}$) -> ()
  pre $?(a \leq b)$
  post $?(V[x \mapsto a] \geq V[x \mapsto b])$ {}
```

The following procedure checks that $[I]$ is a wp-subinvariant with respect to post $[I]$:

```
proc $I\_wp\_subinvariant$ ($x_1^0 \colon \tau_1, \ldots, x_n^0 \colon \tau_n$) -> ($x_1 \colon \tau_1, \ldots, x_n \colon \tau_n$)
  pre $[I_0]$
  post $[I]$
{
  var $x_1 \colon \tau_1$ :≈ $x_1^0$; ... ; var $x_n \colon \tau_1$ :≈ $x_n^0$
  if ($b$) {
    $enc_{\mathrm{wp}} \lfloor C_0 \rfloor$
  }
}
```

The next condition:

```
proc $termination\_condition$ ($x_1 \colon \tau_1, \ldots, x_n \colon \tau_n$) -> () {
  assert $?((\neg b \land I) \longrightarrow (V > 0))$
}
```

Then, we check that ${}^{\mathrm{wp}}\Phi_V(V) \preceq V$ holds:

```
coproc $v\_wp\_superinvariant$ ($x_1^0 \colon \tau_1, \ldots, x_n^0 \colon \tau_n$) -> ($x_1 \colon \tau_1, \ldots, x_n \colon \tau_n$)
  pre $V_0$
  post $V$
{
  var $x_1 \colon \tau_1$ :≈ $x_1^0$; ... ; var $x_n \colon \tau_1$ :≈ $x_n^0$
  if ($b$) {
    $enc_{\mathrm{wp}}^{\mathrm{co}} \lfloor C_0 \rfloor$
  }
}
```

Finally, the progress condition $[I] \cdot [G] \cdot (p \circ V) \preceq \lambda\sigma.\, \mathrm{wp}[\![C_0]\!]([V < V(\sigma) - d(V(\sigma))])(\sigma)$:

```
proc $progress$ ($x_1^0 \colon \tau_1, \ldots, x_n^0 \colon \tau_n$) -> ($x_1 \colon \tau_1, \ldots, x_n \colon \tau_n$)
   pre $[I_0] \cdot [G_0] \cdot (p \circ V_0)$
   post $[V < V_0 - d(V_0)]$
{
   var $x_1 \colon \tau_1$ :≈ $x_1^0$; ...; var $x_n \colon \tau_1$ :≈ $x_n^0$
   $enc_{\mathrm{wp}}\lfloor C_0 \rfloor$
}
```

C.8 PAST Rule

Proof Rule: *PAST from Ranking Superinvariants* [Chakarov and Sankaranarayanan 2013]. Let $C = \texttt{while}\,(\,b\,)\,\{\,C_0\,\}$ be a pGCL loop and let $I \in \mathbb{E}$. Let constants ϵ and K such that $0 < \epsilon < K$. If the following conditions hold, then C terminates universally positively almost-surely:

- $[\neg b] \cdot I \preceq K$,
- $[b] \cdot K \preceq [e] \cdot I + [\neg b]$,
- ${}^{\mathrm{wp}}\Phi_0(I) \preceq [b] \cdot (I - \epsilon)$,

where the *loop-characteristic functional* ${}^{\mathrm{wp}}\Phi_X$ with respect to post X is defined as

$${}^{\mathrm{wp}}\Phi_X(Y) = [b] \cdot \mathrm{wp}[\![C_0]\!](Y) + [\neg b] \cdot X\,.$$

Encoding Input:

- pGCL loop $C = \texttt{while}\,(\,b\,)\,\{\,C_0\,\}$.
- HeyVL encoding $enc^{\mathrm{co}}_{\mathrm{wp}}\lfloor C_0 \rfloor$ that satisfies $\mathrm{wp}[\![C_0]\!] \preceq \mathrm{vp}[\![enc^{\mathrm{co}}_{\mathrm{wp}}\lfloor C_0 \rfloor]\!]$.
- Potential invariant $I \in$ HeyLo.
- Constants ϵ and K such that $0 < \epsilon < K$.

Encoding Output:

- HeyVL encoding S that verifies only if C is PAST.

The first two conditions can be encoded easily via assertions:

```
proc $condition\_1$ ($x_1 \colon \tau_1, \ldots, x_n \colon \tau_n$) -> () {
   assert $?([\neg b] \cdot I \leq K)$
}

proc $condition\_2$ ($x_1 \colon \tau_1, \ldots, x_n \colon \tau_n$) -> () {
   assert $?([b] \cdot K \leq [b] \cdot I + [\neg b])$
}
```

The last condition, ${}^{\mathrm{wp}}\Phi_0(I) \preceq [b] \cdot (I - \epsilon)$, is encoded as another coprocedure:

```
coproc $condition\_3$ ($x_1^0 \colon \tau_1, \ldots, x_n^0 \colon \tau_n$) -> ()
  pre $([b] \cdot (I - \epsilon))[x_1 \mapsto x_1^0] \ldots [x_n \mapsto x_n^0]$
  post $0$
{
  var $x_1 \colon \tau_1 :\approx x_1^0$;  ... ; var $x_n \colon \tau_1 :\approx x_n^0$
  if ($b$) {
    $enc_{\mathrm{wp}}^{\mathrm{co}}\lfloor C_0 \rfloor$
    coassert $I$
    coassume $?(\mathrm{true})$
  }
}
```