Master's Thesis

A Deductive Verifier for Probabilistic Programs

by
Philipp Schroer

First Examiner:
Prof. Dr. Ir. Dr. h.c. Joost-Pieter Katoen (RWTH Aachen University)

Second Examiner:
Prof. Dr. Peter Müller (ETH Zürich)

Supervisors:
Kevin Batz (RWTH Aachen University)
Prof. Dr. Benjamin Kaminski (Saarland University, University College London)
Prof. Dr. Christoph Matheja (Technical University of Denmark, ETH Zürich)

Submitted: 2022-01-25
Revised: 2026-05-04

Abstract

We design and implement a deductive verification infrastructure for probabilistic programs. It consists of a quantitative intermediate verification language (HeyVL) and a quantitative assertion language (HeyLo). HeyLo is a syntax to express expected values of probabilistic programs, with support for quantitative implications based on Gödel logic. Both HeyLo and HeyVL contain lattice-theoretic dual constructs to reason about lower and upper bounds of expected values.

As a case study, we encode weakest pre-expectation and weakest liberal pre-expectation reasoning about the probabilistic programming language pGCL into HeyVL. For loops, we provide encodings of Park induction, *k*-induction, and bounded model checking. Park induction and *k*-induction are both proof rules that require on user-provided invariant candidates.

Furthermore, we discuss the automation of our deductive verification infrastructure. Our implementation Caesar takes a HeyVL program as input, generates and optimizes verification conditions in the form of HeyLo formulas and uses the automated theorem prover Z3 to prove or disprove validity of the verification conditions. In this thesis, we focus on the central optimization of quantifier elimination of HeyLo formulas. We present early promising experimental results.

Finally, we discuss the abstraction of our framework based on Heyting and Gödel algebras to support more domains than expectations.

Contents

1. Introduction

Probabilistic models are at the core of many discussions about the world's current most pressing problems. For example, research and reporting on both climate change and the current COVID-19 pandemic are often about processes and observations that are probabilistic and uncertain. Whether it is a discussion on the likelihood and predicted effects of climate change models or the spread of certain virus variants, probabilistic models play a key role. Especially with respect to the inherent complexity of accurate models for such problems, the accessibility, expressiveness, analysis, and verification of probabilistic models are of substantial interest.

Probabilistic programs are programs whose behavior can depend on uncertain inputs, such as those modeled by probability distributions. Because they can be written in a way familiar to programmers, probabilistic programs can be a widely accessible way to express probabilistic models. Probabilistic programs are able to express probabilistic graphical models that are more powerful than Bayesian networks [BKS20].

However, the analysis of probabilistic programs is hard: It inherits all existing complexities from classical programs such as the undecidability of termination checking and complicates the classical analyses with probabilistic behavior. In the case of termination checking, we can ask new questions about probabilistic programs, such as "what is the probability of termination?" or "does this program always terminate in all executions?" On the arithmetical hierarchy, deciding the latter question for probabilistic programs has been shown to be *even more undecidable* than in the classical case [KKM19].

In this thesis, we focus on questions about the *expected values* of expressions after the execution of a probabilistic program. The expected values depend on how non-termination of the probabilistic program is taken into account. Through expected values, we can analyze termination probabilities and classical verification questions like "when starting in a state that satisfies some property *Pre*, what is the probability of the program terminating in a state that satisfies *Post*?"

The following simple probabilistic program assigns $y + 1$ to x with probability 0.7 and $2 \cdot y$ to x with probability 0.3:

$$\{x := y + 1\}\ [0.7]\ \{x := 2 \cdot y\}\ .$$

When starting in a state where y is even, what is the probability of x being even as well? The answer is $1 - 0.7 = 0.3$, but we would like to formalize and automate such an analysis

as much as possible. This becomes more complicated when the program under scrutiny contains more advanced control flow such as conditional branches or loops.

There are many techniques for classical, non-probabilistic programs that are used to prove some correctness property holds or to find errors in programs. Static program analysis [CC77], model checking [BK08], or deductive verification [Hoa69] have all been used for both non-probabilistic and probabilistic programs. Compared to the former two techniques, deductive verification works with more user input, but allows re-use of already verified programs in a modular fashion [Mül02].

For non-probabilistic programs, there have been many program verifiers that use deductive verification to formally prove properties of programs. Using mathematical logics, such as Hoare logic [Hoa69] or separation logic [Rey02], program verifiers attempt to find proofs of these properties. Most program verifiers generate a set of *verification conditions* in the form of logical formulas that specify whether a program verifies. If these verification conditions are valid, then the correctness of the program is implied. We briefly introduce classical verification conditions in Section 2.1.

Because of the difficulty of formally showing properties of programs in an automated way, program verifiers quickly become very complex. Therefore, *intermediate verification languages* (IVLs) have been developed. They aim to provide a layer between the original program to be verified and the verification conditions. These IVLs are used to encode the program, its specification, and even proof techniques into one program in the IVL. An intermediate verification language is typically simpler than a commonly used programming language. By abstracting from language design details, an IVL can be used for the verification of programs in different programming languages. Proof techniques, optimizations on the IVL, and the verification condition generation itself can thus be shared by different program verifiers. IVLs are often similar in shape to classical programming languages and can be easier to understand than the verification conditions corresponding to it. Even some classical program optimizations like constant value propagation can be adapted to IVLs. All of the following IVLs and associated tools implement such optimizations.

Boogie is a verification infrastructure with a procedural intermediate verification language and a tool to check whether an IVL program verifies [Lei08]. Viper is a verification infrastructure that also uses a procedural IVL [MSS16]. A functional verification language is used by Why3 [FP13]. Based on these infrastructures, program verifiers for a large number of programming languages have been developed. Programs written in the Rust programming language can be verified using Prusti [Ast+19], Nagini is a verifier for Python [EM18], and Gobra is a verifier for Go [Wol+21]. These three verifiers all use the Viper infrastructure. The verifiers Dafny [Lei10] and Chalice [LMS09] use Boogie. The Viper infrastructure supports the use of Boogie as a back-end, further highlighting the versatility of generalized tools for intermediate steps of program verification.

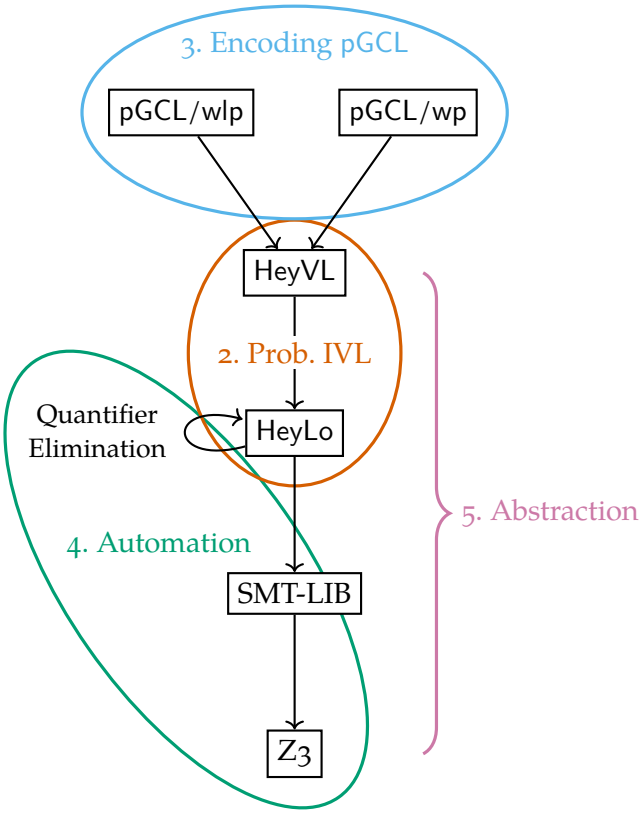

Figure 1.1.: Outline of our deductive verifier infrastructure for probabilistic programs with references to the relevant chapters in this thesis.

The verification of probabilistic programs is actively researched as well. Kozen defined semantics for probabilistic programs [Koz79; Koz81], and more recently probabilistic versions of weakest pre-condition calculi have been developed [MM05]. Martingales have also been used for verification, e.g. [CS13]. Examples for automated verification approaches include [HMM05] and [NCH18].

The goal of this thesis is to develop a deductive verifier infrastructure for probabilistic programs. To our knowledge, such an infrastructure has not been presented before. Figure 1.1 outlines our approach and the relevant chapters in this thesis. In Chapter 2, we present our intermediate verification language HeyVL for probabilistic programs. For this, we first develop our own intermediate logic HeyLo in Section 2.2 to represent the verification conditions of HeyVL and to express probabilistic verification problems with. HeyLo is based on the quantitative assertion language by Batz et al. [Bat+21b] and we add generalizations and lattice-theoretic duals of operators from Gödel logic [Göd32]. Then, we formally define HeyVL in Section 2.3 and show a number of properties and generic encodings that are important for a probabilistic intermediate verification language.

In Chapter 3, we encode the probabilistic programming language pGCL in HeyVL to show how verification problems for a probabilistic language can be embedded in our IVL. We consider both *weakest pre-expectation* (wp) and *weakest liberal pre-expectation* (wlp) semantics for pGCL (Section 3.2). These semantics are defined on *expectations*, representing the expected values mentioned above. For the encoding of loops, we present proof rules for Park induction (Section 3.6.1) and k-induction (Section 3.6.2) to show lower bounds on weakest liberal pre-expectations and to show upper bounds on weakest pre-expectations. In Section 3.7, we show that our two encodings of pGCL programs are correct for the respective semantics.

In Chapter 4, we discuss how to automate the verification of HeyVL programs. For example, the semantics of HeyVL clearly define how to generate verification conditions as formulas in HeyLo. However, HeyLo is an intermediate logic that we defined to describe verification conditions for probabilistic programs. Automated validity checking of HeyLo formulas is non-trivial. We describe a series of steps to transform a HeyLo formula into a representation suitable for the automated theorem prover Z3 [dMBo8].

The occurrence of quantifiers in the verification conditions is a problem. In HeyLo, the quantifiers correspond to least upper bounds and greatest lower bounds. In our experiments, Z3 was often unable to decide validity of verification conditions with our encoding of these quantifiers. Therefore, we develop a *quantifier elimination* technique for HeyLo in Section 4.2. We show that quantifier elimination is always possible in verification conditions for HeyVL programs that we encode for pGCL.

All of these constructions and techniques are used in our implementation of a deductive verifier *Caesar* (Section 4.3). Caesar is a tool that generates HeyLo verification conditions for an intermediate verification program written in an extended version of HeyVL. In contrast to the simplified theoretical presentation of HeyVL we give in the preceding sections, Caesar supports a type system, modularity through *procedures*, and more useful features for an IVL. The quantifier elimination techniques and more optimizations are already implemented in Caesar.

Finally, Chapter 5 takes a step back and reviews our constructions of HeyLo and HeyVL from a more abstract perspective. We redefine HeyLo as a *bi-Gödel logic* with *Heyting algebras* as models. Similarly, HeyVL is redefined over Heyting algebras. With more abstract definitions, we generalize our constructions to other domains than just expectations valued over non-negative reals with infinity. It becomes obvious that HeyVL directly generalizes classical intermediate verification languages and adds syntax for the lattice-theoretic duals of logical operators.

In the conclusion (Chapter 6), we evaluate our intermediate verification language and look at possible future work on HeyLo, HeyVL, and our implementation Caesar.

2. An Intermediate Verification Language for Probabilistic Programs

HeyVL is an intermediate verification language for probabilistic programs that is very similar to classical IVLs. It is based on the same set of statements as classical IVLs, but also includes lattice-theoretic duals to these statements and supports quantitative reasoning. In the next section (Section 2.1), we briefly present a simple intermediate verification language for non-probabilistic verification. This will provide a basic and intuitive understanding for the more complex language HeyVL and the associated assertion language HeyLo. As HeyLo is used in both the syntax of HeyVL and for the representation of verification conditions, we present it next in Section 2.2.

HeyVL is defined in Section 2.3. We introduce syntax and verification condition semantics, as well as healthiness conditions. Based on a deconstruction of HeyLo, we try to give an intuitive understanding of HeyVL statements (Section 2.3.5). Finally, we show how to encode simple verification problems with specifications in Section 2.3.6 and how to replace HeyVL programs with their specification (Section 2.3.7).

2.1. Design of Intermediate Verification Languages

We first look at the general design of a simple, imperative, non-probabilistic intermediate verification language in the style of [Mül19]. Based on its general design, we develop HeyLo and HeyVL in the next sections.

The *simple IVL* is an imperative language with a small number of statements to express verification problems. It is based on Dijkstra's guarded-commands language [Dij75]. There are assignments, sequential composition, and non-deterministic branches. In addition to these well-known basic constructs, the simple IVL supports `havoc`, `assume`, and `assert` statements. The syntax of the simple IVL and a *verification condition* semantics is defined in Figure 2.1. Syntax and verification conditions $\mathsf{vc}[\![S]\!] \colon \mathbb{P} \to \mathbb{P}$ are defined over *predicates* $\mathbb{P}$. For our purposes, these are formulas in first-order propositional logic.

The syntax of the simple IVL consists of an assignment statement $x := a$ that evaluates the expression a in the current state and assigns the result to the variable x. It also has sequential composition $S_1 \,;\, S_2$: The program S_1 is executed before S_2. Additionally, it

The simple IVL has the following syntax:

$$
\begin{aligned}
S \ ::= \ & x := a && \text{(assignment)} \\
\mid \ & \texttt{havoc}\ x && \text{(havoc)} \\
\mid \ & \texttt{assume}\ P && \text{(assume)} \\
\mid \ & \texttt{assert}\ P && \text{(assert)} \\
\mid \ & S\,;\ S && \text{(sequential composition)} \\
\mid \ & \texttt{if (*) \{}S\texttt{\} else \{}S\texttt{\}}\,, && \text{(non-deterministic choice)}
\end{aligned}
$$

where x is a variable, E is an expression, and P is a predicate.

For a statement S of the simple IVL, we define the *verification conditions* $\mathsf{vc}[\![S]\!] : \mathbb{P} \to \mathbb{P}$ as follows:

S	$\mathsf{vc}[\![S]\!](P)$
$x := a$	$P[x \mapsto a]$
$\texttt{havoc}\ x$	$\forall x.\ P$
$\texttt{assume}\ Q$	$Q \Rightarrow P$
$\texttt{assert}\ Q$	$Q \wedge P$
$S_1\,;\ S_2$	$\mathsf{vc}[\![S_1]\!](\mathsf{vc}[\![S_2]\!](P))$
$\texttt{if (*) \{}S_1\texttt{\} else \{}S_2\texttt{\}}$	$\mathsf{vc}[\![S_1]\!](P) \wedge \mathsf{vc}[\![S_2]\!](P)$

Figure 2.1.: A simple intermediate verification language

supports a (demonic) non-deterministic choice $\texttt{if (*) \{}S_1\texttt{\} else \{}S_2\texttt{\}}$. With respect to verification, it can be interpreted as the choice between S_1 and S_2 that refutes the property we want to show if possible.

Out of the remaining three statements for verification, the `assert` statement is the only one with clear operational semantics. Intuitively, $\texttt{assert}\ P$ checks whether P holds in the current state. If not, the program goes to an error state.

The $\texttt{assume}\ P$ statement can seem "magical" [Lei08]: It somehow makes a property P true. In other words, all executions of the program where P is not true in $\texttt{assume}\ P$ are ignored for the purposes of verification.

Finally, the havoc statement invalidates all knowledge about the value of a variable. After $\texttt{havoc}\ x$, the value of x is unknown and correctness must be shown without knowledge about the value of x, i.e. for all values in the domain of the variable x.

For the simple IVL, we say the execution from an initial state *fails* if a condition of an assertion evaluates to false. An *infeasible* execution has a condition of an assumption that evaluates to false. Otherwise, it *succeeds*. If all feasible executions succeed, we say the program is *correct*.

The verification conditions $\mathsf{vc}[\![S]\!](\mathsf{true}) \in \mathbb{P}$ of a simple IVL program S with *post-condition* true are valid iff S is correct. The vc function is based on Dijkstra's *weakest pre-condition transformer*. Assignment corresponds to substitution in the post-condition. A havoc statement introduces a universal quantifier. The `assume` statement creates an implication and the `assert` statement a conjunction. Sequential execution first computes the verification conditions of S_2, then of S_1. Finally, the non-deterministic choice requires that the verification conditions of both branches are true.

Note that this simple IVL does not include a Boolean choice $\texttt{if}\ (b)\ \{S_1\}\ \texttt{else}\ \{S_2\}$. But the Boolean choice can be encoded in the simple IVL using the non-deterministic choice and assumption statements:

$$S = \quad \texttt{if}\ (*)\ \{\texttt{assume}\ b;\ S_1\}\ \texttt{else}\ \{\texttt{assume}\ \neg b;\ S_2\}.$$

Starting in a state $\sigma \in \Sigma$ that satisfies the condition b, an execution through S_2 is infeasible and vice versa for a state satisfying $\neg b$ visiting S_2. The vc of this encoding with respect to post-condition P is given by:

$$\mathsf{vc}[\![S]\!](P) = \quad (b \Rightarrow \mathsf{vc}[\![S_1]\!](P)) \wedge (\neg b \Rightarrow \mathsf{vc}[\![S_2]\!](P)).$$

That is, the verification condition $\mathsf{vc}[\![S]\!](P)$ with respect to post-condition P is valid iff $\mathsf{vc}[\![S_1]\!](P)$ is true when b holds and if $\mathsf{vc}[\![S_2]\!](P)$ is true when b does not hold.

Example 2.2. Simple IVL: Feasible, infeasible, and failing executions

Consider this simple IVL program:

$$S = \quad \texttt{assume}\ (x = 3 \vee x = 5);\ x := x + 1;\ \texttt{assert}\ x = 4\,.$$

The execution starting from state satisfying $x = 3$ is feasible and does not fail, therefore it succeeds. On the other hand, starting from state $x = 5$ is feasible, but it fails. Executions starting from all other states are infeasible. Therefore, the program is not correct. The verification conditions evaluate to:

$$\mathsf{vc}[\![S]\!](\mathsf{true}) = \quad (x = 3 \vee x = 5) \Rightarrow (x + 1 = 4)\,,$$

which is false for $x = 5$.

HeyVL is based on the simple IVL construction. HeyVL has `havoc`, `assume`, and `assert` statements, but with HeyLo formulas instead of predicates. The intuition we provided for infeasible and succeeding executions of this simple IVL can help understanding HeyVL. When the HeyLo formulas represent only predicates, the explanations above apply to HeyVL as well. However, quantitative reasoning requires a shift from individual executions to cumulative probabilities and expected values over multiple executions.

Fortunately, we will see in Chapter 3 that many well-known constructions that work in the classical setting can be generalized quite naturally to probabilistic programs. One important example of encodings which can be generalized is the encoding of loops. Just like the simple IVL, HeyVL does not feature a loop statement. Instead, proof rules for the verification of loops are encoded using the basic statements of the IVL. We discuss the encoding of loops in detail in Section 3.6.

2.2. HeyLo: An Assertion Language for Probabilistic Programs

The central data type our verifier works with is the assertion language HeyLo. The assertion language is used to represent the verification conditions whose validity ultimately indicates whether a HeyVL program is considered correct. It is also used directly within the syntax of HeyVL itself for assertion and assumption statements.

2.2.1. Expectations

In existing deductive verifiers such as the ones mentioned in the previous section, the assertion language is a logic that is used to specify a property on a program state. A simple assertion language might be first-order propositional logic. The verifier infrastructures Viper and Boogie use more complex logics that allow so-called *permissions*. In any case, the logics are *qualitative*: a property holds on a certain program state or it does not.

The verification of probabilistic programs requires *quantitative* reasoning. An assertion does not simply assign either true or false to a state, but instead assigns a value from a larger set of valuations to a state. For the semantics of probabilistic programs, we work with so-called *expectations* that assign a non-negative real number to a state. We introduce the corresponding *weakest pre-expectation calculus* in Section 3.2 for the translation of pGCL to HeyVL. For now, we only consider the assertions themselves formally. We begin with the definition of program states.

Definition 2.3. Program states

Let Vars be a countable set of *program variables*.

The *set of program states* is defined as:

$$\Sigma = \left\{ \sigma \,\middle|\, \sigma \colon \mathsf{Vars} \to \mathbb{Q}_{\geq 0} \right\}$$

We write $[\![E]\!](\sigma)$ for the *evaluation* of an expression E in state σ. Given a program state $\sigma \in \Sigma$ and an expression E, we write $\sigma[x \mapsto E]$ for the state σ with x replaced by E:

$$\sigma[x \mapsto E] = \lambda y. \begin{cases} [\![E]\!](\sigma), & \text{if } x = y \\ \sigma(y), & \text{else} \end{cases}$$

In this thesis, we define program states as mappings from variables to non-negative rational numbers. The restriction to non-negative values does not limit expressiveness as negative numbers can be represented with additional variables in the program. However, it simplifies some details in our presentation because $\mathbb{Q}_{\geq 0}$ is a subset of the set $\mathbb{R}^{\infty}_{\geq 0}$ we

use for expectations. More detailed explanations for this choice can be found in [Bat+21b, Section 11].

Definition 2.4. Expectations

The set $\mathbb{E}$ of *expectations* is defined as:

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

One-bounded expectations $X \in \mathbb{E}_{\leq 1}$ have a domain restricted to $[0,1] \subseteq \mathbb{R}$:

$$\mathbb{E}_{\leq 1} = \{ X \mid X \colon \Sigma \to [0,1] \}.$$

The set $\mathbb{R}^{\infty}_{\geq 0}$ includes non-negative real numbers and infinity: $\mathbb{R}^{\infty}_{\geq 0} = \{ r \in \mathbb{R} \mid r \geq 0 \} \cup \{ \infty \}$. The ability of expectations to map to ∞ is an extension of the work of McIver and Morgan [MM05] where only *bounded expectations* are considered. A bounded expectation X must have a real-valued bound $a \in \mathbb{R}_{\geq 0}$ such that for all $\sigma \in \Sigma$, $X(\sigma) \leq a$ holds. With bounded expectations, it is difficult to ensure that expectations are always well-defined. For example, the expectation $\lambda\, \sigma.\, \sigma(x)$ that returns the current value of variable x, is not bounded.

One frequently used kind of (bounded) expectation is the *Iverson bracket* [Ive62].

Definition 2.5. Iverson bracket

The *Iverson bracket* for a Boolean expression b is a function in $\mathbb{E}_{\leq 1}$ given by:

$$[b] = \lambda\, \sigma. \begin{cases} 1, & \text{if } [\![b]\!](\sigma) = \mathsf{true} \\ 0, & \text{else} \end{cases}$$

The Iverson bracket $[b]$ maps a Boolean value $b \in \mathbb{B}$ to 1 if b is true in the current state and 0 otherwise.

Example 2.6. Expectation with Iverson bracket

The expectation $X = [x = 3] \cdot 0.5 + 0.5$ assigns $X(\sigma) = 1$ to a state $\sigma \in \Sigma$ satisfying $\sigma(x) = 3$ and assigns $X(\sigma) = 0.5$ to all other states σ.

In addition to the Iverson bracket, we define the *Boolean embedding*. It maps true to ∞ and false to 0.

Definition 2.7. Boolean embedding

The *Boolean embedding* for a Boolean expression b is a function in $\mathbb{E}$ given by:

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

The Boolean embedding $?(b)$ is equivalent to $[b]\cdot\infty$. We generally assume that $0\cdot\infty = 0$.

Expectations admit a partial order $(\mathbb{E}, \sqsubseteq)$ where for all $X, Y \in \mathbb{E}$,

$$X \sqsubseteq Y \quad \text{iff} \quad \forall\sigma \in \Sigma. \qquad X(\sigma) \leq Y(\sigma)\ .$$

The structure $(\mathbb{E}, \sqsubseteq)$ forms a *complete lattice* with the bottom element 0 (the constant expectation $\lambda\,\sigma.\,0$) and the top element ∞ (the constant expectation $\lambda\,\sigma.\,\infty$). In a complete lattice, all greatest lower bounds (also known as infima or meets) and least upper bounds (suprema, joins) of its subsets exist. That means for all subsets $S \subseteq \mathbb{E}$, $\inf S \in \mathbb{E}$ and $\sup S \in \mathbb{E}$. We write $X \sqcap Y = \inf\{X, Y\}$ and $X \sqcup Y = \sup\{X, Y\}$ for the point-wise minimum and maximum of two expectations X and Y. We present more general lattice theory in Chapter 5.

Building on the substitution on individual states, we also define the substitution of variables in expectations. For $X \in \mathbb{E}$, the expectation $X[x \mapsto E]$ is the expectation X where each occurrence of the variable $x \in \mathsf{Vars}$ is replaced by the expression E:

$$X[x \mapsto E] = \lambda\,\sigma.\ X(\sigma[x \mapsto E]).$$

2.2.2. Requirements for HeyLo

The sets of expectations and the set of bounded expectations are all defined by mathematical sets of functions and do not have a fixed syntax with which to write them as input to our deductive verifier. Our language *HeyLo* is such a syntax, representing (a subset of) expectations $\mathbb{E}$.

As an assertion language, HeyLo requires a trade-off between opposing ideals. The language needs to be *expressive* enough to be able to state "interesting" properties for program verification. Additionally, it is used for the semantics of HeyVL, so it must be able to syntactically represent the semantics. On the other hand, *simplicity* is important so that analysis and validity checking of HeyLo are tractable.

Recent work by Batz et al. developed a *relatively complete* assertion language for probabilistic programs [Bat+21b]. Relative completeness means that the weakest pre-expectation of a program in the language pGCL with respect to any property written in their language can be represented syntactically in their language again. Batz et al. have shown that their

language is expressive enough for a large number of properties. We designed HeyLo so that it is a superset of the relatively complete language by Batz et al.

HeyLo's additional syntax consists of logical operators for expectations. In particular, we want *implications* and *conjunctions*. Implications are used to compare expectations and conjunctions are used to specify requirements for the program. In HeyVL, implications are used for `assume` statements and conjunctions for `assert` statements. This requirement is not immediately obvious, but we hope to show in this thesis by a number of examples that our logical operators on expectations are well-suited for specifying verification problems for probabilistic programs.

2.2.3. Logical Operators on Expectations

HeyLo includes a large number of operators. Before we overwhelm the reader with a large specification of the assertion language, we introduce the implications on the set of expectations. In total, HeyLo has *four* different implication operators. The *implication* $\to$, the *co-implication* $\leftarrow$, the *hard implication* $\searrow$, and finally the *hard co-implication* $\nwarrow$. All implications are functions that map two expectations to one expectation. However, all these implications can be derived easily from one starting point. This starting point is the *Gödel implication*, initially defined by Kurt Gödel [Göd32], which we have generalized to expectations as follows.

Definition 2.8. Implications on expectations

We define the *implication* $\to$ and the *co-implication* $\leftarrow$ on expectations:

$$\to\colon \mathbb{E} \times \mathbb{E} \to \mathbb{E} \qquad\qquad \leftarrow\colon \mathbb{E} \times \mathbb{E} \to \mathbb{E}$$

$$X \to Y = \lambda\sigma.\begin{cases} \infty, & \text{if } X(\sigma) \leq Y(\sigma) \\ Y(\sigma), & \text{else} \end{cases} \qquad X \leftarrow Y = \lambda\sigma.\begin{cases} 0, & \text{if } X(\sigma) \geq Y(\sigma) \\ Y(\sigma), & \text{else} \end{cases}$$

The implication $\to\colon \mathbb{E} \times \mathbb{E} \to \mathbb{E}$ returns an expectation, i.e. a function that takes a state $\sigma \in \Sigma$ and returns a value in $\mathbb{R}^\infty_{\geq 0}$. In a state σ, the expectation $X \to Y$ evaluates to the greatest element ∞ in $\mathbb{R}^\infty_{\geq 0}$ if $X(\sigma)$ is less than or equal to $Y(\sigma)$. Otherwise, $Y(\sigma)$ is returned. The co-implication is simply its order-theoretic dual: Instead of less than or equal, it checks for greater than or equal. In case the comparison succeeds, the least element 0 in $\mathbb{R}^\infty_{\geq 0}$ is returned by $X \leftarrow Y$.

Example 2.9. Implications on expectations

Consider the expectations $X = [x = 1] \cdot 0.6$ and $Y = 0.5$. With the implication $\to$, the

expectation $X \to Y$ is equivalent to:

$$X \to Y = \lambda\sigma. \begin{cases} \infty, & \text{if } ([x = 1] \cdot 0.6)(\sigma) \leq 0.5 \\ 0.5, & \text{else} \end{cases}$$
$$= \lambda\sigma. \begin{cases} \infty, & \text{if } \sigma(x) \neq 1 \\ 0.5, & \text{else} \end{cases} \qquad (([x = 1] \cdot 0.6)(\sigma) \leq 0.5 \text{ iff } \sigma(x) \neq 1)$$

On the other hand, with the co-implication $\leftarrow$, we get the following:

$$X \leftarrow Y = \lambda\sigma. \begin{cases} 0, & \text{if } ([x = 1] \cdot 0.6)(\sigma) \geq 0.5 \\ 0.5, & \text{else} \end{cases}$$
$$= \lambda\sigma. \begin{cases} 0, & \text{if } \sigma(x) = 1 \\ 0.5, & \text{else} \end{cases} \qquad (([x = 1] \cdot 0.6)(\sigma) \geq 0.5 \text{ iff } \sigma(x) = 1)$$

Most constructions in this thesis require only one side of the duality at a time. In HeyVL, we will associate the implication $\to$ with the "down direction" and the co-implication with the "up direction". Analogously for the hard implications. It is usually sufficient to remember the down variants and then derive the dual up variants from them.

With respect to the implication $\to$, ∞ behaves like true in classical Boolean logic. For all $X, Y \in \mathbb{E}$, we have

$$X \to \infty = \lambda\sigma. \begin{cases} \infty, & \text{if } X(\sigma) \leq \infty \\ \infty, & \text{else} \end{cases} = \infty$$
$$\infty \to Y = \lambda\sigma. \begin{cases} \infty, & \text{if } \infty \leq Y(\sigma) \\ Y(\sigma), & \text{else} \end{cases} = Y$$

Similarly, the constant expectation 0 behaves a lot like false. It holds that $X \to 0 = 0$ and $0 \to Y = \infty$.

As the order-theoretic dual to the implication $\to$, corresponding dual equalities hold for the co-implication $\leftarrow$: We have $X \leftarrow 0 = 0$, $0 \leftarrow Y = Y$, $X \leftarrow \infty = \infty$, and $\infty \leftarrow Y = 0$.

The Gödel implication and co-implication are not equivalent to their respective duals with arguments swapped.[Fixed] Whereas $(A \Rightarrow B) = (B \Leftarrow A)$ holds for $A, B \in \mathbb{B}$, we have seen for $X \in \mathbb{E}$ that $X \to 0 = 0$, but $0 \leftarrow X = X$.

For the point-wise minimum $\sqcap$ and the implication $\to$, there are similar rules as for the propositional $\wedge$ and $\Rightarrow$. The first equality corresponds to the well-known *modus ponens* inference rule from propositional logic.

$$X \sqcap (X \to Y) = X \sqcap Y$$
$$Y \sqcap (X \to Y) = Y$$

Dual equalities hold for the point-wise maximum $\sqcup$ and the co-implication $\leftarrow$. A more general statement is proven later in Chapter 5 (Theorem 5.2).

Since we have implications and corresponding elements that behave like false and true respectively, it is natural to define corresponding negations. There is a *negation* on expectations $\neg X = X \to 0$ and a *co-negation* $\sim X = X \leftarrow \infty$.

Definition 2.10. Negations on expectations

We define the *negation* $\neg$ and the *co-negation* $\sim$ on expectations:

$$\neg\colon \mathbb{E} \to \mathbb{E} \qquad\qquad \sim\colon \mathbb{E} \to \mathbb{E}$$

$$\neg X = \lambda\sigma. \begin{cases} \infty, & \text{if } X(\sigma) = 0 \\ 0, & \text{else} \end{cases} \qquad\qquad \sim X = \lambda\sigma. \begin{cases} 0, & \text{if } X(\sigma) = \infty \\ \infty, & \text{else} \end{cases}$$

Both negations flip between ∞ and 0: $\neg\neg\infty = \neg 0 = \infty$ and $\sim\sim 0 = \sim\infty = 0$. However, double negations cannot be eliminated in general. Consider two negations of the constant expectation 1: $\neg\neg 1 = \neg 0 = \infty$.[Fixed]

Finally, the *hard implications* $\searrow$ and $\nwarrow$ are just syntactic sugar for a combination of negations and implications. It holds that $X \searrow Y = \sim\sim(X \to Y)$ and $X \nwarrow Y = \neg\neg(X \leftarrow Y)$.[Fixed]

Definition 2.11. Hard implications on expectations

We define the *hard implication* $\searrow$ and the *hard co-implication* $\nwarrow$ on expectations:

$$\searrow\colon \mathbb{E} \times \mathbb{E} \to \mathbb{E} \qquad\qquad \nwarrow\colon \mathbb{E} \times \mathbb{E} \to \mathbb{E}$$

$$X \searrow Y = \lambda\sigma. \begin{cases} \infty, & \text{if } X(\sigma) \leq Y(\sigma) \\ 0, & \text{else} \end{cases} \qquad\qquad X \nwarrow Y = \lambda\sigma. \begin{cases} 0, & \text{if } X(\sigma) \geq Y(\sigma) \\ \infty, & \text{else} \end{cases}$$

The names and symbols of these operators are chosen to reflect their similar definitions to the implications $\to$ and $\leftarrow$. Whereas $X \to Y$ returns $Y(\sigma)$ if $X(\sigma) \leq Y(\sigma)$ does not hold in a state σ, $X \searrow Y$ pulls down the result from $Y(\sigma)$ to 0. Similarly, $X \nwarrow Y$ pulls the result up to ∞ from $Y(\sigma)$ in case $X(\sigma) \sqsupseteq Y(\sigma)$ does not hold.

The hard implications always return either ∞ or 0, based on the result of the comparison. They will be used to check whether an expectation is a valid bound of another expectation and returning basically a Boolean result. For example, while $3 \to 2 = 2$, we instead get $3 \searrow 2 = 0$ with a hard implication.

In Chapter 5, we show that the constructions of HeyLo and HeyVL can be generalized to complete lattices via so-called *Heyting algebras*. When defined in such a way, the four implications $\rightarrow$, $\leftarrow$, $\searrow$, and $\nwarrow$ can be instantiated on the lattice of Booleans $(\{\,\mathsf{false}, \mathsf{true}\,\}, \Rightarrow)$ ordered by the Boolean implication $\Rightarrow\colon \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ seen as a relation. On Booleans, $\rightarrow = \searrow = \Rightarrow$. The name *HeyLo* is derived from *Hey*ting and *lo*gic. In Chapter 5, we will discuss HeyLo as a *first-order intuitionistic bi-Gödel logic*, as well as the interpretation of the co-implication $\leftarrow$ on Booleans. However, an understanding of this more general view is not required to understand HeyLo and HeyVL for probabilistic programs.

2.2.4. A Relational View

Let $X, Y \in \mathbb{E}$ be two expectations. In this section, we rewrite the two relations $X \sqsubseteq Y$ and $X \sqsupseteq Y$, by deconstructing Y and leaving X opaque. We deconstruct Y based on the operators $\rightarrow$, $\leftarrow$, $\searrow$, $\nwarrow$, $\sqcap$, and $\sqcup$. This is what we call the *relational view*, because it is focused around the deconstruction of a relation between two expectations X and Y instead of the evaluation of one expectation in a specific state as presented in the previous section.

Why is this interesting? The deconstruction of specific operators on either the left-hand or the right-hand side of the relation $\sqsubseteq$ provides an important insight into when and how to use them. For example, the implication $\rightarrow$ on the right-hand side of $\sqsubseteq$, e.g. $X \sqsubseteq Y \rightarrow Z$, can be deconstructed into the disjunction $X \sqsubseteq Z \lor Y \sqsubseteq Z$ (Theorem 2.12). Therefore, $X \sqsubseteq Y \rightarrow Z$ can be interpreted as "X or Y must be a lower bound of Z".

We revisit these interpretations to explain the semantics of HeyVL in Section 2.3.5. In this context of verification of HeyVL, we check the *validity* of a HeyLo formula (representing an expectation). An understanding of the (recursive) deconstruction of $\infty \sqsubseteq Y$ will help to understand when a HeyVL program "verifies".

Theorem 2.12. Relational view on the logical operators

Let $X, Y, Z \in \mathbb{E}$ and $S \subseteq \mathbb{E}$. The following equivalences hold.

this...		is equivalent to:	this...		is equivalent to:
$Z \sqsubseteq$	∞	true	∞	$\sqsubseteq Z$	$Z = \infty$
$Z \sqsubseteq$	0	$Z = 0$	0	$\sqsubseteq Z$	true
$Z \sqsubseteq$	$X \sqcap Y$	$Z \sqsubseteq X \wedge Z \sqsubseteq Y$	$X \sqcap Y$	$\sqsubseteq Z$	$X \sqsubseteq Z \vee Y \sqsubseteq Z$
$Z \sqsubseteq$	$X \sqcup Y$	$Z \sqsubseteq X \vee Z \sqsubseteq Y$	$X \sqcup Y$	$\sqsubseteq Z$	$X \sqsubseteq Z \wedge Y \sqsubseteq Z$
$Z \sqsubseteq$	$X \to Y$	$X \sqsubseteq Y \vee Z \sqsubseteq Y$	$X \leftarrow Y$	$\sqsubseteq Z$	$Y \sqsubseteq X \vee Y \sqsubseteq Z$
$Z \sqsubseteq$	$X \searrow Y$	$X \sqsubseteq Y \vee Z = 0$	$X \nwarrow Y$	$\sqsubseteq Z$	$Y \sqsubseteq x \vee Z = \infty$
$Z \sqsubseteq$	$\inf S$	$\forall Y \in S.\, Z \sqsubseteq S$	$\sup S$	$\sqsubseteq Z$	$\forall Y \in S.\, Y \sqsubseteq Z$

The proof for Theorem 2.12 is provided in Appendix A on page 117.

The expression $Z \sqsubseteq X \sqcap Y$ is true if and only if both $Z \sqsubseteq Y$ and $Z \sqsubseteq Y$ hold. From this lower bounds perspective, $\sqcap$ behaves just like the Boolean $\wedge$. The expression $Z \sqsubseteq X \to Y$ is true iff $X \sqsubseteq Y$ holds or $Z \sqsubseteq Y$ holds. So Z is a lower bound to $X \to Y$ if X is a lower bound of Y or if Z already is a lower bound of Y.

An infimum $\inf S$ of the right-hand side of a relation $\sqsubseteq$ can be rewritten using a universal quantifier $\forall$. Dually, a supremum $\sup S$ on the left-hand side of a relation can also be rewritten using the universal quantifier. This justifies the view of the infimum as the equivalent of the universal quantifier for lower bounds and the supremum as the universal quantifier for upper bounds.

Example 2.13. Relational view

Consider the expectation $X = \inf \{\, x \to 0.5 \mid x \in \mathbb{Q}_{\geq 0} \,\}$. By Theorem 2.12, the relation $\infty \sqsubseteq X$ can be rewritten as $\forall x \in \mathbb{Q}_{\geq 0}.\ x \sqsubseteq 0.5 \vee \infty \sqsubseteq 0.5$, which is equivalent to false.

Note that there are no general correspondences for the existential quantifier $\exists$. By duality to the quantifier rules above, one might also expect a rule like

$$Z \sqsubseteq \sup S \quad \overset{\not\iff}{\text{iff}} \quad \exists Y \in S.\, Z \sqsubseteq Y.$$

This rule is not valid in general. Consider $\infty \sqsubseteq \sup \{\, x \mid x \in \mathbb{Q}_{\geq 0} \,\}$. The relation holds since the supremum evaluates to ∞. However, there is no $x \in \mathbb{Q}_{\geq 0}$ such that $\infty \sqsubseteq x$. One can also construct a counterexample with lower bound 1 and limit that approaches 1 where the rule fails, even if x was chosen from the larger domain $\mathbb{R}^{\infty}_{\geq 0}$ [Bat+21b,

Section 4.6]. In lattice theory, the properties that would allow the creation of existential quantifiers are called *completely meet/join prime* or *perfect* [GNV05].

In addition to the equivalences in Theorem 2.12, there are also similar, but less pretty equivalences for the co-implications $\leftarrow$ and $\nwarrow$ with $\sqsubseteq$ and also dual statements. Without a proof, we note that the following equivalence holds:

$$Z \sqsubseteq X \leftarrow Y \quad \text{iff} \quad (Y \sqsubseteq X \wedge Z \equiv 0) \vee Z \sqsubseteq Y.$$

These are less intuitive and show that the implications $\rightarrow$ and $\searrow$ behave more naturally with respect to lower bounds, while co-implications $\leftarrow$ and $\nwarrow$ behave more naturally with upper bounds.

As a final note, we observe that the equivalences on expectations $\mathbb{E}$ presented in Theorem 2.12 are also applicable to the underlying domain $\mathbb{R}^{\infty}_{\geq 0}$. In fact, these rules apply in any bi-Gödel algebra (cf. Chapter 5). Therefore, a (potentially recursive) decomposition of an expectation evaluated at a specific state in the same manner is also possible. In our implementation, this state-based recursive decomposition enables the simplification of a validity check from a potentially large number of explicit evaluations to a validity check of a first-order Boolean logic formula composed of simpler relations between atomic expressions occurring in the expectation.

2.2.5. The HeyLo Language

In this section, we formally specify the set HeyLo of HeyLo formulas and their semantics. The elements of HeyLo represent expectations $\mathbb{E}$ with a fixed syntax. Because one of the key features of HeyLo are the implications and corresponding conjunctions and disjunctions, we call the elements of HeyLo *formulas*.

HeyLo formulas can contain arithmetic expressions $a \in \mathsf{ArithExp}$ and Boolean expressions $b \in \mathsf{BExp}$. We now formally define their syntax and semantics. The definitions follow the work of Batz et al. [Bat+21b], so that HeyLo is a superset of their *relatively complete* language for expectations. The syntax of expressions will also be used for the language pGCL (Section 3.1).

Definition 2.14. Arithmetic expressions

An *arithmetic expression* $a \in \mathsf{ArithExp}$ follows the grammar

$$\begin{aligned} a \ ::= & \ r \in \mathbb{Q}_{\geq 0} && \text{(constants)} \\ \mid & \ x \in \mathsf{Vars} && (\mathbb{Q}_{\geq 0}\text{-valued variables}) \\ \mid & \ a + a && \text{(addition)} \\ \mid & \ a \cdot a && \text{(multiplication)} \\ \mid & \ a \dot{-} a \ . && \text{(subtraction truncated at 0 (“monus”))} \end{aligned}$$

An arithmetic expression can contain non-negative rational number constants and variables. On top of these, addition, multiplication, and *monus* operations are allowed. The monus operation is a subtraction that truncates at zero.

Definition 2.15. Boolean expressions

A *Boolean expression* $b \in \mathsf{BExp}$ follows the grammar

$$\begin{aligned} b \ ::= & \ \mathsf{true} \mid \mathsf{false} && \text{(constants)} \\ \mid & \ a = a \mid a < a \mid a > a && \text{(comparisons)} \\ \mid & \ b \wedge b \mid b \vee b \mid \neg b \ , && \text{(Boolean combinations)} \end{aligned}$$

where a is an arithmetic expression.

Boolean expressions can contain the Boolean constants true and false, as well as comparisons of arithmetic expressions with the usual comparison operators $=$, $<$, and $>$. We also support the standard Boolean operators $\wedge$, $\vee$, and $\neg$.

The evaluation of arithmetic and Boolean expressions is defined in Definition 2.16. We use the *monus* operator $\dot{-}$ instead of the usual minus $-$ so that the result of $b \dot{-} c$ is always well-defined on non-negative rationals.

Definition 2.16. Arithmetic and Boolean semantics

The *evaluation* $[\![a]\!](\sigma)$ of an arithmetic expression $a \in \mathsf{ArithExp}$ in a state $\sigma \in \Sigma$ and the *evaluation* $[\![b]\!](\sigma)$ of a Boolean expression $b \in \mathsf{BExp}$ in state σ are given by:

a	$[\![a]\!](\sigma)$	b	$[\![b]\!](\sigma)$
r	r	$c = d$	$[\![c]\!](\sigma) = [\![d]\!](\sigma)$
x	$\sigma(x)$	$c < d$	$[\![c]\!](\sigma) < [\![d]\!](\sigma)$
$b + c$	$[\![b]\!](\sigma) + [\![c]\!](\sigma)$	$c > d$	$[\![c]\!](\sigma) > [\![d]\!](\sigma)$
$b \cdot c$	$[\![b]\!](\sigma) \cdot [\![c]\!](\sigma)$	$c \wedge d$	$[\![c]\!](\sigma) \wedge [\![d]\!](\sigma)$
$b \dot{-} c$	$\begin{cases} [\![b]\!](\sigma) - [\![c]\!](\sigma), & \text{if } [\![b]\!](\sigma) \geq [\![c]\!](\sigma) \\ 0, & \text{else} \end{cases}$	$c \vee d$	$[\![c]\!](\sigma) \vee [\![d]\!](\sigma)$
		$\neg c$	$\neg [\![c]\!](\sigma)$

Definition 2.17 specifies the syntax of HeyLo formulas and Definition 2.18 their semantics. HeyLo formulas represent expectations $\mathbb{E}$ with support for arithmetic and Boolean expressions, addition, multiplication, conjunction, disjunction, and all four presented implications and negations. We also have *quantifiers* $\bigsqcap_x$ (infimum) and $\bigsqcup_x$ (supremum) that generalize the Boolean quantifier $\forall x$ and the Boolean quantifier $\exists x$, respectively.

In this thesis, we mostly use only one of the two dual groups of operators at a time for our encodings. We call the bottom left group in Definition 2.17 "down operators" and we call the bottom right group "up operators".

Definition 2.17. HeyLo syntax

Let $b \in \mathsf{BExp}$ be Boolean expressions and $a \in \mathsf{ArithExp}$ be arithmetic expressions. $x \in \mathsf{Vars}$ is a variable from the countably infinite set of variables Vars. The set HeyLo of formulas is given by the following grammar:

$$
\begin{array}{lll}
\varphi ::= a & \text{(arithmetic expressions)} & \mid \neg\varphi \quad \text{(negation)} \\
\mid \varphi + \varphi & \text{(addition)} & \mid {\sim}\varphi \quad \text{(co-negation)} \\
\mid \varphi \cdot \varphi & \text{(multiplication)} & \\
\mid [b] & \text{(Iverson bracket)} & \\
\mid ?(b) & \text{(Boolean embedding)} & \\
\mid \bigsqcap_x \varphi & \text{(infimum over } x\text{)} & \mid \bigsqcup_x \varphi \quad \text{(supremum over } x\text{)} \\
\mid \varphi \sqcap \varphi & \text{(meet)} & \mid \varphi \sqcup \varphi \quad \text{(join)} \\
\mid \varphi \to \varphi & \text{(implication)} & \mid \varphi \leftarrow \varphi \quad \text{(co-implication)} \\
\mid \varphi \searrow \varphi & \text{(hard implication)} & \mid \varphi \nwarrow \varphi \quad \text{(hard co-implication)}
\end{array}
$$

Definition 2.18. HeyLo semantics

Let ρ be a HeyLo formula. The *evaluation* $[\![\rho]\!] \in \mathbb{E}$ of ρ is defined inductively by:

$\rho \in \mathsf{HeyLo}$	$[\![\rho]\!]$	Recall $[\![\rho]\!](\sigma)$
$a \in \mathsf{ArithExp}$	$[\![a]\!]$	(Definition 2.16)
$\varphi + \psi$	$[\![\varphi]\!] + [\![\psi]\!]$	$[\![\varphi]\!](\sigma) + [\![\psi]\!](\sigma)$
$\varphi \cdot \psi$	$[\![\varphi]\!] \cdot [\![\psi]\!]$	$[\![\varphi]\!](\sigma) \cdot [\![\psi]\!](\sigma)$
$[b]$	$[b]$	$\begin{cases} 1, & \text{if } [\![b]\!](\sigma) \\ 0, & \text{else} \end{cases}$
$?(b)$	$?(b)$	$\begin{cases} \infty, & \text{if } [\![b]\!](\sigma) \\ 0, & \text{else} \end{cases}$
$\bigsqcap_x \varphi$	$\inf \{ [\![\varphi[x \mapsto v]]\!] \mid v \in \mathbb{Q}_{\geq 0} \}$	
$\varphi \sqcap \psi$	$[\![\varphi]\!] \sqcap [\![\psi]\!]$	$\begin{cases} [\![\varphi]\!](\sigma), & \text{if } [\![\varphi]\!](\sigma) \sqsubseteq [\![\psi]\!](\sigma) \\ [\![\psi]\!](\sigma), & \text{else} \end{cases}$
$\varphi \to \psi$	$[\![\varphi]\!] \to [\![\psi]\!]$	$\begin{cases} \infty, & \text{if } [\![\varphi]\!](\sigma) \sqsubseteq [\![\psi]\!](\sigma) \\ [\![\psi]\!](\sigma), & \text{else} \end{cases}$
$\varphi \searrow \psi$	$[\![\varphi]\!] \searrow [\![\psi]\!]$	$\begin{cases} \infty, & \text{if } [\![\varphi]\!](\sigma) \sqsubseteq [\![\psi]\!](\sigma) \\ 0, & \text{else} \end{cases}$
$\bigsqcup_x \varphi$	$\sup \{ [\![\varphi[x \mapsto v]]\!] \mid v \in \mathbb{Q}_{\geq 0} \}$	
$\varphi \sqcup \psi$	$[\![\varphi]\!] \sqcup [\![\psi]\!]$	$\begin{cases} [\![\varphi]\!](\sigma), & \text{if } [\![\varphi]\!](\sigma) \sqsupseteq [\![\psi]\!](\sigma) \\ [\![\psi]\!](\sigma), & \text{else} \end{cases}$
$\varphi \leftarrow \psi$	$[\![\varphi]\!] \leftarrow [\![\psi]\!]$	$\begin{cases} 0, & \text{if } [\![\varphi]\!](\sigma) \sqsupseteq [\![\psi]\!](\sigma) \\ [\![\psi]\!](\sigma), & \text{else} \end{cases}$
$\varphi \nwarrow \psi$	$[\![\varphi]\!] \nwarrow [\![\psi]\!]$	$\begin{cases} 0, & \text{if } [\![\varphi]\!](\sigma) \sqsupseteq [\![\psi]\!](\sigma) \\ \infty, & \text{else} \end{cases}$
$\neg\varphi$	$\neg[\![\varphi]\!]$	$\begin{cases} \infty, & \text{if } [\![\varphi]\!](\sigma) = 0 \\ 0, & \text{else} \end{cases}$
$\sim\varphi$	$\sim[\![\phi]\!]$	$\begin{cases} 0, & \text{if } [\![\varphi]\!](\sigma) = \infty \\ \infty, & \text{else} \end{cases}$

An arithmetic expression $a \in \mathsf{ArithExp}$ is a HeyLo formula. Because a variable $x \in \mathsf{Vars}$ is an arithmetic expression, x is also a HeyLo formula. The quantifiers $\bigsqcap_x$ (infimum) and $\bigsqcup_x$ (supremum) range over all possible values $\mathbb{Q}_{\geq 0}$ of x.

We also write $\bigsqcap_V \varphi$ and $\bigsqcup_V \varphi$ to quantify over a set of variables $V \subseteq \mathsf{Vars}$. Since the number of variables that occur in a formula φ is finite, this notation can be seen as syntactic sugar for the expansion $\bigsqcap_{x_1} \ldots \bigsqcap_{x_n}$ (or its dual) for all $x_1, \ldots, x_n \in V$ that occur in φ. Although the syntax of HeyLo does not include the constant $\infty \in \mathbb{R}^{\infty}_{\geq 0}$, we use this notation as syntactic sugar for $\bigsqcup_x x$ which evaluates to ∞.

In addition to the quantifiers, we also have binary limits: $\sqcap$ and $\sqcup$. We call them meet and join. Finally, all four implications $\to$, $\leftarrow$, $\searrow$, $\nwarrow$, and the negations $\neg$ and $\sim$ are included. Recall that the hard implications $\searrow$ and $\nwarrow$ are just syntactic sugar for the implications combined with negations.

The semantics of a HeyLo formula is given in Definition 2.18. An arithmetic expression $a \in \mathsf{ArithExp}$ is evaluated according to Definition 2.16. The semantics of most operators such as $\sqcap$ and $\to$ simply amount to the recursive evaluation with the eventual application of the corresponding operator on expectations $\mathbb{E}$. The limit operators, i.e. the infimum $\bigsqcap_x \varphi$ and the supremum $\bigsqcup_x \varphi$, evaluate to a limit on expectations where all possible values $v \in \mathbb{Q}_{\geq 0}$ are substituted for x in φ.

Example 2.19. HeyLo formula evaluation

Let $\varphi = \bigsqcap_x (x \to 0.5)$ be a HeyLo formula. For every state $\sigma \in \Sigma$, φ evaluates to

$$
\begin{aligned}
[\![\varphi]\!](\sigma) &= \inf \left\{ [\![x \to 0.5]\!](\sigma[x \mapsto v]) \;\middle|\; v \in \mathbb{Q}_{\geq 0} \right\} && \text{(HeyLo semantics of } \textstyle\bigsqcap_x\text{)} \\
&= \inf \left\{ [\![x]\!](\sigma[x \mapsto v]) \to [\![0.5]\!](\sigma[x \mapsto v]) \;\middle|\; v \in \mathbb{Q}_{\geq 0} \right\} && \text{(HeyLo semantics of } \to\text{)} \\
&= \inf \left\{ v \to 0.5 \;\middle|\; v \in \mathbb{Q}_{\geq 0} \right\} && \text{(HeyLo semantics)} \\
&= \inf \left\{ \begin{cases} \infty, & \text{if } v \sqsubseteq 0.5 \\ 0.5, & \text{else} \end{cases} \;\middle|\; v \in \mathbb{Q}_{\geq 0} \right\} && \text{(Definition 2.8)} \\
&= \inf \left\{ \infty \;\middle|\; v \in \mathbb{Q}_{\geq 0},\, v \sqsubseteq 0.5 \right\} \cup \left\{ 0.5 \;\middle|\; v \in \mathbb{Q}_{\geq 0},\, v \not\sqsubseteq 0.5 \right\} && \text{(case split)} \\
&= 0.5
\end{aligned}
$$

We arbitrarily choose the implication $\to$ as the canonical implication and define *validity* with respect to the truth value ∞ corresponding to $\to$.

Definition 2.20. Validity of a HeyLo formula

A HeyLo formula φ is considered *valid* iff

$$\forall \sigma \in \Sigma. \quad [\![\varphi]\!](\sigma) = \infty.$$

From a dual perspective, the constant formula 0 behaves like true with respect to the co-implication $\leftarrow$ (cf. Section 2.2.3). One might consider a formula that always evaluates to 0 *co-valid* or *up-valid*. This view might help when constructing formulas with up operators. However, we will avoid such terms due to the risk of confusion.

We will not need a notion of *(un-)satisfiability*. Indeed, such terminology might even suggest that validity of a formula φ can be checked by testing unsatisfiability of the negation of φ, as is the case in classical logic. Consider the HeyLo formula $\varphi = 0.5$. It is not valid, but the negation $\neg\varphi$, equivalent to 0, is also not valid.

Definition 2.21. Comparison of HeyLo formulas

Two HeyLo formulas φ, ψ are considered *equivalent*, denoted by $\varphi \equiv \psi$, if

$$\forall \sigma \in \Sigma. \quad [\![\varphi]\!](\sigma) = [\![\psi]\!](\sigma).$$

A formula φ *entails* a formula ψ, denoted by $\varphi \sqsubseteq \psi$, if

$$\forall \sigma \in \Sigma. \quad [\![\varphi]\!](\sigma) \sqsubseteq [\![\psi]\!](\sigma).$$

Two formulas are equivalent when they evaluate to the same value on every state. The entailment relation $\varphi \sqsubseteq \psi$ corresponds to the ordering given by the φ, ψ where $\varphi \to \psi$ evaluates to ∞:

$$\varphi \sqsubseteq \psi \quad \text{iff} \quad \varphi \to \psi \equiv \infty.$$

This a statement is closely related to the well-known *deduction theorem* for logics. Interestingly, among the three most well-known many-valued logic conditionals – Gödel, Łukasiewicz, and product conditionals – the Gödel implication is the only one which satisfies both directions of the statement [Pre10].

Example 2.22. Deduction theorem

Recall from Example 2.9 that the expectation $([x = 1] \cdot 0.6) \to 0.5$ evaluates to 0.5 in a state $\sigma \in \Sigma$ where $\sigma(x) = 1$ holds. Therefore, $([x = 1] \cdot 0.6) \to 0.5 \not\equiv \infty$. By the deduction theorem, $([x = 1] \cdot 0.6) \not\sqsubseteq 0.5$.

Dual to the deduction theorem for $\to$, the reversed entailment $\varphi \sqsupseteq \psi$ corresponds to the ordering given by the co-implication $\leftarrow$ where $\varphi \leftarrow \psi$ evaluates to 0:

$$\varphi \sqsupseteq \psi \quad \text{iff} \quad \varphi \leftarrow \psi \equiv 0.$$

Finally, we define substitution for HeyLo formulas. We write $\varphi[x \mapsto \psi]$ to mean the HeyLo formula φ where each free occurrence of x is replaced by the formula ψ. The set $\mathsf{free}(\varphi) \subseteq \mathsf{Vars}$ of *free variables* in φ is given by all variables occurring in φ, except that $\mathsf{free}(\bigsqcap_x \varphi) = \mathsf{free}(\varphi) \setminus \{ x \}$ and $\mathsf{free}(\bigsqcup_x \varphi) = \mathsf{free}(\varphi) \setminus \{ x \}$.

2.2.6. Expressivity of HeyLo

We propose HeyLo as a language to represent expectations $\mathbb{E}$ for the verification of probabilistic programs. To evaluate how useful HeyLo is, we look at what kinds of expectations can be expressed in HeyLo.

HeyLo is a superset of the assertion language for probabilistic programs by Batz et al. [Bat+21b]. Therefore, HeyLo can be used to represent every expectation that the language by Batz et al. can. Let us look at examples presented in [Bat+21b].

First of all, it is possible to represent a number of irrational functions in HeyLo, even though HeyLo's quantifiers only range over values in $\mathbb{Q}_{\geq 0}$. In addition to formulas with polynomials and irrational numbers, it is possible to encode harmonic numbers [Bat+21b, Section 12].

Example 2.23. HeyLo: Square root

The square root $\sqrt{x}$ of the program variable x is not always rational, e.g. $\sqrt{2} \notin \mathbb{Q}$. But HeyLo can represent the expectation that evaluates to $\sqrt{x}$ by a supremum over all values $y \in \mathbb{Q}_{\geq 0}$ such that $y^2 < x$:

$$\bigsqcup_{y} [y \cdot y < x] \cdot y.$$

Proof. For all states $\sigma \in \Sigma$,

$$\begin{aligned}
[\![\bigsqcup_{y} [y \cdot y < x] \cdot y]\!](\sigma) &= \sup \left\{ [y^2 < \sigma(x)] \cdot y \;\middle|\; y \in \mathbb{Q}_{\geq 0} \right\} && \text{(Definition 2.18)} \\
&= \sup \left\{ y \in \mathbb{Q}_{\geq 0} \;\middle|\; y^2 < \sigma(x) \right\} && ([y^2 < \sigma(x)] = 0 \text{ if } y^2 \not< \sigma(x)) \\
&= \sqrt{\sigma(x)} && \square
\end{aligned}$$

The central contribution in [Bat+21b] is that the presented language is what Batz et al. call *expressive*: If an expectation $X \in \mathbb{E}$ is expressible in the language, then the weakest pre-expectation $\mathsf{wp}[\![C]\!](X)$ of a pGCL program C is also syntactically expressible in the language. We introduce pGCL and the weakest pre-expectation calculus in Section 3.1 and Section 3.2. For now, $\mathsf{wp}[\![C]\!](X) \in \mathbb{E}$ is an expectation that maps an initial state $\sigma \in \Sigma$ to the expected value $\mathsf{wp}[\![C]\!](X)(\sigma)$ of X over the final states reached after starting in σ. When the language satisfies this property, the weakest pre-expectation calculus is *relatively complete* [Coo78].

Because HeyLo is a superset of the assertion language of [Bat+21b], we have a subset of HeyLo that is expressive, so that the weakest pre-expectation of the expectation represented by HeyLo formulas can be represented by a HeyLo formula as well.

In particular, this implies that HeyLo can represent probabilities of termination of a pGCL program, optionally restricted to termination in a state that satisfies some Boolean condition in BExp [Bat+21b, Section 12]. Even more generally, distributions over final states can be expressed by HeyLo.

In addition to the features of the relatively complete language by Batz et al., HeyLo can express *comparisons* of syntactically represented expectations through its implication operators. However, it is not clear whether our implication operators could be encoded into the language by Batz et al.

Finally, HeyLo can easily be extended with more features. For example, HeyLo currently uses the domain $\mathbb{Q}_{\geq 0}$ both for *logical variables* and *program variables* In contrast, our implementation *Caesar* supports program and logical variables with different types (cf. Section 4.3), including support for the use of variables over $\mathbb{R}^{\infty}_{\geq 0}$ and a native Boolean type.

2.3. HeyVL: An Intermediate Verification Language

HeyVL is an intermediate verification language (IVL) for probabilistic programs. Compared to the existing IVLs that we mentioned in Chapter 1, HeyVL features two extensions. First, HeyVL is designed for the verification of probabilistic programs and so supports quantitative reasoning via HeyLo and syntax for probabilistic assignments. Second, it supports reasoning about *upper bounds* of verification conditions, which are not supported in any of the existing IVLs that we looked at. In classical (Boolean) terms, that means existing IVLs only ever allow checks in the style of $\mathit{Pre} \Rightarrow \mathsf{vc}[\![S]\!](\mathit{Post})$ where vc computes the verification conditions with respect to a Boolean predicate *Post*; so only a check whether the predicate *Pre* implies the verification conditions.

For probabilistic verification problems, we also require the opposite direction. An interpretation of the opposite direction in Boolean terms is difficult, we discuss it in Chapter 5. HeyVL supports both lower and upper bounds with verification conditions written in HeyLo.

2.3.1. HeyVL

We now define our intermediate verification language for probabilistic programs, *HeyVL*. HeyVL supports random assignments and uses HeyLo as the assertion language. The `havoc`, `assume`, `assert`, and `negate` statements are available in both a `down` and `up` variant. There is also a `skip` statement that does nothing. We include two `compare` statements for the hard implications. Those are syntactic sugar for `assume` with `negate` statements, in the same way that hard implications are syntactic sugar for implications with negations

(cf. Definition 2.11). The set DistExp of distribution expressions for use in random assignments will be defined in Section 2.3.2.

Definition 2.24. HeyVL syntax

The set HeyVL of HeyVL statements is defined by the following grammar:

$$
\begin{array}{lll}
S ::= \texttt{skip} & \text{(no effect)} & \mid \texttt{down negate} \quad \text{(down negate)} \\
\mid x :\approx \mu & \text{(random assignment)} & \mid \texttt{up negate} \quad \text{(up negate)} \\
\mid S\,;\, S & \text{(sequential composition)} & \\
\mid \texttt{down havoc } x & \text{(down havoc)} & \mid \texttt{up havoc } x \quad \text{(up havoc)} \\
\mid \texttt{down assert } \varphi & \text{(down assert)} & \mid \texttt{up assert } \varphi \quad \text{(up assert)} \\
\mid \texttt{down assume } \varphi & \text{(down assume)} & \mid \texttt{up assume } \varphi \quad \text{(up assume)} \\
\mid \texttt{down compare } \varphi & \text{(down compare)} & \mid \texttt{up compare } \varphi \quad \text{(up compare)} \\
\mid \texttt{if } (\sqcap)\ \{S\}\ \texttt{else}\ \{S\} & \text{(demonic choice)} & \mid \texttt{if } (\sqcup)\ \{S\}\ \texttt{else}\ \{S\} \quad \text{(angelic choice)}
\end{array}
$$

where $x \in \mathsf{Vars}$ is a variable, $\mu \in \mathsf{DistExp}$ is a probability distribution expression, $b \in \mathsf{BExp}$ is a Boolean expression, and $\varphi \in \mathsf{HeyLo}$ is a HeyLo formula.

Just like in HeyLo, we allow quantification over a set of variables $V \subseteq \mathsf{Vars}$ in HeyVL by writing $\texttt{havoc}\ V$. We call the language generated by the two left groups of statements in Definition 2.24 *down fragment* of HeyVL and the language generated by the upper left and bottom right groups the *up fragment* of HeyVL. These fragments correspond to the lower bound and the upper bound versions, respectively, of the simple IVL presented in Section 2.1. More concretely, when restricted to only Boolean predicates with the embed function $?(b)$ and no probability distributions in assignments, the down fragment coincides with the simple IVL. There are no negation statements in classical IVLs, and therefore they are not included in either fragment. For most encodings we present in Chapter 3, working in one of the two fragments suffices.

The verification condition semantics of HeyVL are presented in Definition 2.25. They are very similar to that of the simple IVL (Section 2.1). In analogy to the weakest pre-condition transformer that this semantics is ultimately based on, we call the HeyLo formula $\varphi \in \mathsf{HeyLo}$ in $\mathsf{vc}[\![S]\!](\varphi)$ a *post-expectation*. Note that $\varphi \in \mathsf{HeyLo}$ is not technically an expectation, i.e. $\varphi \notin \mathbb{E}$, but φ represents an expectation. Similarly, we call $\mathsf{vc}[\![S]\!](\varphi) \in \mathsf{HeyLo}$ a *pre-expectation of S with respect to the post-expectation φ*.

Let $\varphi \in \mathsf{HeyLo}$ be the post-expectation of interest. The skip statement does nothing, so $\mathsf{vc}[\![\texttt{skip}]\!](\varphi)$ returns φ unmodified. We allow a *probabilistic* assignment $x :\approx \mu$ with corresponding semantics $E\colon \mathsf{Vars} \times \mathsf{DistExp} \times \mathsf{HeyLo} \to \mathsf{HeyLo}$. The function E maps

Definition 2.25. HeyVL semantics

Let $E\colon \mathsf{Vars} \times \mathsf{DistExp} \times \mathsf{HeyLo} \to \mathsf{HeyLo}$ be the semantics of the random assignment (Section 2.3.2). For a HeyVL statement S, the *verification conditions* $\mathsf{vc}[\![S]\!]\colon \mathsf{HeyLo} \to \mathsf{HeyLo}$ is defined inductively on S by:

S	$\mathsf{vc}[\![S]\!](\varphi)$
`skip`	φ
$x :\approx \mu$	$E(x, \mu, \varphi)$
$S_1\,;\, S_2$	$\mathsf{vc}[\![S_1]\!](\mathsf{vc}[\![S_2]\!](\varphi))$
down havoc x	$\bigsqcap_x \varphi$
down assert ψ	$\psi \sqcap \varphi$
down assume ψ	$\psi \to \varphi$
down compare ψ	$\psi \searrow \varphi$
if $(\sqcap)$ $\{S_1\}$ else $\{S_2\}$	$\mathsf{vc}[\![S_1]\!](\varphi) \sqcap \mathsf{vc}[\![S_2]\!](\varphi)$
up havoc x	$\bigsqcup_x \varphi$
up assert ψ	$\psi \sqcup \varphi$
up assume ψ	$\psi \leftarrow \varphi$
up compare ψ	$\psi \nwarrow \varphi$
if $(\sqcup)$ $\{S_1\}$ else $\{S_2\}$	$\mathsf{vc}[\![S_1]\!](\varphi) \sqcup \mathsf{vc}[\![S_2]\!](\varphi)$
down negate	$\neg\varphi$
up negate	$\sim\varphi$

Recall:

$$(X \to Y)(\sigma) = \begin{cases} \infty, & \text{if } X(\sigma) \sqsubseteq Y(\sigma) \\ Y(\sigma), & \text{else} \end{cases}$$

$$(X \searrow Y)(\sigma) = \begin{cases} \infty, & \text{if } X(\sigma) \sqsubseteq Y(\sigma) \\ 0, & \text{else} \end{cases}$$

$$(\neg X)(\sigma) = \begin{cases} \infty, & \text{if } X(\sigma) = 0 \\ 0, & \text{else} \end{cases}$$

$$(X \leftarrow Y)(\sigma) = \begin{cases} 0, & \text{if } X(\sigma) \sqsupseteq Y(\sigma) \\ Y(\sigma), & \text{else} \end{cases}$$

$$(X \nwarrow Y)(\sigma) = \begin{cases} 0, & \text{if } X(\sigma) \sqsupseteq Y(\sigma) \\ \infty, & \text{else} \end{cases}$$

$$(\sim X)(\sigma) = \begin{cases} 0, & \text{if } X(\sigma) = \infty \\ \infty, & \text{else} \end{cases}$$

a variable x, a distribution expression μ, and a post-expectation φ to corresponding verification conditions. The definition of function E is introduced in the next section.

The verification conditions of the sequential composition $S_1\,;\ S_2$ are obtained by first computing the vc of program S_2 with respect to φ, and then using that result to compute the vc of program S_1.

The semantics of the havoc, `assert`, `assume`, and `compare` statements map to their respective HeyLo operators $\bigsqcap$, $\sqcap$, $\to$, $\searrow$ (or their duals). HeyVL also has two `negate` statements, corresponding to the respective HeyLo negations.

Note that the `negate` statements are not syntactic sugar for `assume` statements. This is in contrast to HeyLo, where the negation $\neg\phi$ can be encoded by $\phi \to 0$ (cf. Section 2.2.3). In HeyVL, the left-hand side ψ of the generated implications $\mathsf{vc}[\![\texttt{down assume}\ \psi]\!](\phi) = \psi \to \varphi$ is always fixed to the HeyLo formula ψ in the program S, so `assume` statements cannot be used to generate negations.

Finally, the two non-deterministic choices `if` $(\sqcap)$ and `if` $(\sqcup)$ map to the point-wise minimum and maximum of the vc of each branch, respectively.

Definition 2.26. Verifying HeyVL programs

Let $S \in \mathsf{HeyVL}$ be a HeyVL program. We say S *verifies*, if

$$\mathsf{vc}[\![S]\!](\infty) \equiv \infty\ .$$

The HeyVL program S *verifies* if the verification conditions with respect to the post-expectation ∞ are equivalent to ∞. This corresponds to the definition of a "correct" simple IVL program (cf. Section 2.1), where the Boolean verification conditions $\mathsf{vc}[\![S]\!](\mathsf{true})$ of a simple IVL program S with respect to the post-condition true must be valid, i.e. equivalent to true.

Example 2.27. A HeyVL program that does not verify

Consider the following HeyVL program S:

$$S = \quad \texttt{down havoc}\ x\,;\ \texttt{down assume}\ x\,;\ \texttt{down assert}\ 0.5\ .$$

The vc semantics with respect to post-expectation ∞ evaluates to

$$\mathsf{vc}[\![S]\!](\infty) = \quad \bigsqcap_x (x \to (0.5 \sqcap \infty))\ . \qquad \text{(Definition 2.25)}$$

The above is equivalent to $\bigsqcap_x (x \to 0.5)$, which was the subject of Example 2.19. By Example 2.19, the formula is equivalent to 0.5. Because $0.5 \not\equiv \infty$, S does not verify.

2.3.2. Distribution Expressions

A *distribution expression* $\mu \in \mathsf{DistExp}$ occurs on the right-hand side of a random assignment $x :\approx \mu$ in HeyLo. To encode pGCL in Chapter 3, we require only the *Bernoulli distribution*. However, more distribution expressions can easily be added to the following definitions.

Definition 2.28. Distribution expressions

A *distribution expression* $\mu \in \mathsf{DistExp}$ follows the grammar

$$\begin{aligned} \mu \ &::= a && \text{(arithmetic expression)} \\ &\mid \mathsf{ber!}(p)\ , && \text{(Bernoulli distribution)} \end{aligned}$$

where $a \in \mathsf{ArithExp}$ and $p \in [0,1] \cap \mathbb{Q}$.

A distribution expression can be either an arithmetic expression (non-probabilistic), or a Bernoulli distribution expression $\mathsf{ber!}(p)$ where $p \in [0,1]$. The assignment $x :\approx \mathsf{ber!}(p)$ assigns the value 1 to x with probability p and the value 0 with probability $1-p$. To ensure that the semantics evaluates to a convex sum, we require that the probability p is a constant in $[0,1]$.

Definition 2.29. Distribution expression semantics

Let $x \in \mathsf{Vars}$, $\mu \in \mathsf{DistExp}$ and $\varphi \in \mathsf{HeyLo}$. The semantics for the random assignment, $E\colon \mathsf{Vars} \times \mathsf{DistExp} \times \mathsf{HeyLo} \to \mathsf{HeyLo}$, is given by:

μ	$E(x, \mu, \varphi)$
a	$\varphi[x \mapsto a]$
$\mathsf{ber!}(p)$	$p \cdot \varphi[x \mapsto 1] + (1-p) \cdot \varphi[x \mapsto 0]$

With this definition, $\mathsf{vc}[\![x :\approx a]\!](\varphi) = E(x, a, \varphi)$ evaluates to the post-expectation φ where x is replaced by the arithmetic expression a. A Bernoulli distribution in a random assignment results in a convex sum where x is set to 1 in φ with probability p and to 0 with the complementary probability $1-p$.

Note that the semantics of an assignment with the Bernoulli distribution is the only place so far where the operators for addition $+$ and multiplication $\cdot$ occur directly in the semantics of HeyVL (and not just in assertions). This suggests that the general constructions of HeyVL and HeyLo can be separated from the semantics of the assignment. We explore this idea further in the chapter on *abstraction*, Chapter 5.

2.3.3. There Is No Observe

The fact that the core HeyVL semantics do not mention any arithmetic also highlights an important difference between HeyVL's `assume` semantics and a probabilistic language construct commonly known as `observe` [Bor+11; Cla+13; Hur+14; Nor+14; Olm+18]. The statement `observe` b as defined by Olmedo et al. accepts a Boolean expression [Olm+18]. Roughly, if the expression b evaluates to false in the current state, the execution is considered *infeasible* and probability distribution of outputs is adjusted accordingly as if $\neg b$ was never in the distribution of initial states to begin with. For example, they present the following program:

$$\{x := 0\}\ [1/2]\ \{x := 1\};\ \texttt{observe}\ (x = 1).$$

It assigns zero to x with probability $\frac{1}{2}$ and one to x with probability $\frac{1}{2}$. Then $x = 1$ is observed. Under their *conditional weakest pre-expectation* semantics, the probability assigned to $x = 1$ is 1 and $x \neq 1$ has probability 0 after execution of this program.

In contrast, the similar HeyVL program which does a probabilistic assignment with the same probabilities and then does an `assume` with the Boolean embedding $?(x = 1)$,

$$S = \quad x :\approx \mathsf{ber}!(1/2);\ \texttt{assume}\ ?(x = 1),$$

is different. The vc of S with respect to post-expectation x evaluates to:[Fixed]

$$\begin{aligned}
\mathsf{vc}[\![S]\!](x) &= \frac{1}{2} \cdot (?(1 = 1) \to 1) + \frac{1}{2} \cdot (?(0 = 1) \to 0) \\
&\equiv \frac{1}{2} \cdot (\infty \to 1) + \frac{1}{2} \cdot (0 \to 0) \\
&\equiv \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot \infty \\
&\equiv \infty
\end{aligned}$$

This is a completely different result with a completely different interpretation than with the `observe` statement. The HeyVL program S can be interpreted as a random assignment, followed by a query whether $?(x = 1)$ is a lower bound to the post-expectation x. In one branch this is the case, and so the truth value ∞ is returned and the result of $\mathsf{vc}[\![S]\!](x)$ evaluates to the constant value ∞.

2.3.4. Healthiness Conditions

There are several properties that many predicate/expectation transformers satisfy. They are referred to as *healthiness conditions* [MM05; Kei15; Hin+16] or *homomorphism properties* in [BvW98]. In general, our vc transformer does not satisfy many of the well-known

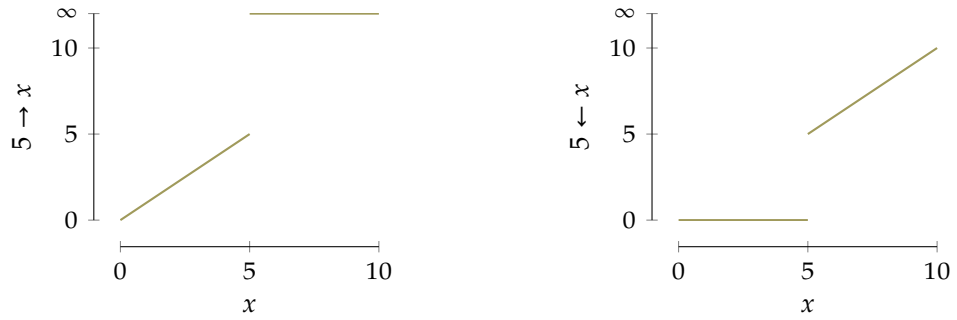

Figure 2.30.: The values of $5 \to x$ and $5 \leftarrow x$ for $x \in [0, 10]$.

healthiness conditions. For example, in contrast to the verification condition transformer for the simple IVL and to weakest pre-condition transformers both for classical and non-probabilistic programs, our verification condition transformer is not necessarily monotonic. However, both the down and up fragments are monotonic, that is, vc is monotonic on all but the negate statements.

In particular, the monotonicity of the semantics of both the down assume and up assume statements may be somewhat surprising. Figure 2.30 contains plots with the values of $\to$ and $\leftarrow$ with a fixed first argument. The plots show the values of $5 \to x$ and $5 \leftarrow x$ for $x \in [0, 10]$. These correspond to $\mathsf{vc}[\![\texttt{down assume } 5]\!](x)$ and $\mathsf{vc}[\![\texttt{up assume } 5]\!](x)$. It is clear that both $\to$ and $\leftarrow$ are monotonic in the second argument.

Theorem 2.31. Monotone fragment of HeyVL

The *monotone fragment* $\mathsf{HeyVL}_{\mathsf{mon}}$ of HeyVL is given by the following grammar:

$$
\begin{aligned}
S \ ::= \ & \texttt{skip} \mid x :\approx \mu \mid S_1 \,;\, S_2 \\
& \mid \texttt{down havoc } x \mid \texttt{down assert } \psi \mid \texttt{down assume } \psi \mid \texttt{down compare } \psi \\
& \mid \texttt{up havoc } x \mid \texttt{up assert } \psi \mid \texttt{up assume } \psi \mid \texttt{up compare } \psi \\
& \mid \texttt{if } (\sqcap)\ \{S\}\ \texttt{else}\ \{S\} \mid \texttt{if } (\sqcup)\ \{S\}\ \texttt{else}\ \{S\}\,.
\end{aligned}
$$

For each $\varphi, \psi \in \mathsf{HeyLo}$ where $\varphi \sqsubseteq \psi$, it holds that:

$$\forall S \in \mathsf{HeyVL}_{\mathsf{mon}}. \quad \mathsf{vc}[\![S]\!](\varphi) \sqsubseteq \mathsf{vc}[\![S]\!](\psi).$$

The proof is provided in Appendix A on page 118.

Continuity is referred to as "perhaps one of the most fundamental properties of expectation transformers" by [Kam19, Section 4.2.1], because of the relation of continuity to the semantics of loops. As not even monotonicity holds for all HeyVL programs S, it is not

surprising that (co-)continuity does not hold for all HeyVL programs S either.

$$\mathsf{vc}[\![S]\!](\bigsqcup_x \varphi) \overset{\lightning}{\equiv} \bigsqcup_x \mathsf{vc}[\![S]\!](\varphi) \qquad \text{(co-continuity)}$$

$$\mathsf{vc}[\![S]\!](\bigsqcap_x \varphi) \overset{\lightning}{\equiv} \bigsqcap_x \mathsf{vc}[\![S]\!](\varphi). \qquad \text{(continuity)}$$

For our purposes, especially in the context of automated verification in Chapter 5, we are more interested in a slightly tangential property: *quantifier elimination* for the verification conditions of a HeyVL program. We investigate quantifier elimination and identification of a fragment of HeyVL where quantifier elimination is possible in detail in Section 4.2. Out of these results, (co-)continuous fragments can be derived.

The *strictness* property was called "Law of the Excluded Miracle" by Dijkstra [Dij75]. In the context of our verification condition calculus, this property reads $\mathsf{vc}[\![S]\!](0) \overset{\lightning}{\equiv} 0$. As we have introduced the `assume` statements as "magical", it is not a surprise this property does not hold in general either.

There is a more general property than strictness called *feasibility*, defined by McIver and Morgan [MM05]. Feasibility implies strictness. By excluding implications and negations, as well as the `compare` statements that are built on these two from the `down` and `up` fragments, we obtain the *feasible* and *co-feasible* fragments of HeyVL. Somewhat surprisingly, the `up assume` statement can be added to the feasible fragment and the `down assume` statement can be added to the co-feasible fragment.

Theorem 2.32. (Co-)feasible fragments of HeyVL

The *feasible fragment* $\mathsf{HeyVL}_{\mathsf{feas}}$ of HeyVL is given by the following grammar:

$$\begin{aligned} S \;::=\; & \texttt{skip} \mid x :\approx \mu \mid S_1 \,;\, S_2 \\ & \mid \texttt{down havoc } x \mid \texttt{down assert } \psi \\ & \mid \texttt{up assume } \psi \\ & \mid \texttt{if } (\sqcap)\ \{S\}\ \texttt{else}\ \{S\} \mid \texttt{if } (\sqcup)\ \{S\}\ \texttt{else}\ \{S\}\,. \end{aligned}$$

The *co-feasible fragment* $\mathsf{HeyVL}_{\mathsf{cof}}$ of HeyVL is given by the following grammar:

$$\begin{aligned} S \;::=\; & \texttt{skip} \mid x :\approx \mu \mid S_1 \,;\, S_2 \\ & \mid \texttt{up havoc } x \mid \texttt{up assert } \psi \\ & \mid \texttt{down assume } \psi \\ & \mid \texttt{if } (\sqcap)\ \{S\}\ \texttt{else}\ \{S\} \mid \texttt{if } (\sqcup)\ \{S\}\ \texttt{else}\ \{S\}\,. \end{aligned}$$

For each $\varphi \in \mathsf{HeyLo}$ and $r \in \mathbb{R}^{\infty}_{\geq 0}$, it holds that:

$$\begin{aligned} \forall S \in \mathsf{HeyVL}_{\mathsf{feas}}.\quad \varphi \sqsubseteq r \quad &\Rightarrow \quad \mathsf{vc}[\![S]\!](\varphi) \sqsubseteq r, \\ \forall S \in \mathsf{HeyVL}_{\mathsf{cof}}.\quad r \sqsubseteq \varphi \quad &\Rightarrow \quad r \sqsubseteq \mathsf{vc}[\![S]\!](\varphi). \end{aligned}$$

Intuitively, the verification conditions of HeyVL programs in the feasible fragments cannot exceed the maximum value $r \in \mathbb{R}^{\infty}_{\geq 0}$ of the provided post-expectation $\varphi \in \mathsf{HeyLo}$. Dually, programs in the co-feasible fragment preserve at least r in their verification conditions.

The proof for Theorem 2.32 is provided in Appendix A on page 120.

2.3.5. The Triple Intuition

Before we start to actually write HeyVL programs in the next sections, let us review the intuition from the HeyLo section *A Relational View* (Section 2.2.4). One can apply the deconstructions of entailment relations between HeyLo formulas to understand the semantics of (some) HeyVL programs. We call this the *triple intuition* because it applies to the triples (ρ, S, ψ) where $\rho, \psi \in \mathsf{HeyLo}$ and $S \in \mathsf{HeyVL}$. These triples represent either

$$\rho \sqsubseteq \mathsf{vc}[\![S]\!](\psi) \quad \text{or} \quad \mathsf{vc}[\![S]\!](\psi) \sqsubseteq \rho$$

and we deconstruct by the semantics of the HeyVL program S.

Theorem 2.33. Entailments of verification conditions
Let $\varphi, \psi, \rho \in \mathsf{HeyLo}$ such that the variable x does not occur in ρ. The following equivalences hold for HeyVL programs S.

S	$\rho \sqsubseteq \mathsf{vc}[\![S]\!](\psi)$	S	$\mathsf{vc}[\![S]\!](\psi) \sqsubseteq \rho$
down assume 0	true	up assert ∞	$\rho \equiv \infty$
down assert 0	$\rho \equiv 0$	up assume ∞	true
down assert φ	$\rho \sqsubseteq \varphi \wedge \rho \sqsubseteq \psi$	down assert φ	$\varphi \sqsubseteq \rho \vee \psi \sqsubseteq \rho$
up assert φ	$\rho \sqsubseteq \varphi \vee \rho \sqsubseteq \psi$	up assert φ	$\varphi \sqsubseteq \rho \wedge \psi \sqsubseteq \rho$
down assume φ	$\varphi \sqsubseteq \psi \vee \rho \sqsubseteq \psi$	up assume φ	$\psi \sqsubseteq \varphi \vee \psi \sqsubseteq \rho$
down compare φ	$\varphi \sqsubseteq \psi \vee \rho \equiv 0$	up compare φ	$\psi \sqsubseteq x \vee \rho \equiv \infty$
down havoc x	$\forall v \in \mathbb{Q}_{\geq 0}.\ \rho \sqsubseteq \psi[x \mapsto v]$	up havoc x	$\forall v \in \mathbb{Q}_{\geq 0}.\ \psi[x \mapsto v] \sqsubseteq \rho$

Proof. Immediate from the definition of vc and Theorem 2.12. □

Both $\mathsf{vc}[\![\text{down assume } 0]\!](\psi)$ and $\mathsf{vc}[\![\text{up assert } \infty]\!](\psi)$ always evaluate to ∞. Similarly, $\mathsf{vc}[\![\text{down assert } 0]\!](\psi)$ and $\mathsf{vc}[\![\text{up assume } \infty]\!](\psi)$ always evaluate to 0.

So when is $\rho \sqsubseteq \mathsf{vc}[\![S]\!](\psi)$ true? For this lower bound perspective, we say *S verifies* with respect to pre-expectation $\rho \in \mathsf{HeyLo}$ and post-expectation $\psi \in \mathsf{HeyLo}$.

The program down assume 0 corresponds to the classical assume false and always verifies.

The assertion that always evaluates to zero, down assert 0, only verifies in case the pre-expectation is 0 as well. In general, the assertion down assert φ verifies if the pre-expectation ρ entails both the assertion φ and the post-expectation ψ. For lower bounds, up assert φ requires only one of the above entailments.

The general assumption down assume φ verifies if either the assumption φ entails the post-expectation ψ or the pre-expectation ρ already entails the post-expectation ψ.

The compare statement requires that the assumption φ entails the post-expectation ψ or that the pre-expectation ρ is already 0. Finally, the down havoc x statement is valid if the pre-expectation ρ entails $\psi[x \mapsto v]$ for all values $v \in \mathbb{Q}_{\geq 0}$.

The same ideas apply dually for the $\mathsf{vc}[\![S]\!](\psi) \sqsubseteq \rho$ cases. For example, the assertion up assert φ verifies if the pre-expectation ρ entails both the assertion φ and the post-expectation ψ.

2.3.6. Verifying Implementations

At the core, our deductive verifier *Caesar* (cf. Section 4.3) only considers a single HeyVL program S and checks whether it *verifies*, that is, whether $\mathsf{vc}[\![S]\!](\infty) \equiv \infty$ holds. That means the verifier only requires a single input, a HeyVL program, without any additional inputs like pre- or post-expectations to check against. But the questions that we are usually interested in are about whether one of the following is true:

$$\text{Does it hold that } \varphi \sqsubseteq \mathsf{vc}[\![S]\!](\psi) \quad \text{or} \quad \mathsf{vc}[\![S]\!](\psi) \sqsubseteq \varphi \,?$$

In this section, we develop encodings $S' \in \mathsf{HeyVL}$ that verify $(\mathsf{vc}[\![S']\!](\infty) \equiv \infty)$ if and only if the HeyVL program *S implements* the specification $\varphi \sqsubseteq \mathsf{vc}[\![S]\!](\psi)$ or $\mathsf{vc}[\![S]\!](\psi) \sqsubseteq \varphi$. The encoding for lower bounds is pretty simple. The encoding for upper bounds is mostly dual, but requires an extra step.

In classical weakest pre-conditions, the proposition $\mathit{Pre} \Rightarrow \mathsf{wp}[\![S]\!](\mathit{Post})$ is valid whenever all executions of the program S that start in a state that satisfies the predicate *Pre* terminate in a state that satisfies *Post*. In other words, we *assume Pre* holds at the start of the program, and then *assert* that *Post* holds after the execution of S. And we can write just such a sequence in the simple IVL (cf. Section 2.1):

```
assume Pre;
S;
assert Post
```

And indeed, by the definition of vc for the simple IVL (Figure 2.1), the above program S' verifies $(\mathsf{vc}[\![S']\!](\mathsf{true}) = \mathsf{true})$ if and only if $\mathit{Pre} \Rightarrow \mathsf{vc}[\![S]\!](\mathit{Post})$ holds.

We can write a matching program in HeyVL. The following HeyVL program S'_{down} verifies $(\mathsf{vc}[\![S'_{\mathsf{down}}]\!](\infty) \equiv \infty)$ iff $\varphi \sqsubseteq \mathsf{vc}[\![S]\!](\psi)$ is a valid:

$$\begin{aligned} S'_{\mathsf{down}} = \quad & \mathsf{down\ assume}\ \varphi; \\ & S; \\ & \mathsf{down\ assert}\ \psi \end{aligned}$$

But what about upper bounds of vc? What does the dual of the previous encoding do?

$$\begin{aligned} S'_{\mathsf{up}} = \quad & \mathsf{up\ assume}\ \varphi; \\ & S; \\ & \mathsf{up\ assert}\ \psi \end{aligned}$$

It turns out that this program S'_{up} satisfies $\mathsf{vc}[\![S'_{\mathsf{up}}]\!](0) \equiv 0$ iff $\mathsf{vc}[\![S]\!](\psi) \sqsubseteq \varphi$. This is not too surprising: as the dual program to the lower bound encoding, it satisfies exactly the dual of the above, which is $\infty \sqsubseteq \mathsf{vc}[\![S'_{\mathsf{down}}]\!](\infty)$. Since 0 can be seen as the "truth value" for upper bounds, we say that S'_{up} *co-verifies* iff $\mathsf{vc}[\![S'_{\mathsf{up}}]\!](0) \equiv 0$.

Now, how do we go from *co-verifying* $(\mathsf{vc}[\![S'_{\mathsf{up}}]\!](0) \equiv 0)$ to *verifying* $(\mathsf{vc}[\![S'']\!](\infty) \equiv \infty)$? This is where we use the `negate` statements.

$$\begin{aligned} S'' = \quad & \mathsf{down\ negate}; \\ & \mathsf{up\ assume}\ \varphi; \\ & S; \\ & \mathsf{up\ assert}\ \psi; \\ & \mathsf{up\ negate} \end{aligned}$$

The program S'' *verifies* $(\mathsf{vc}[\![S'']\!](\infty) \equiv \infty)$ iff $\mathsf{vc}[\![S]\!](\psi) \sqsubseteq \varphi$ holds.

We now prove correctness of these encodings formally. We begin with `assume-assert`.

Theorem 2.34. HeyVL: Assume and assert

Let $\varphi, \psi \in \mathsf{HeyLo}$ and S be a HeyVL program.

$$\begin{aligned} &\mathsf{vc}[\![\mathsf{down\ assume}\ \varphi;\ S;\ \mathsf{down\ assert}\ \psi]\!](\infty) \equiv \infty \quad \text{iff} \quad \varphi \sqsubseteq \mathsf{vc}[\![S]\!](\psi) \\ &\mathsf{vc}[\![\mathsf{up\ assume}\ \varphi;\ S;\ \mathsf{up\ assert}\ \psi]\!](0) \equiv 0 \quad \text{iff} \quad \varphi \sqsupseteq \mathsf{vc}[\![S]\!](\psi) \end{aligned}$$

Proof. Let $\varphi, \psi \in \mathsf{HeyLo}$ and S be a HeyVL program.

$$\begin{aligned}
&\mathsf{vc}[\![\texttt{down assume } \varphi;\ S;\ \texttt{down assert } \psi]\!](\infty) \\
&= \mathsf{vc}[\![\texttt{down assume } \varphi]\!](\mathsf{vc}[\![S]\!](\mathsf{vc}[\![\texttt{down assert } \psi]\!](\infty))) && \text{(definition of ;)} \\
&= \mathsf{vc}[\![\texttt{down assume } \varphi]\!](\mathsf{vc}[\![S]\!](\psi \sqcap \infty)) && \text{(definition of assert)} \\
&= \mathsf{vc}[\![\texttt{down assume } \varphi]\!](\mathsf{vc}[\![S]\!](\psi)) && (\psi \sqcap \infty \equiv \psi) \\
&= \varphi \to \mathsf{vc}[\![S]\!](\psi) && \text{(definition of assume)}
\end{aligned}$$

Now, for every $\sigma \in \Sigma$:

$$\begin{aligned}
&[\![\varphi \to \mathsf{vc}[\![S]\!](\psi)]\!](\sigma) \\
&= \begin{cases} \infty, & \text{if } [\![\varphi]\!](\sigma) \sqsubseteq \mathsf{vc}[\![S]\!](\psi)(\sigma) \\ [\![\psi]\!](\sigma), & \text{else} \end{cases} && \text{(definition of } \to\text{)}
\end{aligned}$$

Therefore $\varphi \to \mathsf{vc}[\![S]\!](\psi) \equiv \infty$ iff $\varphi \sqsubseteq \psi$. The up statement is completely dual. □

Note that it would be equally possible to use compare statements instead of assume statements for Theorem 2.34. Example 2.35 illustrates a case where using assume statements is advantageous: when two assume statements are used in sequence. Sequencing assume statements is often done in classical IVLs.

Example 2.35. Verifying an implementation with two assumptions

Consider the following HeyVL program that assigns 5 to x:

$$S = \quad x :\approx 5\,.$$

It satisfies $0.5 \sqsubseteq \mathsf{vc}[\![S]\!](0.1 \cdot x)$. On the other hand, $1 \not\sqsubseteq \mathsf{vc}[\![S]\!](0.1 \cdot x)$. Still, the following HeyVL program S' verifies:

$$\texttt{down assume } 0.5;\ \texttt{down assume } 1;\ x :\approx 5;\ \texttt{assert } 0.1 \cdot x$$

$$\begin{aligned}
\mathsf{vc}[\![S']\!](\infty) &= 0.5 \to (1 \to 0.1 \cdot 5) && \text{(Definition 2.25)} \\
&\equiv 0.5 \to 0.5 \quad \equiv \infty && \text{(definition of } \to\text{)}
\end{aligned}$$

The same program, but using compare, does not verify. Denote this modified program by S''. Then,

$$\begin{aligned}
\mathsf{vc}[\![S'']\!](\infty) &= 0.5 \searrow (1 \searrow 0.1 \cdot 5) && \text{(Definition 2.25)} \\
&\equiv 0.5 \searrow 0 \quad \equiv 0\,. && \text{(definition of } \searrow\text{)}
\end{aligned}$$

To complete the picture for the encoding for upper bounds, we need to go from *verification* to *co-verification* with a specific combination of negate statements.

Theorem 2.36. HeyVL: Co-verification
Let S be a HeyVL program.

$$\mathsf{vc}[\![\texttt{down negate; } S\texttt{; up negate}]\!](\infty) \equiv \infty \quad \text{iff} \quad \mathsf{vc}[\![S]\!](0) \equiv 0.$$

Proof. Let S be a HeyVL program and $\sigma \in \Sigma$. Then,

$$\begin{aligned}
&\mathsf{vc}[\![\texttt{down negate; } S\texttt{; up negate}]\!](\infty)(\sigma) \\
&= \mathsf{vc}[\![\texttt{down negate}]\!](\mathsf{vc}[\![S]\!](\mathsf{vc}[\![\texttt{up negate}]\!](\infty)))(\sigma) && \text{(definition of ;)} \\
&= \mathsf{vc}[\![\texttt{down negate}]\!](\mathsf{vc}[\![S]\!](\sim\infty))(\sigma) && \text{(definition of negate)} \\
&= \mathsf{vc}[\![\texttt{down negate}]\!](\mathsf{vc}[\![S]\!](0))(\sigma) && (\sim\infty \equiv 0) \\
&= \begin{cases} \infty, & \text{if } \mathsf{vc}[\![S]\!](0)(\sigma) = 0 \\ 0, & \text{else} \end{cases} && \text{(definition of negate)}
\end{aligned}$$

So, $\mathsf{vc}[\![\texttt{down negate; } S\texttt{; up negate}]\!](\infty) \equiv \infty$ holds iff $\mathsf{vc}[\![S]\!](0) \equiv 0$ holds. □

Definition 2.37 contains the full HeyVL encodings S' to check the validity $\phi \sqsubseteq \mathsf{vc}[\![S]\!](\psi)$ or $\mathsf{vc}[\![S]\!](\psi) \sqsubseteq \phi$.

Definition 2.37. HeyVL: Verification encodings
Let $\varphi,\ \psi \in$ HeyLo and $S \in$ HeyVL.

$\underline{impl_{\mathsf{down}}(\varphi,\ S,\ \psi)\text{:}}$

```
down assume φ;
S;
down assert ψ
```

$\underline{impl_{\mathsf{up}}(\varphi,\ S,\ \psi)\text{:}}$

```
down negate;
up assume φ;
S;
up assert ψ;
up negate
```

Correctness of these encodings immediately follows from Theorems 2.34 and 2.36.

Theorem 2.38. HeyVL: Verification encodings
Let $\varphi, \psi \in$ HeyLo and S be a HeyVL program.

$$\begin{aligned}
\mathsf{vc}[\![impl_{\mathsf{down}}(\varphi,\ S,\ \psi)]\!](\infty) \equiv \infty \quad &\text{iff} \quad \varphi \sqsubseteq \mathsf{vc}[\![S]\!](\psi) \\
\mathsf{vc}[\![impl_{\mathsf{up}}(\varphi,\ S,\ \psi)]\!](\infty) \equiv \infty \quad &\text{iff} \quad \varphi \sqsupseteq \mathsf{vc}[\![S]\!](\psi)
\end{aligned}$$

2.3.7. Encoding Specifications

We now know how to verify properties of HeyVL programs by encoding validity of $\varphi \sqsubseteq \mathsf{vc}[\![S]\!](\psi)$ and $\mathsf{vc}[\![S]\!](\psi) \sqsubseteq \varphi$. Assume now that we have already shown validity of $\varphi \sqsubseteq \mathsf{vc}[\![S]\!](\psi)$ for a potentially complex HeyVL program S. Can we use this knowledge in another HeyVL program? Ideally, we would avoid copy-pasting the implementation of S. Instead, we will encode the *specification* of S with just three HeyVL statements: `assert`, `havoc`, and `assume`.

Consider the HeyVL program S' that consists of some part S_1, S, and S_2:

$$S' = \quad S_1 ; S ; S_2 .$$

We know that $\varphi \sqsubseteq \mathsf{vc}[\![S]\!](\psi)$ holds. Recall the meaning of $\varphi \Rightarrow \mathsf{vc}[\![S]\!](\psi)$ in classical terms: When the pre-condition φ holds before the execution of S, we know that ψ holds after executing S. Other than ψ, we know nothing about all other variables because S could have invalidated all other facts from before its execution. In the simple IVL (Section 2.1), we could write this:

$$S_1 ; \texttt{ assert } \varphi ; \texttt{ havoc Vars; assume } \psi ; S_2 .$$

We execute S_1, then assert that φ holds. To represent the execution of S, we `havoc` all variables. After that, we assume ψ holds and execute S_2.

In HeyVL, we can write a similar program. We need to use the hard implication `down compare` instead of `down assume`. There is also a completely dual version for upper bounds.

Definition 2.39. HeyVL: Specification encodings

Let $\varphi \in$ HeyLo and $\psi \in$ HeyLo.

$\underline{spec_{\mathsf{down}}(\varphi,\ \psi)}$:

```
down assert φ;
down havoc Vars;
down compare ψ
```

$\underline{spec_{\mathsf{up}}(\varphi,\ \psi)}$:

```
up assert φ;
up havoc Vars;
up compare ψ
```

As desired, these encodings evaluate to φ in the current state σ if ψ is a lower/upper bound to the post-expectation ρ.

Lemma 2.40. HeyVL: Semantics of specification encodings
Let $\varphi,\ \psi \in \mathsf{HeyLo}$, $S \in \mathsf{HeyVL}$, and $\sigma \in \Sigma$. For all $\rho \in \mathsf{HeyLo}$:

$$\mathsf{vc}[\![\mathit{spec}_{\mathsf{down}}(\varphi,\ \psi)]\!](\rho)(\sigma) = \begin{cases} [\![\varphi]\!](\sigma), & \text{if } \psi \sqsubseteq \rho \\ 0, & \text{else} \end{cases}$$

$$\mathsf{vc}[\![\mathit{spec}_{\mathsf{up}}(\varphi,\ \psi)]\!](\rho)(\sigma) = \begin{cases} [\![\varphi]\!](\sigma), & \text{if } \psi \sqsupseteq \rho \\ \infty, & \text{else} \end{cases}$$

Proof. Let $\varphi, \psi, \rho \in \mathsf{HeyLo}$ and $\sigma \in \Sigma$.

$$\begin{aligned}
&\mathsf{vc}[\![\mathit{spec}_{\mathsf{down}}(\varphi,\ \psi)]\!](\rho)(\sigma) \\
&= \mathsf{vc}[\![\texttt{down assert } \varphi\texttt{; down havoc Vars; down compare } \psi]\!](\rho)(\sigma) && \text{(definition of } \mathit{spec}_{\mathsf{down}}\text{)} \\
&= \mathsf{vc}[\![\texttt{down assert } \varphi]\!](\mathsf{vc}[\![\texttt{down havoc Vars}]\!](\mathsf{vc}[\![\texttt{down compare } \psi]\!](\rho)))(\sigma) \\
& && \text{(definition of ;)} \\
&= [\![\varphi]\!](\sigma) \sqcap \mathsf{vc}[\![\texttt{down havoc Vars}]\!](\mathsf{vc}[\![\texttt{down compare } \psi]\!](\rho))(\sigma) && \text{(definition of down assert)} \\
&= [\![\varphi]\!](\sigma) \sqcap \inf\{\ \mathsf{vc}[\![\texttt{down compare } \psi]\!](\rho)(\sigma') \mid \sigma' \in \Sigma\ \} && \text{(definition of down havoc)} \\
&= [\![\varphi]\!](\sigma) \sqcap \inf\left\{ \left\{ \begin{array}{ll} \infty, & \text{if } [\![\psi]\!](\sigma') \leq [\![\rho]\!](\sigma') \\ 0, & \text{else} \end{array} \right. \middle|\ \sigma' \in \Sigma \right\} && \text{(definition of down compare)} \\
&= [\![\varphi]\!](\sigma) \sqcap \inf\{\ \infty \mid \sigma' \in \Sigma,\ [\![\psi]\!](\sigma') \leq [\![\rho]\!](\sigma')\ \} \cup \{\ 0 \mid \sigma' \in \Sigma,\ [\![\psi]\!](\sigma') \not\leq [\![\rho]\!](\sigma')\ \} \\
& && \text{(case split)} \\
&= [\![\varphi]\!](\sigma) \sqcap \begin{cases} \infty, & \text{if } \forall \sigma' \in \Sigma.\ [\![\psi]\!](\sigma') \leq [\![\rho]\!](\sigma') \\ 0, & \text{else} \end{cases} \\
&= [\![\varphi]\!](\sigma) \sqcap \begin{cases} \infty, & \text{if } \psi \sqsubseteq \rho \\ 0, & \text{else} \end{cases} && \text{(definition of } \sqsubseteq\text{)} \\
&= \begin{cases} [\![\varphi]\!](\sigma) \sqcap \infty, & \text{if } \psi \sqsubseteq \rho \\ [\![\varphi]\!](\sigma) \sqcap 0, & \text{else} \end{cases} && \text{(rewriting)} \\
&= \begin{cases} [\![\varphi]\!](\sigma), & \text{if } \psi \sqsubseteq \rho \\ 0, & \text{else} \end{cases} && (\varphi \sqcap \infty \equiv \varphi,\ \varphi \sqcap 0 \equiv 0)
\end{aligned}$$

The proof for upper bounds is dual and omitted here. □

For lower bounds, we want the encoding $\mathit{spec}_{\mathsf{down}}(\varphi,\ \psi)$ to always *under-approximate* the true verification conditions of S where $\varphi \sqsubseteq \mathsf{vc}[\![S]\!](\psi)$, i.e. $\mathsf{vc}[\![\mathit{spec}_{\mathsf{down}}(\varphi,\ \psi)]\!](\rho) \sqsubseteq \mathsf{vc}[\![S]\!](\rho)$ for all $\rho \in \mathsf{HeyLo}$. The verification condition semantics of S need to satisfy monotonicity (cf. Theorem 2.31), i.e.

$$\psi \sqsubseteq \rho \quad \text{implies} \quad \mathsf{vc}[\![S]\!](\psi) \sqsubseteq \mathsf{vc}[\![S]\!](\rho)\ .$$

Recall that the monotone fragment $\mathsf{HeyVL}_{\mathsf{mon}}$ includes HeyVL programs with all but the negation statements.

Theorem 2.41. HeyVL: Specification encodings are approximations

Let $S \in \mathsf{HeyVL}_{\mathsf{mon}}$ be a HeyVL program from the monotone fragment and $\varphi, \psi \in \mathsf{HeyLo}$. Then:

$$\varphi \sqsubseteq \mathsf{vc}[\![S]\!](\psi) \quad \text{implies} \quad \forall \rho \in \mathsf{HeyLo}.\ \mathsf{vc}[\![\mathit{spec}_{\mathsf{down}}(\varphi,\ \psi)]\!](\rho) \sqsubseteq \mathsf{vc}[\![S]\!](\rho)$$
$$\varphi \sqsupseteq \mathsf{vc}[\![S]\!](\psi) \quad \text{implies} \quad \forall \rho \in \mathsf{HeyLo}.\ \mathsf{vc}[\![\mathit{spec}_{\mathsf{up}}(\varphi,\ \psi)]\!](\rho) \sqsupseteq \mathsf{vc}[\![S]\!](\rho)$$

Proof. Let $S \in \mathsf{HeyVL}_{\mathsf{mon}}$ be a HeyVL program from the monotone fragment and $\varphi, \psi \in \mathsf{HeyLo}$. Assume $\varphi \sqsubseteq \mathsf{vc}[\![S]\!](\psi)$ holds. Let $\rho \in \mathsf{HeyLo}$ and $\sigma \in \Sigma$. By Lemma 2.40,

$$\mathsf{vc}[\![\mathit{spec}_{\mathsf{down}}(\varphi,\ \psi)]\!](\rho)(\sigma) = \begin{cases} [\![\varphi]\!](\sigma), & \text{if } \psi \sqsubseteq \rho \\ 0, & \text{else} \end{cases}$$

In case $[\![\varphi]\!](\sigma)$ is returned, we have $\psi \sqsubseteq \rho$ and by monotonicity of $\mathsf{vc}[\![S]\!]$:

$$[\![\varphi]\!](\sigma) \leq \mathsf{vc}[\![S]\!](\psi)(\sigma) \leq \mathsf{vc}[\![S]\!](\rho)(\sigma).$$

In the other case, $\mathsf{vc}[\![\mathit{spec}_{\mathsf{down}}(\varphi,\ \psi)]\!](\rho)(\sigma) = 0 \leq \mathsf{vc}[\![S]\!](\rho)(\sigma)$.

This concludes the proof for the first implication. The proof for $\mathit{spec}_{\mathsf{up}}$ is dual. □

Example 2.42. Encoding a specification

The program $S = x :\approx 5$ satisfies $0.5 \sqsubseteq \mathsf{vc}[\![S]\!](0.1 \cdot x)$ (cf. Example 2.35). We can replace it by the specification encoding $\mathit{spec}_{\mathsf{down}}(0.5,\ 0.1 \cdot x)$. In the state $\sigma \in \Sigma$ with post-expectation $\rho \in \mathsf{HeyLo}$, that evaluates to:

$$\begin{aligned} &\mathsf{vc}[\![\mathit{spec}_{\mathsf{down}}(0.5,\ 0.1 \cdot x)]\!](\rho)(\sigma) \\ &= \mathsf{vc}[\![\texttt{down assert } 0.5\texttt{; down havoc Vars; down compare } 0.1 \cdot x]\!](\rho)(\sigma) && \text{(definition of } \mathit{spec}_{\mathsf{down}}\text{)} \\ &= \begin{cases} [\![0.5]\!](\sigma), & \text{if } 0.1 \cdot x \sqsubseteq \rho \\ 0, & \text{else} \end{cases} && \text{(Lemma 2.40)} \end{aligned}$$

By Theorems 2.38 and 2.41, the following program verifies:

$$\mathit{impl}_{\mathsf{down}}(0.5,\ \mathit{spec}_{\mathsf{down}}(0.5,\ 0.1 \cdot x),\ 0.1 \cdot x)\ .$$

Finally, we briefly look at the simplest way we found so far to encode specifications: with just a single havoc statement.

Theorem 2.43. HeyVL: Havoc

Let $\varphi \in \mathsf{HeyLo}$. Then:

$$\forall S \in \mathsf{HeyVL}_{\mathsf{feas}}.\quad \mathsf{vc}[\![S]\!](\varphi) \sqsubseteq \mathsf{vc}[\![\mathsf{up\ havoc\ Vars}]\!](\varphi)$$
$$\forall S \in \mathsf{HeyVL}_{\mathsf{cof}}.\quad \mathsf{vc}[\![\mathsf{down\ havoc\ Vars}]\!](\varphi) \sqsubseteq \mathsf{vc}[\![S]\!](\varphi)$$

Proof. Use the feasibility property (Theorem 2.32) with $r \in \mathbb{R}^{\infty}_{\geq 0}$ such that

$$r = \mathsf{vc}[\![\mathsf{up\ havoc\ Vars}]\!](\varphi)(\sigma)$$

for all $\sigma \in \Sigma$. This is well-defined because $\mathsf{vc}[\![\mathsf{up\ havoc\ Vars}]\!](\varphi)$ is constant; we have for all $\sigma \in \Sigma$:

$$\mathsf{vc}[\![\mathsf{up\ havoc\ Vars}]\!](\varphi)(\sigma) = \sup \{\, [\![\varphi]\!](\sigma') \mid \sigma' \in \Sigma \,\} \ .$$

By feasibility, $\varphi \sqsubseteq \mathsf{vc}[\![\mathsf{up\ havoc\ Vars}]\!](\varphi)$ holds. The co-feasibility case is analogous. □

3. Encoding pGCL into HeyVL

HeyVL is an intermediate verification language, aimed at proving properties of encoded programs. It is explicitly not designed as an easy-to-use (probabilistic) programming language, but instead as the intermediate layer that verification problems for probabilistic programs are encoded in. For example, neither the conditional choice `if` with a Boolean condition nor loops are included in HeyVL.

In this chapter, we encode the probabilistic programming language *pGCL* into HeyVL. pGCL was defined by McIver and Morgan as a probabilistic extension of Dijkstra's guarded-commands language (GCL) [MM05; Dij75]. We will consider *weakest pre-expectation* and *weakest liberal pre-expectation* semantics.

After defining pGCL in Section 3.1, we present the *weakest pre-expectation* and *weakest liberal pre-expectation transformers* for pGCL (Section 3.2). They are similar to the verification condition transformer we defined for HeyVL (Section 2.3.1). We then define encodings for all language constructs with respect to both weakest pre-expectation transformers wp and wlp. More formally, the goal is to provide (four different) encodings $S' \in \mathsf{HeyVL}$ such that S' verifies if verification statements about a pGCL program $C \in \mathsf{pGCL}$ of the following form are true:

1. Upper bounds on weakest pre-expectation semantics:

   Is it true that $\mathsf{wp}[\![C]\!](X) \sqsubseteq Y$ for expectations $X, Y \in \mathbb{E}$?

2. Lower bounds on weakest liberal pre-expectation semantics:

   Is it true that $Y \sqsubseteq \mathsf{wlp}[\![C]\!](X)$ for one-bounded expectations $X, Y \in \mathbb{E}_{\leq 1}$?

3. Lower bounds on weakest pre-expectation semantics:

   Is it true that $Y \sqsubseteq \mathsf{wp}[\![C]\!](X)$ for expectations $X, Y \in \mathbb{E}$?

4. Upper bounds on weakest liberal pre-expectation semantics:

   Is it true that $\mathsf{wlp}[\![C]\!](X) \sqsubseteq Y$ for one-bounded expectations $X, Y \in \mathbb{E}_{\leq 1}$?

The encoding of loops is the difficult part of this task. For problems (1) and (2) we provide the encodings of Park (co-)induction (Section 3.6.1) and k-(co-)induction (Section 3.6.2). For both encodings, the user needs to provide a so-called *loop invariant*. Importantly, the

encodings for (1) and (2) are always *valid approximations* with respect to the original semantics of the program, regardless of whether the loop invariant is actually valid. We discuss this property in Section 3.3.

For problems (3) and (4), we briefly demonstrate a *bounded model checking* encoding in Section 3.6.3. However, bounded model checking does not use invariants and does not use induction. The reason is that there are no corresponding versions of Park (co-)induction and k-(co-)induction for these problems. There are other proof rules for these problems, for example the one presented in [Har+19], which deals with lower bounds on weakest pre-expectations (problem (3)). We leave the encoding of such proof rules in HeyVL for problems (3) and (4) as future work.

We use the encodings of pGCL in this chapter to evaluate our implementation of a verifier of HeyVL, *Caesar*. In the section *Automation* (Chapter 4), we discuss how we implement the automated tool with the goal of efficiently verifying HeyVL programs that we encode pGCL with. Ultimately, we are able to give similar guarantees about decidability and complexities as Batz et al. report for their tool that is specifically built for the proof rule *k-induction* [Bat+21a]. A HeyVL encoding of k-induction for a fixed $k \in \mathbb{N}$ will be provided in Section 3.6.2.

This chapter serves as an example of the expressiveness of HeyVL. By encoding pGCL, we show that HeyVL can encode probabilistic programs and verify their properties. It is our hope that HeyVL can serve as a reusable foundation for the verification of probabilistic programs written in other languages. All encodings we show in this chapter can easily be reused in other contexts. In particular, since pGCL programs can also represent non-probabilistic programs, we immediately get encodings for the non-probabilistic GCL.

In the chapter *Abstraction* (Chapter 5), we will see that all our constructions can be generalized to semantics not just on *expectations*, but on valuations that map a program state σ to values in lattices, like the Boolean lattice $(\mathbb{B}, \Rightarrow)$ that is used for classical verification, or numbers in $([0,1], \leq)$ to represent *bounded* expectations, or natural numbers with infinity $(\mathbb{N} \cup \{\infty\}, \leq)$ for reasoning about runtimes of non-probabilistic programs.

3.1. pGCL

pGCL is a probabilistic programming language defined by McIver and Morgan as an extension of Dijkstra's GCL [MM05; Dij75]. Although semantics of probabilistic programs were studied already by Kozen long before McIver and Morgan in [Koz79; Koz81; Koz85], only McIver and Morgan reintroduced the non-deterministic choices that were present in Dijkstra's GCL, but that Kozen replaced by a probabilistic choice. McIver and Morgan consider the *demonic* non-deterministic choice $\sqcap$ the "very basis of what is now known

as *refinement* and *abstraction* of programs". "An implementation of a program refines its specification; a specification abstracts from its implementation." [MM05, p. 5]

The history of abstraction and refinement for probabilistic programs is interesting to us because of its relation to HeyVL. The work of McIver and Morgan recovered abstraction and refinement for probabilistic programs. For non-probabilistic programs, these ideas were introduced by Back [Bac78]. This led to the *refinement calculus* [Mor88; Bac88; Mor87; Mor94; BvW98].

Although McIver and Morgan defined the notion of refinement for probabilistic programs, they did not add a *specification statement* [Mor88] from the refinement calculus to pGCL. The specification statement can be used to replace a program by its *specification* that includes a pre- and post-condition. We conjecture that the encoding of specifications in HeyVL (Section 2.3.7) allows the encoding of this missing specification statement for probabilistic programs. Further questions relate to the sound verification of recursive probabilistic programs [Olm+16]. The definition of a specification statement for pGCL is out of scope of this thesis. Therefore, we only encode pGCL as defined by McIver and Morgan and leave a specification statement for pGCL as interesting future work.

Let us now look at the syntax of the pGCL language we want to encode.

Definition 3.1. pGCL syntax

The set pGCL of pGCL commands is defined by the following grammar:

$$
\begin{aligned}
C \;::=\; & \texttt{skip} && \text{(no effect)}\\
\mid\; & x := a && \text{(assignment)}\\
\mid\; & C;\, C && \text{(sequential composition)}\\
\mid\; & \{C\}\ [p]\ \{C\} && \text{(probabilistic choice)}\\
\mid\; & \texttt{if}\ (\sqcap)\ \{C\}\ \texttt{else}\ \{C\} && \text{(demonic choice)}\\
\mid\; & \texttt{if}\ (b)\ \{C\}\ \texttt{else}\ \{C\} && \text{(Boolean choice)}\\
\mid\; & \texttt{while}\ (b)\ \{C\}\,, && \text{(while loop)}
\end{aligned}
$$

where $x \in \mathsf{Vars}$ is a variable, $a \in \mathsf{ArithExp}$ is an arithmetic expression, $b \in \mathsf{BExp}$ is a Boolean expression, and $p \in [0,1] \cap \mathbb{Q}$ is a rational probability.

For the arithmetic expressions $a \in \mathsf{ArithExp}$ and the Boolean expressions $b \in \mathsf{BExp}$, we use the syntax already defined for HeyLo and HeyVL in Section 2.2.5. With the addition of the demonic choice, this definition specifies a superset of the language allowed in the paper about *relatively complete* verification by Batz et al. [Bat+21b].

For now, we consider the semantics of the language without the demonic choice. Kozen defined the semantics of such programs by *measure transformers* that move forwards

through the program [Koz79; Koz81]. An initial state $\sigma \in \Sigma$ is transformed into a probability distribution over final states. We will look at *weakest pre-expectation semantics* in the Section 3.2. In addition to these denotational semantics, there are also various types of operational semantics. Examples for these are semantics in the form of probabilistic transition systems [GKM14; Kam19] or trace semantics [CM12; PW15; Kam19].

The `skip` statement does nothing. The assignment statement $x := a$ assigns the result of evaluating the arithmetic expression a in the current state to the variable x. Note that we have a *non-probabilistic* assignment here, in contrast to HeyVL, where the assignment can be probabilistic. A sequence $C_1\,;\, C_2$ of two programs executes C_1 and then C_2. Probabilistic behavior is introduced by the probabilistic choice $\{C_1\}\ [p]\ \{C_2\}$ which executes C_1 with probability p and C_2 with probability $1 - p$. The demonic non-deterministic choice is a difficult to interpret with just operational semantics, so we will just define its weakest pre-expectation semantics later. The Boolean choice $\texttt{if}\ (b)\ \{C_1\}\ \texttt{else}\ \{C_2\}$ executes C_1 if the Boolean expression b evaluates to `true` in the current state and executes C_2 otherwise. The loop $\texttt{while}\ (b)\ \{C\}$ executes C if b is true in the current state and then repeats C until b is false.

As for HeyLo and HeyVL, we use the same definition of program state (Definition 2.3). A program state assigns a non-negative rational number in $\mathbb{Q}_{\geq 0}$ to each variable $x \in \mathsf{Vars}$.

3.2. Weakest Pre-Expectations

The previously discussed semantics were all *forward semantics*, proceeding from an initial state forwards towards final states. The family of *weakest pre-expectation transformers* moves *backwards* through the program, starting with a *post-expectation* and computing a *pre-expectation*. This notion is based on the weakest *pre-condition* calculus for non-probabilistic programs by Dijkstra [Dij76]. The resulting pre-condition is logically *weakest* in the sense that all other valid pre-conditions are implied by it. The *backwards-moving random variable transformers* were originally defined by Kozen [Koz83; Koz85]. McIver and Morgan invented the name *weakest pre-expectation calculus* to emphasize the relation to Dijkstra's weakest pre-condition calculus and added support for non-determinism [MM05]. Our verification condition semantics for HeyVL (Section 2.3.1) is based on this weakest pre-expectation calculus.

There exist a number of very similar weakest pre-expectation semantics for probabilistic programs [Kam19]. In this thesis, we focus on the *weakest pre-expectation* and *weakest liberal pre-expectation semantics* for pGCL. There are also specifically *angelic* weakest pre-expectation transformers [MM05]. To reason about *expected runtimes*, Kaminski et al. presented the *expected runtime calculus* [Kam+16; Kam+18]. Finally, a *mixed-sign* semantics also allows negative values to be assigned to each state instead of just non-negative

extended reals $\mathbb{R}^{\infty}_{\geq 0}$ [KK17]. We believe the extension of HeyVL to support the encoding of these additional calculi is an interesting avenue for further research.

Definition 3.2. pGCL: Weakest pre-expectations

Let C be a pGCL command. The *weakest pre-expectation transformer* $\mathsf{wp}[\![C]\!] \colon \mathbb{E} \to \mathbb{E}$ is given by:

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

Let $X \in \mathbb{E}$ be a post-expectation. The semantics of the `skip` command, $\mathsf{wp}[\![\texttt{skip}]\!](X)$, does nothing and returns the unmodified post-expectation X. An assignment $x := a$ results in the variable x being replaced by a in the post-expectation X. Sequential composition $C_1 \,;\, C_2$ computes the weakest pre-expectation of C_2 with respect to X, then uses this intermediate result to compute the weakest pre-expectation with that. The weakest pre-expectation semantics of the probabilistic choice amounts to a convex sum of weakest pre-expectations of both branches. For the demonic choice, the pointwise minimum of both branches is used. This is the worst case ("demonic") with respect to lower bounds. The Boolean choice uses Iverson brackets to compute the weakest pre-expectation of either C_1 or C_2 depending on whether the condition b is true in the current state.

The semantics of the loop is defined as a least fixed point of the unrollings. We define the *characteristic function* ${}^{\mathsf{wp}}\Phi_X$ of the loop `while` (b) $\{C'\}$ with respect to the post-expectation $X \in \mathbb{E}$ as

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

The weakest pre-expectation of the loop is defined as a least fixed point:

$$ \mathsf{wp}[\![\texttt{while}\ (b)\ \{C'\}]\!](X) = \mathrm{lfp}\, Y.\ {}^{\mathsf{wp}}\Phi_X(Y). $$

By monotonicity of ${}^{\mathsf{wp}}\Phi_X$ and the fact that $(\mathbb{E}, \sqsubseteq)$ is a complete lattice, the Knaster-Tarski theorem guarantees that the fixed point exists [Tar55].

But what does the weakest pre-expectation $\mathsf{wp}[\![C]\!](X)$ actually mean? For programs C without non-determinism, we have the *Kozen duality* [Koz83; Koz85]. Let $p(\sigma') \in [0,1]$

be the probability of C terminating in state $\sigma' \in \Sigma$ when starting in the initial state $\sigma \in \Sigma$. For any post-expectation $X \in \mathbb{E}$:

$$\sum_{\sigma' \in \Sigma} p(\sigma') \cdot X(\sigma') = \mathsf{wp}[\![C]\!](X)(\sigma).$$

So $\mathsf{wp}[\![C]\!](X)(\sigma)$ computes the *expected value* of X after termination when starting in state σ. If $X = [b]$, then $\mathsf{wp}[\![C]\!](X)(\sigma)$ is the probability of termination in state where b is true when starting in state σ. This equality can be extended with *schedulers* to incorporate non-determinism as well [Kam19, page 81].

The dual to the weakest pre-expectation calculus is the *weakest liberal pre-expectation calculus*. It uses a greatest fixed point for the loop semantics. Intuitively, it represents the expected value of the *one-bounded* post-expectation $X \in \mathbb{E}_{\leq 1}$ plus the probability of the program not terminating. For this to be well-defined, the weakest liberal pre-expectation calculus is defined over one-bounded expectations $\mathbb{E}_{\leq 1}$ only.

Definition 3.3. pGCL: Weakest liberal pre-expectations

Let C be a pGCL command. The *weakest liberal pre-expectation transformer* $\mathsf{wlp}[\![C]\!] \colon \mathbb{E}_{\leq 1} \to \mathbb{E}_{\leq 1}$ is given by:

C	$\mathsf{wlp}[\![C]\!](X)$
`skip`	X
$x := a$	$X[x \mapsto a]$
$C_1 \,;\, C_2$	$\mathsf{wlp}[\![C_1]\!](\mathsf{wlp}[\![C_2]\!](X))$
$\{C_1\}\ [p]\ \{C_2\}$	$p \cdot \mathsf{wlp}[\![C_1]\!](X) + (1-p) \cdot \mathsf{wlp}[\![C_2]\!](X)$
`if` $(\sqcap)$ $\{C_1\}$ `else` $\{C_2\}$	$\mathsf{wlp}[\![C_1]\!](X) \sqcap \mathsf{wlp}[\![C_2]\!](X)$
`if` (b) $\{C_1\}$ `else` $\{C_2\}$	$[b] \cdot \mathsf{wlp}[\![C_1]\!](X) + [\neg b] \cdot \mathsf{wlp}[\![C_2]\!](X)$
`while` (b) $\{C'\}$	$\mathsf{gfp}\, Y.\ [b] \cdot \mathsf{wlp}[\![C']\!](Y) + [\neg b] \cdot X$

Apart from being restricted to one-bounded expectations $\mathbb{E}_{\leq 1}$ and using the greatest fixed point, the definitions of the wp and wlp coincide.

Throughout this chapter, we will use the *geometric loop* as an example program. The encoding in HeyVL will use all the encodings we present in this chapter. This example will also be used to illustrate the difference between the Park induction and the more powerful k-induction encoding [Bat+21b].

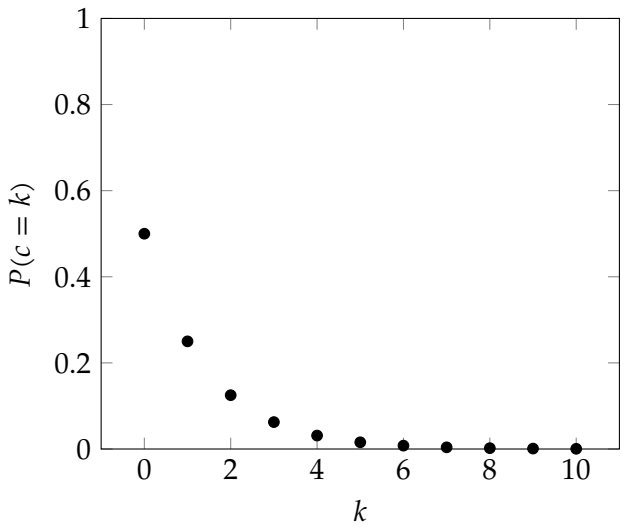

Figure 3.5.: Probability of $c = k$ for the geometric distribution with $p = 0.5$.

Example 3.4. Geometric loop

As an example, we define the *geometric loop* program $C_{\text{geo}} \in \mathsf{pGCL}$ as follows:

$$C_{\text{geo}} = \quad \texttt{while}\,(x = 1)\,\{\,\{x := 0\}\,[0.5]\,\{c := c + 1\}\,\}$$

This program flips a fair coin in each loop iteration until $x \neq 1$ holds. In each iteration where x is not set to 0, the variable c is incremented by one. This program is called *geometric loop* because the distribution of the variable c after the termination of C_{geo} follows a geometric distribution. Starting from the initial state with $c = 0$ and $x = 1$, the probability of c being assigned the value $k \in \mathbb{N}$ after termination is given by $(1 - 0.5)^k \cdot 0.5 = 0.5^{k+1}$. The probability distribution is depicted in Figure 3.5.

For arbitrary initial values of c, the expected value of c after execution evaluates to

$$c + \sum_{k \geq 1} 0.5^{k+1} \cdot k \quad = \quad c + 1.$$

One can verify that this is indeed true using the weakest pre-expectation calculus by computing $\mathsf{wp}[\![C_{\text{geo}}]\!](c)$. At the end of this chapter, in Example 3.38, we show that $\mathsf{wp}[\![C_{\text{geo}}]\!](c) \sqsubseteq c + 1$ holds using an encoding based on Park induction (Section 3.6.1).

3.3. Approximations

Now that we have defined the weakest (liberal) pre-expectation semantics, we come back to the question of how to encode the validity query $\mathsf{wp}[\![C]\!](X) \sqsubseteq Y$ and the three variations

from the introduction of this chapter. In the ideal case, we would have an encoding $S \in \mathsf{HeyVL}$ that is completely equivalent to C, i.e. that $\mathsf{wp}[\![C]\!]([\![\varphi]\!])(\sigma) = \mathsf{vc}[\![S]\!](\varphi)(\sigma)$ holds for all states and all $\varphi \in \mathsf{HeyLo}$.

For the purposes of verification, we instead only *approximate* the original pGCL program C by the HeyVL program S. For upper bounds, we say S *(up-)approximates* C if

$$\forall \varphi \in \mathsf{HeyLo}.\ \forall \sigma \in \Sigma.\quad \mathsf{wp}[\![C]\!]([\![\varphi]\!])(\sigma) \leq \mathsf{vc}[\![S]\!](\varphi)(\sigma).$$

Additionally, if we show some post-expectation $\rho \in \mathsf{HeyLo}$ is a valid upper-bound for the vc of the HeyVL program S with respect to pre-expectation $\varphi \in \mathsf{HeyLo}$, i.e.

$$\mathsf{vc}[\![S]\!](\varphi) \sqsubseteq \rho,$$

then we also know by transitivity that

$$\forall \sigma \in \Sigma.\quad \mathsf{wp}[\![C]\!]([\![\varphi]\!])(\sigma) \leq \mathsf{vc}[\![S]\!](\varphi)(\sigma) \leq [\![\rho]\!](\sigma).$$

On the other hand, if the latter inequality does not hold, i.e. $\mathsf{vc}[\![S]\!](\varphi) \not\sqsubseteq \rho$, then we *cannot* conclude that $\mathsf{wp}[\![C]\!]([\![\varphi]\!])(\sigma) \not\leq [\![\rho]\!](\sigma)$ because $\mathsf{vc}[\![S]\!](\varphi)(\sigma)$ might just be a too coarse approximation. For example, the HeyVL program $S = \texttt{down assume } 0$ is always a valid up-approximation for all pGCL programs C, because $\mathsf{vc}[\![\texttt{down assume } 0]\!](\varphi) \equiv \infty$ for all $\varphi \in \mathsf{HeyLo}$.

Recall that we can encode the property

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

with the HeyVL program $\mathit{impl}_{\mathsf{up}}(\varphi,\ S,\ \rho)$ (cf. Section 2.3.6):

$$\texttt{down negate; up assume } \varphi\texttt{; } S\texttt{; up assert } \rho\texttt{; up negate.}$$

If this encoding verifies, i.e. if $\mathsf{vc}[\![\mathit{impl}_{\mathsf{up}}(\varphi,\ S,\ \rho)(\infty)]\!] \equiv \infty$, then $\mathsf{vc}[\![S]\!](\rho) \sqsubseteq \varphi$ holds. For the dual $\varphi \sqsubseteq \mathsf{vc}[\![S]\!](\rho)$, we used

$$\texttt{down assume } \varphi\texttt{; } S\texttt{; down assert } \rho.$$

Definition 3.6. pGCL approximations

Let $S \in \mathsf{HeyVL}$ and $C \in \mathsf{pGCL}$. We say S *down-approximates* $\mathsf{wp}[\![C]\!]$ if

$$\forall \varphi \in \mathsf{HeyLo}.\ \forall \sigma \in \Sigma.\quad \mathsf{vc}[\![S]\!](\varphi)(\sigma) \leq \mathsf{wp}[\![C]\!]([\![\varphi]\!])(\sigma).$$

We say S *up-approximates* $\mathsf{wp}[\![C]\!]$ if

$$\forall \varphi \in \mathsf{HeyLo}.\ \forall \sigma \in \Sigma.\quad \mathsf{wp}[\![C]\!]([\![\varphi]\!])(\sigma) \leq \mathsf{vc}[\![S]\!](\varphi)(\sigma).$$

The definitions for wlp are analogous with the restriction $\varphi \sqsubseteq 1$.

We define the notions of *down-* and *up-approximations* for wp and wlp. For wlp, we require that the post-expectation $\varphi \in \mathsf{HeyVL}$ is bounded from above by 1, so that $\mathsf{wlp}[\![C]\!]([\![\varphi]\!])$ is defined.

Thus, to solve the four problems from the introduction, we want four encodings with the respective approximation properties: $trans^{\mathsf{wp}}_{\mathsf{up}}(C)$, $trans^{\mathsf{wlp}}_{\mathsf{down}}(C)$, $trans^{\mathsf{wp}}_{\mathsf{down}}(C)$, and $trans^{\mathsf{wlp}}_{\mathsf{up}}(C)$ that all map a pGCL command C to a HeyVL program S such that

1. $trans^{\mathsf{wp}}_{\mathsf{up}}(C)$ up-approximates $\mathsf{wp}[\![C]\!]$,
2. $trans^{\mathsf{wlp}}_{\mathsf{down}}(C)$ down-approximates $\mathsf{wlp}[\![C]\!]$,
3. $trans^{\mathsf{wp}}_{\mathsf{down}}(C)$ down-approximates $\mathsf{wp}[\![C]\!]$,
4. $trans^{\mathsf{wlp}}_{\mathsf{up}}(C)$ up-approximates $\mathsf{wlp}[\![C]\!]$.

All four encodings coincide for loop-free $C \in \mathsf{pGCL}$. For loops, the Park induction and k-induction encodings are used for problems (1) and (2), and bounded model checking is used for problems (3) and (4).

The encodings for the `skip`, assignment, sequential composition $C_1\,;\,C_2$ and demonic choice `if` $(\sqcap)$ are trivial. They simply translate from pGCL to the corresponding equivalent statement in HeyVL, recursively translating sub-statements. The encodings of the Boolean choice, probabilistic choice, and those for loops will be shown in the next sections.

Definition 3.7. Encoding basic statements

Let $C_1, C_2 \in \mathsf{pGCL}$. For $trans \in \left\{ trans^{\mathsf{wp}}_{\mathsf{down}},\ trans^{\mathsf{wp}}_{\mathsf{up}},\ trans^{\mathsf{wlp}}_{\mathsf{down}},\ trans^{\mathsf{wlp}}_{\mathsf{up}} \right\}$, we define

C	$trans(C)$
`skip`	`skip`
$x := a$	$x :\approx a$
$C_1\,;\,C_2$	$trans(C_1)\,;\,trans(C_2)$
`if` $(\sqcap)$ $\{C_1\}$ `else` $\{C_2\}$	`if` $(\sqcap)$ $\{trans(C_1)\}$ `else` $\{trans(C_2)\}$

The deterministic assignment $x := a$ of pGCL is translated to a probabilistic assignment $x :\approx a$ in HeyVL. Recall that arithmetic expressions $a \in \mathsf{ArithExp}$ are valid distribution expressions (Section 2.3.2). The result of $x :\approx a$ is the evaluation of a in the current state σ with probability 1, i.e. a probabilistic assignment with the Dirac distribution that assigns probability 1 to $\sigma(a)$.

3.4. Encoding If-Then-Else

The first statement to encode is the Boolean choice

$$\texttt{if } (b)\ \{C_1\}\ \texttt{else}\ \{C_2\}.$$

Recall the weakest pre-expectation semantics of the Boolean choice (Definition 3.2):

$$\begin{aligned}&\mathsf{wp}[\![\texttt{if } (b)\ \{C_1\}\ \texttt{else}\ \{C_2\}]\!](X)\\ &= [b]\cdot \mathsf{wp}[\![C_1]\!](X) + [\neg b]\cdot \mathsf{wp}[\![C_2]\!](X) \qquad \text{(definition of wp)}\end{aligned}$$

Evaluated in a state σ, we get:

$$= \lambda\,\sigma.\ \begin{cases} \mathsf{wp}[\![C_1]\!](X)(\sigma), & \text{if } [\![b]\!](\sigma)\\ \mathsf{wp}[\![C_2]\!](X)(\sigma), & \text{else}\end{cases}$$

How do we encode this in HeyVL? We can use a non-deterministic choice and assume either b holds, and then execute C_1, or assume $\neg b$ holds, then execute C_2.

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

where S_1 and S_2 are the HeyVL translations of the pGCL statements C_1 and C_2 and $?(b)$ is the Boolean embedding of b (Section 2.2.1). Recall:

$$?(b)(\sigma) = \begin{cases}\infty, & \text{if } [\![b]\!](\sigma)\\ 0, & \text{else}\end{cases}$$

Definition 3.8. Encoding of if-then-else

Let $b \in \mathsf{BExp}$ and $C_1, C_2 \in \mathsf{pGCL}$. For $\mathit{trans} \in \left\{ \mathit{trans}^{\mathsf{wp}}_{\mathsf{down}},\ \mathit{trans}^{\mathsf{wp}}_{\mathsf{up}},\ \mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}},\ \mathit{trans}^{\mathsf{wlp}}_{\mathsf{up}} \right\}$, we define

$$\begin{aligned}&\mathit{trans}(\texttt{if } (b)\ \{C_1\}\ \texttt{else}\ \{C_2\})\\ &\quad = \texttt{if } (\sqcap)\ \{\texttt{down assume } ?(b);\ S_1\}\ \texttt{else}\ \{\texttt{down assume } ?(\neg b);\ S_2\}\,,\end{aligned}$$

where $S_1 = \mathit{trans}(C_1)$ and $S_2 = \mathit{trans}(C_2)$.

Lemma 3.9 states that the vc semantics of the encoding evaluates to a Boolean choice.

Lemma 3.9. Semantics of if-then-else encoding

Let $b \in \mathsf{BExp}$ and $C_1, C_2 \in \mathsf{pGCL}$. For any $\varphi \in \mathsf{HeyLo}$ and $\sigma \in \Sigma$,

$$\mathsf{vc}[\![\mathit{trans}(\texttt{if } (b) \{C_1\} \texttt{ else } \{C_2\})]\!](\varphi)(\sigma) = \begin{cases} \mathsf{vc}[\![\mathit{trans}(C_1)]\!](\varphi)(\sigma), & \text{if } [\![b]\!](\sigma) \\ \mathsf{vc}[\![\mathit{trans}(C_2)]\!](\varphi)(\sigma), & \text{else} \end{cases}$$

where $\mathit{trans} \in \left\{ \mathit{trans}^{\mathsf{wp}}_{\mathsf{down}}, \mathit{trans}^{\mathsf{wp}}_{\mathsf{up}}, \mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}, \mathit{trans}^{\mathsf{wlp}}_{\mathsf{up}} \right\}$.

Proof. Let $b \in \mathsf{BExp}$, $C_1, C_2 \in \mathsf{pGCL}$, $\varphi \in \mathsf{HeyLo}$, and $\sigma \in \Sigma$. Denote $S_1 = \mathit{trans}(C_1)$ and $S_2 = \mathit{trans}(C_2)$ with the respective *trans* function.

$$\begin{aligned}
&\mathsf{vc}[\![\texttt{if } (b) \{S_1\} \texttt{ else } \{S_2\}]\!](\varphi)(\sigma) \\
&= \mathsf{vc}[\![\texttt{if } (\sqcap) \{\texttt{down assume } ?(b)\texttt{; } S_1\} \texttt{ else } \{\texttt{down assume } ?(\neg b)\texttt{; } S_2\}]\!](\varphi)(\sigma) && \text{(Definition 3.8)} \\
&= ((?(b) \to \mathsf{vc}[\![S_1]\!](\varphi)) \sqcap (?(\neg b) \to \mathsf{vc}[\![S_2]\!](\varphi)))(\sigma) && \text{(definition of vc)} \\
&= \begin{cases} (\infty \to \mathsf{vc}[\![S_1]\!](\varphi)(\sigma)) \sqcap (0 \to \mathsf{vc}[\![S_1]\!](\varphi)(\sigma)), & \text{if } [\![b]\!](\sigma) \\ (0 \to \mathsf{vc}[\![S_1]\!](\varphi)(\sigma)) \sqcap (\infty \to \mathsf{vc}[\![S_1]\!](\varphi)(\sigma)), & \text{else} \end{cases} && \text{(case distinction)}
\end{aligned}$$

We have seen in Section 2.2.3 that $0 \to \psi \equiv \infty$ holds for all $\psi \in \mathsf{HeyLo}$.

$$\begin{aligned}
&= \begin{cases} (\mathsf{vc}[\![S_1]\!](\varphi)(\sigma) \sqcap \infty), & \text{if } [\![b]\!](\sigma) \\ (\infty \sqcap \mathsf{vc}[\![S_2]\!](\varphi)(\sigma)), & \text{else} \end{cases} \\
&= \begin{cases} \mathsf{vc}[\![S_1]\!](\varphi)(\sigma), & \text{if } [\![b]\!](\sigma) \\ \mathsf{vc}[\![S_2]\!](\varphi)(\sigma), & \text{else} \end{cases} && (\psi \sqcap \infty \equiv \psi)
\end{aligned}$$

Thus, the encoding of the Boolean choice indeed evaluates to a conditional on b. □

Example 3.10. Geometric loop: Boolean choice

This Boolean choice in pGCL,

$$\texttt{if } (x = 1) \{C_1\} \texttt{ else } \{C_2\},$$

is encoded in HeyVL by $trans^{\mathsf{wp}}_{\mathsf{up}}$ as follows:

```
if (⊓) {
  down assume ?(x = 1);
  S_1
} else {
  down assume ?(¬(x = 1));
  S_2
}
```

where $S_1 = trans^{\mathsf{wp}}_{\mathsf{down}}(C_1)$ and $S_2 = trans^{\mathsf{wp}}_{\mathsf{down}}(C_2)$.

In the following sections, we will use

$$\texttt{if } (b)\ \{S_1\}\ \texttt{else}\ \{S_2\}$$

as syntactic sugar in HeyVL programs to mean

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

3.5. Encoding Probabilistic Choice

The encoding of the probabilistic choice in pGCL,

$$\{C_1\}\ [p]\ \{C_2\}\ ,$$

into HeyVL is simple. Although HeyVL does not have such a probabilistic choice statement, we have the $\mathsf{ber}!(p)$ distribution expression (Section 2.3.2) that evaluates to 1 with probability p and to 0 with probability $1 - p$. Therefore, we can encode the probabilistic choice by a random assignment to a temporary variable and then branch on the result:

$$\mathit{choice} :\approx \mathsf{ber}!(p);\ \texttt{if } (\mathit{choice} = 1)\ \{S_1\}\ \texttt{else}\ \{S_2\}\ ,$$

where S_1 and S_2 are the HeyVL translations of the pGCL statements C_1 and C_2. The `if` statement is encoded as described in the previous section (Section 3.4). We assume the variable *choice* is not used anywhere else in the program as this encoding overwrites its value. One fixed variable *choice* suffices to encode multiple probabilistic choices in the same program as the value of *choice* is immediately read and then not used anymore.

Definition 3.11. Encoding of probabilistic choice

Let $p \in [0,1]$ and $C_1, C_2 \in \mathsf{pGCL}$. For $\mathit{trans} \in \left\{ \mathit{trans}^{\mathsf{wp}}_{\mathsf{down}}, \mathit{trans}^{\mathsf{wp}}_{\mathsf{up}}, \mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}, \mathit{trans}^{\mathsf{wlp}}_{\mathsf{up}} \right\}$, we define

$$\begin{aligned}&\mathit{trans}(\{C_1\}\ [p]\ \{C_2\})\\ &\quad = \mathit{choice} :\approx \mathsf{ber!}(p);\ \mathit{trans}(\texttt{if}\ (\mathit{choice} = 1)\ \{C_1\}\ \texttt{else}\ \{C_2\}).\end{aligned}$$

We use the probabilistic choice of Example 3.4 to illustrate the encoding.

Example 3.12. Geometric loop: Encoding probabilistic choice

The probabilistic choice in C_{geo},

$$\{x := 0\}\ [0.5]\ \{c := c + 1\}\,,$$

is encoded in HeyVL by $\mathit{trans}^{\mathsf{wp}}_{\mathsf{up}}$ as follows:

```
choice :≈ ber!(0.5);
if (⊓) {
  down assume ?(choice = 1);
  x :≈ 0
} else {
  down assume ?(¬(choice = 1));
  c :≈ c + 1
}
```

The encoding always evaluates to a convex sum over both branches.

Lemma 3.13. Semantics of probabilistic choice encoding

Let $p \in [0,1]$ and $C_1, C_2 \in \mathsf{pGCL}$. For any $\varphi \in \mathsf{HeyLo}$,

$$\begin{aligned}&\mathsf{vc}[\![\mathit{trans}(\{C_1\}\ [p]\ \{C_2\})]\!](\varphi)\\ &\quad \equiv p \cdot \mathsf{vc}[\![\mathit{trans}(C_1)]\!](\varphi) + (1-p) \cdot \mathsf{vc}[\![\mathit{trans}(C_2)]\!](\varphi)\,,\end{aligned}$$

where $\mathit{trans} \in \left\{ \mathit{trans}^{\mathsf{wp}}_{\mathsf{down}}, \mathit{trans}^{\mathsf{wp}}_{\mathsf{up}}, \mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}, \mathit{trans}^{\mathsf{wlp}}_{\mathsf{up}} \right\}$.

Proof. Let $p \in [0,1] \cap \mathbb{Q}$, $C_1, C_2 \in \mathsf{pGCL}$, $\varphi \in \mathsf{HeyLo}$, and $\sigma \in \Sigma$.

$$
\begin{aligned}
&\mathsf{vc}[\![\mathit{trans}(\{C_1\}\ [p]\ \{C_2\})]\!](\varphi)(\sigma) \\
&= \mathsf{vc}[\![\mathit{choice} :\approx \mathsf{ber}!(p)\,;\ \mathit{trans}(\texttt{if}\ (\mathit{choice} = 1)\ \{C_1\}\ \texttt{else}\ \{C_2\})]\!](\varphi)(\sigma) && \text{(Definition 3.11)} \\
&= \mathsf{vc}[\![\mathit{choice} :\approx \mathsf{ber}!(p)]\!](\mathsf{vc}[\![\mathit{trans}(\texttt{if}\ (\mathit{choice} = 1)\ \{C_1\}\ \texttt{else}\ \{C_2\})]\!](\varphi))(\sigma) \\
& && \text{(definition of ;)} \\
&= \begin{array}{l} p \cdot \mathsf{vc}[\![\mathit{trans}(\texttt{if}\ (\mathit{choice} = 1)\ \{C_1\}\ \texttt{else}\ \{C_2\})]\!](\varphi)(\sigma[\mathit{choice} \mapsto 1]) \\ \quad +(1-p) \cdot \mathsf{vc}[\![\mathit{trans}(\texttt{if}\ (\mathit{choice} = 1)\ \{C_1\}\ \texttt{else}\ \{C_2\})]\!](\varphi)(\sigma[\mathit{choice} \mapsto 0]) \end{array} \\
& && \text{(definition of ber}!(p)\text{)} \\
&\equiv p \cdot \mathsf{vc}[\![\mathit{trans}(C_1)]\!](\varphi)(\sigma[\mathit{choice} \mapsto 1]) + (1-p) \cdot \mathsf{vc}[\![\mathit{trans}(C_2)]\!](\varphi)(\sigma[\mathit{choice} \mapsto 0]) && \text{(Lemma 3.9)} \\
&= p \cdot \mathsf{vc}[\![\mathit{trans}(C_1)]\!](\varphi)(\sigma) + (1-p) \cdot \mathsf{vc}[\![\mathit{trans}(C_2)]\!](\varphi)(\sigma) \\
& && (\mathit{choice} \text{ does not occur in } C_1, C_2)
\end{aligned}
$$

Therefore, this encoding evaluates to a probabilistic choice, as desired. □

3.6. Encoding Loops

The verification of programs becomes especially challenging in the presence of loops. For example, Alan Turing famously showed that the problem of termination checking is undecidable in 1937 [Tur37]. Since already the classical weakest pre-*condition* can be used to prove the termination of programs, it is no surprise that the computation of the weakest pre-*expectation* is at least as hard. In fact, the complexity of verifying bounds on the weakest pre-expectation of a probabilistic program with loops has been shown to be even harder than the undecidability of the Halting problem [KKM19].

For our deductive verifier, we instead encode *proof rules* for loops. In this thesis, we present two proof rules and corresponding HeyVL encodings for loops: *Park Induction* (Section 3.6.1) and *k-Induction* (Section 3.6.2). k-induction is a generalization of Park induction (Park induction is 1-induction), but we present the simpler Park induction encoding first. The more general k-induction proof rule has a very similar encoding.

Both encodings require additional input from the user to encode a loop: A so-called *invariant*. In classical verification, an invariant is a predicate that holds before the loop and does not change after the loop's execution. So when we can guarantee that a predicate is a valid invariant, we know that if the invariant holds before the loop, it also holds after its execution. The proof rule looks similar for probabilistic programs, but now there are *sub-* and *super-invariants* for lower and upper bounds, respectively.

The task of our proof rule encoding is therefore to check that a user-provided invariant is valid, and only then use that invariant for verification. For verification conditions, this

means that if the invariant cannot be shown valid in HeyVL, the encoding must return the trivial approximation of the semantics that is always valid. For down-approximations, 0 is always a valid lower bound and for up-approximations, ∞ is always a valid upper bound.

We write $C = \texttt{while}\ (b)\ \texttt{invariant}\ I\ \{C'\}$ to denote a loop $\texttt{while}\ (b)\ \{C'\}$ with an attached invariant $I \in \mathsf{HeyLo}$. By a similar reasoning as above, there is always a trivial invariant that leads to a valid down- or up-approximation. The validity of an invariant I is always only given with respect to a post-expectation $\varphi \in \mathsf{HeyLo}$. So for the up-approximation of $\mathsf{wp}[\![C]\!]$, we want an encoding $\mathit{trans}^{\mathsf{wp}}_{\mathsf{up}}(C)$ such that

$$\mathsf{vc}[\![\mathit{trans}^{\mathsf{wp}}_{\mathsf{up}}(\texttt{while}\ (b)\ \texttt{invariant}\ I\ \{C'\})]\!](\varphi)$$

evaluates to I if I is a valid super-invariant for C with respect to φ, and to ∞ otherwise.

Both the Park induction and k-induction encodings work in this way. For $k > 1$, the k-induction encoding accepts more invariants than Park induction does.

At the end of this section, we also briefly discuss *bounded model checking* (Section 3.6.3). While Park induction and k-induction are both encodings for problems (1) and (2) of the introduction, bounded model checking is used for problems (3) and (4).

3.6.1. Park Induction

Park induction, also known as *Park's lemma*, or *Park's theorem*, is a proof rule for complete lattices $(D, \sqsubseteq)$ [Par69]. By proving one inequality, Park induction provides a bound on a least or greatest fixed point.

> Theorem 3.14. Park induction
> Let $(D, \sqsubseteq)$ be a complete lattice. Let $d \in D$ and let $\Phi : D \to D$ be a monotonic function. Then
>
> $$d \sqsubseteq \Phi(d) \quad \text{implies} \quad d \sqsubseteq \mathrm{gfp}\, x.\, \Phi(x) \qquad \text{(Park co-induction)}$$
> $$\Phi(d) \sqsubseteq d \quad \text{implies} \quad \mathrm{lfp}\, x.\, \Phi(x) \sqsubseteq d. \qquad \text{(Park induction)}$$

The lattice of expectations $(\mathbb{E}, \sqsubseteq)$ is a complete lattice and the characteristic functional ${}^{\mathsf{wp}}\Phi_X$ of the while loop $\texttt{while}\ (b)\ \{C'\}$ is monotonic. Therefore, we can apply Park induction to obtain an upper bound $I \in \mathbb{E}$ on the least fixed point of the characteristic functional ${}^{\mathsf{wp}}\Phi_X$ of the loop:

$$ {}^{\mathsf{wp}}\Phi_X(I) \sqsubseteq I \quad \text{implies} \quad \mathsf{wp}[\![\texttt{while}\ (b)\ \{C'\}]\!](X) = \mathrm{lfp}\, Y.\, {}^{\mathsf{wp}}\Phi_X(Y) \sqsubseteq I.$$

Dually, by showing a lower bound $I \in \mathbb{E}_{\leq 1}$ on ${}^{\mathsf{wlp}}\Phi_X(I)$, we can prove a lower bound on the greatest fixed point of ${}^{\mathsf{wlp}}\Phi_X$ in its loop semantics.

We first define *invariants* I for loops as those expectations that satisfy either $I \sqsubseteq \Phi_X(I)$ or $\Phi_X(I) \sqsubseteq I$.

Definition 3.15. Invariants

Let ${}^{\mathsf{wp}}\Phi_X$ be the characteristic function of $C = \texttt{while}\ (b)\ \{C'\}$ with respect to the post-expectation $X \in \mathbb{E}$. Let $I \in \mathbb{E}$. I is a wp-*superinvariant* of C with respect to X iff ${}^{\mathsf{wp}}\Phi_X(I) \sqsubseteq I$.

Let ${}^{\mathsf{wlp}}\Phi_X$ be the characteristic function of $C = \texttt{while}\ (b)\ \{C'\}$ with respect to the post-expectation $X \in \mathbb{E}_{\leq 1}$. Let $I \in \mathbb{E}_{\leq 1}$. I is a wlp-*subinvariant* of C with respect to X iff $I \sqsubseteq {}^{\mathsf{wlp}}\Phi_X(I)$.

These definitions are directly related to Park induction. As Kaminski notes, related literature uses different definitions of invariants [Kam19, Remark 5.2]. For example McIver and Morgan require the stronger $\mathsf{wp}[\![C]\!](I) \sqsubseteq [b] \cdot I$ for wp [MM05, Definition 2.2.1]. There, post-expectations of the form $[\neg b] \cdot I$ are required for the rule to be sound. We use the same definition as Kaminski. The superinvariants correspond to *supermartingales* as used in e.g. [CS14; FH15].

Let us now encode the invariant check in HeyVL. We first define encodings for the characteristic functionals ${}^{\mathsf{wlp}}\Phi$ and ${}^{\mathsf{wlp}}\Phi$ of the loop. These constructions will be used for both Park induction and k-induction.

Let $C = \texttt{while}\ (b)\ \texttt{invariant}\ I\ \{C'\}$ be a pGCL program. Recall that the wlp-characteristic functional ${}^{\mathsf{wlp}}\Phi_X$ of the loop C with respect to post-expectation $X \in \mathbb{E}_{\leq 1}$ is given by

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

We can rewrite ${}^{\mathsf{wlp}}\Phi_X(Y)$ with a conditional expression. Let $\sigma \in \Sigma$.

$$ {}^{\mathsf{wlp}}\Phi_X(Y)(\sigma) = \begin{cases} \mathsf{wlp}[\![C']\!](Y)(\sigma), & \text{if } [\![b]\!](\sigma) \\ X(\sigma), & \text{else} \end{cases} $$

For ${}^{\mathsf{wp}}\Phi$, this can be done analogously.

Definition 3.16. HeyVL: Encoding of characteristic functional

Let $C = \texttt{while}\ (b)\ \texttt{invariant}\ I\ \{C'\}$ be a pGCL program. We define the encodings of the characteristic functionals as follows:

$$ \begin{aligned} \mathit{iter}^{\mathsf{wlp}}_{\mathsf{down}},\ \mathit{iter}^{\mathsf{wp}}_{\mathsf{up}} &\colon \mathsf{pGCL} \times \mathsf{HeyVL} \to \mathsf{HeyVL} \\ \mathit{iter}^{\mathsf{wlp}}_{\mathsf{down}}(C,\ S) &= \quad \texttt{if}\ (b)\ \{\ \mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C')\texttt{;}\ S\ \}\ \texttt{else}\ \{\ \texttt{skip}\ \} \\ \mathit{iter}^{\mathsf{wp}}_{\mathsf{up}}(C,\ S) &= \quad \texttt{if}\ (b)\ \{\ \mathit{trans}^{\mathsf{wp}}_{\mathsf{up}}(C')\texttt{;}\ S\ \}\ \texttt{else}\ \{\ \texttt{skip}\ \} \end{aligned} $$

We define the HeyVL encoding as a conditional choice over the loop condition b. If b is true, we execute the HeyVL encoding of the body C' and then continue with a HeyVL program $S \in \mathsf{HeyVL}$. If b is not true, we do nothing (`skip`).

Lemma 3.17. Semantics of the characteristic functional encoding

Let $C = \texttt{while}\ (b)\ \texttt{invariant}\ I\ \{C'\}$ be a pGCL program and let $S \in \mathsf{HeyVL}$ and $\varphi \in \mathsf{HeyLo}$. For all $\sigma \in \Sigma$,

$$\mathsf{vc}[\![\mathit{iter}^{\mathsf{wlp}}_{\mathsf{down}}(C,\ S)]\!](\varphi)(\sigma) = \begin{cases} \mathsf{vc}[\![\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C')]\!](\mathsf{vc}[\![S]\!](\varphi))(\sigma), & \text{if } [\![b]\!](\sigma) \\ [\![\varphi]\!](\sigma), & \text{else} \end{cases}$$

$$\mathsf{vc}[\![\mathit{iter}^{\mathsf{wp}}_{\mathsf{up}}(C,\ S)]\!](\varphi)(\sigma) = \begin{cases} \mathsf{vc}[\![\mathit{trans}^{\mathsf{wp}}_{\mathsf{up}}(C')]\!](\mathsf{vc}[\![S]\!](\varphi))(\sigma), & \text{if } [\![b]\!](\sigma) \\ [\![\varphi]\!](\sigma), & \text{else} \end{cases}$$

Proof. Follows from Lemma 3.9 and the definition of sequential composition. □

We need to encode a program whose verification conditions always evaluates to I.

Definition 3.18. Constant program

Let $I \in \mathsf{HeyLo}$. We define the *I-constant program* $\mathit{const}(I) \in \mathsf{HeyVL}$ as:

$$\mathit{const}(I) = \quad \mathsf{down}\ \texttt{assert}\ I;\ \mathsf{down}\ \texttt{assume}\ 0\ .$$

Lemma 3.19. Constant program

Let $I \in \mathsf{HeyLo}$. The verification conditions of the constant program $\mathit{const}(I)$ always evaluates to I:

$$\forall \varphi \in \mathsf{HeyLo}.\quad \mathsf{vc}[\![\mathit{const}(I)]\!](\varphi) \equiv I$$

Proof. Let $I \in \mathsf{HeyLo}$ and $\varphi \in \mathsf{HeyLo}$.

$$\begin{aligned} &\mathsf{vc}[\![\mathit{const}(I)]\!](\varphi) \\ &= \mathsf{vc}[\![\mathsf{down}\ \texttt{assert}\ I;\ \mathsf{down}\ \texttt{assume}\ 0]\!](\varphi) && \text{(definition)} \\ &\equiv I \sqcap (0 \to \varphi) && \text{(definition of vc)} \\ &\equiv I \sqcap \infty \quad \equiv I && \square \end{aligned}$$

The constant program can also be written in a variety of other ways (cf. Section 2.2.3). For example:

$$\begin{aligned} const(I) &= \texttt{up assert } I\texttt{; up assume } \infty\,, \\ const(I) &= \texttt{up assert } I\texttt{; down assert } 0\,, \\ const(I) &= \texttt{down assert } I\texttt{; up assert } \infty\,. \end{aligned}$$

We encode the characteristic functionals ${}^{\mathsf{wlp}}\Phi(I)$ and ${}^{\mathsf{wp}}\Phi(I)$ for some $I \in \mathbb{E}$ using Lemmas 3.17 and 3.19. Let $C = \texttt{while } (b) \texttt{ invariant } I\ \{C'\}$ be a pGCL program and $\varphi \in \mathsf{HeyLo}$ and $\sigma \in \Sigma$. Then, the semantics of characteristic functional encodings $iter^{\mathsf{wlp}}_{\mathsf{down}}(C,\ S)$ and $iter^{\mathsf{wp}}_{\mathsf{up}}(C,\ S)$ with $S = const(I)$ evaluate as desired:

$$\mathsf{vc}[\![iter^{\mathsf{wlp}}_{\mathsf{down}}(C,\ const(I))]\!](\varphi)(\sigma) = \begin{cases} \mathsf{vc}[\![trans^{\mathsf{wlp}}_{\mathsf{down}}(C')]\!](I)(\sigma), & \text{if } [\![b]\!](\sigma) \\ [\![\varphi]\!](\sigma), & \text{else} \end{cases}$$

$$\mathsf{vc}[\![iter^{\mathsf{wp}}_{\mathsf{up}}(C,\ const(I))]\!](\varphi)(\sigma) = \begin{cases} \mathsf{vc}[\![trans^{\mathsf{wp}}_{\mathsf{up}}(C')]\!](I)(\sigma), & \text{if } [\![b]\!](\sigma) \\ [\![\varphi]\!](\sigma), & \text{else} \end{cases}$$

The next example shows the encoding of the geometric loop program of Example 3.4 with some invariant $I \in \mathsf{HeyLo}$. The encoding of the probabilistic choice was already demonstrated in Example 3.12.

Example 3.20. Geometric loop: Encoding the characteristic functional

Let $I \in \mathsf{HeyLo}$. The encoding of C_{geo},

$$C_{\text{geo}} = \quad \texttt{while } (x = 1)\ \{\ C'\ \} \qquad \text{where } C' = \{x := 0\}\ [0.5]\ \{c := c + 1\}\,,$$

in HeyVL by $iter^{\mathsf{wp}}_{\mathsf{up}}(C_{\text{geo}},\ const(I))$ evaluates to the following, where $P' = trans^{\mathsf{wp}}_{\mathsf{up}}(C')$:

```
if (⊓) {
  down assume ?(x = 1);
  P';
  down assert I;
  down assume 0
} else {
  down assume ?(¬(x = 1))
}
```

When we add the two statements

$$\texttt{down havoc Vars; down compare } I$$

in front of $\mathit{iter}^{\mathsf{wlp}}_{\mathsf{down}}(C,\ \mathit{const}(I))$, we obtain an encoding that evaluates to the top value ∞ only if $\mathsf{vc}[\![\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C')]\!](I) \sqsubseteq I$ holds and to 0 otherwise. Finally, prepend a down `assert` I and we get the desired encoding for lower bounds. The program below returns $I \sqcap \infty \equiv I$ if $\mathsf{vc}[\![\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C')]\!](I) \sqsubseteq I$ and $I \sqcap 0 \equiv 0$ otherwise:

$$\mathsf{down}\ \texttt{assert}\ I;\ \mathsf{down}\ \mathsf{havoc}\ \mathsf{Vars};\ \mathsf{down}\ \texttt{compare}\ I; \quad \mathit{iter}^{\mathsf{wlp}}_{\mathsf{down}}(C,\ \mathit{const}(I))\ .$$

Does the combination of these first three statements look familiar? They are exactly the same as in the encoding of specifications from Section 2.3.7, $\mathit{spec}_{\mathsf{down}}(I,\ I)$.

Definition 3.21. HeyVL: Encoding Park induction

Let $C =$ `while` (b) `invariant` $I\ \{C'\}$ be a pGCL program. We define the encodings for loops using Park induction as follows:

$$\begin{aligned} \mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C) = &\quad \mathit{spec}_{\mathsf{down}}(I,\ I);\ \mathit{iter}^{\mathsf{wlp}}_{\mathsf{down}}(C',\ \mathit{const}(I)) \\ \mathit{trans}^{\mathsf{wp}}_{\mathsf{up}}(C) = &\quad \mathit{spec}_{\mathsf{up}}(I,\ I);\ \mathit{iter}^{\mathsf{wp}}_{\mathsf{up}}(C',\ \mathit{const}(I)) \end{aligned}$$

A very informal, but intuitive interpretation reads as follows: The specification encoding represents arbitrarily many loop iterations that were already executed. And the loop body that we append represents the *last* loop execution. At its end, we only need to `assert` the invariant I. The `assume` 0 encodes the assumption that we do not need to verify additional executions.

Figure 3.22 shows the complete expansion of both encodings for general programs. Recall that there are multiple different encodings for the if-then-else statement and the constant program, so that the $\mathit{trans}^{\mathsf{wp}}_{\mathsf{up}}(S)$ encoding can be written equivalently with the angelic choice `if` $(\sqcup)$ and only up statements. In this way, both encodings are completely dual.

Example 3.23. Geometric loop: Park induction encoding

Let $I \in \mathsf{HeyLo}$ be a HeyLo formula. The encoding with invariant I of C_{geo},

$$C_{\text{geo}} = \quad \texttt{while}\ (x = 1)\ \{\ C'\ \} \qquad \text{where } C' = \{x := 0\}\ [0.5]\ \{c := c + 1\}\ ,$$

$trans^{\mathsf{wlp}}_{\mathsf{down}}(C)$	$trans^{\mathsf{wp}}_{\mathsf{up}}(C)$
down assert I;	up assert I;
down havoc Vars;	up havoc Vars;
down compare I;	up compare I;
if (⊓) {	if (⊓) {
down assume ?(b);	down assume ?(b);
$trans^{\mathsf{wlp}}_{\mathsf{down}}(C')$;	$trans^{\mathsf{wp}}_{\mathsf{up}}(C')$;
down assert I;	down assert I;
down assume 0	down assume 0
} else {	} else {
down assume ?($\neg b$)	down assume ?($\neg b$)
}	}

Figure 3.22.: Complete expansions of loop encodings with Park induction for a pGCL loop with an invariant, $C =$ `while` (b) `invariant` I $\{C'\}$.

in HeyVL by $trans^{\mathsf{wp}}_{\mathsf{up}}(C_{\text{geo}})$ evaluates to the following, where $P' = trans^{\mathsf{wp}}_{\mathsf{up}}(C')$:

```
up assert I;
up havoc Vars;
up assume I;
if (⊓) {
  down assume ?(x = 1);
  P';
  down assert I;
  down assume 0
} else {
  down assume ?(¬(x = 1))
}
```

We define the notion of *sub-* and *super-invariants* on HeyVL.

Definition 3.24. HeyVL: Park induction invariants

Let $C = \texttt{while}\ (b)\ \texttt{invariant}\ I\ \{C'\}$ be a pGCL program.

For any $\varphi \in \mathsf{HeyLo}$, we say $I \sqsubseteq 1$ is a *sub-invariant* for the wlp-down-encoding of C with respect to φ if

$$\forall \sigma \in \Sigma.\quad [\![I]\!](\sigma) \leq \begin{cases} \mathsf{vc}[\![\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C')]\!](I)(\sigma), & \text{if } [\![b]\!](\sigma) \\ [\![\varphi]\!](\sigma), & \text{else} \end{cases}$$

Similarly, we say I is a *super-invariant* for the wp-up-encoding of C if

$$\forall \sigma \in \Sigma.\quad [\![I]\!](\sigma) \geq \begin{cases} \mathsf{vc}[\![\mathit{trans}^{\mathsf{wp}}_{\mathsf{up}}(C')]\!](I)(\sigma), & \text{if } [\![b]\!](\sigma) \\ [\![\varphi]\!](\sigma), & \text{else} \end{cases}$$

Lemma 3.25 states that the semantics of the Park induction encodings evaluate to the expected invariant check.

Lemma 3.25. Semantics of Park induction encodings

Let $C = \texttt{while}\ (b)\ \texttt{invariant}\ I\ \{C'\}$ be a pGCL program.

$$\mathsf{vc}[\![\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C)]\!](\varphi)(\sigma) = \begin{cases} [\![I]\!](\sigma), & \text{if } I \text{ is a sub-invariant for } C \text{ w.r.t. } \varphi \\ 0, & \text{else} \end{cases}$$

$$\mathsf{vc}[\![\mathit{trans}^{\mathsf{wp}}_{\mathsf{up}}(C)]\!](\varphi)(\sigma) = \begin{cases} [\![I]\!](\sigma), & \text{if } I \text{ is a super-invariant for } C \text{ w.r.t. } \varphi \\ \infty, & \text{else} \end{cases}$$

Proof. Let $C = \texttt{while}\ (b)\ \texttt{invariant}\ I\ \{C'\}$, $\varphi \in \mathsf{HeyLo}$, and $\sigma \in \Sigma$.

$$\begin{aligned}
&\mathsf{vc}[\![\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C)]\!](\varphi)(\sigma) \\
&= \mathsf{vc}[\![\mathit{spec}_{\mathsf{down}}(I,\ I)\,;\ \mathit{iter}^{\mathsf{wlp}}_{\mathsf{down}}(C,\ \mathit{const}(I))]\!](\varphi)(\sigma) && \text{(Definition 3.21)} \\
&= \mathsf{vc}[\![\mathit{spec}_{\mathsf{down}}(I,\ I)]\!](\mathsf{vc}[\![\mathit{iter}^{\mathsf{wlp}}_{\mathsf{down}}(C,\ \mathit{const}(I))]\!](\varphi))(\sigma) && \text{(definition of ;)} \\
&= \begin{cases} [\![I]\!](\sigma), & \text{if } I \sqsubseteq \mathsf{vc}[\![\mathit{iter}^{\mathsf{wlp}}_{\mathsf{down}}(C',\ \mathit{const}(I))]\!](\varphi) \\ 0, & \text{else} \end{cases} && \text{(Lemma 2.40)}
\end{aligned}$$

We now rewrite the condition of the conditional expression:

$$I \sqsubseteq \mathsf{vc}[\![\mathit{iter}^{\mathsf{wlp}}_{\mathsf{down}}(C',\ \mathit{const}(I))]\!](\varphi)$$

$$\text{iff} \quad \forall \sigma \in \Sigma.\ [\![I]\!](\sigma) \leq \begin{cases} \mathsf{vc}[\![\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C')]\!](\mathsf{vc}[\![\mathit{const}(I)]\!](\varphi))(\sigma), & \text{if } [\![b]\!](\sigma) \\ [\![\varphi]\!](\sigma), & \text{else} \end{cases}$$

(Lemma 3.17)

$$\text{iff} \quad \forall \sigma \in \Sigma.\ [\![I]\!](\sigma) \leq \begin{cases} \mathsf{vc}[\![\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C')]\!](I)(\sigma), & \text{if } [\![b]\!](\sigma) \\ [\![\varphi]\!](\sigma), & \text{else} \end{cases}$$

(Lemma 3.19)

iff I is a sub-invariant for the wlp-down-encoding of C with respect to φ.

Thus, the encoding $\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C)$ behaves as expected. The case for $\mathit{trans}^{\mathsf{wp}}_{\mathsf{up}}(C)$ is dual. □

Let $C = \texttt{while}\ (b)\ \texttt{invariant}\ I\ \{C'\}$. Finally, we show that the Park induction encodings are always valid approximations, i.e. that $\mathsf{vc}[\![\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C)]\!] \sqsubseteq \mathsf{wlp}[\![C]\!]$ and $\mathsf{wp}[\![C]\!] \sqsubseteq \mathsf{vc}[\![\mathit{trans}^{\mathsf{wp}}_{\mathsf{up}}(C)]\!]$. For this, we need to assume that the encodings of the loop body C' are valid approximations. With this assumption, we can apply the Park induction theorem to obtain our approximation theorem.

Theorem 3.26. Park induction encodings are approximations

Let $C = \texttt{while}\ (b)\ \texttt{invariant}\ I\ \{C'\}$. Let $\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C')$ be a down-approximation of $\mathsf{wlp}[\![C']\!]$ and $\mathit{trans}^{\mathsf{wp}}_{\mathsf{up}}(C')$ be an up-approximation of $\mathsf{wp}[\![C']\!]$.

For all $\varphi \in \mathsf{HeyLo}$ such that $\varphi \sqsubseteq 1$ and $I \sqsubseteq 1$,

$$\mathsf{vc}[\![\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C)]\!](\varphi) \sqsubseteq \mathsf{wlp}[\![C]\!]([\![\varphi]\!])\ .$$

For all $\varphi \in \mathsf{HeyLo}$,

$$\mathsf{vc}[\![\mathit{trans}^{\mathsf{wp}}_{\mathsf{up}}(C)]\!](\varphi) \sqsupseteq \mathsf{wp}[\![C]\!]([\![\varphi]\!])\ .$$

Proof. Let $C = \texttt{while}\ (b)\ \texttt{invariant}\ I\ \{C'\}$ and let $\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C')$ be a down-approximation of $\mathsf{wlp}[\![C']\!]$. Assume $\varphi \sqsubseteq 1$ and $I \sqsubseteq 1$. By Lemma 3.25, if I is *not* a sub-invariant for the encoding of C with respect to $\varphi \in \mathsf{HeyLo}$, we have for all $\sigma \in \Sigma$,

$$\begin{aligned} \mathsf{vc}[\![\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C)]\!](\varphi)(\sigma) &= 0 && \text{(Lemma 3.25)} \\ &\leq \mathsf{wlp}[\![C]\!]([\![\varphi]\!])(\sigma) && (\forall r \in \mathbb{R}^{\infty}_{\geq 0}.\ 0 \leq r) \end{aligned}$$

Now assume I *is* a sub-invariant for the encoding of C with respect to $\varphi \in \mathsf{HeyLo}$. For all states $\sigma \in \Sigma$,

$$\begin{aligned} [\![I]\!](\sigma) &\leq \mathsf{vc}[\![\mathit{iter}^{\mathsf{wlp}}_{\mathsf{down}}(C',\ \mathit{const}(I))]\!](I)(\sigma) && (I \text{ is a sub-invariant}) \\ &= \begin{cases} \mathsf{vc}[\![\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C')]\!]([\![I]\!])(\sigma), & \text{if } [\![b]\!](\sigma) \\ [\![\varphi]\!](\sigma), & \text{else} \end{cases} \\ & && \text{(Lemmas 3.17 and 3.19)} \end{aligned}$$

$trans^{\mathsf{wlp}}_{\mathsf{down}}(C')$ is a down-approximation of $\mathsf{wlp}[\![C']\!]$, therefore:

$$\begin{aligned}
&\leq \begin{cases} \mathsf{wlp}[\![C']\!]([\![I]\!])(\sigma), & \text{if } [\![b]\!](\sigma) \\ [\![\varphi]\!](\sigma), & \text{else} \end{cases} \\
&= {}^{\mathsf{wlp}}\Phi_{[\![\varphi]\!]}([\![I]\!])(\sigma) && \text{(cf. definition of } {}^{\mathsf{wlp}}\Phi)
\end{aligned}$$

With Park induction, we get:

$$\begin{aligned}
[\![I]\!](\sigma) &\leq [\![I]\!](\sigma) \\
&\leq (\mathrm{gfp}\, Y.\ {}^{\mathsf{wlp}}\Phi_{[\![\varphi]\!]}(Y))(\sigma) && \text{(Park induction)} \\
&= \mathsf{wlp}[\![C]\!]([\![\varphi]\!])(\sigma) && \text{(definition of wlp)}
\end{aligned}$$

So when I is a sub-invariant, we have:

$$\begin{aligned}
\mathsf{vc}[\![trans^{\mathsf{wlp}}_{\mathsf{down}}(C)]\!](\varphi)(\sigma) &= [\![I]\!](\sigma) && \text{(Lemma 3.25)} \\
&\leq \mathsf{wlp}[\![C]\!]([\![\varphi]\!])(\sigma)
\end{aligned}$$

Therefore, $\mathsf{vc}[\![trans^{\mathsf{wlp}}_{\mathsf{down}}(C)]\!]$ is a valid down-approximation for $\mathsf{wlp}[\![C]\!]$. The proof for $\mathsf{vc}[\![trans^{\mathsf{wp}}_{\mathsf{up}}(C)]\!]$ is analogous. □

3.6.2. k-Induction

k-induction [SSS00] is a generalization of induction, where k transition steps in a program are used to verify a property instead of just 1. For $k = 1$, k-induction is the same as Park induction, therefore Park induction is also called 1-induction. Originally invented for hardware verification [SSS00], k-induction has also been used in software verification, e.g. [Don+11]. Recently, Batz et al. have extended k-induction to the verification of probabilistic programs by generalizing k-induction to lattices [Bat+21b].

We now provide an encoding for k-induction in HeyVL for a fixed $k \in \mathbb{N}$. As for Park (co-)induction, we have encodings $trans^{\mathsf{wp}}_{\mathsf{up}}(C)$ and $trans^{\mathsf{wlp}}_{\mathsf{down}}(C)$, but there are no (valid) encodings for the other two directions. In [Bat+21b], the k-induction operator Ψ_f is only defined for upper bounds. We also use the dual of Ψ_f to prove lower bounds for wlp.

> **Definition 3.27.** *k-induction operators*
>
> Let $\Phi : D \to D$ be a function on a lattice $(D, \sqsubseteq)$ and let $f \in D$. The *k-induction operators* ${}^{\mathsf{down}}\Psi_f$ *and* ${}^{\mathsf{up}}\Psi_f$ *with respect to f and Φ* are defined as:
>
> $$\begin{aligned} {}^{\mathsf{down}}\Psi_f &: D \to D, \quad d \mapsto \Phi(d) \sqcup f \\ {}^{\mathsf{up}}\Psi_f &: D \to D, \quad d \mapsto \Phi(d) \sqcap f \end{aligned}$$

The k-induction theorem looks very similar to the one of Park induction (Theorem 3.14). Now we have $\Phi({}^{\text{down}}\Psi_d^k(d))$ or $\Phi({}^{\text{up}}\Psi_d^k(d))$ instead of just $\Phi(d)$. The expression ${}^{\text{down}}\Psi_d^k(d)$ refers to the k-fold application of ${}^{\text{down}}\Psi_d$ to d.

Theorem 3.28. k-induction

Let $(D, \sqsubseteq)$ be a complete lattice. Let $d \in D$ and let $\Phi : D \to D$ be a monotonic function. For $k \in \mathbb{N}$,

$$d \sqsubseteq \Phi({}^{\text{down}}\Psi_d^k(d)) \quad \text{implies} \quad d \sqsubseteq \operatorname{gfp} x.\, \Phi(x) \qquad (k\text{-co-induction})$$
$$\Phi({}^{\text{up}}\Psi_d^k(d)) \sqsubseteq d \quad \text{implies} \quad \operatorname{lfp} x.\, \Phi(x) \sqsubseteq d \qquad (k\text{-induction})$$

Proof. The least fixed point statement is shown in [Bat+21b, Theorem 2]. The statement about greatest fixed points is dual. □

We get the following statement for lower bounds on weakest liberal pre-expectations of loops $C = \texttt{while}\ (b)\ \texttt{invariant}\ I\ \{C'\}$. Let $k \in \mathbb{N}$, and $I, X \in \mathbb{E}_{\leq 1}$. The Φ used in ${}^{\text{down}}\Psi$ is the wlp-characteristic functional ${}^{\text{wlp}}\Phi_X$.

$$I \sqsubseteq {}^{\text{wlp}}\Phi_X({}^{\text{down}}\Psi_I^k(I)) \quad \text{implies} \quad I \sqsubseteq \operatorname{gfp} Y.\, {}^{\text{wlp}}\Phi_X(Y) = \text{wlp}[\![\texttt{while}\ (b)\ \{C'\}]\!](X).$$

And there is also a dual statement for upper bounds of wp.

The encoding for k-induction is similar to the one of Park induction. We start with the `assert`, `havoc`, and `compare` statements for the specification encoding. Then we encode the characteristic functional Φ_X, but now instead of encoding $\Phi_X(I)$, we encode the $(k-1)$-fold iteration of $\Psi_I(I)$ wrapped in one Φ-encoding. Let $C = \texttt{while}\ (b)\ \texttt{invariant}\ I\ \{C'\}$ For lower bounds of wlp and $k = 2$, that amounts to one application of Ψ_I to I:

$$\mathit{spec}_{\text{down}}(I,\ I)\,;\ \mathit{iter}^{\text{wlp}}_{\text{down}}(C,\ \mathit{extend}^{\text{wlp}}_{\text{down}}(C,\ \mathit{const}(I)))\ ,$$

where $\mathit{extend}^{\text{wlp}}_{\text{down}}(C,\ \mathit{const}(I))$ encodes $\Psi_I(I)$ for lower bounds of wlp.

Definition 3.29. HeyVL: k-induction operator encodings

Let $C = \texttt{while}\ (b)\ \texttt{invariant}\ I\ \{C'\}$ be a pGCL program and let $k \in \mathbb{N}$. Then, the k-induction operator encodings are defined as follows:

$$\mathit{extend}^{\text{wlp}}_{\text{down}}(C)\colon \text{HeyVL} \to \text{HeyVL}, \quad S \mapsto \text{up}\ \texttt{assert}\ I;\ \mathit{iter}^{\text{wlp}}_{\text{down}}(C,\ S)\ ,$$
$$\mathit{extend}^{\text{wp}}_{\text{up}}(C)\colon \text{HeyVL} \to \text{HeyVL}, \quad S \mapsto \text{down}\ \texttt{assert}\ I;\ \mathit{iter}^{\text{wp}}_{\text{up}}(C,\ S)\ .$$

Notice how we simply use the up `assert` statement for *lower* bounds and the down `assert` for upper bounds. With $\varphi \in \text{HeyLo}$, the semantics of this encoding evaluates to:

$$\text{vc}[\![\mathit{extend}^{\text{wlp}}_{\text{down}}(C)(S)]\!](\varphi) = I \sqcup \text{vc}[\![\mathit{iter}^{\text{wlp}}_{\text{down}}(C,\ S)]\!](\varphi)\ .$$

This is a simple encoding, but there is another equivalent encoding that may be somewhat more intuitive from an operational perspective. We might define the encoding $\mathit{extend}^{\mathsf{wlp}}_{\mathsf{down}}(C,\ S)$ as

$$\texttt{if (}\sqcup\texttt{) \{ }\mathit{iter}^{\mathsf{wlp}}_{\mathsf{down}}(C,\ S)\texttt{ \} else \{ }\mathit{const}(I)\texttt{ \}}\ .$$

Informally, this operator encoding can be interpreted as an angelic choice that chooses the branch whichever maximizes the pre-expectation, and so goes towards the *greatest* pre-expectation possible, i.e. that makes the verification of lower bounds most likely. In one branch, we run the next iteration, S, and in the other, we assert I holds and assume that this is the final iteration. Similar to the intuition from the previous section on Park induction, the k-induction encoding can be read as the encoding of *one of the last k iterations* of the loop after arbitrarily many before it.

We write $\mathit{extend}^{\mathsf{wlp}}_{\mathsf{down}}(C)^{(k)}(S)$ for the k-fold application of $\mathit{extend}^{\mathsf{wlp}}_{\mathsf{down}}(C)$ to $S \in \mathsf{HeyVL}$:

$$\begin{aligned}\mathit{extend}^{\mathsf{wlp}}_{\mathsf{down}}(C)^{(0)}(S) &= S\\ \mathit{extend}^{\mathsf{wlp}}_{\mathsf{down}}(C)^{(k+1)}(S) &= \mathit{extend}^{\mathsf{wlp}}_{\mathsf{down}}(C,\ \mathit{extend}^{\mathsf{wlp}}_{\mathsf{down}}(C)^{(k)}(S))\end{aligned}$$

Same for $\mathit{extend}^{\mathsf{wp}}_{\mathsf{up}}$.

Definition 3.30. HeyVL: Encoding k-induction

Let $C = \texttt{while (}b\texttt{) invariant }I\ \{C'\}$ be a pGCL program and $k \geq 1$. We define the encodings for loops using k-induction as follows:

$$\begin{aligned}\mathit{kind}^{\mathsf{wlp}}_{\mathsf{down}}(k,\ C) = &\quad \mathit{spec}_{\mathsf{down}}(I,\ I)\,;\ \mathit{iter}^{\mathsf{wlp}}_{\mathsf{down}}(C,\ \mathit{extend}^{\mathsf{wlp}}_{\mathsf{down}}(C)^{(k-1)}(\mathit{const}(I)))\\ \mathit{kind}^{\mathsf{wp}}_{\mathsf{up}}(k,\ C) = &\quad \mathit{spec}_{\mathsf{up}}(I,\ I)\,;\ \mathit{iter}^{\mathsf{wp}}_{\mathsf{up}}(C,\ \mathit{extend}^{\mathsf{wp}}_{\mathsf{up}}(C)^{(k-1)}(\mathit{const}(I)))\end{aligned}$$

Note that *k-induction* encodings require $k-1$ applications of the *extend* encoding. This is the same terminology as used by Batz et al. [Bat+21b]. When $k = 1$ holds, the encoding contains no actual *extend* encoding because $\mathit{extend}(C)^0(S) = S$. In this case, the encodings of k-induction and Park induction are completely identical. Figure 3.31 shows the expansion of the encoding of 2-induction for a loop. For readability, we use the Boolean choice notation `if (`b`)` in HeyVL.

$kind^{\mathsf{wlp}}_{\mathsf{down}}(2,\ C)$	$kind^{\mathsf{wp}}_{\mathsf{up}}(2,\ C)$
down assert I;	up assert I;
down havoc Vars;	up havoc Vars;
down compare I;	up compare I;
if (b) {	if (b) {
$trans^{\mathsf{wlp}}_{\mathsf{down}}(C')$;	$trans^{\mathsf{wp}}_{\mathsf{up}}(C')$;
up assert I;	down assert I;
if (b) {	if (b) {
$trans^{\mathsf{wlp}}_{\mathsf{down}}(C')$;	$trans^{\mathsf{wp}}_{\mathsf{up}}(C')$;
down assert I;	down assert I;
down assume 0	down assume 0
}	}
}	}

Figure 3.31.: Loop encodings with 2-induction for $C =$ `while` (b) `invariant` $I\ \{C'\}$.

Lemma 3.32. Semantics of k-induction encodings

Let $C =$ `while` (b) `invariant` $I\ \{C'\}$ be a pGCL program and $k \geq 1$. Let $\varphi \in$ HeyLo. For all $\sigma \in \Sigma$,

$$\mathsf{vc}[\![kind^{\mathsf{wlp}}_{\mathsf{down}}(k,\ C)]\!](\varphi)(\sigma) = \begin{cases} [\![I]\!](\sigma), & \text{if } I \sqsubseteq \mathsf{vc}[\![iter^{\mathsf{wlp}}_{\mathsf{down}}(C,\ extend^{\mathsf{wlp}}_{\mathsf{down}}(C)^{(k-1)}(const(I)))]\!](\varphi) \\ 0, & \text{else} \end{cases}$$

$$\mathsf{vc}[\![kind^{\mathsf{wp}}_{\mathsf{up}}(k,\ C)]\!](\varphi)(\sigma) = \begin{cases} [\![I]\!](\sigma), & \text{if } I \sqsupseteq \mathsf{vc}[\![iter^{\mathsf{wp}}_{\mathsf{up}}(C,\ extend^{\mathsf{wp}}_{\mathsf{up}}(C)^{(k-1)}(const(I)))]\!](\varphi) \\ \infty, & \text{else} \end{cases}$$

Proof. Follows immediately from Lemma 2.40. □

Theorem 3.33. *k*-induction encodings are approximations

Let $C = \texttt{while}\ (b)\ \texttt{invariant}\ I\ \{C'\}$. Let $\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C')$ be a down-approximation of $\mathsf{wlp}[\![C']\!]$ and $\mathit{trans}^{\mathsf{wp}}_{\mathsf{up}}(C')$ be an up-approximation of $\mathsf{wp}[\![C']\!]$.

For all $\varphi \in \mathsf{HeyLo}$ such that $\varphi \sqsubseteq 1$, $I \sqsubseteq 1$, and $k \geq 1$,

$$\mathsf{vc}[\![\mathit{kind}^{\mathsf{wlp}}_{\mathsf{down}}(k,\ C)]\!](\varphi) \sqsubseteq \mathsf{wlp}[\![C]\!]([\![\varphi]\!])\ .$$

For all $\varphi \in \mathsf{HeyLo}$ and $k \geq 1$,

$$\mathsf{vc}[\![\mathit{kind}^{\mathsf{wp}}_{\mathsf{up}}(k,\ C)]\!](\varphi) \sqsupseteq \mathsf{wp}[\![C]\!]([\![\varphi]\!])\ .$$

Proof. Let $C = \texttt{while}\ (b)\ \texttt{invariant}\ I\ \{C'\}$ and let $\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C')$ be a down-approximation of $\mathsf{wlp}[\![C']\!]$. Furthermore, let $\varphi \in \mathsf{HeyLo}$ such that $\varphi \sqsubseteq 1$, $I \sqsubseteq 1$, and $k \geq 1$. Let $\sigma \in \Sigma$. By Lemma 3.32,

$$\begin{aligned}
&\mathsf{vc}[\![\mathit{kind}^{\mathsf{wlp}}_{\mathsf{down}}(k,\ C)]\!](\varphi)(\sigma) \\
&= \begin{cases} [\![I]\!](\sigma), & \text{if } I \sqsubseteq \mathsf{vc}[\![\mathit{iter}^{\mathsf{wlp}}_{\mathsf{down}}(C,\ \mathit{extend}^{\mathsf{wlp}}_{\mathsf{down}}(C)^{(k-1)}(\mathit{const}(I)))]\!](\varphi) \\ 0, & \text{else} \end{cases}
\end{aligned}$$

The case when 0 is returned is trivial, since $0 \sqsubseteq \mathsf{wlp}[\![C]\!]([\![\varphi]\!])$.

Assume $I \sqsubseteq \mathsf{vc}[\![\mathit{iter}^{\mathsf{wlp}}_{\mathsf{down}}(C,\ \mathit{extend}^{\mathsf{wlp}}_{\mathsf{down}}(C)^{(k-1)}(\mathit{const}(I)))]\!](\varphi)$ holds. For all $\sigma' \in \Sigma$,

$$\begin{aligned}
[\![I]\!](\sigma') &\leq \mathsf{vc}[\![\mathit{iter}^{\mathsf{wlp}}_{\mathsf{down}}(C,\ \mathit{extend}^{\mathsf{wlp}}_{\mathsf{down}}(C)^{(k-1)}(\mathit{const}(I)))]\!](\varphi)(\sigma') \\
&\leq {}^{\mathsf{wlp}}\Phi_{[\![\varphi]\!]}([\![\mathsf{vc}[\![\mathit{extend}^{\mathsf{wlp}}_{\mathsf{down}}(C)^{(k-1)}(\mathit{const}(I))]\!]](\varphi))(\sigma') && \text{(cf. Theorem 3.26)}
\end{aligned}$$

If we show $\mathsf{vc}[\![\mathit{extend}^{\mathsf{wlp}}_{\mathsf{down}}(C)^{(k-1)}(\mathit{const}(I))]\!](\varphi) \sqsubseteq {}^{\mathsf{down}}\Psi^{k-1}_{[\![I]\!]}([\![I]\!])$ for all $k \in \mathbb{N}$ where ${}^{\mathsf{down}}\Psi$ is the k-induction operator with respect to ${}^{\mathsf{wlp}}\Phi_{[\![\varphi]\!]}$, we get by monotonicity of ${}^{\mathsf{wlp}}\Phi$:

$$\leq {}^{\mathsf{wlp}}\Phi_{[\![\varphi]\!]}({}^{\mathsf{down}}\Psi^{k-1}_{[\![I]\!]})(\sigma')$$

Then, we can apply the k-induction theorem (Theorem 3.28):

$$\begin{aligned}
[\![I]\!](\sigma) &\leq \mathrm{gfp}\ Y.\ {}^{\mathsf{wlp}}\Phi_{[\![\varphi]\!]}(Y) \\
&= \mathsf{wlp}[\![C]\!]([\![\varphi]\!])(\sigma) && \text{(definition of wlp)}
\end{aligned}$$

We now show by induction over $k \geq 1$, for all $\sigma' \in \Sigma$:

$$\mathsf{vc}[\![\mathit{extend}^{\mathsf{wlp}}_{\mathsf{down}}(C)^{(k-1)}(\mathit{const}(I))]\!](\varphi)(\sigma') \sqsubseteq {}^{\mathsf{down}}\Psi^{k-1}_{[\![I]\!]}([\![I]\!])(\sigma')$$

For the base case $k = 1$, we have:

$$\mathsf{vc}[\![\mathit{extend}^{\mathsf{wlp}}_{\mathsf{down}}(C)^{(k-1)}(\mathit{const}(I))]\!](\varphi)(\sigma')$$

$$
\begin{aligned}
&= \mathsf{vc}[\![const(I)]\!](\varphi)(\sigma') && (k-1=0)\\
&= [\![I]\!](\sigma') && (\text{Lemma } 3.19)\\
&= {}^{\mathsf{down}}\Psi^{0}_{[\![I]\!]}([\![I]\!])(\sigma') && (f^0(x)=x)\\
&= {}^{\mathsf{down}}\Psi^{k-1}_{[\![I]\!]}([\![I]\!])(\sigma') && (k-1=0)
\end{aligned}
$$

Now assume the statement holds for an arbitrary but fixed $k \geq 1$. Then,

$$
\begin{aligned}
&\mathsf{vc}[\![extend^{\mathsf{wlp}}_{\mathsf{down}}(C)^{(k)}(const(I))]\!](\varphi)(\sigma')\\
&= \mathsf{vc}[\![extend^{\mathsf{wlp}}_{\mathsf{down}}(C)^{(1)}(extend^{\mathsf{wlp}}_{\mathsf{down}}(C)^{(k-1)}(const(I)))]\!](\varphi)(\sigma')\\
&\sqsubseteq \mathsf{vc}[\![\mathsf{up}\ \texttt{assert}\ I;\ iter^{\mathsf{wlp}}_{\mathsf{down}}(C,\ extend^{\mathsf{wlp}}_{\mathsf{down}}(C)^{(k-1)}(const(I)))]\!](\varphi)(\sigma') && (\text{Definition } 3.29)\\
&= [\![I]\!](\sigma') \sqcup \mathsf{vc}[\![iter^{\mathsf{wlp}}_{\mathsf{down}}(C,\ extend^{\mathsf{wlp}}_{\mathsf{down}}(C)^{(k-1)}(const(I)))]\!](\varphi)(\sigma') && (\text{definition of } \texttt{assert})
\end{aligned}
$$

We use the fact that $trans^{\mathsf{wlp}}_{\mathsf{down}}(C')$ is a down-approximation of $\mathsf{wlp}[\![C']\!]$ to rewrite with ${}^{\mathsf{wlp}}\Phi$, just like in Theorem 3.26 for Park induction.

$$
\begin{aligned}
&\sqsubseteq [\![I]\!](\sigma') \sqcup {}^{\mathsf{wlp}}\Phi_{[\![I]\!]}([\![\mathsf{vc}[\![extend^{\mathsf{wlp}}_{\mathsf{down}}(C)^{(k-1)}(const(I))]\!](\varphi)]\!])(\sigma') && (\text{cf. Theorem } 3.26)\\
&= {}^{\mathsf{down}}\Psi_{[\![I]\!]}([\![\mathsf{vc}[\![extend^{\mathsf{wlp}}_{\mathsf{down}}(C)^{(k-1)}(const(I))]\!](\varphi)]\!])(\sigma') && (\text{Definition } 3.27)
\end{aligned}
$$

We use monotonicity of ${}^{\mathsf{down}}\Psi$ and the induction hypothesis:

$$
\begin{aligned}
&\sqsubseteq {}^{\mathsf{down}}\Psi_{[\![I]\!]}({}^{\mathsf{down}}\Psi^{k-1}_{[\![I]\!]}([\![I]\!]))(\sigma')\\
&= {}^{\mathsf{down}}\Psi^{k}_{[\![I]\!]}([\![I]\!])(\sigma')
\end{aligned}
$$

We have now shown everything required to show the theorem for $kind^{\mathsf{wlp}}_{\mathsf{down}}(k,\ C)$. The proof for $kind^{\mathsf{wp}}_{\mathsf{up}}(k,\ C)$ is dual. □

3.6.3. Bounded Model Checking

Bounded model checking [Bie+99] is a technique to refute properties. It was generalized by Batz et al. [Bat+21b] to lattices. In this setting, bounded model checking simply amounts to computing the k-th iteration of the characteristic functional of the loop, starting at 1 (for lower bounds on wlp) or 0 (for upper bounds).[Fixed]

Definition 3.34. HeyVL: Encoding bounded model checking

Let $C = \texttt{while}\ (b)\ \{C'\}$ be a pGCL program. The k-th iteration of the bounded model checking encodings is defined as follows:

$$
\begin{aligned}
bmc^{\mathsf{wp}}_{\mathsf{down}},\ bmc^{\mathsf{wlp}}_{\mathsf{up}} &: \mathsf{pGCL} \times \mathbb{N} \to \mathsf{HeyVL}\\
bmc^{\mathsf{wp}}_{\mathsf{down}}(C,k) &= iter^{\mathsf{wp}}_{\mathsf{down}}(C)^{(k)}(const(0))\\
bmc^{\mathsf{wlp}}_{\mathsf{up}}(C,k) &= iter^{\mathsf{wlp}}_{\mathsf{up}}(C)^{(k)}(const(1))
\end{aligned}
$$

The repeated iterations $iter^{\mathsf{wp}}_{\mathsf{down}}$ and $iter^{\mathsf{wlp}}_{\mathsf{up}}$ are defined analogously to Definition 3.16.

For wlp, the k-th iteration is an up-approximation and for wp, the k-th iteration is a down-approximation. These corresponds to problems (3) and (4) respectively from the introduction of this chapter.

Theorem 3.35. Bounded model checking encodings are approximations

Let $C = \texttt{while}\ (b)\ \texttt{invariant}\ I\ \{C'\}$. Let $trans^{\mathsf{wp}}_{\mathsf{up}}(C')$ be an up-approximation of $\mathsf{wp}[\![C']\!]$ and $trans^{\mathsf{wlp}}_{\mathsf{down}}(C')$ be a down-approximation of $\mathsf{wp}[\![C']\!]$.

For all $\varphi \in \mathsf{HeyLo}$ and $k \geq 1$,

$$\mathsf{vc}[\![bmc^{\mathsf{wp}}_{\mathsf{down}}(k, C)]\!](\varphi) \sqsubseteq \mathsf{wp}[\![C]\!]([\![\varphi]\!])$$

For all $\varphi \in \mathsf{HeyLo}$ such that $\varphi \sqsubseteq 1$, $I \sqsubseteq 1$, and $k \geq 1$,

$$\mathsf{vc}[\![bmc^{\mathsf{wlp}}_{\mathsf{up}}(k, C)]\!](\varphi) \sqsupseteq \mathsf{wlp}[\![C]\!]([\![\varphi]\!])$$

The proof is simple and omitted here. One merely needs to show that $bmc^{\mathsf{wp}}_{\mathsf{down}}(C, k)$ encodes ${}^{\mathsf{wp}}\Phi^{k}_{[\![\varphi]\!]}(0)$ and that $bmc^{\mathsf{wlp}}_{\mathsf{up}}(C, k)$ encodes ${}^{\mathsf{wlp}}\Phi^{k}_{[\![\varphi]\!]}(1)$. Then, use the facts ${}^{\mathsf{wp}}\Phi^{k}_{[\![\varphi]\!]}(0) \sqsubseteq \operatorname{lfp} X.\ {}^{\mathsf{wp}}\Phi_{[\![\varphi]\!]}(X)$ and ${}^{\mathsf{wlp}}\Phi^{k}_{[\![\varphi]\!]}(1) \sqsupseteq \operatorname{gfp} X.\ {}^{\mathsf{wlp}}\Phi_{[\![\varphi]\!]}(X)$.

To refute an invariant $I \in \mathsf{HeyLo}$ for a particular post-expectation $\varphi \in \mathsf{HeyLo}$, we need to show that I *is not* a bound for the bounded model checking encoding, e.g.

$$\mathsf{vc}[\![bmc^{\mathsf{wp}}_{\mathsf{down}}(k, C)]\!](\varphi) \not\sqsubseteq I$$

Together with Theorem 3.35, the above statement contradicts I being a valid super-invariant, since valid super-invariants always satisfy

$$\mathsf{wp}[\![C]\!]([\![\varphi]\!]) \sqsubseteq I\,.$$

3.7. Complete Encodings

We now summarize the our encodings for pGCL programs. Figure 3.36 shows the encoding $trans^{\mathsf{wlp}}_{\mathsf{down}}(C)$ for a pGCL program C. The encoding $trans^{\mathsf{wp}}_{\mathsf{up}}(C)$ is dual. At the end of this chapter, Example 3.38 illustrates the complete encoding with Park induction for the geometric loop example program.

Theorem 3.37 is the central correctness theorem of this chapter. For wlp, we require that all invariants $I \in \mathsf{HeyLo}$ that occur in loops are 1-bounded.

Theorem 3.37. HeyVL encodings are approximations

For all $C \in \mathsf{pGCL}$,

- $trans^{\mathsf{wlp}}_{\mathsf{down}}(C)$ is a down-approximation of $\mathsf{wlp}[\![C]\!]$ and
- $trans^{\mathsf{wp}}_{\mathsf{up}}(C)$ is an up-approximation of $\mathsf{wp}[\![C]\!]$.

C	$trans^{\mathsf{wlp}}_{\mathsf{down}}(C)$
`skip`	`skip`
$x := a$	$x :\approx a$
$C_1 ; C_2$	$S_1 ; S_2$
$\{C_1\}\ [p]\ \{C_2\}$	$\mathit{choice} :\approx \mathsf{ber!}(p)$; $trans^{\mathsf{wlp}}_{\mathsf{down}}(\texttt{if}\ (\mathit{choice} = 1)\ \{C_1\}\ \texttt{else}\ \{C_2\})$
`if` $(\sqcap)$ $\{C_1\}$ `else` $\{C_2\}$	`if` $(\sqcap)$ $\{S_1\}$ `else` $\{S_2\}$
`if` (b) $\{C_1\}$ `else` $\{C_2\}$	`if` $(\sqcap)$ {`down assume ?`(b); S_1} `else` {`down assume ?`$(\neg b)$; S_2}
`while` (b) `invariant` I $\{C'\}$	$\mathit{spec}_{\mathsf{down}}(I,\ C')I$; $\mathit{iter}^{\mathsf{wlp}}_{\mathsf{down}}(C',\ \mathit{const}(I))$

Figure 3.36.: Summary of pGCL encodings for lower bounds of wlp. We use the abbreviations $S_1 = trans^{\mathsf{wlp}}_{\mathsf{down}}(C_1)$ and $S_2 = trans^{\mathsf{wlp}}_{\mathsf{down}}(C_2)$. For wlp, we require $I \sqsubseteq 1$.

Proof. Let $C \in \mathsf{pGCL}$. We show that $trans^{\mathsf{wlp}}_{\mathsf{down}}(C)$ is a down-approximation of $\mathsf{wlp}[\![C]\!]$, i.e.

$$\forall \varphi \sqsubseteq 1. \quad \mathsf{vc}[\![trans^{\mathsf{wlp}}_{\mathsf{down}}(C)]\!](\varphi) \sqsubseteq \mathsf{wlp}[\![C]\!]([\![\varphi]\!]) \ .$$

We do the proof by induction over the structure of C. Let $\varphi \in \mathsf{HeyLo}$ such that $\varphi \sqsubseteq 1$ and let $\sigma \in \Sigma$.

Base cases:

- $C = \texttt{skip}$:

$$\mathsf{vc}[\![\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(\texttt{skip})]\!](\varphi)(\sigma) = \mathsf{vc}[\![\texttt{skip}]\!](\varphi)(\sigma) = [\![\varphi]\!](\sigma) = \mathsf{wlp}[\![\texttt{skip}]\!]([\![\varphi]\!]).$$

- $C = x := a$:

$$\begin{aligned}
&\mathsf{vc}[\![\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(x := a)]\!](\varphi)(\sigma) \\
&= \mathsf{vc}[\![x :\approx a]\!](\varphi)(\sigma) && \text{(definition of } \mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}\text{)} \\
&= [\![\varphi[x \mapsto a]]\!](\sigma) && \text{(definition of vc)} \\
&= \mathsf{wlp}[\![x := a]\!]([\![\varphi]\!]). && \text{(definition of wlp)}
\end{aligned}$$

Now assume the induction hypothesis holds for arbitrary but fixed C_1, C_2 and $C' \in \mathsf{pGCL}$.

Induction step.

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

$$\begin{aligned}
&\mathsf{vc}[\![\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C_1\,;\, C_2)]\!](\varphi)(\sigma) \\
&= \mathsf{vc}[\![\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C_1)\,;\, \mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C_2)]\!](\varphi)(\sigma) && \text{(definition of } \mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}\text{)} \\
&= \mathsf{vc}[\![\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C_1)]\!](\mathsf{vc}[\![\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C_2)]\!](\varphi))(\sigma) && \text{(definition of vc)} \\
&\leq \mathsf{wlp}[\![C_1]\!](\mathsf{vc}[\![\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C_2)]\!](\varphi))(\sigma) && \text{(induction hypothesis)} \\
&\leq \mathsf{wlp}[\![C_1]\!](\mathsf{wlp}[\![C_2]\!]([\![\varphi]\!]))(\sigma) && \text{(induction hypothesis)} \\
&= \mathsf{wlp}[\![C_1\,;\, C_2]\!]([\![\varphi]\!])(\sigma) && \text{(definition of wlp)}
\end{aligned}$$

- $C = \{C_1\}\ [p]\ \{C_2\}$.

$$\begin{aligned}
&\mathsf{vc}[\![\{C_1\}\ [p]\ \{C_2\}]\!](\varphi)(\sigma) \\
&= (p \cdot \mathsf{vc}[\![\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C_1)]\!](\varphi) + (1-p) \cdot \mathsf{vc}[\![\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C_2)]\!](\varphi))(\sigma) && \text{(Lemma 3.13)} \\
&\leq (p \cdot \mathsf{wlp}[\![C_1]\!]([\![\varphi]\!]) + (1-p) \cdot \mathsf{vc}[\![\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C_2)]\!](\varphi))(\sigma) && \text{(induction hypothesis)} \\
&\leq (p \cdot \mathsf{wlp}[\![C_1]\!]([\![\varphi]\!]) + (1-p) \cdot \mathsf{wlp}[\![C_2]\!]([\![\varphi]\!]))(\sigma) && \text{(induction hypothesis)} \\
&= \mathsf{wlp}[\![\{C_1\}\ [p]\ \{C_2\}]\!]([\![\varphi]\!])(\sigma) && \text{(definition of wlp)}
\end{aligned}$$

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

$$\begin{aligned}
&\mathsf{vc}[\![\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(\texttt{if}\ (\sqcap)\ \{C_1\}\ \texttt{else}\ \{C_2\})]\!](\varphi)(\sigma) \\
&= \mathsf{vc}[\![\texttt{if}\ (\sqcap)\ \{\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C_1)\}\ \texttt{else}\ \{\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C_2)\}]\!](\varphi)(\sigma) && \text{(definition of } \mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}\text{)} \\
&= \mathsf{vc}[\![\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C_1)]\!](\varphi)(\sigma) \sqcap \mathsf{vc}[\![\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C_2)]\!](\varphi)(\sigma) && \text{(definition of vc)} \\
&\leq \mathsf{wlp}[\![C_1]\!]([\![\varphi]\!])(\sigma) \sqcap \mathsf{vc}[\![\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C_2)]\!](\varphi)(\sigma) && \text{(induction hypothesis)} \\
&\leq \mathsf{wlp}[\![C_1]\!]([\![\varphi]\!])(\sigma) \sqcap \mathsf{wlp}[\![C_2]\!]([\![\varphi]\!])(\sigma) && \text{(induction hypothesis)} \\
&= \mathsf{wlp}[\![\texttt{if}\ (\sqcap)\ \{C_1\}\ \texttt{else}\ \{C_2\}]\!]([\![\varphi]\!])(\sigma) && \text{(definition of wlp)}
\end{aligned}$$

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

$$
\begin{aligned}
&\mathsf{vc}[\![\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(\texttt{if}\ (b)\ \{C_1\}\ \texttt{else}\ \{C_2\})]\!](\varphi)(\sigma) \\
&= \begin{cases} \mathsf{vc}[\![\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C_1)]\!](\varphi)(\sigma), & \text{if } [\![b]\!](\sigma) \\ \mathsf{vc}[\![\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C_2)]\!](\varphi)(\sigma), & \text{else} \end{cases} && \text{(Lemma 3.9)} \\
&\leq \begin{cases} \mathsf{wlp}[\![C_1]\!]([\![\varphi]\!])(\sigma), & \text{if } [\![b]\!](\sigma) \\ \mathsf{vc}[\![\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C_2)]\!](\varphi)(\sigma), & \text{else} \end{cases} && \text{(induction hypothesis)} \\
&\leq \begin{cases} \mathsf{wlp}[\![C_1]\!]([\![\varphi]\!])(\sigma), & \text{if } [\![b]\!](\sigma) \\ \mathsf{wlp}[\![C_2]\!]([\![\varphi]\!])(\sigma), & \text{else} \end{cases} && \text{(induction hypothesis)} \\
&= ([b] \cdot \mathsf{wlp}[\![C_1]\!]([\![\varphi]\!]) + [\neg b] \cdot \mathsf{wlp}[\![C_2]\!]([\![\varphi]\!]))(\sigma) && \text{(rewriting)} \\
&= \mathsf{wlp}[\![\texttt{if}\ (b)\ \{C_1\}\ \texttt{else}\ \{C_2\}]\!]([\![\varphi]\!])(\sigma) && \text{(definition of wlp)}
\end{aligned}
$$

- $C = \texttt{while}\ (b)\ \texttt{invariant}\ I\ \{C'\}$.

$$
\begin{aligned}
&\mathsf{vc}[\![\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(\texttt{while}\ (b)\ \texttt{invariant}\ I\ \{C'\})]\!](\varphi)(\sigma) \\
&\leq \mathsf{wlp}[\![\texttt{while}\ (b)\ \{C'\}]\!]([\![\varphi]\!])(\sigma) && \text{(Theorem 3.26 with induction hypothesis)}
\end{aligned}
$$

Therefore, $\mathit{trans}^{\mathsf{wlp}}_{\mathsf{down}}(C)$ is a down-approximation of $\mathsf{wlp}[\![C]\!]$ for all $C \in \mathsf{pGCL}$ by the principle of induction over the program structure of C. The proof for $\mathit{trans}^{\mathsf{wp}}_{\mathsf{up}}(C)$ is analogous. □

As the conclusion of this chapter, we show a complete HeyVL program that can be used to show that $c + 1$ is a valid upper bound for $\mathsf{wp}[\![C_{\text{geo}}]\!](c)$, where C_{geo} is the geometric loop program from Example 3.4.

Example 3.38. Geometric loop: Verifying an invariant with Park induction

The following HeyVL program S verifies $(\mathsf{vc}[\![S]\!](\infty) \equiv \infty)$ if and only if $c + 1$ is a valid upper bound for $\mathsf{wp}[\![C_{\text{geo}}]\!](c)$.

Because we are verifying upper bounds, we enclose the program by `down negate` and `up negate` (cf. Theorem 2.36). The `up assume` $c + 1$ at the start and the `up assert` c at the end are used to encode the query $\mathsf{vc}[\![S']\!](c) \sqsubseteq c + 1$ (cf. Theorem 2.34).

For the loop encoding, we use Park induction with the invariant

$$I = \quad [x = 1] \cdot (c + 1) + [x \neq 1] \cdot c.$$

The evaluation of $\mathsf{vc}[\![S]\!](\infty)$ is provided in the comments in between the lines and starts from the bottom, proceeding upwards. Since $\mathsf{vc}[\![S]\!](\infty) \equiv \infty$, we have shown that $c + 1$ is a valid upper bound for $\mathsf{wp}[\![C_{\text{geo}}]\!](c)$.

```
// ∞
```

```
// ¬0
down negate;
// 0
// (c + 1) ← ([x = 1] · (c + 1) + [x ≠ 1] · c)
up assume c + 1;
// [x = 1] · (c + 1) + [x ≠ 1] · c
// ([x = 1] · (c + 1) + [x ≠ 1] · c) ⊔ ⨆_{c,x} 0
up assert [x = 1] · (c + 1) + [x ≠ 1] · c;
// ⨆_{Vars} 0
up havoc Vars;
// 0
// ([x = 1] · (c + 1) + [x ≠ 1] · c) ↖ ([x = 1] · (c + 1) + [x ≠ 1] · c)
up compare [x = 1] · (c + 1) + [x ≠ 1] · c;
// [x = 1] · (c + 1) + [x ≠ 1] · c
// (?(x = 1) → (c + 1) ⊓ (?(x ≠ 1) → c)
if (⊓) {
  // ?(x = 1) → (c + 1)
  // ?(x = 1) → (c + 0.5 · (1 + [x = 1]))
  down assume ?(x = 1);
  // c + 0.5 · (1 + [x = 1])
  // 0.5 · c + 0.5 · (c + 1 + [x = 1])
  choice :≈ ber!(0.5);
  // (?(choice = 1) → c) ⊓ (?(choice ≠ 1) → (c + 1 + [x = 1]))
  if (⊓) {
    // ?(choice = 1) → c
    down assume ?(choice = 1);
    // c
    // [0 = 1] · (c + 1) + [0 ≠ 1] · c
    x :≈ 0
    // [x = 1] · (c + 1) + [x ≠ 1] · c
  } else {
    // ?(choice ≠ 1) → (c + 1 + [x = 1])
    down assume ?(choice ≠ 1);
    // c + 1 + [x = 1]
    // [x = 1] · (c + 2) + [x ≠ 1] · (c + 1)
    c :≈ c + 1
```

```
      // [x = 1] · (c + 1) + [x ≠ 1] · c
    }
    // [x = 1] · (c + 1) + [x ≠ 1] · c
    // ([x = 1] · (c + 1) + [x ≠ 1] · c) ⊓ ∞
    down assert [x = 1] · (c + 1) + [x ≠ 1] · c;
    // ∞
    // 0 → c
    down assume 0
    // c
  } else {
    // ?(x ≠ 1) → c
    down assume ?(x ≠ 1)
    // c
  };

  // c
  // c ⊔ 0
  up assert c;
  // 0
  // ~∞
  up negate
  // ∞
```

Using 2-induction with the simple invariant $I = c{+}1$, one can also show that $\mathsf{wp}[\![C_{\text{geo}}]\!](c) \sqsubseteq c + 1$ holds. The 2-induction encoding is one of our examples we use in the evaluation of our implementation *Caesar* in Section 4.3.

4. Automation

In the previous chapters, we introduced HeyLo and HeyVL and have seen how to encode verification problems of pGCL programs in HeyVL. In this chapter, we look at the *automation* of deductive verification of HeyVL programs with the goal of implementing an automated deductive verifier for HeyVL programs. We develop *Caesar*, a tool written in Rust that takes a HeyVL program, generates verification conditions, and checks their validity using the theorem prover Z3 [dMB08].

While the generation of verification conditions in Caesar mostly follows the definition of the vc transformer (Definition 2.25), the validity checking of HeyLo formulas is more involved. Recall that a HeyLo formula $\varphi \in \mathsf{HeyLo}$ is *valid* if and only if $\varphi \equiv \infty$. Because of the complexity of automated theorem proving, we want to utilize the considerable engineering effort that went into automated theorem provers like Z3. In Section 4.1, we present an encoding of this validity problem for Z3. However, the encoding of quantifiers ($\bigsqcup_x$ and $\bigsqcap_x$) in HeyLo formulas is naive and led to Z3 returning "unknown" on many inputs.

Therefore, we develop *quantifier elimination* for HeyLo in Section 4.2: We show that for many formulas $\varphi \in \mathsf{HeyLo}$, there is a quantifier-free formula $\varphi' \in \mathsf{HeyLo}$ that is valid if and only if φ is valid. We also identify a fragment of HeyVL for which quantifier elimination of the corresponding verification conditions is always possible. This fragment includes the HeyVL encodings for pGCL programs presented in the previous section, with a restriction on the quantifiers that may occur in the assertions. Finally, we discuss the implementation of Caesar in Section 4.3.

4.1. Encoding HeyLo into SMT-LIB

Z3 is an automated theorem prover that is widely used in software verification [dMB08]. On a high level, Z3 accepts a mathematical formula and tries to determine whether it is satisfiable or unsatisfiable. In some cases, Z3 is not complete and returns "unknown". Z3 accepts input in the *SMT-LIB* format, a language to specify theories, logics, and inputs to SMT solvers like Z3 [BFT17].

Our goal is to provide an encoding in SMT-LIB that is unsatisfiable if and only if the verification conditions in the form of a HeyLo formula are valid. More generally, we

develop an encoding that is unsatisfiable if and only if $\varphi \equiv \infty$ for a given $\varphi \in \mathsf{HeyLo}$. Our approach is simple: We encode the HeyLo formula φ directly in SMT-LIB, so that there is an SMT-LIB formula f that evaluates to $[\![\varphi]\!](\sigma) \in \mathbb{R}^{\infty}_{\geq 0}$ in state $\sigma \in \Sigma$. For this, we define a custom datatype `RealPlus` in SMT-LIB that represents values in $\mathbb{R}^{\infty}_{\geq 0}$. Then, the final query checks encodes $\neg(\infty \sqsubseteq f)$ – which is unsatisfiable if and only if $\varphi \equiv \infty$.

Notation Throughout this section, we avoid the use of SMT-LIB syntax as much as possible for readability. For example, instead of `(forall ((x RealPlus)) f)`, we write $\forall x \in \texttt{RealPlus}.\, f$. For function applications, we write $f(x, y, z)$ instead of `(f x y z)`. Similarly, we use

$$\begin{cases} g, & \text{if } f \\ h, & \text{else} \end{cases} \quad \text{instead of} \quad \texttt{(ite f g h)}\,.$$

The RealPlus Datatype *Z3* does not support a *sort* (i.e. a type) that represents values in $\mathbb{R}^{\infty}_{\geq 0}$ out of the box. We define a custom datatype that either represents infinity or a (non-negative) real value. The following SMT-LIB declaration defines `RealPlus`, our representation of $\mathbb{R}^{\infty}_{\geq 0}$:

```
(declare-datatype RealPlus
  (  (infinity)
     (finite (value Real)) )
```

This declaration creates a sort `RealPlus` with two associated symbols:

$$\texttt{infinity}\colon \texttt{RealPlus}$$
$$\texttt{finite}\colon \mathbb{R} \to \texttt{RealPlus}$$

That means the formula `infinity` has sort `RealPlus` and represents ∞. `(finite x)` with $x \in \mathbb{R}$ also has sort `RealPlus` and represents a finite value. We always assume that x is non-negative. For this, we add constraints to free variables ensuring that $x \geq 0$. We write x with $x \in \mathbb{R}$ to mean `(finite x)` and we write ∞ instead of `infinity`.

Furthermore, the declaration creates the *selector* symbol `value` that has sort `Real`. The SMT-LIB formula `(value (finite x))` evaluates to `x`. However, `(value infinity)` is undefined.

To test whether a formula of sort `RealPlus` evaluates to infinity or a finite value, we can use the automatically declared *testers*. They are written as `(_ is infinity)` and `(_ is finite)`. The SMT-LIB formula `((_ is infinity) x)` evaluates to `true` when `x` evaluates to infinity. Similar for `(_ is finite)`. We abbreviate `((_ is infinity) x)` by $x = \infty$. The notation $x \neq \infty$ is used to abbreviate `((_ is value) x)`.

Ordering RealPlus We define the ordering on `RealPlus` in a natural way:

$$\begin{aligned} \sqsubseteq &: \texttt{RealPlus} \times \texttt{RealPlus} \to \texttt{Bool} \\ x \sqsubseteq y &= (y = \infty) \vee (x \neq \infty \wedge \texttt{value}(x) \leq \texttt{value}(y)) \end{aligned}$$

This definition respects the ordering of $\mathbb{R}^{\infty}_{\geq 0}$: $x \sqsubseteq \infty$ always evaluates to true. $\infty \sqsubseteq y$ evaluates to true if and only if $y = \infty$. And when $x \neq \infty$ and $y \neq \infty$, then $x \sqsubseteq y$ evaluates to true if and only if $\texttt{value}(x) \leq \texttt{value}(y)$.

Arithmetic on RealPlus For probabilistic choices, HeyLo supports the addition operator $+$ and the multiplication operator $\cdot$. We encode them in SMT-LIB using case distinctions.

$$+ : \texttt{RealPlus} \times \texttt{RealPlus} \to \texttt{RealPlus}$$

$$x + y = \begin{cases} \infty, & \text{if } x = \infty \vee y = \infty \\ \texttt{value}(x) + \texttt{value}(y), & \text{else} \end{cases}$$

$$\cdot : \texttt{RealPlus} \times \texttt{RealPlus} \to \texttt{RealPlus}$$

$$x \cdot y = \begin{cases} \texttt{value}(x) \cdot \texttt{value}(y), & \text{if } x \neq \infty \wedge y \neq \infty \\ 0, & \text{if } x = 0 \vee y = 0 \\ \infty, & \text{otherwise} \end{cases}$$

In SMT-LIB syntax, $\texttt{value}(x) + \texttt{value}(y)$ is written as `(finite (+ (value x) (value y)))`. The case distinction with three cases is to be read as a nested `ite`.

The encoding of the addition operator $+$ is simple: If any operand is infinite, the result is infinity. If both operands are finite, the result is given by the addition of their real number values.

For the multiplication, we distinguish between three cases. If both operands are finite, the result is given by the multiplication on real numbers. If any operand is zero (and the other is infinite), then the result is zero. Otherwise, both operands are infinite and therefore the result is as well.

Unary and Binary HeyLo Operators on RealPlus The encodings of the unary and binary HeyLo operators like $\to$ and $\leftarrow$ quite trivially follow the semantics provided in Definition 2.18. For example, the implication is defined as follows:

$$\to : \texttt{RealPlus} \times \texttt{RealPlus} \to \texttt{RealPlus}$$

$$x \to y = \begin{cases} \infty, & \text{if } x \sqsubseteq y \\ y, & \text{else} \end{cases}$$

The negation $\neg x$ is defined like this:

$$\neg\colon \texttt{RealPlus} \to \texttt{RealPlus}$$
$$\neg x = \begin{cases} \infty, & \text{if } x = 0 \\ 0, & \text{else} \end{cases}$$

All other unary and binary operators are encoded analogously. In particular, the binary infimum $\sqcap$ and supremum $\sqcup$ operators are also encoded with an if-then-else, using the fact that $\mathbb{R}^{\infty}_{\geq 0}$ is totally ordered. For example,

$$\sqcap\colon \texttt{RealPlus} \times \texttt{RealPlus} \to \texttt{Bool}$$
$$x \sqcap y = \begin{cases} x, & \text{if } x \sqsubseteq y \\ y, & \text{else} \end{cases}$$

The encoding of the Iverson bracket $[b]$ and the embedding function $?(b)$ is also done using if-then-else.

Limits on RealPlus For the encoding of the HeyLo quantifiers ($\bigsqcup_x$ and $\bigsqcap_x$), we need to encode the supremum and infimum over `RealPlus`, respectively. For each occurrence φ_i of a HeyLo quantifier, we generate a new symbol $limit_i$ of sort `RealPlus`. Then we add constraints to the problem that specify that $limit_i$ evaluates to the corresponding limit. For example, $\varphi_i = \bigsqcup_x \varphi$ results in a new symbol $limit_i$ with the following two additional constraints:

1. φ_i is a lower bound on φ for all x:

$$\forall x \in \mathbb{Q}_{\geq 0}. \quad limit_i \sqsubseteq \varphi$$

2. φ_i is the greatest lower bound:

$$\forall x \in \mathbb{Q}_{\geq 0}.\ \forall other \in \texttt{RealPlus}. \quad other \sqsubseteq \varphi \Rightarrow other \sqsubseteq limit_i$$

   where *other* is another fresh symbol of sort `RealPlus` to quantify over all lower bounds of φ.

Note that Z3 does not support a non-negative rational number type directly to represent $x \in \mathbb{Q}_{\geq 0}$, but we can implement it using the rational number type with the additional constraints $x \geq 0$ and that x can be represented as the result of a division of two integers. Since the two constraints above directly implement the definition of an infimum, it is obvious that this encoding is correct. The encoding of $\bigsqcap_x \varphi$ is dual.

4.2. Quantifier Elimination for HeyLo

HeyLo formulas that contain quantifiers, i.e. $\bigsqcap_x$ and $\bigsqcup_x$, have been problematic with our SMT-LIB encoding and Z3. When checking validity by using the encoding of the previous section, Z3 was often unable to prove either satisfiability or unsatisfiability. In this section, we present a general method to eliminate (some) quantifiers that occur in HeyLo formulas. More formally, given a HeyLo formula $\varphi \in \mathsf{HeyLo}$, we have a quantifier elimination method that returns a formula $\varphi' \in \mathsf{HeyLo}$ without or with fewer quantifiers than before. Importantly, φ is valid ($\varphi \equiv \infty$) if and only if φ' is valid as well. Therefore, we can use our quantifier elimination procedure as a pre-processing step before the translation to SMT-LIB to simplify the validity checking.

This section uses the important observation that

$$\bigsqcap_x \varphi \text{ is valid} \quad \text{iff} \quad \varphi \text{ is valid.} \tag{Theorem 4.2}$$

By repeatedly moving infimum operators to the front of a formula, we can extract the body of the formula without the leading quantifier. Note that there is no valid logical equivalence for $\bigsqcup_x$ (cf. Section 2.2.4).

In the following subsections, we state various theorems that are used to shift HeyLo quantifiers to the front of a formula while preserving equivalence to the original HeyLo formula. For example, if $y \notin \mathsf{free}(\varphi)$:

$$\varphi \to \bigsqcap_y \psi \equiv \bigsqcap_y (\varphi \to \psi) \tag{Theorem 4.4}$$

Collecting these rules, we describe a subset of HeyLo formulas (Theorem 4.14) where all quantifiers can be moved to the front such that the resulting formula starts with a sequence of infimum operators and then a quantifier-free body. This subset is called *infimum-prenexable*. Using Theorem 4.2, we can thus eliminate all quantifiers. Finally, we identify a corresponding subset of HeyVL and post-expectation $\varphi \in \mathsf{HeyLo}$ such that $\mathsf{vc}[\![S]\!](\varphi)$ is infimum-prenexable (Definition 4.16). Note that our infimum-prenexable subset is not maximal: there are more HeyLo formulas where prenexing is possible. However, we have found this fragment sufficient for our examples so far.

For the above equivalence, as well as for many of the following proofs, we use the following lemma about complete lattices.

Lemma 4.1. Complete lattice quantifier rules

Let $(L, \sqsubseteq)$ be a complete lattice, and let $x \in L$ and $S \subseteq L$. Then the following equivalences hold:

$$\begin{aligned} x \sqsubseteq \inf S \quad &\text{iff} \quad \forall s \in S.\ \ x \sqsubseteq s \\ \sup S \sqsubseteq x \quad &\text{iff} \quad \forall s \in S.\ \ s \sqsubseteq x \end{aligned}$$

Proof. Let $(L, \sqsubseteq)$ be a complete lattice, and let $x \in L$ and $S \subseteq L$.

Case inf**:** ($\Rightarrow$) Assume $x \sqsubseteq \inf S$. By the definition of infimum, $\bigsqcap S$ is a lower bound for all $s \in S$, i.e. $\inf S \sqsubseteq s$. By transitivity of $\sqsubseteq$, $x \sqsubseteq s$ for all $s \in S$.

($\Leftarrow$) Assume $\forall s \in S.\ x \sqsubseteq s$. That means x is a lower bound for S. Since $\inf S$ is the *greatest* lower bound of S, it follows that $x \sqsubseteq \inf S$.

Case sup**:** ($\Rightarrow$) Assume $\sup S \sqsubseteq x$. By the definition of supremum, $\sup S$ is an upper bound for all $s \in S$, i.e. $s \sqsubseteq \sup S$. By transitivity of $\sqsubseteq$, $s \sqsubseteq x$ for all $s \in S$.

($\Leftarrow$) Assume $\forall s \in S.\ s \sqsubseteq x$. That means x is an upper bound for S. Since $\sup S$ is the *least* upper bound of S, it follows that $\sup S \sqsubseteq x$. □

The theorem about quantifier elimination follows from Lemma 4.1 and the definition of HeyLo semantics.

Theorem 4.2. Quantifier elimination for validity

Let $\varphi \in \mathsf{HeyLo}$.

$$\bigsqcap_x \varphi \text{ is valid} \quad \text{iff} \quad \varphi \text{ is valid.}$$

Proof. Let $\varphi \in \mathsf{HeyLo}$. Then:

$$\begin{aligned}
& \bigsqcap_x \varphi \text{ is valid} \\
\text{iff} \quad & [\![\bigsqcap_x \varphi]\!](\sigma) = \infty && \forall \sigma \in \Sigma && \text{(definition of validity)} \\
\text{iff} \quad & \infty \sqsubseteq [\![\bigsqcap_x \varphi]\!](\sigma) && \forall \sigma \in \Sigma \\
\text{iff} \quad & \infty \sqsubseteq \inf \{\, [\![\varphi]\!](\sigma[x \mapsto v]) \mid v \in \mathbb{Q}_{\geq 0} \,\} && \forall \sigma \in \Sigma && \text{(definition of } \textstyle\bigsqcap_x\text{)} \\
\text{iff} \quad & \infty \sqsubseteq [\![\varphi]\!](\sigma[x \mapsto v]) && \forall \sigma \in \Sigma,\ \forall v \in \mathbb{Q}_{\geq 0} && \text{(Lemma 4.1)} \\
\text{iff} \quad & \infty \sqsubseteq [\![\varphi]\!](\sigma) && \forall \sigma \in \Sigma \\
\text{iff} \quad & [\![\varphi]\!](\sigma) = \infty && \forall \sigma \in \Sigma \\
\text{iff} \quad & \varphi \text{ is valid}
\end{aligned}$$

□

The following lemma will be used often to rewrite limits over sets with a guard P. The guard is a function $P\colon S \to \mathbb{B}$ that decides whether element $s \in S$ should not be included in the limit.[Fixed] An if-then-else that returns $\top$ respectively $\bot$ when $P(s)$ is true can also be used. We write $\mathbf{ite}(P,\ a,\ b)$ as shorthand notation for an if-then-else:

$$\mathbf{ite}(P,\ a,\ b) = \begin{cases} a, & \text{if } P \\ b, & \text{else} \end{cases}$$

Lemma 4.3. Limits with filters to if-then-else
Let L be a complete lattice, $S \subseteq L$ and $P\colon S \to \mathbb{B}$ a function. Then:

$$\inf \{\, s \in S \mid \neg P(s) \,\} = \inf \{\, \mathbf{ite}(P(s),\ \top,\ s) \mid s \in S \,\}$$
$$\sup \{\, s \in S \mid \neg P(s) \,\} = \sup \{\, \mathbf{ite}(P(s),\ \bot,\ s) \mid s \in S \,\}$$

Proof. Let L be a complete lattice, $S \subseteq L$ and $P\colon S \to \mathbb{B}$.

$$\begin{aligned} \inf \{\, s \in S \mid \neg P(s) \,\} &= \inf (\{\, s \in S \mid \neg P(s) \,\} \cup \{\, \top \mid s \in S,\ P(s) \,\}) && \text{(add } \top\text{)} \\ &= \inf \{\, \mathbf{ite}(P(s),\ \top,\ s) \mid s \in S \,\} \end{aligned}$$

The case for sup is dual. □

We now start with various lemmas with equivalences that will be used to identify the infimum-prenexable fragment of HeyLo (Theorem 4.14).

4.2.1. Prenexing in Positive Positions

We now show several prenexing rules to extract quantifiers out of so-called *positive positions* in HeyLo formulas. Positive positions are those where the quantifier can be extracted without changing the quantifier itself. This is contrasted to *negative positions* where, e.g. an inner $\bigsqcup_x$ becomes an outer $\bigsqcap_x$. We proceed with rules for negative positions in Section 4.2.2.

Theorem 4.4. Prenexing in positive positions
Let $\varphi, \psi \in \mathsf{HeyLo}$ and $x, y \in \mathsf{Vars}$ where $x \notin \mathsf{free}(\psi)$ and $y \notin \mathsf{free}(\varphi)$.

$$\begin{aligned} \bigsqcap_x \varphi \sqcap \psi &\equiv \bigsqcap_x (\varphi \sqcap \psi) & \bigsqcup_x \varphi \sqcup \psi &\equiv \bigsqcup_x (\varphi \sqcup \psi) \\ \varphi \sqcap \bigsqcap_y \psi &\equiv \bigsqcap_y (\varphi \sqcap \psi) & \varphi \sqcup \bigsqcup_y \psi &\equiv \bigsqcup_y (\varphi \sqcup \psi) \\ \bigsqcap_x \varphi \sqcup \psi &\equiv \bigsqcap_x (\varphi \sqcup \psi) & \bigsqcup_x \varphi \sqcap \psi &\equiv \bigsqcup_x (\varphi \sqcap \psi) \\ \varphi \sqcup \bigsqcap_y \psi &\equiv \bigsqcap_y (\varphi \sqcup \psi) & \varphi \sqcap \bigsqcup_y \psi &\equiv \bigsqcup_y (\varphi \sqcap \psi) \\ \varphi \to \bigsqcap_y \psi &\equiv \bigsqcap_y (\varphi \to \psi) & \varphi \leftarrow \bigsqcup_y \psi &\equiv \bigsqcup_y (\varphi \leftarrow \psi) \\ \varphi \searrow \bigsqcap_y \psi &\equiv \bigsqcap_y (\varphi \searrow \psi) & \varphi \nwarrow \bigsqcup_y \psi &\equiv \bigsqcup_y (\varphi \nwarrow \psi) \end{aligned}$$

For the proof, we first show the high-level structural induction and refer to smaller lemmas below to show equivalences of individual cases.[Fixed]

Proof. Let $\varphi, \psi \in \mathsf{HeyLo}$ and $x, y \in \mathsf{Vars}$ where $x \notin \mathsf{free}(\psi)$ and $y \notin \mathsf{free}(\varphi)$, and $\sigma \in \Sigma$. Recall that $\varphi \equiv \psi$ iff $[\![\varphi]\!](\sigma) = [\![\psi]\!](\sigma)$ for all $\sigma \in \Sigma$.

$$
\begin{aligned}
&[\![\textstyle\bigsqcap_x \varphi \sqcap \psi]\!](\sigma) \\
&= \inf\left\{ [\![\textstyle\bigsqcap_x \varphi]\!](\sigma),\ [\![\psi]\!](\sigma) \right\} && \text{(definition of } \sqcap) \\
&= \inf\left\{ \inf\left\{ [\![\varphi]\!](\sigma[x \mapsto v]) \mid v \in \mathbb{Q}_{\geq 0} \right\},\ [\![\psi]\!](\sigma) \right\} && \text{(definition of } \textstyle\bigsqcap_x) \\
&= \inf\left\{ [\![\varphi]\!](\sigma[x \mapsto v]),\ [\![\psi]\!](\sigma) \mid v \in \mathbb{Q}_{\geq 0} \right\} && \text{(merge inf)} \\
&= \inf\left\{ [\![\varphi]\!](\sigma[x \mapsto v]),\ [\![\psi]\!](\sigma[x \mapsto v]) \mid v \in \mathbb{Q}_{\geq 0} \right\} && (x \notin \mathsf{free}(\psi)) \\
&= \inf\left\{ \inf\left\{ [\![\varphi]\!](\sigma[x \mapsto v]),\ [\![\psi]\!](\sigma[x \mapsto v]) \right\} \mid v \in \mathbb{Q}_{\geq 0} \right\} && \text{(introduce inf)} \\
&= \inf\left\{ [\![\varphi \sqcap \psi]\!](\sigma[x \mapsto v]) \mid v \in \mathbb{Q}_{\geq 0} \right\} && \text{(definition of } \sqcap) \\
&= [\![\textstyle\bigsqcap_x (\varphi \sqcap \psi)]\!](\sigma) && \text{(definition of } \textstyle\bigsqcap_x)
\end{aligned}
$$

The proof of $\varphi \sqcap \bigsqcap_y \psi \equiv \bigsqcap_y (\varphi \sqcap \psi)$ is analogous and omitted here.

In the next case, we use distributivity. We will show this as a more general property in Chapter 5 (cf. Theorem 5.4).

$$
\begin{aligned}
&[\![\textstyle\bigsqcap_x \varphi \sqcup \psi]\!](\sigma) \\
&= \sup\left\{ [\![\textstyle\bigsqcap_x \varphi]\!](\sigma),\ [\![\psi]\!](\sigma) \right\} && \text{(definition of } \sqcup) \\
&= \sup\left\{ \inf\left\{ [\![\varphi]\!](\sigma[x \mapsto v]) \mid v \in \mathbb{Q}_{\geq 0} \right\},\ [\![\psi]\!](\sigma) \right\} && \text{(definition of } \textstyle\bigsqcap_x) \\
&= \inf\left\{ \sup\left\{ [\![\varphi]\!](\sigma[x \mapsto v]),\ [\![\psi]\!](\sigma) \right\} \mid v \in \mathbb{Q}_{\geq 0} \right\} && \text{(distributivity)} \\
&= \inf\left\{ \sup\left\{ [\![\varphi]\!](\sigma[x \mapsto v]),\ [\![\psi]\!](\sigma[x \mapsto v]) \right\} \mid v \in \mathbb{Q}_{\geq 0} \right\} && (x \notin \mathsf{free}(\psi)) \\
&= \inf\left\{ [\![\varphi \sqcup \psi]\!](\sigma[x \mapsto v]) \mid v \in \mathbb{Q}_{\geq 0} \right\} && \text{(definition of } \sqcup) \\
&= [\![\textstyle\bigsqcap_x (\varphi \sqcup \psi)]\!](\sigma) && \text{(definition of } \textstyle\bigsqcap_x)
\end{aligned}
$$

The proof of $\varphi \sqcup \bigsqcap_y \psi \equiv \bigsqcap_y (\varphi \sqcup \psi)$ is analogous and omitted here.

The infimum can be moved out of the positive position of the implication $\to$ as well.

$$
\begin{aligned}
&[\![\varphi \to \textstyle\bigsqcap_y \psi]\!](\sigma) \\
&= \begin{cases} \infty, & \text{if } [\![\varphi]\!](\sigma) \sqsubseteq [\![\textstyle\bigsqcap_y \psi]\!](\sigma) \\ [\![\textstyle\bigsqcap_y \psi]\!](\sigma), & \text{else} \end{cases} && \text{(definition of } \to)
\end{aligned}
$$

$$
\begin{aligned}
&= \begin{cases} [\![\bigsqcap_y(\varphi \to \psi)]\!](\sigma), & \text{if } [\![\varphi]\!](\sigma) \sqsubseteq [\![\bigsqcap_y \psi]\!](\sigma) \\ [\![\bigsqcap_y \psi]\!](\sigma), & \text{else} \end{cases} && \text{(Lemma 4.5)} \\
&= \begin{cases} [\![\bigsqcap_y(\varphi \to \psi)]\!](\sigma), & \text{if } [\![\varphi]\!](\sigma) \sqsubseteq [\![\bigsqcap_y \psi]\!](\sigma) \\ [\![\bigsqcap_y(\varphi \to \psi)]\!](\sigma), & \text{else} \end{cases} && \text{(Lemma 4.6)} \\
&= [\![\bigsqcap_y(\varphi \to \psi)]\!](\sigma) && \text{(same value in both cases)}
\end{aligned}
$$

The same holds for the hard implication $\searrow$.

$$
\begin{aligned}
&[\![\varphi \searrow \bigsqcap_y \psi]\!](\sigma) \\
&= \begin{cases} \infty, & \text{if } [\![\varphi]\!](\sigma) \sqsubseteq [\![\bigsqcap_y \psi]\!](\sigma) \\ 0, & \text{else} \end{cases} && \text{(definition of } \searrow) \\
&= \begin{cases} [\![\bigsqcap_y(\varphi \searrow \psi)]\!](\sigma), & \text{if } [\![\varphi]\!](\sigma) \sqsubseteq [\![\bigsqcap_y \psi]\!](\sigma) \\ 0, & \text{else} \end{cases} && \text{(Lemma 4.5)} \\
&= \begin{cases} [\![\bigsqcap_y(\varphi \searrow \psi)]\!](\sigma), & \text{if } [\![\varphi]\!](\sigma) \sqsubseteq [\![\bigsqcap_y \psi]\!](\sigma) \\ [\![\bigsqcap_y(\varphi \searrow \psi)]\!](\sigma), & \text{else} \end{cases} && \text{(Lemma 4.7)} \\
&= [\![\bigsqcap_y(\varphi \searrow \psi)]\!](\sigma) && \text{(same value in both cases)}
\end{aligned}
$$

The other cases are dual and omitted here. □

With the high-level proof done, we proceed with the individual lemmas for equivalences in the cases.

Lemma 4.5. Positive prenexing: implications if case

Let $\varphi, \psi \in \mathsf{HeyLo}$ and $\sigma \in \Sigma$ and $y \in \mathsf{Vars}$ where $y \notin \mathsf{free}(\varphi)$.

If $[\![\varphi]\!](\sigma) \sqsubseteq [\![\bigsqcap_y \psi]\!](\sigma)$:

$$\infty = [\![\bigsqcap_y(\varphi \to \psi)]\!](\sigma) = [\![\bigsqcap_y(\varphi \searrow \psi)]\!](\sigma)$$

If $[\![\bigsqcup_y \psi]\!](\sigma) \sqsubseteq [\![\varphi]\!](\sigma)$:

$$0 = [\![\bigsqcup_y(\varphi \leftarrow \psi)]\!](\sigma) = [\![\bigsqcup_y(\varphi \nwarrow \psi)]\!](\sigma)$$

Proof. Let $\varphi, \psi \in \mathsf{HeyLo}$ and $\sigma \in \Sigma$ and $y \in \mathsf{Vars}$ where $y \notin \mathsf{free}(\varphi)$. The proof is based on the fact that $\inf \emptyset = \infty$ (and dually, $\sup \emptyset = 0$). Assume $[\![\varphi]\!](\sigma) \sqsubseteq [\![\bigsqcap_y \psi]\!](\sigma)$.

From the assumption, we can deduce

$$\begin{aligned} & [\![\varphi]\!](\sigma) \sqsubseteq \inf\{\, [\![\psi]\!](\sigma[y \mapsto v]) \mid v \in \mathbb{Q}_{\geq 0} \,\} && \text{(definition of } \textstyle\bigsqcap_y\text{)} \\ \Rightarrow\quad & [\![\varphi]\!](\sigma) \sqsubseteq [\![\psi]\!](\sigma[y \mapsto v]) \quad \forall v \in \mathbb{Q}_{\geq 0} && \text{(Lemma 4.1)} \end{aligned}$$

Thus:

$$\begin{aligned} \infty &= \inf \emptyset \\ &= \inf\{\, [\![\psi]\!](\sigma[y \mapsto v]) \mid v \in \mathbb{Q}_{\geq 0},\ \neg([\![\varphi]\!](\sigma) \sqsubseteq [\![\psi]\!](\sigma[y \mapsto v])) \,\} && \text{(set is empty)} \\ &= \inf\{\, \mathbf{ite}([\![\varphi]\!](\sigma) \sqsubseteq [\![\psi]\!](\sigma[y \mapsto v]),\ \infty,\ [\![\psi]\!](\sigma[y \mapsto v])) \mid v \in \mathbb{Q}_{\geq 0} \,\} && \text{(Lemma 4.3)} \\ &= \inf\{\, \mathbf{ite}([\![\varphi]\!](\sigma[y \mapsto v]) \sqsubseteq [\![\psi]\!](\sigma[y \mapsto v]),\ \infty,\ [\![\psi]\!](\sigma[y \mapsto v])) \mid v \in \mathbb{Q}_{\geq 0} \,\} \\ & && (y \notin \mathsf{free}(\varphi)) \\ &= \inf\{\, [\![\varphi \to \psi]\!](\sigma[y \mapsto v]) \mid v \in \mathbb{Q}_{\geq 0} \,\} && \text{(definition of } \to\text{)} \\ &= [\![\textstyle\bigsqcap_y (\varphi \to \psi)]\!](\sigma) && \text{(definition of } \textstyle\bigsqcap_y\text{)} \end{aligned}$$

The same approach works for $\bigsqcap_y(\varphi \searrow \psi)$ and dually for the second statement. □

Lemma 4.6. Positive prenexing: implications else case

Let $\varphi, \psi \in \mathsf{HeyLo}$ and $\sigma \in \Sigma$ and $y \in \mathsf{Vars}$ where $y \notin \mathsf{free}(\varphi)$.

If $[\![\varphi]\!](\sigma) \sqsubseteq [\![\bigsqcap_y \psi]\!](\sigma)$ does not hold:

$$[\![\textstyle\bigsqcap_y \psi]\!](\sigma) = [\![\textstyle\bigsqcap_y (\varphi \to \psi)]\!](\sigma)$$

If $[\![\bigsqcup_y \psi]\!](\sigma) \sqsubseteq [\![\varphi]\!](\sigma)$ does not hold:

$$[\![\textstyle\bigsqcup_y \psi]\!](\sigma) = [\![\textstyle\bigsqcup_y (\varphi \leftarrow \psi)]\!](\sigma)$$

Proof. Let $\varphi, \psi \in \mathsf{HeyLo}$ and $\sigma \in \Sigma$ and $y \in \mathsf{Vars}$ where $y \notin \mathsf{free}(\varphi)$. Assume $[\![\varphi]\!](\sigma) \sqsubseteq [\![\bigsqcap_y \psi]\!](\sigma)$ does not hold.

We use some abbreviations:

$$\begin{aligned} S &= \{\, [\![\psi]\!](\sigma[y \mapsto v]) \mid v \in \mathbb{Q}_{\geq 0} \,\} \\ S_{<\varphi} &= \{\, [\![\psi]\!](\sigma[y \mapsto v]) \mid v \in \mathbb{Q}_{\geq 0},\ [\![\psi]\!](\sigma[y \mapsto v]) \sqsubsetneq [\![\varphi]\!](\sigma) \,\} \\ S_{\geq\varphi} &= \{\, [\![\psi]\!](\sigma[y \mapsto v]) \mid v \in \mathbb{Q}_{\geq 0},\ [\![\varphi]\!](\sigma) \sqsubseteq [\![\psi]\!](\sigma[y \mapsto v]) \,\} \end{aligned}$$

From the assumption, we can deduce the following:

$$\begin{aligned} & \neg([\![\varphi]\!](\sigma) \sqsubseteq \inf\{\, [\![\psi]\!](\sigma[y \mapsto v]) \mid v \in \mathbb{Q}_{\geq 0} \,\}) && \text{(definition of } \textstyle\bigsqcap_y\text{)} \\ \Rightarrow\quad & [\![\psi]\!](\sigma[y \mapsto v]) \sqsubsetneq [\![\varphi]\!](\sigma) \quad \exists v \in \mathbb{Q}_{\geq 0} && \text{(Lemma 4.1)} \\ \Rightarrow\quad & \inf S_{<\varphi} \sqsubseteq \inf S_{\geq\varphi} \end{aligned}$$

Now we rewrite $[\![\bigsqcap_y \psi]\!](\sigma)$ as $[\![\bigsqcap_y (\varphi \to \psi)]\!](\sigma)$.

$$
\begin{aligned}
&[\![\bigsqcap_y \psi]\!](\sigma) \\
&= \inf S && \text{(definition of } \textstyle\bigsqcap_y\text{)} \\
&= \inf(S_{<\varphi} \cup S_{\geq\varphi}) && (S_{<\varphi} \text{ and } S_{\geq\varphi} \text{ partition } S) \\
&= \inf S_{<\varphi} \sqcap \inf S_{\geq\varphi} && \text{(distributivity)} \\
&= \inf S_{<\varphi} && (\inf S_{<\varphi} \sqsubseteq \inf S_{\geq\varphi}) \\
&= \inf \{\, [\![\psi]\!](\sigma[y \mapsto v]) \mid v \in \mathbb{Q}_{\geq 0},\ \neg([\![\varphi]\!](\sigma[y \mapsto v]) \sqsubseteq [\![\psi]\!](\sigma[y \mapsto v])) \,\} && (y \notin \mathsf{free}(\varphi)) \\
&= \inf \{\, \mathbf{ite}([\![\varphi]\!](\sigma[y \mapsto v]) \sqsubseteq [\![\psi]\!](\sigma[y \mapsto v]),\ \infty,\ [\![\psi]\!](\sigma[y \mapsto v])) \mid v \in \mathbb{Q}_{\geq 0} \,\} \\
& && \text{(Lemma 4.3)} \\
&= \inf \{\, [\![\varphi \to \psi]\!](\sigma[y \mapsto v]) \mid v \in \mathbb{Q}_{\geq 0} \,\} && \text{(definition of } \to\text{)} \\
&= [\![\bigsqcap_y (\varphi \to \psi)]\!](\sigma) && \text{(definition of } \textstyle\bigsqcap_y\text{)}
\end{aligned}
$$

The proof for $\bigsqcup_y (\varphi \leftarrow \psi)$ with the other assumption is dual. □

Lemma 4.7. Positive prenexing: hard implications else case

Let $\varphi, \psi \in \mathsf{HeyLo}$ and $\sigma \in \Sigma$ and $y \in \mathsf{Vars}$ where $y \notin \mathsf{free}(\varphi)$.

If $[\![\varphi]\!](\sigma) \sqsubseteq [\![\bigsqcap_y \psi]\!](\sigma)$ does not hold:

$$0 = [\![\bigsqcap_y (\varphi \searrow \psi)]\!](\sigma)$$

If $[\![\bigsqcup_y \psi]\!](\sigma) \sqsubseteq [\![\varphi]\!](\sigma)$ does not hold:

$$\infty = [\![\bigsqcup_y (\varphi \nwarrow \psi)]\!](\sigma)$$

Proof. Let $\varphi, \psi \in \mathsf{HeyLo}$ and $\sigma \in \Sigma$ and $y \in \mathsf{Vars}$ where $y \notin \mathsf{free}(\varphi)$. Assume $[\![\varphi]\!](\sigma) \sqsubseteq [\![\bigsqcap_y \psi]\!](\sigma)$ does not hold.

From the assumption, we can deduce the following:

$$
\begin{aligned}
& \neg([\![\varphi]\!](\sigma) \sqsubseteq \inf \{\, [\![\psi]\!](\sigma[y \mapsto v]) \mid v \in \mathbb{Q}_{\geq 0} \,\}) && \text{(definition of } \textstyle\bigsqcap_y\text{)} \\
\Rightarrow\quad & \neg([\![\varphi]\!](\sigma) \sqsubseteq [\![\psi]\!](\sigma[y \mapsto v])) \quad \exists v \in \mathbb{Q}_{\geq 0} && \text{(Lemma 4.1)}
\end{aligned}
$$

With this fact, we can rewrite 0 as $[\![\bigsqcap_y(\varphi \searrow \psi)]\!](\sigma)$.

$$
\begin{aligned}
0 &= \inf\{\, 0 \mid v \in \mathbb{Q}_{\geq 0}, \neg([\![\varphi]\!](\sigma) \sqsubseteq [\![\psi]\!](\sigma[y \mapsto v])) \,\} \\
&= \inf\{\, 0 \mid v \in \mathbb{Q}_{\geq 0}, \neg([\![\varphi]\!](\sigma[y \mapsto v]) \sqsubseteq [\![\psi]\!](\sigma[y \mapsto v])) \,\} && (y \notin \mathsf{free}(\varphi)) \\
&= \inf\{\, \mathbf{ite}([\![\varphi]\!](\sigma[y \mapsto v]) \sqsubseteq [\![\psi]\!](\sigma[y \mapsto v]),\ \infty,\ 0) \mid v \in \mathbb{Q}_{\geq 0} \,\} && (\text{Lemma } 4.3) \\
&= \inf\{\, [\![\varphi \searrow \psi]\!](\sigma[y \mapsto v]) \mid v \in \mathbb{Q}_{\geq 0} \,\} && (\text{definition of } \searrow) \\
&= [\![\textstyle\bigsqcap_y(\varphi \searrow \psi)]\!](\sigma)
\end{aligned}
$$

The proof for the second statement is dual. □

4.2.2. Prenexing in Negative Positions

In addition to the prenexing rules for positive positions (Theorem 4.4) which allow to extend the scope of a quantifier, there are also three more rules that allow moving quantifiers from *negative* positions in the formula. *Negative positions* are the operands of negations and the left-hand sides of implications. In classical first-order logic, they correspond to the terms which are negated. Recall that we have $a \to b \equiv \neg a \vee b$ in classical logic and thus only need quantifier rules for negation. Quantifiers in negative positions can be moved out by transforming universals to existentials and vice versa. Again there are similar rules for HeyLo as in classical logic, but not for all quantifiers. For example, we have

$$(\bigsqcup_x \varphi) \to \psi \equiv \bigsqcap_x(\varphi \to \psi),$$

but not

$$(\bigsqcap_x \varphi) \to \psi \overset{\lightning}{\equiv} \bigsqcup_x(\varphi \to \psi).$$

Theorem 4.8. Prenexing in negative positions

Let $\varphi, \psi \in \mathsf{HeyLo}$ and $x, y \in \mathsf{Vars}$ where $x \notin \mathsf{free}(\psi)$ and $y \notin \mathsf{free}(\varphi)$.[Fixed]

$$
\begin{aligned}
(\textstyle\bigsqcup_x \varphi) \to \psi &\equiv \bigsqcap_x(\varphi \to \psi) & \qquad (\textstyle\bigsqcap_x \varphi) \leftarrow \psi &\equiv \bigsqcup_x(\varphi \leftarrow \psi) \\
(\textstyle\bigsqcup_x \varphi) \searrow \psi &\equiv \bigsqcap_x(\varphi \searrow \psi) & \qquad (\textstyle\bigsqcap_x \varphi) \nwarrow \psi &\equiv \bigsqcup_x(\varphi \nwarrow \psi) \\
\neg(\textstyle\bigsqcup_x \varphi) &\equiv \bigsqcap_x(\neg\varphi) & \qquad \sim(\textstyle\bigsqcap_x \varphi) &\equiv \bigsqcup_x(\sim\varphi)
\end{aligned}
$$

Proof. Let $\varphi, \psi \in \mathsf{HeyLo}$ and $x, y \in \mathsf{Vars}$ where $x \notin \mathsf{free}(\psi)$ and $y \notin \mathsf{free}(\varphi)$.

$$
\begin{aligned}
&[\![(\bigsqcup_x \varphi) \to \psi]\!](\sigma) \\
&= \begin{cases} \infty, & \text{if } [\![\bigsqcup_x \varphi]\!](\sigma) \sqsubseteq [\![\psi]\!](\sigma) \\ [\![\psi]\!](\sigma), & \text{else} \end{cases} && \text{(definition of } \to) \\
&= \begin{cases} [\![\bigsqcap_x (\varphi \to \psi)]\!](\sigma), & \text{if } [\![\bigsqcup_x \varphi]\!](\sigma) \sqsubseteq [\![\psi]\!](\sigma) \\ [\![\psi]\!](\sigma), & \text{else} \end{cases} && \text{(Lemma 4.9)} \\
&= \begin{cases} [\![\bigsqcap_x (\varphi \to \psi)]\!](\sigma), & \text{if } [\![\bigsqcup_x \varphi]\!](\sigma) \sqsubseteq [\![\psi]\!](\sigma) \\ [\![\bigsqcap_x (\varphi \to \psi)]\!](\sigma), & \text{else} \end{cases} && \text{(Lemma 4.10)} \\
&= [\![\bigsqcap_x (\varphi \to \psi)]\!](\sigma) && \text{(same value in both cases)}
\end{aligned}
$$

The case for the hard implication $\searrow$ is almost identical. The cases for $\leftarrow$ and $\nwarrow$ are dual.

For the negation, we can simply use the fact that $\neg\varphi \equiv \varphi \to 0$:

$$[\![\neg \bigsqcup_x \varphi]\!](\sigma) = [\![(\bigsqcup_x \varphi) \to 0]\!](\sigma) = [\![\bigsqcap_x (\varphi \to 0)]\!](\sigma) = [\![\bigsqcap_x \neg\varphi]\!](\sigma).$$

For the co-negation, we use $\sim\varphi \equiv \varphi \leftarrow \infty$:

$$[\![\sim \bigsqcap_x \varphi]\!](\sigma) = [\![(\bigsqcap_x \varphi) \leftarrow \infty]\!](\sigma) = [\![\bigsqcup_x (\varphi \leftarrow \infty)]\!](\sigma) = [\![\bigsqcup_x \sim\varphi]\!](\sigma).$$

□

Lemma 4.9. Negative prenexing: implications if case

Let $\varphi, \psi \in \mathsf{HeyLo}$ and $\sigma \in \Sigma$ and $x \in \mathsf{Vars}$ where $x \notin \mathsf{free}(\psi)$.

If $[\![\bigsqcup_x \varphi]\!](\sigma) \sqsubseteq [\![\psi]\!](\sigma)$:

$$\infty = [\![\bigsqcap_x (\varphi \to \psi)]\!](\sigma) = [\![\bigsqcap_x (\varphi \searrow \psi)]\!](\sigma)$$

If $[\![\psi]\!](\sigma) \sqsubseteq [\![\bigsqcap_x \varphi]\!](\sigma)$:

$$0 = [\![\bigsqcup_x (\varphi \leftarrow \psi)]\!](\sigma) = [\![\bigsqcup_x (\varphi \nwarrow \psi)]\!](\sigma)$$

Proof. Let $\varphi, \psi \in \mathsf{HeyLo}$ and $\sigma \in \Sigma$ and $x \in \mathsf{Vars}$ where $x \notin \mathsf{free}(\psi)$.
Assume $[\![\bigsqcup_x \varphi]\!](\sigma) \sqsubseteq [\![\psi]\!](\sigma)$.

From the assumption, we can deduce

$$
\begin{aligned}
& \sup \{\, [\![\varphi]\!](\sigma[x \mapsto v]) \mid v \in \mathbb{Q}_{\geq 0} \,\} \sqsubseteq [\![\psi]\!](\sigma) && \text{(definition of } \bigsqcup_x) \\
\Rightarrow \quad & [\![\varphi]\!](\sigma[x \mapsto v]) \sqsubseteq [\![\psi]\!](\sigma) \quad \forall v \in \mathbb{Q}_{\geq 0} && \text{(Lemma 4.1)}
\end{aligned}
$$

Thus:

$$\begin{aligned}
\infty &= \inf \emptyset \\
&= \inf \left\{ [\![\psi]\!](\sigma) \;\middle|\; v \in \mathbb{Q}_{\geq 0},\ \neg([\![\varphi]\!](\sigma[x \mapsto v]) \sqsubseteq [\![\psi]\!](\sigma)) \right\} && \text{(set is empty)} \\
&= \inf \left\{ \mathbf{ite}([\![\varphi]\!](\sigma[x \mapsto v]) \sqsubseteq [\![\psi]\!](\sigma),\ \infty,\ [\![\psi]\!](\sigma)) \;\middle|\; v \in \mathbb{Q}_{\geq 0} \right\} && \text{(Lemma 4.3)} \\
&= \inf \left\{ \mathbf{ite}([\![\varphi]\!](\sigma[x \mapsto v]) \sqsubseteq [\![\psi]\!](\sigma[x \mapsto v]),\ \infty,\ [\![\psi]\!](\sigma)) \;\middle|\; v \in \mathbb{Q}_{\geq 0} \right\} && (x \notin \mathsf{free}(\psi)) \\
&= \inf \left\{ [\![\varphi \to \psi]\!](\sigma[x \mapsto v]) \;\middle|\; v \in \mathbb{Q}_{\geq 0} \right\} && \text{(definition of } \to) \\
&= [\![\textstyle\bigsqcap_x (\varphi \to \psi)]\!](\sigma) && \text{(definition of } \textstyle\bigsqcap_x)
\end{aligned}$$

Similar for $\bigsqcap_x(\varphi \searrow \psi)$ and dual for the other assumption. □

Lemma 4.10. Negative prenexing: implications else case

Let $\varphi, \psi \in \mathsf{HeyLo}$ and $\sigma \in \Sigma$ and $x \in \mathsf{Vars}$ where $x \notin \mathsf{free}(\psi)$.

If $[\![\bigsqcup_x \varphi]\!](\sigma) \sqsubseteq [\![\psi]\!](\sigma)$ does not hold:

$$\begin{aligned}
[\![\psi]\!](\sigma) &= [\![\textstyle\bigsqcap_x (\varphi \to \psi)]\!](\sigma) \\
0 &= [\![\textstyle\bigsqcap_x (\varphi \searrow \psi)]\!](\sigma)
\end{aligned}$$

If $[\![\psi]\!](\sigma) \sqsubseteq [\![\bigsqcap_x \varphi]\!](\sigma)$ does not hold:

$$\begin{aligned}
[\![\psi]\!](\sigma) &= [\![\textstyle\bigsqcup_x (\varphi \leftarrow \psi)]\!](\sigma) \\
\infty &= [\![\textstyle\bigsqcup_x (\varphi \nwarrow \psi)]\!](\sigma)
\end{aligned}$$

Fixed

Proof. Let $\varphi, \psi \in \mathsf{HeyLo}$ and $\sigma \in \Sigma$ and $x \in \mathsf{Vars}$ where $x \notin \mathsf{free}(\psi)$.
Assume $[\![\bigsqcup_x \varphi]\!](\sigma) \sqsubseteq [\![\psi]\!](\sigma)$ does not hold.

$$\begin{aligned}
& \sup \left\{ [\![\varphi]\!](\sigma[x \mapsto v]) \;\middle|\; v \in \mathbb{Q}_{\geq 0} \right\} \not\sqsubseteq [\![\psi]\!](\sigma) && \text{(definition of } \textstyle\bigsqcup_x) \\
\Rightarrow\quad & [\![\varphi]\!](\sigma[x \mapsto v]) \not\sqsubseteq [\![\psi]\!](\sigma) \quad \exists v \in \mathbb{Q}_{\geq 0} && \text{(Lemma 4.1)}
\end{aligned}$$

Therefore, the set $\left\{ [\![\psi]\!](\sigma) \;\middle|\; [\![\varphi]\!](\sigma[x \mapsto v]) \not\sqsubseteq [\![\psi]\!](\sigma),\ v \in \mathbb{Q}_{\geq 0} \right\}$ is not empty. Thus,

$$\begin{aligned}
[\![\psi]\!](\sigma) &= \inf \left\{ [\![\psi]\!](\sigma) \;\middle|\; [\![\varphi]\!](\sigma[x \mapsto v]) \not\sqsubseteq [\![\psi]\!](\sigma),\ v \in \mathbb{Q}_{\geq 0} \right\} && \text{(set is not empty)} \\
&= \inf \left\{ \mathbf{ite}([\![\varphi]\!](\sigma[x \mapsto v]) \sqsubseteq [\![\psi]\!](\sigma),\ \infty,\ [\![\psi]\!](\sigma)) \;\middle|\; v \in \mathbb{Q}_{\geq 0} \right\} && \text{(Lemma 4.3)} \\
&= \inf \left\{ \mathbf{ite}([\![\varphi]\!](\sigma[x \mapsto v]) \sqsubseteq [\![\psi]\!](\sigma[x \mapsto v]),\ \infty,\ [\![\psi]\!](\sigma[x \mapsto v])) \;\middle|\; v \in \mathbb{Q}_{\geq 0} \right\} \\
& && (x \notin \mathsf{free}(\psi)) \\
&= \inf \left\{ [\![\varphi \to \psi]\!](\sigma[x \mapsto v]) \;\middle|\; v \in \mathbb{Q}_{\geq 0} \right\} && \text{(definition of } \to) \\
&= [\![\textstyle\bigsqcap_x (\varphi \to \psi)]\!](\sigma) && \text{(definition of } \textstyle\bigsqcap_x)
\end{aligned}$$

Similar for $\bigsqcap_x(\varphi \searrow \psi)$ (with 0 instead of ψ) and dual for the other assumption. □

4.2.3. Prenexing Arithmetic

Theorem 4.11 provides prenexing rules for the arithmetic operators. Note that prenexing is only proven sound for *scalar* multiplication, i.e. multiplication of a HeyLo formula by a constant $a \in \mathbb{R}_{\geq 0}$.

Theorem 4.11. Prenexing arithmetic functions

Let $a \in \mathbb{R}_{\geq 0}{}^{\text{Fixed}}$ and $\varphi, \psi \in \mathsf{HeyLo}$ and $x, y \in \mathsf{Vars}$ where $x \notin \mathsf{free}(\psi)$ and $y \notin \mathsf{free}(\varphi)$.

$$a \cdot (\prod_y \psi) \equiv \prod_y (a \cdot \psi) \qquad a \cdot (\bigsqcup_y \psi) \equiv \bigsqcup_y (a \cdot \psi)$$

$$(\prod_x \varphi) + \psi \equiv \prod_x (\varphi + \psi) \qquad (\bigsqcup_x \varphi) + \psi \equiv \bigsqcup_x (\varphi + \psi)$$

$$\varphi + (\prod_y \psi) \equiv \prod_y (\varphi + \psi) \qquad \varphi + (\bigsqcup_y \psi) \equiv \bigsqcup_y (\varphi + \psi)$$

Proof. Let $a \in \mathbb{R}_{\geq 0}$ and $\varphi, \psi \in \mathsf{HeyLo}$ and $x, y \in \mathsf{Vars}$ where $x \notin \mathsf{free}(\psi)$ and $y \notin \mathsf{free}(\varphi)$. Let $\sigma \in \Sigma$.

$$\begin{aligned}
&[\![a \cdot (\prod_y \psi)]\!](\sigma) \\
&= [\![a]\!](\sigma) \cdot [\![\prod_y \psi]\!](\sigma) && \text{(definition of } \cdot) \\
&= a \cdot [\![\prod_y \psi]\!](\sigma) && (a \in \mathbb{R}_{\geq 0}) \\
&= a \cdot \inf \{\, [\![\psi]\!](\sigma[y \mapsto v]) \mid v \in \mathbb{Q}_{\geq 0} \,\} && \text{(definition of } \textstyle\prod) \\
&= \inf \{\, a \cdot [\![\psi]\!](\sigma[y \mapsto v]) \mid v \in \mathbb{Q}_{\geq 0} \,\} \\
&= \inf \{\, [\![a]\!](\sigma) \cdot [\![\psi]\!](\sigma[y \mapsto v]) \mid v \in \mathbb{Q}_{\geq 0} \,\} \\
&= \inf \{\, [\![a \cdot \psi]\!](\sigma[y \mapsto v]) \mid v \in \mathbb{Q}_{\geq 0} \,\} && (y \notin \mathsf{free}(a)) \\
&= [\![\prod_y (a \cdot \psi)]\!](\sigma)
\end{aligned}$$

The other equivalence, $a \cdot (\bigsqcup_y \psi) \equiv \bigsqcup_y (a \cdot \psi)$, follows by a similar argument.

$$\begin{aligned}
&[\![(\prod_x \varphi) + \psi]\!](\sigma) \\
&= [\![\prod_x \varphi]\!](\sigma) + [\![\psi]\!](\sigma) && \text{(definition of } +) \\
&= \inf \{\, [\![\varphi]\!](\sigma[x \mapsto v]) \mid v \in \mathbb{Q}_{\geq 0} \,\} + [\![\psi]\!](\sigma) && \text{(definition of } \textstyle\prod)
\end{aligned}$$

$$
\begin{aligned}
&= \inf\{\, [\![\varphi]\!](\sigma[x \mapsto v]) + [\![\psi]\!](\sigma) \mid v \in \mathbb{Q}_{\geq 0} \,\} \\
&= \inf\{\, [\![\varphi]\!](\sigma[x \mapsto v]) + [\![\psi]\!](\sigma[x \mapsto v]) \mid v \in \mathbb{Q}_{\geq 0} \,\} && (x \notin \mathsf{free}(\psi)) \\
&= \inf\{\, [\![\varphi + \psi]\!](\sigma[x \mapsto v]) \mid v \in \mathbb{Q}_{\geq 0} \,\} && (\text{definition of } +) \\
&= [\![\textstyle\bigsqcap_x (\varphi + \psi)]\!](\sigma) && (\text{definition of } \textstyle\bigsqcap)
\end{aligned}
$$

The other equivalences for + again follow by similar arguments. □

Note that it is possible to extend the definition of Boolean expressions BExp so that the Iverson bracket $[b]$ and the Boolean embedding $?(b)$ functions can contain ordinary, Boolean quantifiers. One can derive similar prenexing rules for these as well, using e.g. $?(\forall x.\, b) \equiv \bigsqcap_x ?(b)$.

4.2.4. Prenex Normal Form

To fully eliminate HeyLo quantifiers in a validity check using Theorem 4.2, the only allowed quantifiers in the HeyLo formula are infimum operators at the beginning of the formula. We define the subset $\mathsf{HeyLo}^{\mathsf{PNF}}_{\sqcap} \subset \mathsf{HeyLo}$ below, as well as the subset $\mathsf{HeyLo}^{\mathsf{PNF}}_{\sqcup} \subset \mathsf{HeyLo}$ with only suprema, and the most general $\mathsf{HeyLo}^{\mathsf{PNF}} \subset \mathsf{HeyLo}$ that allows both quantifiers.

Definition 4.12. HeyLo: Prenex normal form

The set of HeyLo formulas in *prenex normal form* $\mathsf{HeyLo}^{\mathsf{PNF}} \subset \mathsf{HeyLo}$ consists of formulas v from the grammar below.

$$
\begin{aligned}
v \;&::= \textstyle\bigsqcap_x v \mid \bigsqcup_x v \mid \varphi \\
\varphi \;&::= a \mid \varphi + \varphi \mid \varphi \cdot \varphi \mid [b] \mid ?(b) \\
&\quad \mid \varphi \sqcap \varphi \mid \varphi \to \varphi \mid \varphi \searrow \varphi \\
&\quad \mid \varphi \sqcup \varphi \mid \varphi \leftarrow \varphi \mid \varphi \nwarrow \varphi \\
&\quad \mid \neg\varphi \mid {\sim}\varphi
\end{aligned}
$$

The subset $\mathsf{HeyLo}^{\mathsf{PNF}}_{\sqcap} \subset \mathsf{HeyLo}^{\mathsf{PNF}}$ does not contain $\bigsqcup_x$ and the subset $\mathsf{HeyLo}^{\mathsf{PNF}}_{\sqcup}$ does not contain $\bigsqcap_x$.

A HeyLo formula in prenex normal form consists of a sequence of limits called *prefix* and a formula without limits called *matrix*.

Our goal is now to identify a fragment of HeyLo formulas φ where quantifier elimination is possible by converting to an equivalent formula $\varphi' \in \mathsf{HeyLo}_{\sqcap}^{\mathsf{PNF}}$ and applying Theorem 4.2. The first step is the somewhat trivial observation that one prenexing rule allows repeated prenexing in the following lemma.

Lemma 4.13. Repeated prenexing

Let $f\colon \mathsf{HeyLo} \to \mathsf{HeyLo}$.

If $\forall \varphi \in \mathsf{HeyLo}.\, f(\bigsqcap_x \varphi) \equiv \bigsqcap_x f(\varphi)$:

$$\forall n \in \mathbb{N}_0. \quad f(\bigsqcap_{x_1} \dots \bigsqcap_{x_n} \varphi) \equiv \bigsqcap_{x_1} \dots \bigsqcap_{x_n} f(\varphi).$$

If $\forall \varphi \in \mathsf{HeyLo}.\, f(\bigsqcup_x \varphi) \equiv \bigsqcup_x f(\varphi)$:

$$\forall n \in \mathbb{N}_0. \quad f(\bigsqcup_{x_1} \dots \bigsqcup_{x_n} \varphi) \equiv \bigsqcup_{x_1} \dots \bigsqcup_{x_n} f(\varphi).$$

Proof. We show the first statement of Lemma 4.13 by induction over $n \in \mathbb{N}_0$. The second statement is analogous and omitted here.

For the base case ($n = 0$), we have $f(\varphi) \equiv f(\varphi)$. Now assume the proposition holds for an arbitrary but fixed $n \in \mathbb{N}_0$. Then $f(\bigsqcap_{x_1} \dots \bigsqcap_{x_n} \bigsqcap_{x_{n+1}} \varphi) \equiv \bigsqcap_{x_1} \dots \bigsqcap_{x_n} f(\bigsqcap_{x_{n+1}} \varphi)$ by the induction hypothesis. Using the original assumption, we get $\bigsqcap_{x_1} \dots \bigsqcap_{x_n} f(\bigsqcap_{x_{n+1}} \varphi) \equiv \bigsqcap_{x_1} \dots \bigsqcap_{x_{n+1}} f(\varphi)$.

Therefore, the lemma holds for arbitrary $n \in \mathbb{N}_0$. □

Theorem 4.14 identifies HeyLo formulas that can be converted into prenex normal form by applying the rules above. We distinguish between *infimum-prenexable* and *supremum-prenexable* formulas. Note that these fragments are not maximal and there are more HeyLo formulas where quantifier elimination is possible.

In addition, the prenexable fragments only specify subsets of HeyLo where *complete* quantifier elimination is possible. However, since the theorems above do not actually require sub-formulas to be quantifier-free in any way, it is also possible to partially eliminate quantifiers in HeyLo. The following theorem is more limited.

Theorem 4.14. Prenexable fragments of HeyLo

A HeyLo formula φ is *infimum-prenexable*, i.e. is equivalent to a formula $\varphi \equiv \varphi' \in \mathsf{HeyLo}_{\sqcap}^{\mathsf{PNF}}$, when its syntax is generated by the $\varphi_{\sqcap}$ rule from the grammar below. Similarly, φ is *supremum-prenexable* when it is generated by the $\varphi_{\sqcup}$ rule. Let φ_{QF} denote a quantifier-free HeyLo formula, $a \in \mathsf{ArithExp}$, $b \in \mathsf{BExp}$, and $c \in \mathbb{R}_{\geq 0}$.[Fixed]

$$
\begin{array}{ll}
\varphi_{\sqcap} ::= a & \varphi_{\sqcup} ::= a \\
\quad \mid \varphi_{\sqcap} + \varphi_{\sqcap} & \quad \mid \varphi_{\sqcup} + \varphi_{\sqcup} \\
\quad \mid c \cdot \varphi_{\sqcap} & \quad \mid c \cdot \varphi_{\sqcup} \\
\quad \mid [b] & \quad \mid [b] \\
\quad \mid ?(b) & \quad \mid ?(b) \\
\quad \mid \bigsqcap_x \varphi_{\sqcap} & \quad \mid \varphi_{\sqcup} \sqcap \varphi_{\sqcup} \\
\quad \mid \varphi_{\sqcap} \sqcap \varphi_{\sqcap} & \quad \mid \varphi_{QF} \to \varphi_{QF} \\
\quad \mid \varphi_{\sqcup} \to \varphi_{\sqcap} & \quad \mid \varphi_{QF} \searrow \varphi_{QF} \\
\quad \mid \varphi_{\sqcup} \searrow \varphi_{\sqcap} & \quad \mid \bigsqcup_x \varphi_{\sqcup} \\
\quad \mid \varphi_{\sqcap} \sqcup \varphi_{\sqcap} & \quad \mid \varphi_{\sqcup} \sqcup \varphi_{\sqcup} \\
\quad \mid \varphi_{QF} \leftarrow \varphi_{QF} & \quad \mid \varphi_{\sqcap} \leftarrow \varphi_{\sqcup} \\
\quad \mid \varphi_{QF} \nwarrow \varphi_{QF} & \quad \mid \varphi_{\sqcap} \nwarrow \varphi_{\sqcup} \\
\quad \mid \neg \varphi_{\sqcup} & \quad \mid \neg \varphi_{QF} \\
\quad \mid \sim \varphi_{QF} & \quad \mid \sim \varphi_{\sqcap}
\end{array}
$$

Proof. We show by induction over the structure of HeyLo formulas $f_{\sqcap}, f_{\sqcup} \in \mathsf{HeyLo}$ generated by the respective rules $\varphi_{\sqcap}$ and $\varphi_{\sqcup}$ that there are equivalent formulas $f'_{\sqcap} \in \mathsf{HeyLo}_{\sqcap}^{\mathsf{PNF}}$ and $f'_{\sqcup} \in \mathsf{HeyLo}_{\sqcup}^{\mathsf{PNF}}$.

Base cases. If $\varphi_{\sqcap} = \varphi_{\sqcup} \in \{\, a,\ [b],\ ?(b) \,\}$, then $\varphi_{\sqcap}, \varphi_{\sqcup} \in \mathsf{HeyLo}_{\sqcap}^{\mathsf{PNF}} \cap \mathsf{HeyLo}_{\sqcup}^{\mathsf{PNF}}$.

Induction hypothesis. Assume $\varphi_{\sqcap}, \psi_{\sqcap} \in \mathsf{HeyLo}$ are generated by the $\varphi_{\sqcap}$ rule. Then $\varphi_{\sqcap}, \psi_{\sqcap}$ are infimum-prenexable and respectively equivalent to $(\bigsqcap_{x_1} \dots \bigsqcap_{x_n} \varphi'_{\sqcap}) \in \mathsf{HeyLo}_{\sqcap}^{\mathsf{PNF}}$ and $(\bigsqcap_{y_1} \dots \bigsqcap_{y_m} \psi'_{\sqcap}) \in \mathsf{HeyLo}_{\sqcap}^{\mathsf{PNF}}$. Assume similarly $\varphi_{\sqcup}, \psi_{\sqcup} \in \mathsf{HeyLo}$ are generated by the corresponding rule, being supremum-prenexable and having an equivalent $(\bigsqcup_{x_1} \dots \bigsqcup_{x_n} \varphi'_{\sqcup}) \in \mathsf{HeyLo}_{\sqcup}^{\mathsf{PNF}}$ and $(\bigsqcup_{y_1} \dots \bigsqcup_{y_m} \psi'_{\sqcup}) \in \mathsf{HeyLo}_{\sqcup}^{\mathsf{PNF}}$.

Induction step. Without loss of generality, assume the sets of variables $\{\, x_1, \dots, x_n \,\}$ and $\{\, y_1, \dots, y_m \,\}$ are disjoint and $x_1, \dots, x_n \notin \mathsf{free}(\psi_{\sqcap}) \cup \mathsf{free}(\psi_{\sqcup})$ and $y_1, \dots, y_m \notin \mathsf{free}(\varphi_{\sqcap}) \cup$

$\mathsf{free}(\varphi_{\sqcup})$.

Let $f_{\sqcap}$ be generated by the $\varphi_{\sqcap}$ rule.

- $f_{\sqcap} = \bigsqcap_x \varphi_{\sqcap} \equiv \bigsqcap_x (\bigsqcap_{x_1} \dots \bigsqcap_{x_n} \varphi'_{\sqcap}) \in \mathsf{HeyLo}_{\sqcap}^{\mathsf{PNF}}$ by the induction hypothesis.
- If $f_{\sqcap} = \varphi_{\sqcap} \sqcap \psi_{\sqcap}$:

$$\begin{aligned} f_{\sqcap} &= \varphi_{\sqcap} \sqcap \psi_{\sqcap} \\ &\equiv (\bigsqcap_{x_1} \dots \bigsqcap_{x_n} \varphi'_{\sqcap}) \sqcap \psi_{\sqcap} && \text{(induction hypothesis)} \end{aligned}$$

Repeatedly apply rule $\bigsqcap_x \varphi \sqcap \psi \equiv \bigsqcap_x (\varphi \sqcap \psi)$ of Theorem 4.4.

$$\begin{aligned} &\equiv \bigsqcap_{x_1} \dots \bigsqcap_{x_n} \varphi'_{\sqcap} \sqcap \psi_{\sqcap} && \text{(Theorem 4.4 and Lemma 4.13)} \\ &\equiv \bigsqcap_{x_1} \dots \bigsqcap_{x_n} \varphi'_{\sqcap} \sqcap (\bigsqcap_{y_1} \dots \bigsqcap_{y_m} \psi'_{\sqcap}) && \text{(induction hypothesis)} \end{aligned}$$

Repeatedly apply rule $\varphi \sqcap \bigsqcap_y \psi \equiv \bigsqcap_y (\varphi \sqcap \psi)$ of Theorem 4.4.

$$\begin{aligned} &\equiv \bigsqcap_{x_1} \dots \bigsqcap_{x_n} \bigsqcap_{y_1} \dots \bigsqcap_{y_m} \varphi'_{\sqcap} \sqcap \psi'_{\sqcap} && \text{(Theorem 4.4 and Lemma 4.13)} \\ &\in \mathsf{HeyLo}_{\sqcap}^{\mathsf{PNF}} \end{aligned}$$

All proofs for the other cases have the same structure as the $\varphi_{\sqcap} \sqcap \psi_{\sqcap}$ case. We abbreviate in the following and just state the required rules for prenexing one quantifier.

- $f_{\sqcap} = \varphi_{\sqcup} \to \psi_{\sqcap}$:
  - $(\bigsqcup_x \varphi) \to \psi \equiv \bigsqcap_x (\varphi \to \psi)$ (Theorem 4.8),
  - $\varphi \to \bigsqcap_y \psi \equiv \bigsqcap_y (\varphi \to \psi)$ (Theorem 4.4).
- $f_{\sqcap} = \varphi_{\sqcup} \searrow \psi_{\sqcap}$:
  - $(\bigsqcup_x \varphi) \searrow \psi \equiv \bigsqcap_x (\varphi \searrow \psi)$ (Theorem 4.8),
  - $\varphi \searrow \bigsqcap_y \psi \equiv \bigsqcap_y (\varphi \searrow \psi)$ (Theorem 4.4).
- $f_{\sqcap} = \varphi_{\sqcap} \sqcup \psi_{\sqcap}$:
  - $\bigsqcap_x \varphi \sqcup \psi \equiv \bigsqcap_x (\varphi \sqcup \psi)$,
  - $\varphi \sqcup \bigsqcap_y \psi \equiv \bigsqcap_y (\varphi \sqcup \psi)$ (Theorem 4.4).
- $f_{\sqcap} = \varphi_{QF} \leftarrow \psi_{QF}$ is quantifier-free and therefore in $\mathsf{HeyLo}_{\sqcap}^{\mathsf{PNF}}$.
- $f_{\sqcap} = \varphi_{QF} \nwarrow \psi_{QF}$ is quantifier-free and therefore in $\mathsf{HeyLo}_{\sqcap}^{\mathsf{PNF}}$.
- $f_{\sqcap} = \neg \varphi_{\sqcup}$:
  - $\neg(\bigsqcup_x \varphi) \equiv \bigsqcap_x (\neg \varphi)$ (Theorem 4.8).
- $f_{\sqcap} = {\sim}\varphi_{QF}$ is quantifier-free and therefore in $\mathsf{HeyLo}_{\sqcap}^{\mathsf{PNF}}$.

- $f_\sqcap = a, f_\sqcap = [b]$, and $f_\sqcap = ?(b)$ are quantifier-free and therefore in $\mathsf{HeyLo}^{\mathsf{PNF}}_\sqcap$.
- $f_\sqcap = \varphi_\sqcap + \psi_\sqcap$:
  - $(\bigsqcap_x \varphi) + \psi \equiv \bigsqcap_x (\varphi + \psi)$,
  - $\varphi + (\bigsqcap_y \psi) \equiv \bigsqcap_y (\varphi + \psi)$ (Theorem 4.11).
- $f_\sqcap = c \cdot \varphi_\sqcap$:
  - $a \cdot (\bigsqcap_x \varphi) \equiv \bigsqcap_x (a \cdot \varphi)$ (Theorem 4.11).

The cases for $f_\sqcup$ generated by the $\varphi_\sqcup$ rule are completely dual and the same theorems apply. We therefore omit them here.

In conclusion, the theorem about prenexable fragments is shown by structural induction over the formulas $f_\sqcap$ and $f_\sqcup$. □

Lemma 4.15. HeyLo: Validity check of prenexable fragment

For every infimum-prenexable HeyLo formula $\varphi_\sqcap$, there is a quantifier-free formula $\varphi_{QF} \in \mathsf{HeyLo}$ such that

$$\varphi_\sqcap \text{ is valid} \quad \text{iff} \quad \varphi_{QF} \text{ is valid.}$$

Proof. By Theorem 4.14, every infimum-prenexable formula $\varphi_\sqcap$ is equivalent to a formula $\varphi' \in \mathsf{HeyLo}^{\mathsf{PNF}}_\sqcap$. Let $\varphi' = \bigsqcap_{x_1} \dots \bigsqcap_{x_n} \varphi_{QF}$ for a quantifier-free φ_{QF}. By induction over $n \in \mathbb{N}_0$, φ' is valid iff φ_{QF} is valid: For $n = 0$, $\varphi' = \varphi_{QF}$. Assume the statement holds for arbitrary but fixed $n \in \mathbb{N}_0$. By the induction hypothesis, $\varphi' = \bigsqcap_{x_1} \dots \bigsqcap_{x_n} \bigsqcap_{x_{n+1}} \varphi_{QF}$ is valid iff $\bigsqcap_{x_{n+1}} \varphi_{QF}$ is valid. By Theorem 4.2, $\bigsqcap_{x_{n+1}} \varphi_{QF}$ is valid iff φ_{QF} is valid. □

The proofs of Theorem 4.14 and Lemma 4.15 make it clear that a quantifier-free $\varphi_{QF} \in \mathsf{HeyLo}$ such that φ is valid iff φ_{QF} is valid can be computed by iteratively applying the equivalence rules above so that quantifiers are always shifted to the front of the formula. In this way, an algorithm can create sub-formulas in prenex normal form starting from the bottom (most nested) to the top of the formula. Eventually, the resulting formula is in prenex normal form and all leading infimum quantifiers can be removed for validity checking. This algorithm runs in linear time in the size of the formula.[Fixed]

4.2.5. HeyVL and Quantifier Elimination

Theorem 4.14 specifies a subset of HeyLo where complete quantifier elimination is possible. Is there a fragment of HeyVL where we can guarantee that the verification conditions with respect to a subset of HeyLo formulas are prenexable? The answer is yes. The verification conditions of programs in the *infimum-prenexable fragment* of HeyVL are always infimum-prenexable when computed with respect to an infimum-prenexable post-expectation.

Definition 4.16. Prenexable fragments of HeyVL

Let $f_\sqcap \in \mathsf{HeyLo}_\sqcap^{\mathsf{PNF}}$ and $f_\sqcup \in \mathsf{HeyLo}_\sqcup^{\mathsf{PNF}}$.[Fixed] The set of *prenexable HeyVL statements* is generated by the following two rules. We call statements generated by $S_\sqcap$ *infimum-prenexable* and those generated by $S_\sqcup$ *supremum-prenexable*.

$$
\begin{aligned}
S_\sqcap \;::=\; & \texttt{skip} \\
& \mid x :\approx a \\
& \mid x :\approx \mathsf{ber}!(p) \\
& \mid S_\sqcap ;\, S_\sqcap \\
& \mid \texttt{down havoc } x \\
& \mid \texttt{down assert } f_\sqcap \\
& \mid \texttt{down assume } f_\sqcup \\
& \mid \texttt{down compare } f_\sqcup \\
& \mid \texttt{if } (\sqcap)\ \{S_\sqcap\}\ \texttt{else}\ \{S_\sqcap\} \\
& \mid \texttt{up assert } f_\sqcap \\
& \mid \texttt{if } (\sqcup)\ \{S_\sqcap\}\ \texttt{else}\ \{S_\sqcap\} \\
& \mid S_\sqcup ;\, \texttt{up negate}
\end{aligned}
\qquad
\begin{aligned}
S_\sqcup \;::=\; & \texttt{skip} \\
& \mid x :\approx a \\
& \mid x :\approx \mathsf{ber}!(p) \\
& \mid S_\sqcup ;\, S_\sqcup \\
& \mid \texttt{down assert } f_\sqcup \\
& \mid \texttt{if } (\sqcap)\ \{S_\sqcup\}\ \texttt{else}\ \{S_\sqcup\} \\
& \mid \texttt{up havoc } x \\
& \mid \texttt{up assert } f_\sqcup \\
& \mid \texttt{up assume } f_\sqcap \\
& \mid \texttt{up compare } f_\sqcap \\
& \mid \texttt{if } (\sqcup)\ \{S_\sqcup\}\ \texttt{else}\ \{S_\sqcup\} \\
& \mid S_\sqcap ;\, \texttt{down negate}
\end{aligned}
$$

Theorem 4.17. Validity of prenexable HeyVL

Let S be an infimum-prenexable HeyVL program. Then there is a quantifier-free formula $\varphi_{QF} \in \mathsf{HeyLo}$ such that

$$\varphi_{QF} \text{ is valid} \quad \text{iff} \quad S \text{ is valid.}$$

Proof. S is valid iff $\mathsf{vc}[\![S]\!](\infty)$ is valid. We show $\mathsf{vc}[\![S]\!](\infty)$ is infimum-prenexable, and by Lemma 4.15 there is a quantifier-free formula $\varphi_{QF} \in \mathsf{HeyLo}$ that is valid iff $\mathsf{vc}[\![S]\!](\infty)$ is

valid.

To show that $\mathsf{vc}[\![S]\!](\infty)$ is infimum-prenexable, we show a more general statement: For any infimum-prenexable $\varphi_\sqcap \in \mathsf{HeyLo}$ and $S_\sqcap \in \mathsf{HeyVL}$ generated by the $S_\sqcap$ rule of Definition 4.16, $\mathsf{vc}[\![S_\sqcap]\!](\varphi_\sqcap)$ is infimum-prenexable. That statement in turn requires the dual statement for supremum-prenexability. We only show the cases for infimum-prenexability and omit those for supremum-prenexability because of their duality.

Let $\varphi_\sqcap \in \mathsf{HeyLo}_\sqcap^{\mathsf{PNF}}$ and let $S \in \mathsf{HeyVL}$ be generated by the $S_\sqcap$ rule of Definition 4.16.[Fixed]

Base cases:

- $S = \mathtt{skip}$.
$$\mathsf{vc}[\![\mathtt{skip}]\!](\varphi_\sqcap) = \varphi_\sqcap \in \mathsf{HeyLo}_\sqcap^{\mathsf{PNF}}.$$
- $S = x :\approx a$.
$$\mathsf{vc}[\![x :\approx a]\!](\varphi_\sqcap) = \varphi_\sqcap[x \mapsto a] \in \mathsf{HeyLo}_\sqcap^{\mathsf{PNF}}.$$
- $S = x :\approx \mathsf{ber!}(p)$.
$$\mathsf{vc}[\![x :\approx \mathsf{ber!}(p)]\!](\varphi_\sqcap) = p \cdot \varphi_\sqcap[x \mapsto 1] + (1-p) \cdot \varphi_\sqcap[x \mapsto 0] \in \mathsf{HeyLo}_\sqcap^{\mathsf{PNF}}.$$
- $S = \mathsf{down\ havoc}\ x$.
$$\mathsf{vc}[\![\mathsf{down\ havoc}\ x]\!](\varphi_\sqcap) = \bigsqcap_x \varphi_\sqcap \in \mathsf{HeyLo}_\sqcap^{\mathsf{PNF}}.$$
- $S = \mathsf{down}\ \mathtt{assert}\ f_\sqcap$.
$$\mathsf{vc}[\![\mathsf{down}\ \mathtt{assert}\ f_\sqcap]\!](\varphi_\sqcap) = f_\sqcap \sqcap \varphi_\sqcap \in \mathsf{HeyLo}_\sqcap^{\mathsf{PNF}}.$$
- $S = \mathsf{down}\ \mathtt{assume}\ f_\sqcup$.
$$\mathsf{vc}[\![\mathsf{down}\ \mathtt{assume}\ f_\sqcup]\!](\varphi_\sqcap) = f_\sqcup \to \varphi_\sqcap \in \mathsf{HeyLo}_\sqcap^{\mathsf{PNF}}.$$
- $S = \mathsf{down}\ \mathtt{compare}\ f_\sqcup$.
$$\mathsf{vc}[\![\mathsf{down}\ \mathtt{compare}\ f_\sqcup]\!](\varphi_\sqcap) = f_\sqcup \searrow \varphi_\sqcap \in \mathsf{HeyLo}_\sqcap^{\mathsf{PNF}}.$$
- $S = \mathsf{up}\ \mathtt{assert}\ f_\sqcap$.
$$\mathsf{vc}[\![\mathsf{up}\ \mathtt{assert}\ f_\sqcap]\!](\varphi_\sqcap) = f_\sqcap \sqcup \varphi_\sqcap \in \mathsf{HeyLo}_\sqcap^{\mathsf{PNF}}.$$

Now assume that the induction hypothesis holds:

- For $\varphi_\sqcap \in \mathsf{HeyLo}_\sqcap^{\mathsf{PNF}}$, $\mathsf{vc}[\![S_\sqcap]\!](\varphi_\sqcap) \in \mathsf{HeyLo}_\sqcap^{\mathsf{PNF}}$.
- For $\varphi_\sqcup \in \mathsf{HeyLo}_\sqcup^{\mathsf{PNF}}$, $\mathsf{vc}[\![S_\sqcup]\!](\varphi_\sqcup) \in \mathsf{HeyLo}_\sqcup^{\mathsf{PNF}}$.

Induction step.

- $S = S_\sqcap^1 ; S_\sqcap^2$. By the induction hypothesis, $\mathsf{vc}[\![S_\sqcap^2]\!](\varphi_\sqcap) \in \mathsf{HeyLo}_\sqcap^{\mathsf{PNF}}$. And using the induction hypothesis again, $\mathsf{vc}[\![S_\sqcap^1]\!](\mathsf{vc}[\![S_\sqcap^2]\!](\varphi_\sqcap)) \in \mathsf{HeyLo}_\sqcap^{\mathsf{PNF}}$. Therefore,

$$\begin{aligned}
&\mathsf{vc}[\![S_\sqcap^1 ; S_\sqcap^2]\!](\varphi_\sqcap) \\
&= \mathsf{vc}[\![S_\sqcap^1]\!](\mathsf{vc}[\![S_\sqcap^2]\!](\varphi_\sqcap)) && \text{(definition of ;)} \\
&\in \mathsf{HeyLo}_\sqcap^{\mathsf{PNF}}
\end{aligned}$$

- $S = \mathtt{if}\ (\sqcap)\ \{S_\sqcap^1\}\ \mathtt{else}\ \{S_\sqcap^2\}$. By the induction hypothesis, $\mathsf{vc}[\![S_\sqcap^1]\!](\varphi_\sqcap) \in \mathsf{HeyLo}_\sqcap^{\mathsf{PNF}}$ and $\mathsf{vc}[\![S_\sqcap^2]\!](\varphi_\sqcap) \in \mathsf{HeyLo}_\sqcap^{\mathsf{PNF}}$ hold. Therefore,

$$\begin{aligned}
&\mathsf{vc}[\![\mathtt{if}\ (\sqcap)\ \{S_\sqcap^1\}\ \mathtt{else}\ \{S_\sqcap^2\}]\!](\varphi_\sqcap) \\
&= \mathsf{vc}[\![S_\sqcap^1]\!](\varphi_\sqcap) \sqcap \mathsf{vc}[\![S_\sqcap^2]\!](\varphi_\sqcap) && \text{(definition of } \mathtt{if}\ (\sqcap)) \\
&\in \mathsf{HeyLo}_\sqcap^{\mathsf{PNF}}
\end{aligned}$$

- $S = \mathtt{if}\ (\sqcup)\ \{S_\sqcap^1\}\ \mathtt{else}\ \{S_\sqcap^2\}$. By the induction hypothesis, $\mathsf{vc}[\![S_\sqcap^1]\!](\varphi_\sqcap) \in \mathsf{HeyLo}_\sqcap^{\mathsf{PNF}}$ and $\mathsf{vc}[\![S_\sqcap^2]\!](\varphi_\sqcap) \in \mathsf{HeyLo}_\sqcap^{\mathsf{PNF}}$ hold. Therefore,

$$\begin{aligned}
&\mathsf{vc}[\![\mathtt{if}\ (\sqcup)\ \{S_\sqcap^1\}\ \mathtt{else}\ \{S_\sqcap^2\}]\!](\varphi_\sqcap) \\
&= \mathsf{vc}[\![S_\sqcap^1]\!](\varphi_\sqcap) \sqcup \mathsf{vc}[\![S_\sqcap^2]\!](\varphi_\sqcap) && \text{(definition of } \mathtt{if}\ (\sqcup)) \\
&\in \mathsf{HeyLo}_\sqcap^{\mathsf{PNF}}
\end{aligned}$$

- $S = S_\sqcup ; \mathsf{up}\ \mathsf{negate}$.

$$\begin{aligned}
&\mathsf{vc}[\![S_\sqcup ; \mathsf{up}\ \mathsf{negate}]\!](\varphi_\sqcap) \\
&= \mathsf{vc}[\![S_\sqcup]\!](\mathsf{vc}[\![\mathsf{up}\ \mathsf{negate}]\!](\varphi_\sqcap)) && \text{(definition of ;)} \\
&= \mathsf{vc}[\![S_\sqcup]\!](\sim\varphi_\sqcap) && \text{(definition of up negate)} \\
&\in \mathsf{HeyLo}_\sqcap^{\mathsf{PNF}} && (\sim\varphi_\sqcap \in \mathsf{HeyLo}_\sqcup^{\mathsf{PNF}}\text{, induction hypothesis})
\end{aligned}$$

All cases for $S_\sqcup$ are dual and the same theorems apply. Therefore, we have shown that $\mathsf{vc}[\![S_\sqcap]\!](\varphi_\sqcap) \in \mathsf{HeyLo}_\sqcap^{\mathsf{PNF}}$ and $\mathsf{vc}[\![S_\sqcup]\!](\varphi_\sqcup) \in \mathsf{HeyLo}_\sqcup^{\mathsf{PNF}}$ by induction over the structures of $S_\sqcap$ and $S_\sqcup$.

Applying this to S and $\infty \in \mathsf{HeyLo}_\sqcap^{\mathsf{PNF}}$, it follows by Lemma 4.15 that there is a quantifier-free $\varphi_{QF} \in \mathsf{HeyLo}$ that is valid if and only if $\mathsf{vc}[\![S]\!](\infty)$ is valid. □

As we have noted for Lemma 4.15, the quantifier-free $\varphi_{QF} \in \mathsf{HeyLo}$ that is valid if and only if $\mathsf{vc}[\![S_\sqcap]\!](\varphi_\sqcap)$ is valid can be computed as a bottom-up traversal of the HeyLo formula in linear time.

These results allow us to eliminate quantifiers in the verification conditions for a large number of HeyVL programs. In particular, the encodings we have developed for pGCL (cf. Section 3.7) are also covered by the infimum-prenexable fragment of HeyVL if the pre- and post-expectations are prenexable as well.

4.3. The Implementation Caesar

Caesar is our implementation of a deductive verifier for probabilistic programs. The name is inspired by the adapted quote "veni, vidi, vc". In spite of this slogan, Caesar is only a prototype. Caesar takes a HeyVL program S as input and checks whether it verifies $(\mathsf{vc}[\![S]\!](\infty) \equiv \infty)$. To do this, it generates the HeyLo verification conditions $\mathsf{vc}[\![S]\!](\infty)$ of S and encodes the verification query $\mathsf{vc}[\![S]\!](\infty) \equiv \infty$ in SMT-LIB. It uses Z3 to prove that S verifies, to find a counter-example, or to return "unknown".

Our implementation Caesar is written in Rust and we have a Python tool to generate HeyVL encodings for pGCL programs (cf. Chapter 3). Caesar and our Python tool consist of around 7000 lines of code in total. In addition to the verification condition generation and SMT-LIB encoding, Caesar implements quantifier elimination for HeyLo and supports different *types* for program and logical variables that occur in HeyLo and HeyVL. Based on the encoding for the verification of implementations (Section 2.3.6) and the encoding of specifications (Section 2.3.7), we also implement simple *procedures* to enable modular verification. In this section, we give a high-level overview of the architecture of Caesar and present results of benchmarks.

4.3.1. Extensions to HeyVL and HeyLo

Types and Declarations In this thesis, we gave a simplified presentation of HeyVL and HeyLo. Both the variables that occur in HeyVL programs, as well as logical variables that only occur in HeyLo formulas, were quantified over $\mathbb{Q}_{\geq 0}$. We also did not require the *declaration* of variables before their use, implicitly assuming all variables were declared and of type $\mathbb{Q}_{\geq 0}$.

In Caesar, each variable has to be declared before use and a type annotation has to be given. At the moment, Caesar supports three basic types: `bool`, `nat`, and `realplus`. For example, `bool x;` declares a variable x with domain $\mathbb{B}$. `nat` represents the natural numbers $\mathbb{N}$ and `realplus` represents $\mathbb{R}^\infty_{\geq 0}$.

Procedures Caesar also supports *procedures* for modular verification of HeyVL programs. A procedure is a block of code that has annotations with pre- and post-expectations, and optionally parameters and a return value. Procedures are a standard feature in deductive verifiers and our implementation is similar to the standard designs, but supports both lower and upper bound proofs. One input file to Caesar can contain multiple procedures, which are independently verified. Procedures can be used in other procedures, where only their specification via pre- and post-expectations, as well as parameters and return value is available, but not their definition.

Example 4.18. Geometric loop: Procedure declaration

Recall the geometric loop example (Example 3.4). Let S_{geo} be the HeyVL encoding of C_{geo}, for example using k-induction with $k = 3$. The verification of the following procedure checks $\text{vc}[\![S_{\text{geo}}]\!](c) \sqsubseteq c + 1$:

```
proc geo(x: bool, c: nat)
    returns (c: nat),
    up requires c + 1,
    up ensures c
{
    S_geo
}
```

For the verification of geo, the following HeyVL program is generated and verified:

$$\texttt{down negate; up assume } c+1\texttt{; } S_{\text{geo}}\texttt{; up assert } c\texttt{; up negate}$$

Caesar automatically determines that an upper bound proof is required and uses `up assume` and `assert` statements and inserts the proper negations (cf. Example 3.38).

In a procedure declaration, only either `down` or `up` annotations may be used. From these, the proper encoding for the verification of the implementation is generated and verified. We explained the verification of implementations in Section 2.3.6. The implementation of so-called *old expressions* is planned (cf. [Leio8, Section 4.3]).

Caesar also supports *procedure calls* by encoding specifications (cf. Section 2.3.7). When a procedure is called, Caesar checks that the implementation is monotone so that Theorem 2.41 can be applied. The check is done purely syntactically, by checking that the implementation is contained in the monotone fragment $\text{HeyVL}_{\text{mon}}$ (Theorem 2.31).

Example 4.19. Geometric loop: Procedure call

The *procedure call* $g :\approx geo(0, 0)$ is converted by Caesar into the following HeyVL

program:

$$\begin{aligned}&\texttt{up assert } (c+1)[x \mapsto 0][c \mapsto 0];\\&\texttt{up havoc } x, c;\\&\texttt{up assume } c[c \mapsto g]\end{aligned}$$

A detailed explanation and formal definition of procedures and procedure calls is out of scope for this thesis and a next step for future work.

Domains, Functions, and Axioms One simple, but powerful feature that most IVLs support is the definition of user-defined *domains*, uninterpreted functions, and custom *axioms*. A user-defined domain creates a new type in HeyVL. *Uninterpreted functions* are functions that operate on types, but do not have a definition. Instead, their meaning is specified by *axioms* that are added by the user.

Example 4.20. User-defined three-valued domain

The following declarations create a new type `ternary` with three "constructor" uninterpreted functions `one`, `two`, and `three`, as well as testing functions `is_one`, `is_two`, and `is_three`.

```
domain ternary {
    func one() -> ternary;
    func two() -> ternary;
    func three() -> ternary;
    func is_one(x: ternary) -> bool;
    func is_two(x: ternary) -> bool;
    func is_three(x: ternary) -> bool;
    axiom forall(x: ternary). is_one(x) == (x == one());
    axiom forall(x: ternary). is_two(x) == (x == two());
    axiom forall(x: ternary). is_three(x) == (x == three());
}
```

A domain is translated to a new SMT-LIB *sort*. Functions are declared as uninterpreted function symbols and axioms are added to the SMT-LIB query. Note that axioms are always specified in first-order Boolean logic, not in HeyLo.

Boolean Verification Domain Motivated by the generalization of HeyLo and HeyVL in Chapter 5, Caesar also supports a setting so that the default verification domain uses Booleans $\mathbb{B}$ instead of $\mathbb{R}^{\infty}_{\geq 0}$. Then, all formulas used in `assume`, `assert`, and `compare` statements must evaluate to Booleans. Through this, we can compare Caesar to classical verification based on first-order Boolean logic.

4.3.2. Architecture of Caesar

We described a high-level architecture of our deductive verifier in Chapter 1. Refer to Figure 1.1. The architecture of our deductive verifier Caesar roughly follows it, but includes various optimization passes in-between.

Caesar accepts HeyVL programs as input. The *parser* takes the input file and generates an abstract syntax tree. The *resolver* is responsible for name resolution, i.e. associating each identifier with the unique declaration it refers to. The *type checker* traverses the syntax tree and annotates expressions with their types. A small number of type inference rules are implemented, for example automatically promoting natural number types to non-negative real numbers if needed. In the *desugaring* phase, procedure calls and procedure declarations are encoded (cf. Section 4.3.1).

Now, we generate the verification conditions for the HeyVL program. The *vcgen* phase implements the vc transformer as presented in Definition 2.25. Importantly, our representation of HeyLo formulas supports explicit substitutions and sharing of common sub-expressions. These features allow the *vcgen* phase itself to run in linear time in the size of the HeyVL program.

However, we need a representation of HeyLo formulas without substitutions for the next phases. We support both a *strict* and *lazy* "unfolding" of the verification conditions. Both methods apply substitutions recursively, and here the resulting formulas may require memory with size exponential in the size of the HeyVL program. While the strict unfolding simply applies all substitutions recursively, the lazy unfolding keeps track of Boolean conditions for reachability of the current sub-formula. For example, it can infer that the sub-formula $?(\neg b) \to \varphi$ of $?(b) \to (?(\neg b) \to \varphi)$ can be replaced by the constant 0. This saves the recursive unfolding of φ. The reachability checks are done using an incremental SMT encoding. Only a limited number of rules are implemented, and therefore the lazy unfolding does not constitute a complete decision procedure for the validity of HeyLo formulas.

Next, we run *quantifier elimination* on the verification conditions. We described this phase in detail in Section 4.2.

Given the simplified verification conditions $\varphi \in \mathsf{HeyLo}$, the last phases work on the Boolean formula $\varphi \equiv \infty$. Optional optimizations based on the *relational view* (Section 2.2.4) and other lattice theorems are implemented. For example, $(a \to b) \equiv \infty$ is transformed into $\infty \sqsubseteq (a \to b)$ and then into $a \sqsubseteq b$. Our benchmarks use a simple recursive implementation, but Caesar also includes an experimental optimizer based on *e-graphs* using the Rust library *egg* [Wil+21]. Finally, we encode the query of the validity of the verification conditions and ask Z3 to run the deciding SAT check.

4.3.3. Empirical Evaluation

We compare Caesar to KIPRO2, the k-induction tool of [Bat+21a]. Tests were executed on an Intel Core i5-4690 with a memory limit of 16 GB and a timeout of 400 seconds. In the following table, "TO" indicates a timeout and "OOM" an out-of-memory condition.

Our first set of benchmarks is based on the set of inductive examples in [Bat+21a, Table 2]. The examples are all pGCL programs consisting of a while loop with a loop-free loop body and the task is the verification of upper bounds of weakest pre-expectations (wp). Note that KIPRO2 is limited to the verification of such programs and cannot handle pGCL programs that contain nested loops.

To generate HeyVL input for our tool Caesar from the pGCL programs, we developed a fully automatic tool that converts pGCL programs with the specification of the loop and a fixed $k \in \mathbb{N}$ into HeyVL programs according to our encoding presented in Chapter 3. We run this tool as a pre-processing step and its execution time is not included in our measurements.

For a fair comparison between Caesar and KIPRO2, the latter was modified to skip k-inductivity checks for values below the respective k given in the table. Therefore, KIPRO2 does not do more k-inductivity checks than Caesar.

Figure 4.21 shows the results of our benchmarks, comparing total execution times. We tested Caesar with both the lazy and strict unfolding of the verification conditions. For the lazy unfolding, we additionally measured the time taken for the lazy unfolding itself ("Unfold") and for the final SAT check ("SAT").

With the exception of the brp2 and brp3 benchmarks, Caesar (with lazy unfolding) is consistently as fast or significantly faster than KIPRO2. In most benchmarks, the time for lazy unfolding dominates the overall execution time. Compared to strict unfolding, the lazy unfolding is significantly faster. The examples brp2 and unif_gen4 do not successfully terminate with the strict unfolding, but run out of memory. Especially noteworthy is the time required by Caesar (with lazy unfolding) for unif_gen4, which is more than forty times faster than KIPRO2. Finally, Caesar does not terminate on brp3 within 400 seconds. The timeout occurs in the unfolding phase. Future work will include evaluation of KIPRO2's optimizations to include them in Caesar.

To test Caesar's performance on HeyVL programs that encode pGCL programs with nested loops, we encoded Rabin's mutual exclusion protocol in Example 4.22. The pGCL program contains a nested loop and therefore its verification requires two invariants. rabin1 and rabin2 in Figure 4.21 are manually rewritten versions of the same program that use a single loop and consequently a different invariant. Caesar (with lazy unfolding) verifies the HeyVL encoding in 0.1s.

Benchmark	k	Lazy			Strict	kipro2
		Unfold	SAT	Total	Total	Total
brp1	5	0.03s	0.02s	0.1s	0.29s	0.43s
brp2	11	1.57s	1.14s	2.93s	OOM	2.74s
brp3	23	TO	/	TO	OOM	355.3s
geo1	2	0.01s	0.01s	0.04s	0.15s	0.12s
rabin1	1	0.01s	0.01s	0.06s	0.31s	0.27s
rabin2	5	0.09s	0.02s	0.15s	1.38s	1.97s
unif_gen1	2	0.03s	0.01s	0.08s	0.31s	0.75s
unif_gen2	3	0.11s	0.03s	0.19s	2.77s	2.77s
unif_gen3	3	0.11s	0.03s	0.16s	2.43s	2.64s
unif_gen4	5	2.97s	0.28s	3.54s	OOM	153.24s

Figure 4.21.: Empirical results comparing Caesar's lazy unfolding, Caesar's strict unfolding, and KIPRO2 of [Bat+21a].

Example 4.22. Rabin's mutual exclusion algorithm

Assume n processors are working with some shared data. Rabin's mutual exclusion algorithm is an algorithm to select a unique processor that receives exclusive access to it [KR92]. The winner is determined by the following simple protocol: Each processor randomly flips fair coins until heads is shown. The processor that requires the most coin flips is the winner. It is possible that this winner is not unique, therefore the algorithm can fail with a certain probability.

Based on [KR92], Hurd et al. encode this algorithm as a sequential pGCL program to reason about the algorithm's success probability [HMM05]. We present it below with invariant annotations. The outer loop terminates if $i = 1$ (indicating success) or $i = 0$ (indicating failure). In each iteration, n is initialized with the number of processors i that are still competing. Then, each remaining processor flips a fair coin and retires if

the result is 1 (heads).

$$
\begin{array}{l}
\texttt{while}\ (1 < i)\ \texttt{invariant}\ I_1\ \{ \\
\quad n := i; \\
\quad \texttt{while}\ (0 < n)\ \texttt{invariant}\ I_2\ \{ \\
\quad\quad \{d := 0\}\ [0.5]\ \{d := 1\}; \\
\quad\quad i := i - d; \\
\quad\quad n := n - 1 \\
\quad \} \\
\}
\end{array}
$$

The post-expectation $post = [1 = i]$ is used to check success. Therefore, $\mathsf{wlp}[\![C_{\text{rabin}}]\!](post)$ encodes the probability of the algorithm succeeding.

Hurd et al. show that the success probability is independent of the number of processors n by proving that $I_1 \sqsubseteq \mathsf{wlp}[\![C_{\text{rabin}}]\!](post)$ holds, where the pre-expectation I_1 does not depend on n.

For a bounded number of processors $n \leq N$, we can prove this fact with HeyVL:

$$
\begin{array}{l}
\texttt{down assume}\ I_1\,; \\
\mathit{trans}^{\mathsf{wlp}}_{\texttt{down}}(C_{\text{rabin}})\,; \\
\texttt{down assert}\ \mathit{post}
\end{array}
$$

The pre-expectation (and invariant) I_1 and the other invariant I_2 are given by the following equations. More detailed explanations can be found in [HMM05].

$$
\begin{aligned}
B &= n \leq N \land i \leq N && \text{(We only support values up to } N\text{)} \\
I_1 &= [B] \cdot ([1 = i] + [1 < i] \cdot (2/3)) && \text{(Invariant for outer loop)} \\
I_2 &= [B] \cdot [0 \leq n \land n \leq i] \cdot ((2/3) \cdot \mathit{invar}1 + \mathit{invar}2) && \text{(Invariant for inner loop)} \\
\mathit{invar}1 &= 1 - ([i = n] \cdot (n+1) \cdot \mathit{rptwo} + [i = n+1] \cdot \mathit{rptwo}) \\
\mathit{invar}2 &= [i = n] \cdot n \cdot \mathit{rptwo} + [i = n+1] \cdot \mathit{rptwo}
\end{aligned}
$$

Because Caesar does not support general division nor exponentials, we represent the reciprocals of powers of two $\frac{1}{2^n}$ by *rptwo*, a case distinction for finite values of $n \leq N$.

$$
\mathit{rptwo} = \sum_{i=0}^{n} [i = n] \cdot \frac{1}{2^n}
$$

Finally, we tested an encoding of the rdwalk example of the benchmark set by Ngo, Carbonneaux, and Hoffmann [NCH18]. This requires reasoning about lower bounds of expected runtimes. We added support for the associated expected runtime calculus

ert [Kam+16] to our HeyVL translation tool. For this, we added a `tick` x statement to Caesar's HeyVL syntax with semantics $\mathsf{vc}[\![\texttt{tick}\ x]\!](\varphi) = x + \varphi$. Note that we have not formally shown correctness of our ert encoding in this thesis. Caesar verifies the HeyVL program in $0.04s$, compared to $0.18s$ for KIPRO2 and the reported $0.012s$ for the tool presented in [NCH18].

Overall, these results are promising. Our prototypical implementation Caesar is competitive with the specialized k-induction tool KIPRO2. This is remarkable because Caesar itself does not implement k-induction, but accepts generic HeyVL programs with k-induction encodings.

5. Abstraction

So far, HeyLo and HeyVL have been defined to work on the domain of expectations $\mathbb{E}$ only, i.e. mappings $X \colon \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$. In this chapter, we take a step back and redefine (most of) HeyLo using *Heyting algebras* and *Gödel algebras*. Gödel algebras are a sub-class of Heyting algebras.

With this algebraic foundation, we are able to use HeyLo on other domains than $\mathbb{R}^{\infty}_{\geq 0}$. For example, we obtain first-order Boolean logic by instantiating HeyLo with the natural Gödel algebra on $\mathbb{B}$. With the generalization of HeyLo to other domains, it is quite easy to generalize HeyVL as well. Using the Boolean domain $\mathbb{B}$, we directly obtain a classical verification framework in the style of the simple IVL we presented back in Section 2.1.

The structure of this chapter is similar to the structure of Chapter 2 where we introduced HeyLo and HeyVL. We first define the logical operators for HeyLo using Heyting algebras (Section 5.1) and Gödel algebras (Section 5.2). After that, we discuss the generalization of HeyLo using Heyting and Gödel algebras and relate our work to the broader research context in mathematical logic (Section 5.3). Finally, we discuss the abstraction of HeyVL based on the abstraction of HeyLo in Section 5.4 with a number of ideas for future work.

5.1. Heyting Algebras

Heyting algebras were developed by Arend Heyting in 1930 to formalize intuitionistic logic [Hey30]. A Heyting algebra is a bounded lattice with an implication operator $\to$.

> Definition 5.1. Heyting algebra
>
> A *Heyting algebra* $(H, \sqsubseteq, \sqcup, \sqcap, \to)$ is a bounded lattice $(H, \sqsubseteq, \sqcup, \sqcap)$ with a binary implication operator $\to \colon H \times H \to H$ such that for all $a, b, c \in H$:
>
> $$(c \sqcap a) \sqsubseteq b \quad \text{iff} \quad c \sqsubseteq (a \to b).$$
>
> The *negation* $\neg \colon H \to H$ is defined as $\neg a = a \to \bot$.

The lattice over Booleans, $\mathbb{B} = (\{\, \mathsf{true}, \mathsf{false} \,\}, \Rightarrow, \vee, \wedge, \Rightarrow)$, is a Heyting algebra. Here, the ordering is given by the implication $\Rightarrow \colon \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ itself. We will see in the next section

on Gödel algebras that our construction on the domain $\mathbb{R}^{\infty}_{\geq 0}$ with the Gödel implication also forms a Heyting algebra.

The condition for the implication of Definition 5.1 states that the conjunction $\sqcap$ and implication $\rightarrow$ are *adjoint*. Instead, we choose to work with a set of four equivalent properties on bounded lattices. With these, the relation to the classical implication becomes more apparent.

Theorem 5.2. Heyting implication

A bounded lattice $(H, \sqsubseteq, \sqcup, \sqcap)$ with a binary implication $\rightarrow : H \times H \rightarrow H$ is a Heyting algebra $(H, \sqsubseteq, \sqcup, \sqcap, \rightarrow)$ iff all four of the following properties hold:

1. $a \rightarrow a = \top$,
2. $a \sqcap (a \rightarrow b) = a \sqcap b$,
3. $b \sqcap (a \rightarrow b) = b$,
4. $a \rightarrow (b \sqcap c) = (a \rightarrow b) \sqcap (a \rightarrow c)$.

Proof. A proof can be found in [Joh82, p. 8]. □

All Heyting algebras are distributive. Distributivity is an intuitive property when $\sqcup$ is seen as the logical disjunction $\vee$ and $\sqcap$ is seen as the logical conjunction $\wedge$.

Theorem 5.3. Distributivity of Heyting algebras

Any Heyting algebra $(H, \sqsubseteq, \sqcup, \sqcap, \rightarrow)$ satisfies, for all $a, b, c \in H$,

$$a \sqcup (b \sqcap c) = (a \sqcup b) \sqcap (a \sqcup c)$$
$$a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c)$$

Proof. See [Joh82, p. 8]. □

We use the stronger *infinite distributivity* property in the section on quantifier elimination (Section 4.2). Heyting algebras that are complete as a lattice automatically satisfy the infinite distributivity law. These algebras are also known as *frames* or *complete Heyting algebras* [Joh82, Chapter 2]. For example, the lattice on Booleans $\mathbb{B}$ and the lattice on $\mathbb{R}^{\infty}_{\geq 0}$ are complete and the Heyting algebras previously defined on the respective domains satisfy the infinite distributivity property.

Theorem 5.4. Infinite distributivity of complete Heyting algebras

Any Heyting algebra $(H, \sqsubseteq, \sqcup, \sqcap, \to)$ that is a complete lattice is also *infinitely distributive*, i.e. for all $x \in H$ and $S \subseteq H$:

$$x \sqcap \bigsqcup S = \bigsqcup \{ x \sqcap s \mid s \in S \}$$

Proof. See [Joh82, p. 39]. □

Note that the dual statement for infinite distributivity does not automatically hold, i.e.

$$x \sqcup \bigsqcap S \overset{\not}{=} \bigsqcap \{ x \sqcup s \mid s \in S \}$$

However, it is true when the *dual* lattice of $(H, \sqsubseteq, \sqcup, \sqcap)$ is also a Heyting algebra (by Theorem 5.4). This is the case for our examples on $\mathbb{B}$ and $\mathbb{R}^{\infty}_{\geq 0}$.

The lattice-theoretic dual of a Heyting algebra is called *co-Heyting algebra*.

Definition 5.5. Co-Heyting algebra

Let $(H, \sqsubseteq, \sqcup, \sqcap)$ be a bounded lattice. A *co-Heyting algebra* $(H, \sqsubseteq, \sqcup, \sqcap, \leftarrow)$ has a binary *co-implication* $\leftarrow : H \times H \to H$ such that for all $a, b, c \in H$:

$$b \sqsubseteq (c \sqcup a) \quad \text{iff} \quad (a \leftarrow b) \sqsubseteq c.$$

The *co-negation* $\sim : H \to H$ is defined as $\sim a = a \leftarrow \top$.

The set $\mathbb{R}^{\infty}_{\geq 0}$ with the co-implication we introduced for HeyLo in Section 2.2 is a co-Heyting algebra. Note that $(\mathbb{B}, \Rightarrow, \vee, \wedge, \Leftarrow)$ is not a co-Heyting algebra because $\Leftarrow$ is not a proper co-implication. For example, $\mathsf{true} \Rightarrow (\mathsf{false} \vee \mathsf{false})$ does not hold when $\Rightarrow$ is seen as a relation, but $(\mathsf{false} \Leftarrow \mathsf{true}) \Rightarrow \mathsf{false}$ does hold. Instead, the *converse nonimplication*, sometimes written as $\not\Leftarrow$, is a proper co-implication for a co-Heyting algebra on Booleans. The converse nonimplication $a \not\Leftarrow b$ is logically equivalent to $\neg(a \Leftarrow b)$.

A Heyting algebra with both an implication and a co-implication is called *bi-Heyting algebra*. If it is complete as a lattice, we call it a *complete bi-Heyting algebra*.

Definition 5.6. Bi-Heyting algebra

Let $(H, \sqsubseteq, \sqcup, \sqcap)$ be a bounded lattice. $(H, \sqsubseteq, \sqcup, \sqcap, \to, \leftarrow)$ is a *bi-Heyting algebra* when $(H, \sqsubseteq, \sqcup, \sqcap, \to)$ is a Heyting algebra and $(H, \sqsubseteq, \sqcup, \sqcap, \leftarrow)$ is a co-Heyting algebra.

In a bi-Heyting algebra H, both distributivity laws hold: For all $x \in H$ and $S \subseteq H$:

$$x \sqcap \bigsqcup S = \bigsqcup \{ x \sqcap s \mid s \in S \} \qquad \text{(Theorem 5.4)}$$
$$x \sqcup \bigsqcap S = \bigsqcap \{ x \sqcup s \mid s \in S \} \qquad \text{(Theorem 5.4)}$$

Bi-Heyting algebras provide almost all necessary operators to define semantics for HeyLo (cf. Definition 2.18). One notable exception are the hard implications $\searrow$ and $\nwarrow$. These are nonstandard and our own invention. There are several possible ways one could define them on Heyting algebras, but it is not clear which definition is "most natural" in general. In Gödel algebras, which we define in the next section, there is a natural definition of hard implications.

For general Heyting algebras, one possibility is the definition by $a \searrow b = {\sim}{\sim}(a \to b)$ and $a \nwarrow b = \neg\neg(a \leftarrow b)$ (cf. Section 2.2.3). In this thesis, the hard implications $\searrow$ and $\nwarrow$ are only used for the encoding of specifications (cf. Section 2.3.7) which in turn is used for the encoding of loops in pGCL (cf. Section 3.6). We believe this could be a starting point for possible future research on the further generalization of HeyLo and HeyVL to Heyting algebras which are not Gödel algebras. Although many proofs in this thesis depend on the particular structure of the implication in Gödel algebras, we conjecture that many proofs can be generalized to Heyting algebras.

5.2. Gödel Algebras

Gödel algebras are a subset of Heyting algebras where the implication has the particular structure that we have seen for HeyLo. Additionally, the underlying order of the algebra must be *linear* or *totally ordered* for the Gödel algebra to be a proper Heyting algebra. For every totally ordered and bounded lattice, there is a corresponding Gödel algebra. The set $\mathbb{R}^{\infty}_{\geq 0}$ used throughout this thesis admits a natural total order, as do Booleans $\mathbb{B}$.

Definition 5.7. Gödel algebra

Let $(H, \sqsubseteq, \sqcup, \sqcap)$ be a bounded lattice where $(H, \sqsubseteq)$ is a total order, i.e. for all $a, b \in H$:

$$a \sqsubseteq b \quad \text{or} \quad b \sqsubseteq a \,.$$

We call $(H, \sqsubseteq, \sqcup, \sqcap, \to)$ a *Gödel algebra* where $\to$ is the *Gödel implication*:

$$\to : H \times H \to H$$
$$a \to b = \begin{cases} \top, & \text{if } a \sqsubseteq b \\ b, & \text{else} \end{cases}$$

The *co-* and *bi-*versions are defined analogously.

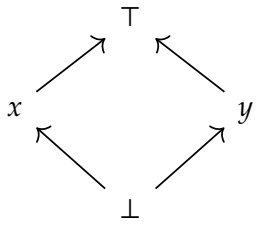

Figure 5.8.: Hasse diagram of a simple bounded order that is not linear.

When the underlying order is not linear, the Gödel implication may not be a proper Heyting implication. Consider Figure 5.8 which depicts a bounded order with four elements: $\top$, x, y, and $\bot$. The elements x and y are not directly comparable, and therefore the order is not linear. This partial order with the Gödel implication violates property 4 of Theorem 5.2. While $x \to (x \sqcap y) = x \to \bot = \bot$, we have $(x \to x) \sqcap (x \to y) = \top \sqcap y = y$. Since $\bot \neq y$, we have a contradiction.

Theorem 5.9. Gödel algebras are Heyting algebras

Let $\mathcal{H} = (H, \sqsubseteq, \sqcup, \sqcap, \to)$ be a Gödel algebra. Then $\mathcal{H}$ is a Heyting algebra.

Proof. Let $(H, \sqsubseteq, \sqcup, \sqcap, \to)$ be a Gödel algebra and $a, b, c \in H$. We show that

$$(c \sqcap a) \sqsubseteq b \quad \text{iff} \quad c \sqsubseteq (a \to b).$$

($\Rightarrow$). Assume $(c \sqcap a) \sqsubseteq b$ holds. Because $(H, \sqsubseteq)$ is a linear order, we have $c \sqsubseteq b \vee a \sqsubseteq b$. In case $c \sqsubseteq b$ holds, we have

$$c \sqsubseteq (a \to b) = \begin{cases} \top, & \text{if } a \sqsubseteq b \\ b, & \text{else} \end{cases}$$

In case $a \sqsubseteq b$ holds, we have $c \sqsubseteq (a \to b) = \top$.

($\Leftarrow$). Assume $c \sqsubseteq (a \to b)$ holds. Then,

$$c \sqsubseteq \begin{cases} \top, & \text{if } a \sqsubseteq b \\ b, & \text{else} \end{cases}$$

Thus, $a \not\sqsubseteq b$ implies $c \sqsubseteq b$. Therefore, $a \sqsubseteq b \vee c \sqsubseteq b$ and we have $(c \sqcap a) \sqsubseteq b$. □

We define the hard implications $\searrow$ and $\nwarrow$ just as we did in HeyLo.

Definition 5.10. Hard implications

$(H, \sqsubseteq, \sqcup, \sqcap)$ be a bounded lattice.
The *hard implication* $\searrow$ and *hard co-implication* $\nwarrow$ are defined as:

$$\searrow : H \times H \to H \qquad\qquad \nwarrow : H \times H \to H$$

$$a \searrow b = \begin{cases} \top, & \text{if } a \sqsubseteq b \\ \bot, & \text{else} \end{cases} \qquad\qquad a \nwarrow b = \begin{cases} \bot, & \text{if } a \sqsupseteq b \\ \top, & \text{else} \end{cases}$$

The hard implications can be seen as syntactic sugar for the non-hard implications with corresponding dual negations.

Theorem 5.11. Hard implications are syntactic sugar

$(H, \sqsubseteq, \sqcup, \sqcap)$ be a Gödel algebra. For all $a, b \in H$:

$$a \searrow b = {\sim}{\sim}(a \to b) \qquad\qquad a \nwarrow b = \neg\neg(a \leftarrow b)$$

Proof. Let $a, b \in H$.

$$a \searrow b = \begin{cases} \top, & \text{if } a \sqsubseteq b \\ \bot, & \text{else} \end{cases} \qquad \text{(definition of } \to\text{)}$$

We have ${\sim}{\sim}\top = \top$. $\neg(a \sqsubseteq b)$ implies $b \neq \top$, then ${\sim}{\sim}b = \neg\top = \bot$.

$$\begin{aligned} &= \begin{cases} {\sim}{\sim}\top, & \text{if } a \sqsubseteq b \\ {\sim}{\sim}b, & \text{else} \end{cases} \\ &= {\sim}{\sim} \begin{cases} \top, & \text{if } a \sqsubseteq b \\ b, & \text{else} \end{cases} \\ &= {\sim}{\sim}(a \to b) \qquad \text{(definition of } \to\text{)} \end{aligned}$$

Similar for $\nwarrow$: $\neg\neg\bot = \bot$ and if $b \neq \bot$, then $\neg\neg b = \neg\bot = \top$. □

The double co-negation ${\sim}{\sim}a$ is equivalent to the *Baaz delta* $\Delta(a)$ [Baa96]. Baaz denotes the dual operator equivalent to the double negation $\neg\neg a$ by $\nabla(a)$. With this notation, we could have defined $a \searrow b = \Delta(a \to b)$ and $a \nwarrow b = \nabla(a \leftarrow b)$.

5.3. Abstraction of HeyLo

Bi-Gödel algebras provide almost all necessary operators for a generalization of the definition of the semantics of HeyLo (Definition 2.18). Only the arithmetic operators $+$, $\cdot$, and the Iverson bracket $[\cdot]$ are specific to the domain $\mathbb{R}^{\infty}_{\geq 0}$. Many results in this thesis do not depend on these operators. In particular, this includes *A Relational View* (Section 2.2.4), our SMT-LIB encoding in Section 4.1 (provided the values and the ordering of the domain can be encoded in SMT-LIB), and our results for quantifier elimination (Section 4.2). In our implementation Caesar (Section 4.3), HeyLo is already implemented with support for the domain $\mathbb{R}^{\infty}_{\geq 0}$ and the Booleans $\mathbb{B}$.

Furthermore, the abstraction of HeyLo to Gödel algebras enables the connection to a rich and well-studied field of intuitionistic logics and Gödel logics, whose models are Gödel algebras. In the following, we briefly outline the history of research on Gödel logics in the hope that future work on HeyLo might benefit from it.

Kurt Gödel was one of the first to provide examples of intermediate logics as far back as the year 1932 [Göd32]. He described what is now called Gödel logic [BPZ07], although there are claims that Thoralf Skolem discovered what is now known as Gödel logic in as far back as 1913 [vPla03]. In 1959, Dummett worked on infinite-valued propositional Gödel logics with the axiom $(A \to B) \vee (B \to A)$ [Dum59]. Infinite-valued propositional logic is also sometimes known as *Gödel-Dummett logic*. According to [BPZ07], first-order Gödel logic on the real interval $[0, 1]$ has been discovered independently by a few people. Horn developed a logic on a linearly ordered Heyting algebra [Hor69] and Takeuti and Titani called it *intuitionistic fuzzy logic* [TT84]. There is also more recent work by Baaz et al. on Gödel logic, especially with quantifiers [BCZ00; Baa+01; BI06; BPZ07; BCP11; BCF12]. The recent thesis of Martins [Mar21] provides an overview of research on properties of general bi-Gödel logics.

HeyLo is a first-order logic on complete bi-Heyting algebras. We believe that our practical work on validity checking HeyLo with a working implementation using our quantifier elimination theorems is a contribution to the field that we have not found in existing literature on Gödel logics. In addition, our use of a Gödel logic for automated deductive verification is new. In this thesis, we have limited our focus on HeyLo on a simple semantics and practical validity checking. A deeper connection to the literature in mathematical logic is a desirable topic for future work.

5.4. Abstraction of HeyVL

Since the semantics of HeyVL are based on HeyLo syntax only, abstracting HeyVL with an abstracted HeyLo is trivial (cf. Definition 2.25). *Distribution expressions* (Section 2.3.2)

are necessarily specific to the domain. The constructions for verifying implementations (Section 2.3.6) and encoding specifications (Section 2.3.7) work with any Gödel algebra. Similarly, we expect most of the chapter on encoding pGCL (Chapter 3) to be applicable in other contexts as well.

There is a number of additional domains that can be investigated for use in HeyVL. Our implementation already supports the Booleans $\mathbb{B}$ as the Heyting algebra backing HeyLo and HeyVL. With the down fragment of HeyVL on $\mathbb{B}$, we recover the simple IVL presented in Section 2.1. Future research could investigate the relation of the up fragment to deductive verification on $\mathbb{B}$.

When HeyVL is used on the $[0, 1]$ domain of real numbers between zero and one, we can directly embed the wlp calculus without any possible invalid valuations. In the encodings of pGCL with wlp (Chapter 3), particular care had to be taken that the encoding remained within the $[0, 1]$ domain and did not accidentally evaluate to values like ∞ at certain places. With HeyVL semantics limited to $[0, 1]$, this cannot happen.

The domain of natural numbers $\mathbb{N}$ could be used for deductive verification of (non-probabilistic) amortized analysis. For probabilistic programs, the extension to *mixed-sign expectations* [KK17] is interesting. We believe that a definition of mixed-sign expectations as a complete lattice is possible. However, such a definition is unlikely to fulfill HeyLo's current requirement of a linear order, further motivating the generalization of HeyLo and HeyVL to domains with orders that are not linear.

6. Conclusion

In this thesis, we have presented the theoretical foundations and a prototypical implementation of a deductive verifier for probabilistic programs. HeyLo is an assertion language for probabilistic programs based on Gödel logic, adding an implication and a co-implication to expectations. It is a conservative extension of first-order Boolean logic and is has a *relatively complete* subset with respect to wp. Our implementation uses an SMT encoding of HeyLo together with quantifier elimination in HeyLo to implement validity checks of HeyLo formulas.

HeyVL is an intermediate verification language for probabilistic programs. It is based on HeyLo and allows the verification of both lower and upper bound problems. At the moment, only simple distribution expressions are supported, but HeyVL can be easily extended with new distribution expressions. With minor restrictions, the verification condition semantics satisfies well-known healthiness conditions, such as monotonicity, continuity, and (co-)feasibility. We have presented generic encodings for specifications and the verification of implementations in HeyVL.

As an example for the application of HeyVL, we have encoded pGCL into HeyVL with respect to both its wp and wlp semantics. Based on a composable encoding based on down- and up-approximations of semantics, we have shown that HeyVL can be used to encode not only primitive pGCL constructs such as random assignments, but also encode more complex proof rules like Park induction, k-induction, and bounded model checking. As our encodings are modular, we are also able to encode proofs with nested loops in our framework without any additional work.

Our prototypical implementation shows that our framework does not only work in theory, but that it is possible to efficiently check the validity of the verification conditions for a set of HeyVL programs.

Finally, the abstraction of HeyLo and HeyVL based on Gödel algebras connects our work with the existing research on Gödel logics and opens the possibility of re-use of HeyLo and HeyVL for other domains in a very natural way.

We discussed some possible future work in the chapter on abstraction already (Chapter 5). There are many exciting directions for further research.

The implementation is so far only a prototype and can benefit from the large number of optimizations that are already implemented in tools such as Boogie or Viper. Additional

features, such as user-defined datatypes, procedures, or parallelization are also of interest. Further generalization of the verification domain from $\mathbb{R}^{\infty}_{\geq 0}$ to other domains such as mixed-sign expectations [KK17] might be possible. More built-in functions in HeyLo and HeyVL are also planned. Finally, formal validation of the verification condition generation in the style of [PMS21] is interesting.

The theory of HeyLo and HeyVL could be developed further to not only support additional domains, but also to encode ideas like *sensitivity analysis* [Agu+21]. Finally, we are interested in the encoding of proof rules for `observe`-like statements, for example based on conditional pre-expectation semantics [Jan+15; Olm+18].

A. Omitted Proofs

A Relational View (Theorem 2.12)

Proof. For readability, we provide the proof on the domain $\mathbb{R}^{\infty}_{\geq 0}$ instead of on $\mathbb{E} = \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$. From these results, one merely needs to lift the statements to functions over states Σ and the relation $\sqsubseteq$. Let $X, Y, Z \in \mathbb{R}^{\infty}_{\geq 0}$ and $S \subseteq \mathbb{R}^{\infty}_{\geq 0}$.

- $Z \leq \infty$ is true in any lattice, because ∞ is the top element of $\mathbb{R}^{\infty}_{\geq 0}$.
- $Z \leq 0$ iff $Z = 0$:

  ($\Rightarrow$) $Z \leq 0 \Rightarrow Z = 0$, because 0 is the bottom element of $\mathbb{R}^{\infty}_{\geq 0}$.

  ($\Leftarrow$) $Z = 0 \Rightarrow p \leq 0$ is trivial.
- $Z \leq X \sqcap Y$ iff $Z \leq X \wedge Z \leq Y$:

  ($\Rightarrow$) $Z \leq X \sqcap Y \Rightarrow Z \leq X \sqcap Y \leq X \wedge Z \leq X \sqcap Y \leq Y$.

  ($\Leftarrow$) $Z \leq X \wedge Z \leq Y \Rightarrow Z \leq X \sqcap Y$, because $X \sqcap Y$ is the greatest lower bound of both X and Y.
- $Z \leq X \sqcup Y$ iff $Z \leq X \vee Z \leq Y$:

  ($\Rightarrow$) $Z \leq X \sqcup Y \Rightarrow Z \leq X \vee Z \leq Y$, because $\mathbb{R}^{\infty}_{\geq 0}$ is *join prime*.

  ($\Leftarrow$) $Z \leq X \wedge Z \leq Y \Rightarrow Z \leq X \sqcup Y$, because $X \leq X \sqcup Y$ and $Y \leq X \sqcup Y$.
- $Z \leq X \to Y$ iff $X \leq Y \vee Z \leq Y$:

  ($\Rightarrow$) If $X \leq Y$, $Z \leq X \to Y = \infty \Leftrightarrow \mathsf{true}$.
  On the other hand, if $X \not\sqsubseteq Y$, then $Z \leq X \to Y = Y$.

  ($\Leftarrow$) If $X \leq Y$, then $Z \leq X \to Y = \infty$. If $Z \leq Y$, then $Z \leq X \to Y \in \{ \infty, Y \}$.
- $Z \leq X \searrow Y$ iff $X \leq Y \vee Z = 0$:

  ($\Rightarrow$) If $X \leq Y$, $Z \leq X \searrow Y = \infty \Leftrightarrow \mathsf{true}$.
  On the other hand, if $X \not\sqsubseteq Y$, then $Z \leq X \to Y = 0$.

  ($\Leftarrow$) If $X \leq Y$, then $Z \leq X \searrow Y = \infty$. If $Z = 0$, then $0 = Z \leq X \to Y$.

- We have the lattice-theoretic statement for $\inf S \Leftrightarrow \forall Y \in S.\ Z \leq Y$ and its dual in Lemma 4.1.

The dual statements admit dual proofs and are omitted here. □

Monotone Fragment (Theorem 2.31)

Proof. We do an induction over the structure of $S \in \mathsf{HeyVL}_{\mathsf{mon}}$. Let $\varphi, \varphi' \in \mathsf{HeyLo}$ such that $\varphi \sqsubseteq \varphi'$.

Base cases:

- $S = \texttt{skip}$:

$$\begin{aligned}
&\mathsf{vc}[\![\texttt{skip}]\!](\varphi) \\
&= \varphi && \text{(definition of skip)} \\
&\sqsubseteq \varphi' = \mathsf{vc}[\![\texttt{skip}]\!](\varphi') && \text{(definition of skip)}
\end{aligned}$$

- $S = x :\approx a$ where $a \in \mathsf{ArithExp}$:

$$\begin{aligned}
&\mathsf{vc}[\![x :\approx a]\!](\varphi) \\
&= \varphi[x \mapsto a] && \text{(definition of } :\approx) \\
&\sqsubseteq \varphi'[x \mapsto a] && (\varphi \sqsubseteq \varphi') \\
&= \mathsf{vc}[\![x :\approx a]\!](\varphi') && \text{(definition of } :\approx)
\end{aligned}$$

- $S = x :\approx \mathsf{ber!}(p)$:

$$\begin{aligned}
&\mathsf{vc}[\![x :\approx \mathsf{ber!}(p)]\!](\varphi) \\
&= p \cdot \varphi[x \mapsto 1] + (1-p) \cdot \varphi[x \mapsto 0] && \text{(definition of ber!}(p)\text{)} \\
&\sqsubseteq p \cdot \varphi'[x \mapsto 1] + (1-p) \cdot \varphi'[x \mapsto 0] && (\varphi \sqsubseteq \varphi') \\
&= \mathsf{vc}[\![x :\approx \mathsf{ber!}(p)]\!](\varphi') && \text{(definition of ber!}(p)\text{)}
\end{aligned}$$

- $S = \mathsf{down\ havoc}\ x$:

$$\begin{aligned}
&\mathsf{vc}[\![\mathsf{down\ havoc}\ x]\!](\varphi) \\
&= \inf \left\{ \varphi[x \mapsto v] \;\middle|\; v \in \mathbb{Q}_{\geq 0} \right\} && \text{(definition of havoc)} \\
&\sqsubseteq \inf \left\{ \varphi'[x \mapsto v] \;\middle|\; v \in \mathbb{Q}_{\geq 0} \right\} && (\varphi \sqsubseteq \varphi') \\
&= \mathsf{vc}[\![\mathsf{down\ havoc}\ x]\!](\varphi')
\end{aligned}$$

- $S = \mathsf{down}\ \mathtt{assert}\ \psi$:

$$\begin{aligned}
&\mathsf{vc}[\![\mathsf{down}\ \mathtt{assert}\ \psi]\!](\varphi) \\
&= \psi \sqcap \varphi && \text{(definition of assert)} \\
&\sqsubseteq \psi \sqcap \varphi' && (\varphi \sqsubseteq \varphi') \\
&= \mathsf{vc}[\![\mathsf{down}\ \mathtt{assert}\ \psi]\!](\varphi') && \text{(definition of assert)}
\end{aligned}$$

- $S = \mathsf{down}\ \mathtt{assume}\ \psi$:

$$\begin{aligned}
&\mathsf{vc}[\![\mathsf{down}\ \mathtt{assume}\ \psi]\!](\varphi) \\
&= \psi \to \varphi && \text{(definition of assume)}
\end{aligned}$$

Let $\sigma \in \Sigma$.[Fixed] If $\psi \sqsubseteq \varphi$, then also $\psi \sqsubseteq \varphi'$ because $\varphi \sqsubseteq \varphi'$. Then, $[\![\psi \to \varphi]\!](\sigma) \equiv \infty \equiv [\![\psi \to \varphi']\!](\sigma)$. Otherwise, $[\![\psi \to \varphi]\!](\sigma) \equiv \varphi \sqsubseteq [\![\psi \to \varphi']\!](\sigma) \in \infty, [\![\psi']\!](\sigma)$. Also compare Figure 2.30.

$$\begin{aligned}
&\sqsubseteq \psi \to \varphi' \\
&= \mathsf{vc}[\![\mathsf{down}\ \mathtt{assume}\ \psi]\!](\varphi') && \text{(definition of assume)}
\end{aligned}$$

- $S = \mathsf{down}\ \mathtt{compare}\ \psi$:

$$\begin{aligned}
&\mathsf{vc}[\![\mathsf{down}\ \mathtt{compare}\ \psi]\!](\varphi) \\
&= \psi \searrow \varphi && \text{(definition of compare)}
\end{aligned}$$

Let $\sigma \in \Sigma$. If $\psi \sqsubseteq \varphi$, then also $\psi \sqsubseteq \varphi'$ because $\varphi \sqsubseteq \varphi'$. Then, $[\![\psi \searrow \varphi]\!](\sigma) \equiv \infty \equiv [\![\psi \searrow \varphi']\!](\sigma)$. Otherwise, $[\![\psi \searrow \varphi]\!](\sigma) \equiv \bot \sqsubseteq [\![\psi \searrow \varphi']\!](\sigma)$.

$$\begin{aligned}
&\sqsubseteq \psi \searrow \varphi' \\
&= \mathsf{vc}[\![\mathsf{down}\ \mathtt{compare}\ \psi]\!](\varphi') && \text{(definition of compare)}
\end{aligned}$$

The up cases are dual, but we show the up assume case for illustration:

- $S = \mathsf{up}\ \mathtt{assume}\ \psi$:

$$\begin{aligned}
&\mathsf{vc}[\![\mathsf{up}\ \mathtt{assume}\ \psi]\!](\varphi) \\
&\equiv \psi \leftarrow \varphi && \text{(definition of assume)}
\end{aligned}$$

Let $\sigma \in \Sigma$. If $\varphi' \sqsubseteq \psi$, then also $\varphi \sqsubseteq \psi$ because $\varphi \sqsubseteq \varphi'$. Then, $[\![\psi \to \varphi']\!](\sigma) \equiv 0 \equiv [\![\psi \to \varphi']\!](\sigma)$. Otherwise, $[\![\psi \to \varphi']\!](\sigma) \equiv \varphi' [\![\psi \to \varphi]\!](\sigma) \in 0, [\![\psi]\!](\sigma)$. Also compare Figure 2.30.

$$\begin{aligned}
&\sqsubseteq \psi \leftarrow \varphi' \\
&= \mathsf{vc}[\![\mathsf{up}\ \mathtt{assume}\ \psi]\!](\varphi') && \text{(definition of assume)}
\end{aligned}$$

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

Induction step:

- $S = S_1 \,;\, S_2$:
  We use the induction hypothesis for S_2: Because $\varphi \sqsubseteq \varphi'$ holds, we have

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

  Now use the induction hypothesis for S_1 with $\varphi \mapsto \mathsf{vc}[\![S_2]\!](\varphi)$ and $\varphi' \mapsto \mathsf{vc}[\![S_2]\!](\varphi')$. Because $\mathsf{vc}[\![S_2]\!](\varphi) \sqsubseteq \mathsf{vc}[\![S_2]\!](\varphi')$ holds, we get:

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

  Applying definitions, we get:

  $$\begin{aligned}
  \mathsf{vc}[\![S_1 \,;\, S_2]\!](\varphi) &= \mathsf{vc}[\![S_1]\!](\mathsf{vc}[\![S_2]\!](\varphi)) && \text{(definition of ;)} \\
  &\sqsubseteq \mathsf{vc}[\![S_1]\!](\mathsf{vc}[\![S_2]\!](\varphi')) && \text{(above)} \\
  &= \mathsf{vc}[\![S_1 \,;\, S_2]\!](\varphi') && \text{(definition of ;)}
  \end{aligned}$$

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

  $$\begin{aligned}
  &\mathsf{vc}[\![S_1 \,;\, S_2]\!](\varphi) \\
  &\qquad = \mathsf{vc}[\![S_1]\!](\varphi) \sqcap \mathsf{vc}[\![S_2]\!](\varphi) && \text{(definition of ;)} \\
  &\qquad \sqsubseteq \mathsf{vc}[\![S_1]\!](\varphi') \sqcap \mathsf{vc}[\![S_2]\!](\varphi) && \text{(induction hypothesis)} \\
  &\qquad \sqsubseteq \mathsf{vc}[\![S_1]\!](\varphi') \sqcap \mathsf{vc}[\![S_2]\!](\varphi') && \text{(induction hypothesis)} \\
  &\qquad = \mathsf{vc}[\![S_1 \,;\, S_2]\!](\varphi') && \text{(definition of ;)}
  \end{aligned}$$

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

By the principle of induction over the program structure, Theorem 2.31 holds. □

(Co-)Feasible Fragments (Theorem 2.32)

Proof. Let $\varphi \in \mathsf{HeyLo}$ and $r \in \mathbb{R}^\infty_{\geq 0}$. Instead of the statements

$$\begin{aligned}
\forall S \in \mathsf{HeyVL}_{\mathsf{feas}}.\quad \varphi \sqsubseteq r \quad &\Rightarrow \quad \mathsf{vc}[\![S]\!](\varphi) \sqsubseteq r, \\
\forall S \in \mathsf{HeyVL}_{\mathsf{cof}}.\quad r \sqsubseteq \varphi \quad &\Rightarrow \quad r \sqsubseteq \mathsf{vc}[\![S]\!](\varphi) \,,
\end{aligned}$$

we show the following simpler statements. From the statements below, the above statements follow by monotonicity of $\mathsf{vc}[\![S]\!]$, using e.g. $\mathsf{vc}[\![S]\!](\varphi) \sqsubseteq \mathsf{vc}[\![S]\!](r)$ for the $\mathsf{HeyVL}_{\mathsf{feas}}$

case. We know $\mathsf{vc}[\![S]\!]$ is monotone because both $\mathsf{HeyVL}_{\mathsf{feas}}$ and $\mathsf{HeyVL}_{\mathsf{cof}}$ are subsets of $\mathsf{HeyVL}_{\mathsf{mon}}$ (cf. Theorem 2.31).

$$\forall S \in \mathsf{HeyVL}_{\mathsf{feas}}. \quad \mathsf{vc}[\![S]\!](r) \sqsubseteq r,$$
$$\forall S \in \mathsf{HeyVL}_{\mathsf{cof}}. \quad r \sqsubseteq \mathsf{vc}[\![S]\!](r).$$

We start with $\mathsf{HeyVL}_{\mathsf{feas}}$. We do a proof over the structure of $S \in \mathsf{HeyVL}_{\mathsf{feas}}$.

Base cases:

- $S = \texttt{skip}$: We have $\mathsf{vc}[\![\texttt{skip}]\!](r) = r$.
- $S = x :\approx a$ where $a \in \mathsf{ArithExp}$: $\mathsf{vc}[\![x :\approx a]\!](r) = r[x \mapsto a] = r$.
- $S = x :\approx \mathsf{ber!}(p)$: $\mathsf{vc}[\![x :\approx \mathsf{ber!}(p)]\!](r) = 0.5 \cdot r[x \mapsto 1] + 0.5 \cdot r[x \mapsto 0] = 0.5 \cdot r + 0.5 \cdot r = r$.
- $S = \mathsf{down\ havoc}\ x$: $\mathsf{vc}[\![\mathsf{down\ havoc}\ x]\!](r) = \inf \{\, r[x \mapsto v] \mid v \in \mathbb{Q}_{\geq 0} \,\} = \inf \{\, r \,\} = r$.
- $S = \mathsf{down}\ \texttt{assert}\ \psi$: $\mathsf{vc}[\![\mathsf{down}\ \texttt{assert}\ \psi]\!](r) = \psi \sqcap r \sqsubseteq r$.
- $S = \mathsf{up}\ \texttt{assume}\ \psi$: $\mathsf{vc}[\![\mathsf{up}\ \texttt{assume}\ \psi]\!](r) = \psi \leftarrow r \sqsubseteq r$ since $[\![\psi \leftarrow r]\!](\sigma) \in \{\, 0, r \,\}$ for all $\sigma \in \Sigma$.

Assume the induction hypothesis holds for arbitrary, but fixed $S_1, S_2 \in \mathsf{HeyVL}_{\mathsf{feas}}$.

Induction step:

- $S = S_1 \,;\, S_2$: $\mathsf{vc}[\![S_1 \,;\, S_2]\!](r) = \mathsf{vc}[\![S_1]\!](\mathsf{vc}[\![S_2]\!](r))$. By the induction hypothesis, $\mathsf{vc}[\![S_2]\!](r) \sqsubseteq r$ and $\mathsf{vc}[\![S_1]\!](\mathsf{vc}[\![S_2]\!](r)) \sqsubseteq \mathsf{vc}[\![S_1]\!](r) \sqsubseteq r$.
- $S = \texttt{if}\ (\sqcap)\ \{S_1\}\ \texttt{else}\ \{S_2\}$: $\mathsf{vc}[\![\texttt{if}\ (\sqcap)\ \{S_1\}\ \texttt{else}\ \{S_2\}]\!](r) = \mathsf{vc}[\![S_1]\!](r) \sqcap \mathsf{vc}[\![S_2]\!](r) \sqsubseteq r \sqcap r = r$ by the induction hypothesis.
- $S = \texttt{if}\ (\sqcup)\ \{S_1\}\ \texttt{else}\ \{S_2\}$: $\mathsf{vc}[\![\texttt{if}\ (\sqcup)\ \{S_1\}\ \texttt{else}\ \{S_2\}]\!](r) = \mathsf{vc}[\![S_1]\!](r) \sqcup \mathsf{vc}[\![S_2]\!](r) \sqsubseteq r \sqcap r = r$ by the induction hypothesis.

By the principle of induction over the structure of the program and Theorem 2.31, the statement for $\mathsf{HeyVL}_{\mathsf{feas}}$ holds. The proof for $\mathsf{HeyVL}_{\mathsf{cof}}$ is dual. □

B. Bibliography

[Agu+21] A. Aguirre et al. "A Pre-Expectation Calculus for Probabilistic Sensitivity". In: *Proceedings of the ACM on Programming Languages* 5 (POPL 2021), 52:1–52:28.

[Ast+19] V. Astrauskas et al. "Leveraging Rust Types for Modular Specification and Verification". In: *Proceedings of the ACM on Programming Languages* 3 (OOPSLA 2019), 147:1–147:30.

[Baa+01] M. Baaz et al. "A Guide to Quantified Propositional Gödel Logic". In: *IJCAR Workshop QBF*. 2001.

[Baa96] M. Baaz. "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. Springer, 1996, pp. 23–33.

[Bac78] R. J. Back. *On the Correctness of Refinement Steps in Program Development*. Department of Computer Science, University of Helsinki Helsinki, Finland, 1978.

[Bac88] R. J. R. Back. "A Calculus of Refinements for Program Derivations". In: *Acta Informatica* 25.6 (1988), pp. 593–624.

[Bat+21a] K. Batz et al. "Latticed K-Induction with an Application to Probabilistic Programs". In: *CAV (2)*. Lecture Notes in Computer Science. Springer, 2021, pp. 524–549.

[Bat+21b] K. Batz et al. "Relatively Complete Verification of Probabilistic Programs: An Expressive Language for Expectation-Based Reasoning". In: *Proceedings of the ACM on Programming Languages* 5 (POPL 2021), 39:1–39:30.

[BCF12] M. Baaz, A. Ciabattoni, and C. G. Fermüller. "Theorem Proving for Prenex Gödel Logic with Delta: Checking Validity and Unsatisfiability". In: *Logical Methods in Computer Science* 8.1 (2012), p. 20.

[BCP11] M. Baaz, A. Ciabattoni, and N. Preining. "First-Order Satisfiability in Gödel Logics: An NP-complete Fragment". In: *Theoretical Computer Science* 412.47 (2011), pp. 6612–6623.

[BCZ00] M. Baaz, A. Ciabattoni, and R. Zach. "Quantified Propositional Gödel Logics". In: *Logic for Programming and Automated Reasoning*. Lecture Notes in Artificial Intelligence. Springer, 2000, pp. 240–256.

[BFT17] C. Barrett, P. Fontaine, and C. Tinelli. *The SMT-LIB Standard: Version 2.6*. Department of Computer Science, The University of Iowa, 2017.

[BI06] M. Baaz and R. Iemhoff. "The Skolemization of Existential Quantifiers in Intuitionistic Logic". In: *Annals of Pure and Applied Logic* 142.1 (2006), pp. 269–295.

[Bie+99] A. Biere et al. "Symbolic Model Checking without BDDs". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science. Springer, 1999, pp. 193–207.

[BK08] C. Baier and J.-P. Katoen. *Principles of Model Checking*. Illustrated Edition. The MIT Press, 2008. 984 pp.

[BKS20] G. Barthe, J.-P. Katoen, and A. Silva, eds. *Foundations of Probabilistic Programming*. Cambridge University Press, 2020.

[Bor+11] J. Borgström et al. "Measure Transformer Semantics for Bayesian Machine Learning". In: *Programming Languages and Systems*. Lecture Notes in Computer Science. Springer, 2011, pp. 77–96.

[BPZ07] M. Baaz, N. Preining, and R. Zach. "First-Order Gödel Logics". In: *Annals of Pure and Applied Logic* 147.1 (2007), pp. 23–47.

[BvW98] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. 1998.

[CC77] P. Cousot and R. Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '77. ACM, 1977, pp. 238–252.

[Cla+13] G. Claret et al. "Bayesian Inference Using Data Flow Analysis". In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*. The 2013 9th Joint Meeting. ACM Press, 2013, p. 92.

[CM12] P. Cousot and M. Monerau. "Probabilistic Abstract Interpretation". In: *Programming Languages and Systems*. Lecture Notes in Computer Science. Springer, 2012, pp. 169–193.

[Coo78] S. A. Cook. "Soundness and Completeness of an Axiom System for Program Verification". In: *SIAM Journal on Computing* 7.1 (1978), pp. 70–90.

[CS13] A. Chakarov and S. Sankaranarayanan. "Probabilistic Program Analysis with Martingales". In: *Computer Aided Verification*. Red. by D. Hutchison et al. Vol. 8044. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 511–526.

[CS14] A. Chakarov and S. Sankaranarayanan. "Expectation Invariants for Probabilistic Program Loops as Fixed Points". In: *Static Analysis*. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 85–100.

[Dij75] E. W. Dijkstra. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs". In: *Communications of the ACM* 18.8 (1975), pp. 453–457.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. 246 pp.

[dMB08] L. de Moura and N. Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340.

[Don+11] A. F. Donaldson et al. "Software Verification Using K-Induction". In: *Static Analysis*. Vol. 6887. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 351–368.

[Dum59] M. Dummett. "A Propositional Calculus with Denumerable Matrix". In: *The Journal of Symbolic Logic* 24.2 (1959), pp. 97–106.

[EM18] M. Eilers and P. Müller. "Nagini: A Static Verifier for Python". In: *Computer Aided Verification*. Lecture Notes in Computer Science. Springer International Publishing, 2018, pp. 596–603.

[FH15] L. M. Ferrer Fioriti and H. Hermanns. "Probabilistic Termination: Soundness, Completeness, and Compositionality". In: *ACM SIGPLAN Notices* 50.1 (2015), pp. 489–501.

[FP13] J.-C. Filliâtre and A. Paskevich. "Why3 — Where Programs Meet Provers". In: *Programming Languages and Systems*. Lecture Notes in Computer Science. Springer, 2013, pp. 125–128.

[GKM14] F. Gretz, J.-P. Katoen, and A. McIver. "Operational versus Weakest Pre-Expectation Semantics for the Probabilistic Guarded Command Language". In: *Performance Evaluation*. Special Issue on the 9th International Conference on Quantitative Evaluation of Systems 73 (2014), pp. 110–132.

[GNV05] M. Gehrke, H. Nagahashi, and Y. Venema. "A Sahlqvist Theorem for Distributive Modal Logic". In: *Annals of Pure and Applied Logic* 131.1 (2005), pp. 65–102.

[Göd32] K. Gödel. "Zum Intuitionistischen Aussagenkalkül". In: *Anzeiger der Akademie der Wissenschaften in Wien* 69 (1932), pp. 65–66.

[Har+19] M. Hark et al. "Aiming Low Is Harder: Induction for Lower Bounds in Probabilistic Program Verification". In: *Proceedings of the ACM on Programming Languages* 4 (POPL 2019), 37:1–37:28.

[Hey30] A. Heyting. *Die formalen Regeln der intuitionistischen Logik*. 1930.

[Hin+16] W. Hino et al. “Healthiness from Duality”. In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS ’16. Association for Computing Machinery, 2016, pp. 682–691.

[HMM05] J. Hurd, A. McIver, and C. Morgan. “Probabilistic Guarded Commands Mechanized in HOL”. In: *Electron. Notes Theor. Comput. Sci.* (2005).

[Hoa69] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Commun. ACM* 12.10 (1969), p. 6.

[Hor69] A. Horn. “Logic with Truth Values in a Linearly Ordered Heyting Algebra”. In: *The Journal of Symbolic Logic* 34.3 (1969), pp. 395–408.

[Hur+14] C.-K. Hur et al. “Slicing Probabilistic Programs”. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’14: ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, 2014, pp. 133–144.

[Ive62] K. E. Iverson. “A Programming Language”. In: *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*. AIEE-IRE ’62 (Spring). Association for Computing Machinery, 1962, pp. 345–351.

[Jan+15] N. Jansen et al. “Conditioning in Probabilistic Programming”. In: *Electron. Notes Theor. Comput. Sci.* 319.C (2015), pp. 199–216.

[Joh82] P. Johnstone. *Stone Spaces*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1982.

[Kam+16] B. L. Kaminski et al. “Weakest Precondition Reasoning for Expected Run–Times of Probabilistic Programs”. In: *Programming Languages and Systems*. Lecture Notes in Computer Science. Springer, 2016, pp. 364–389.

[Kam+18] B. L. Kaminski et al. “Weakest Precondition Reasoning for Expected Run-times of Randomized Algorithms”. In: *Journal of the ACM* 65.5 (2018), 30:1–30:68.

[Kam19] B. L. Kaminski. “Advanced Weakest Precondition Calculi for Probabilistic Programs”. PhD thesis. RWTH Aachen University, 2019.

[Kei15] K. Keimel. “Healthiness Conditions for Predicate Transformers”. In: *Electronic Notes in Theoretical Computer Science*. The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI). 319 (2015), pp. 255–270.

[KK17] B. L. Kaminski and J.-P. Katoen. “A Weakest Pre-Expectation Semantics for Mixed-Sign Expectations”. In: *Proceedings of the 32nd Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS ’17. IEEE Press, 2017, pp. 1–12.

[KKM19] B. L. Kaminski, J.-P. Katoen, and C. Matheja. “On the Hardness of Analyzing Probabilistic Programs”. In: *Acta Informatica* 56.3 (2019), pp. 255–285.

[Koz79] D. Kozen. "Semantics of Probabilistic Programs". In: *20th Annual Symposium on Foundations of Computer Science (Sfcs 1979)*. 20th Annual Symposium on Foundations of Computer Science (Sfcs 1979). 1979, pp. 101–114.

[Koz81] D. Kozen. "Semantics of Probabilistic Programs". In: *Journal of Computer and System Sciences* 22.3 (1981), pp. 328–350.

[Koz83] D. Kozen. "A Probabilistic PDL". In: *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*. STOC '83. Association for Computing Machinery, 1983, pp. 291–297.

[Koz85] D. Kozen. "A Probabilistic PDL". In: *Journal of Computer and System Sciences* 30.2 (1985), pp. 162–178.

[KR92] E. Kushilevitz and M. O. Rabin. "Randomized Mutual Exclusion Algorithms Revisited". In: *Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing*. PODC '92. Association for Computing Machinery, 1992, pp. 275–283.

[Lei08] K. R. M. Leino. *This Is Boogie 2*. 2008.

[Lei10] K. R. M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Lecture Notes in Computer Science. Springer, 2010, pp. 348–370.

[LMS09] K. R. M. Leino, P. Müller, and J. Smans. "Verification of Concurrent Programs with Chalice". In: *Foundations of Security Analysis and Design V*. Vol. 5705. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 195–222.

[Mar21] M. Martins. "Bi-Gödel Algebras and Co-Trees". MA thesis. University of Amsterdam, 2021. 116 pp.

[MM05] A. McIver and C. C. Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Monographs in Computer Science. Springer-Verlag, 2005.

[Mor87] J. M. Morris. "A Theoretical Basis for Stepwise Refinement and the Programming Calculus". In: *Science of Computer Programming* 9.3 (1987), pp. 287–306.

[Mor88] C. Morgan. "The Specification Statement". In: *ACM Transactions on Programming Languages and Systems* 10.3 (1988), pp. 403–419.

[Mor94] C. Morgan. *Programming from Specifications (2nd Ed.)* Prentice Hall International (UK) Ltd., 1994. 332 pp.

[MSS16] P. Müller, M. Schwerhoff, and A. J. Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning". In: *Verification, Model Checking, and Abstract Interpretation*. Lecture Notes in Computer Science. Springer, 2016, pp. 41–62.

[Mülo2] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. Lecture Notes in Computer Science. Springer-Verlag, 2002.

[Mül19] P. Müller. "Building Deductive Program Verifiers - Lecture Notes". In: *Engineering Secure and Dependable Software Systems* (2019), pp. 189–206.

[NCH18] V. C. Ngo, Q. Carbonneaux, and J. Hoffmann. "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, 2018, pp. 496–512.

[Nor+14] A. V. Nori et al. "R2: An Efficient MCMC Sampler for Probabilistic Programs". In: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*. AAAI'14. AAAI Press, 2014, pp. 2476–2482.

[Olm+16] F. Olmedo et al. "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, 2016, pp. 672–681.

[Olm+18] F. Olmedo et al. "Conditioning in Probabilistic Programming". In: *ACM Transactions on Programming Languages and Systems* 40.1 (2018), 4:1–4:50.

[Par69] D. Park. "Fixpoint Induction and Proofs of Program Properties". In: *Machine Intelligence* 5 (1969).

[PMS21] G. Parthasarathy, P. Müller, and A. J. Summers. "Formally Validating a Practical Verification Condition Generator". In: *Computer Aided Verification*. Vol. 12760. Lecture Notes in Computer Science. Springer International Publishing, 2021, pp. 704–727.

[Pre10] N. Preining. "Gödel Logics – A Survey". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Lecture Notes in Computer Science. Springer, 2010, pp. 30–51.

[PW15] A. Pierro and H. Wiklicky. "Probabilistic Abstract Interpretation: From Trace Semantics to DTMC's and Linear Regression". In: *Essays Dedicated to Hanne Riis Nielson and Flemming Nielson on the Occasion of Their 60th Birthdays on Semantics, Logics, and Calculi - Volume 9560*. Springer-Verlag, 2015, pp. 111–139.

[Rey02] J. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 17th Annual IEEE Symposium on Logic in Computer Science. IEEE Comput. Soc, 2002, pp. 55–74.

[SSS00] M. Sheeran, S. Singh, and G. Stålmarck. "Checking Safety Properties Using Induction and a SAT-Solver". In: *FMCAD*. 2000.

[Tar55] A. Tarski. "A Lattice-Theoretical Fixpoint Theorem and Its Applications." In: *Pacific Journal of Mathematics* 5.2 (1955), pp. 285–309.

[TT84] G. Takeuti and S. Titani. "Intuitionistic Fuzzy Logic and Intuitionistic Fuzzy Set Theory". In: *The Journal of Symbolic Logic* 49.3 (1984), pp. 851–866.

[Tur37] A. M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* s2-42.1 (1937), pp. 230–265.

[vPla03] J. von Plato. "Skolem's Discovery of Gödel-Dummett Logic". In: *Studia Logica* 73.1 (2003), pp. 153–157.

[Wil+21] M. Willsey et al. "Egg: Fast and Extensible Equality Saturation". In: *Proceedings of the ACM on Programming Languages* 5 (POPL 2021), 23:1–23:29.

[Wol+21] F. A. Wolf et al. "Gobra: Modular Specification and Verification of Go Programs". In: *Computer Aided Verification*. Lecture Notes in Computer Science. Springer International Publishing, 2021, pp. 367–379.

C. Index

C.1. Definitions

C.2. Theorems

C.3. Lemmas

C.4. Examples

C.5. Figures

D. Errata

This document is a revised version of the original submitted master's thesis. Below is a list of changes made since the submission. Changes are marked in the document by FIXED.

Theory.

- Section 2.2, Page 12: Removed a sentence suggesting $\Leftarrow$ is the lattice-theoretic dual to $\Rightarrow$ on Booleans. The lattice-theoretic dual to $A \Rightarrow B$ is the converse non-implication $\neg(A \Leftarrow B)$.

- Section 3.6, Page 68: Bounded model checking iteration for wlp should start at 1, not at ∞.

- Section 4.2, Page 89 and Page 92: The scalars a in Theorem 4.11 and c in Theorem 4.14 must range over finite nonnegative reals $\mathbb{R}_{\geq 0}$, not over $\mathbb{R}^{\infty}_{\geq 0}$. As a counterexample for $a = \infty$, let $\psi = y + ?(y = 0)$. Then $\psi(0) = \infty$ and $\psi(y) = y$ for $y > 0$, so ψ has infimum 0 without ever attaining 0. Therefore $\infty \cdot (\bigsqcap_y \psi) = 0$, but $\infty \cdot \psi(y) = \infty$ for all y, and hence $\bigsqcap_y(\infty \cdot \psi) = \infty$.

- Section 4.2, Page 94: Clarified that the computed quantifier-free formula preserves validity, while only the prenexing step preserves equivalence.

- Section 4.2, Page 95 and Page 96: Definition 4.16 was restricted. From $S_{\sqcap}$, the alternatives up assume f_{QF} and up compare f_{QF} were removed, and from $S_{\sqcup}$, the alternatives down assume f_{QF} and down compare f_{QF} were removed.

  - These rules would require invalid rules such as $f_{QF} \leftarrow (\bigsqcap_y \psi) \equiv \bigsqcap_y (f_{QF} \leftarrow \psi)$. For example, with $f_{QF} = 1$ and $\psi = y + ?(y \leq 1)$, we have $\bigsqcap_y \psi = 1$, but $\psi(y) > 1$ for all y. Thus $1 \leftarrow (\bigsqcap_y \psi) = 0$, while $1 \leftarrow \psi(y) = \psi(y)$ for all y, so $\bigsqcap_y(1 \leftarrow \psi) = 1$.
  - In the proof of Theorem 4.17, membership in $\mathsf{HeyLo}^{\mathsf{PNF}}_{\sqcap}$ or $\mathsf{HeyLo}^{\mathsf{PNF}}_{\sqcup}$ is only used up to the equivalence of Theorem 4.14.

Typos.

- Section 2.2, Page 13: The calculation $\neg\neg 1 = \neg\infty = 0$ was corrected to $\neg\neg 1 = \neg 0 = \infty$.
- Section 2.2, Page 13: The negations used in the equivalent formulations of hard implications were swapped.
- Section 2.3, Page 28: Fixed a typo in the calculation.
- Section 4.2, Page 82: Fixed typos in the proof of Theorem 4.4: an infimum was printed as a supremum, and two bound variables were printed incorrectly.
- Section 4.2, Page 80: In Lemma 4.3, concrete endpoints ∞ and 0 were corrected to $\top$ and $\bot$.
- Section 4.2, Page 86: Fixed a typo in the co-negation rule of Theorem 4.8; it should be $\sim(\bigsqcap_x \varphi) \equiv \bigsqcup_x (\sim\varphi)$.
- Section 4.2, Page 88: Fixed the hard-implication else-case values in Lemma 4.10.
- Appendix A, Page 119: Fixed a typo.