Semantics and Loop Invariant Synthesis for Probabilistic Programs

by

Friedrich Gretz

A thesis submitted in fulfilment
of the requirements for the degree of
Doctor of Philosophy
in the
Department of Computing
Faculty of Science and Engineering
Macquarie University

Supervisor: Prof. Annabelle McIver

2014

Abstract

In this thesis we consider sequential probabilistic programs. Such programs are a means to model randomised algorithms in computer science. They facilitate the formal analysis of performance and correctness of algorithms or security aspects of protocols.

We develop an operational semantics for probabilistic programs and show it to be equivalent to the expectation transformer semantics due to McIver and Morgan. This connection between the two kinds of semantics provides a deeper understanding of the behaviour of probabilistic programs and is instrumental to transfer results between communities that use transition systems such as Markov decision processes to reason about probabilistic behaviour and communities that focus on deductive verification techniques based on expectation transformers.

As a next step, we add the concept of observations and extend both semantics to facilitate the calculation of expectations which are *conditioned* on the fact that no observation is violated during the program's execution. Our main contribution here is to explore issues that arise with non-terminating, non-deterministic or infeasible programs and provide semantics that are generally applicable. Additionally, we discuss several program transformation to facilitate the understanding of conditioning in probabilistic programming.

In the last part of the thesis we turn our attention to the automated verification of probabilistic programs. We are interested in automating inductive verification techniques. As usual the main obstacle in program analysis are loops which require either the calculation of fixed points or the generation of inductive invariants for their analysis. This task, which is already hard for standard, i.e. non-probabilistic, programs, becomes even more challenging as our reasoning becomes quantitative. We focus on a technique to generate quantitative loop invariants from user defined templates. This approach is implemented in a software tool called Prinsys and evaluated on several examples.

Statement

The research has been carried out under a Cotutelle agreement between the RWTH Aachen University and Macquarie University. To fulfil this agreement, this thesis entitled "Semantics and Loop Invariant Synthesis for Probabilistic Programs" will be submitted to RWTH Aachen University in a modified form to meet the requirements there.

I certify that the work in this thesis has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree to any other university or institution.

I also certify that the thesis is an original piece of research and it has been written by me. Any help and assistance that I have received in my research work and the preparation of the thesis itself have been appropriately acknowledged.

In addition, I certify that all information sources and literature used are indicated in the thesis.

Date: Signature:

Acknowledgements

Foremost I wish to thank my supervisors Annabelle McIver and Joost-Pieter Katoen. They have patiently guided me and helped with good advice whenever I needed it. I was granted the opportunity to pursue my research at two universities which not only gave a me a more diverse and richer research training but also has been an exciting experience. I feel that my time as a PhD candidate had a great impact on me in many ways and I am grateful for that.

I am indebted to Federico Olmedo, Benjamin Kaminski and Nils Jansen at RWTH for the fruitful collaboration, particularly on conditional expectations. Additionally, Tahiry Rabehaja at Macquarie has helped me a lot through discussions and valuable feedback on proofs.

Special thanks go to Souymodip Chakraborty for pointing out a result on 1-counter MDPs that led to a much better understanding of the behaviour of weakest liberal pre-expectations; Christian Dehnert who helped a lot to gain overview over model checking techniques; Damian Barsotti for patiently introducing me to his fixed point approximation method; and Aleksandar Chakarov who showed interest in our invariant generation techniques and readily helped to understand the differences to and connection with his inductive expectation invariants.

Finally, I thank all colleagues, friends and last but not least my parents who have supported me throughout my nearly four year long journey.

Publications

- Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. Operational versus weakest precondition semantics for the probabilistic guarded command language. In *Ninth International Conference on Quantitative Evaluation of Systems, QEST 2012, London, United Kingdom, September 17-20, 2012*, pages 168–177. IEEE Computer Society, 2012

- Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. Operational versus weakest pre-expectation semantics for the probabilistic guarded command language. *Perform. Eval.*, 73:110–132, 2014

- Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. Prinsys - on a quest for probabilistic loop invariants. In Kaustubh R. Joshi, Markus Siegle, Mariëlle Stoelinga, and Pedro R. D'Argenio, editors, *Quantitative Evaluation of Systems - 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*, volume 8054 of *Lecture Notes in Computer Science*, pages 193–208. Springer, 2013

- Friedrich Gretz, Nils Jansen, Benjamin Kaminski, Joost-Pieter Katoen, Annabelle McIver, and Federico Olmedo. Conditioning in probabilistic programming. *To appear in MFPS 2015.*

Author's Statement

For the following articles I was responsible for the conceptualisation of work, its realisation and its documentation.

- Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. Operational versus weakest precondition semantics for the probabilistic guarded command language. In *Ninth International Conference on Quantitative Evaluation of Systems, QEST 2012, London, United Kingdom, September 17-20, 2012*, pages 168–177. IEEE Computer Society, 2012

- Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. Operational versus weakest pre-expectation semantics for the probabilistic guarded command language. *Perform. Eval.*, 73:110–132, 2014

- Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. Prinsys - on a quest for probabilistic loop invariants. In Kaustubh R. Joshi, Markus Siegle, Mariëlle Stoelinga, and Pedro R. D'Argenio, editors, *Quantitative Evaluation of Systems - 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*, volume 8054 of *Lecture Notes in Computer Science*, pages 193–208. Springer, 2013

For the following article I contributed to the definition of semantics, worked out analysis examples and suggested the two program transformations, including the notion of iid loops. Additionally, I provided the correctness proof of the transformation that turns an observation into a loop and its reverse.

- Friedrich Gretz, Nils Jansen, Benjamin Kaminski, Joost-Pieter Katoen, Annabelle McIver, and Federico Olmedo. Conditioning in probabilistic programming. *To appear in MFPS 2015.*

Contents

1. Introduction **1**

1.1. Motivation – probabilistic systems 1

1.2. Expressiveness of probabilistic programs 3

1.3. Benefits and challenges of probabilistic programs 8

1.4. Research questions and our contributions 10

1.4.1. Linking operational and denotational semantics 10

1.4.2. Conditional probabilities and expectations 12

1.4.3. Automated analysis 12

2. Linking operational and denotational semantics **15**

2.1. The probabilistic Guarded Command Language 15

2.2. Operational semantics 17

2.2.1. Markov decision processes 17

2.3. Denotational semantics 23

2.3.1. Distribution based – forward 24

2.3.2. Expectation based – backward 24

2.4. Transfer theorem 27

3. Conditional probabilities and expectations **29**

3.1. Operational semantics for programs with conditioning 29

3.2. Expectation transformer semantics for programs with conditioning 33

3.2.1. Infeasible programs 35

3.2.2. Alternative definition 37

3.2.3. Expectation transformers and non-determinism 37

3.3. Reasoning with conditioning 40

3.3.1. Replacing observations by loops 40

3.3.2. Replacing loops by observations 43

3.3.3. Observation hoisting 46
3.3.4. iid loops and hoisting – a case study 49
3.3.5. Conditional expectations in loopy programs – the Crowds protocol . . 54

4. Automated analysis **57**
4.1. Proving properties of probabilistic programs 57
4.1.1. Computing fixed points 57
4.1.2. Invariants 61
4.2. Feasible level of automation 65
4.3. Prinsys 66
4.3.1. Methodology 67
4.3.2. Examples 68
4.3.3. Problems 70

5. Conclusion and future work **73**

Appendix **83**

A. Operational versus Weakest Precondition Semantics for the Probabilistic Guarded Command Language **85**

B. Operational versus weakest pre-expectation semantics for the probabilistic guarded command language **97**

C. Conditioning in Probabilistic Programming **137**

D. Prinsys – On a Quest for Probabilistic Loop Invariants **155**

1. Introduction

1.1. Motivation – probabilistic systems

This thesis is situated within the very broad topic of analysis of probabilistic systems. In general, the term *probabilistic systems* denotes any formal representation of a mechanism, process or algorithm, that evolves over time and whose behaviour depends on random events. The study of such probabilistic mechanisms provides key insights in a large number of fields such as chemistry [11], quantum physics [65] and economy [10], just to name a few. Our focus lies on probabilistic systems within the context of computer science. Because of their omnipresence and importance, there is a general interest to formalise instances of probabilistic systems using some formal description languages and thereby allow for the formal analysis of such systems. However there is always a trade-off between the expressiveness of a formal language and its aptitude for automated analysis. For instance, consider the well known Chomsky hierarchy of grammars [16]. There we know that e.g. unrestricted grammars are more expressive than context-free grammars. This means the set of languages that can be described is larger for unrestricted grammars. Meanwhile there are analysis questions – e.g. the word problem – that can be answered automatically for any context-free grammar but cannot be answered for an unrestricted grammar in general.

Returning to our discussion of probabilistic systems, there are several features that we may control: is the underlying state-space of a process allowed to be only finite or infinite? Can it be only discrete or continuous? Can our process be parameterised or do all values have to be numerical? Do we permit the parallel execution of several processes or only execute one process sequentially? Do we model continuous time? In the past, various approaches to formalise probabilistic systems have emerged and each approach facilitates a particular analysis technique. Here we mention just a few.

CHURCH[1] is a language where a user can describe generative models in a functional programming style. Generative models are used to model joint probability distributions over

[1] http://projects.csail.mit.edu/church/wiki/Church

observable data. In the field of computer science, prominent examples of generative models are Hidden Markov models or probabilistic context-free grammars. The CHURCH language comes with a programming environment WEBCHURCH[2] which can simulate any given program and display a resulting histogram. This is an example of a very expressive language with only limited analysis capabilities as the produced histograms vary for each run and while they may convey some intuition about the modelled process they do not serve as a rigorous proof of any particular property. There are many other probabilistic programming environments which are based on sampling techniques such as Gibbs sampling [48], Metropolis-Hastings [54] and other variants of Markov chain Monte Carlo sampling [38]. The probabilistic-programming.org website provides an extensive overview.

On the other side of the spectrum we find more restricted languages that were designed with some particular verification technique in mind. Thus any program in that language is amenable to automated verification. One of the most prominent examples is the PRISM language [44] which allows the specification of finite state Markov chains or Markov decision processes[3]. Given a description of such a Markovian process and a specification in probabilistic computation tree logic (PCTL), the model checker PRISM can automatically decide whether the model meets the specification. From the verification point of view this is a powerful tool, as it needs no user interaction in the verification process and allows the quantitative analysis of many interesting properties. However the systems that can be described in PRISM may appear limited in practice as it has to have a finite state space and transition probabilities between states have to be specified as numerical values rather than samples from some possibly parameterised distribution.

Our area of research can be positioned somewhere in between the two approaches described above. We use a language so expressive that systems specified in it cannot be exhaustively checked by a model checker in general, yet it is structured enough to allow for rigorous proofs about the behaviour of the system. The next section treats what we call probabilistic programs in greater detail and prepares us for the discussion of research questions that motivate this work.

[2] https://probmods.org/play-space.html

[3] not be confused with a programming language of the same name for statistical modelling, cf. http://sato-www.cs.titech.ac.jp/prism/

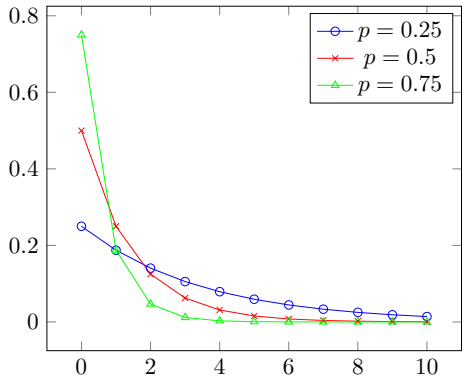
1.2. Expressiveness of probabilistic programs

In this thesis we study *probabilistic programs*, which are a special kind of probabilistic systems. Probabilistic programs are written in an imperative language – just like standard programs usually are – but the language is enriched with a statement that allows random samples to be drawn from some distribution. These programs are executed sequentially and as they proceed step by step, their outcome may depend on the samples drawn during the execution. One can think of various languages for describing probabilistic programs. In this thesis we choose to work with the *probabilistic Guarded Command Language* (pGCL), which is a probabilistic extension of Dijkstra's GCL [23], and was introduced by McIver and Morgan [50]. A program written in this language can draw a sample from a Bernoulli distribution and, depending on the outcome, executes one or the other branch of a choice statement. A common illustration of a Bernoulli experiment is a coin flip which has two outcomes "heads" or "tails". Of course, a Bernoulli experiment need not have equal probabilities for both outcomes and then in our illustration we speak of a *biased* coin flip. The language pGCL is simple enough so we do not have to care about complex data structures, objects or other implementation details in our arguments and at the same time it is expressive enough to succinctly capture programs which are interesting both, theoretically and practically.

In what follows we illustrate the possibilities of pGCL. Having only Bernoulli trials at our disposal might seem to be a severe limitation, but in fact this is sufficient to write subprograms that produce a sample from other important distributions. For example, the geometric distribution gives the probability of encountering the first success in a series of independent Bernoulli trials. Figure 1.1a shows a program whose outcomes are distributed according to the geometric distribution with parameter p. In Chapter 2 we will make precise what each statement in this program means. All we need for now is that we have a program that repeatedly can choose to increase a variable x with probability p or to stop with probability $1-p$. The set of possible values of x upon termination is the set of all natural numbers. And for each number k, the probability to terminate with $x = k$ is $(1-p)^k p$ which is precisely how the geometric distribution is defined. This is a simple example where a distribution is implicitly encoded by a probabilistic program. Similarly, it is possible to write pGCL programs that produce samples distributed according to a binomial distribution which gives the probability to have k successes within a series of independent Bernoulli trials of length n. With slight modifications one obtains programs for the hypergeometric distribution and

```
1  x := 0;
2  flip := 0;
3  while (flip = 0) {
4      ( flip := 1 [p] x := x + 1 );
5  }
```

(a) Program text

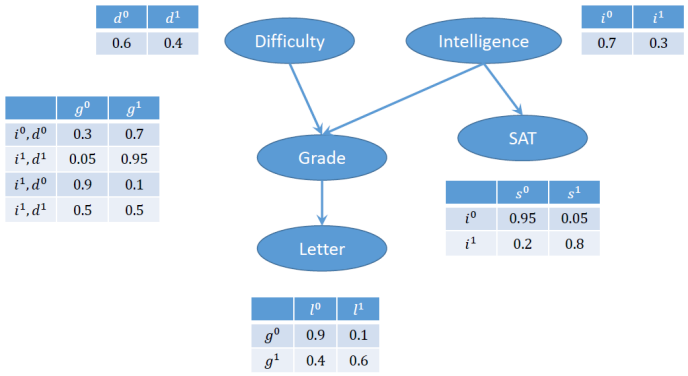


(b) Generated distributions over x for various values of p

Figure 1.1.: A probabilistic program implicitly models a distribution.

the negative binomial (Pascal) distribution. Finally, it is also known how to obtain a discrete uniform distribution using repeated fair coin flips [47]. This shows that in fact we have access to a variety of discrete distributions and are able to describe all systems that draw samples from these distributions.

Another interesting aspect of probabilistic programs is the possibility of conditioning the generated distribution using *observe* statements in the program. An *observe* statement is equipped with a boolean guard and behaves like a "filter" that selects runs that pass the guard. Thereby the distribution which is encoded by the program becomes conditioned on the fact that all observations have been passed during the program's execution. To motivate this let us briefly consider Bayesian Networks, which are frequently used in the area of artificial intelligence, to concisely represent probability distributions. They are graphs in which each node represents an event and arrows between nodes represent dependencies, e.g. $A \to B$ indicates that the probability of B being true is conditioned on the truth value of A. Finally each node is labelled with a table giving these conditional probabilities. Consider for example the Bayesian network in Figure 1.2a taken from [27]. It models the likelihood that a student will receive a recommendation letter based on his performance. Bayesian inference [20] allows to extract the probability of some (possibly conditioned) events from the network. Let us assume we have *observed* the "Grade" event in the Bayesian network, i.e. the student has passed his exams with good grades, and given this information, we need to *infer* the likelihood of the other events. Instead of working with the network we may translate



(a) Bayesian network [27] which implicitly represents the joint probability over the five events Difficulty, Intelligence, Grade, SAT and Letter. The marginal distributions of the first two are independent of any other events while the marginal distributions of the latter three are given as conditional distributions.

```
1  i := 1 [0.3] i := 0;
2  d := 1 [0.4] d := 0;
3  if (i = 0 and d = 0)
4          g := 1 [0.7] g := 0;
5  else if (i = 0 and d = 1)
6          g := 1 [0.95] g := 0;
7  else if (i = 1 and d = 0)
8          g := 1 [0.1] g := 0;
9  else
10         g := 1 [0.5] g := 0;
11 observe(g = 1);
12 if (i = 0)
13         s := 1 [0.05] s := 0;
14 else
15         s := 1 [0.8] s := 0;
16 if (g = 0)
17         l := 1 [0.1] l := 0;
18 else
19         l := 1 [0.6] l := 0;
```

(b) Program adapted from [27] that represents the above network and allows to infer the probability of all variables i, d, g, s, l conditioned on the fact that g has been set to 1.

Figure 1.2.: A probabilistic program that models a conditional probability distribution.

```
1  counter := 0;
2  while (x > 0) {
3      (x := x + 1 [p] x := x - 1);
4      counter := counter + 1;
5  }
```

Figure 1.3.: An unbounded one-dimensional random walk.

it into a probabilistic program in a straightforward way, cf. Figure 1.2b and analyse that. Here each variable takes values 0 or 1 to indicate whether the corresponding event in the Bayesian network has occurred. The observation is built into the program, cf. line 11, and admits only those runs that satisfy the predicate $g = 1$, according to our assumption. As we will see in Chapter 3, the conditional probability of any of the variables being 1 can easily be determined for this program. Querying a Bayesian network by analysing a probabilistic program is just one of the possible applications of observations. In Chapter 3 we will see various other use cases. What the example in Figure 1.2 nicely shows – and what is also emphasised in [27] – is that probabilistic programs encompass other modelling formalisms. This allows to transfer results between different communities such as formal methods and AI. For example, a successful program analysis technique thus becomes also an inference method for Bayesian networks. From a programmer's point of view, *observe* can be seen as the probabilistic extension of the *assert* statement known from most standard programming languages. We will explain in detail what *observe* means and how we can reason about conditional probabilities and expectations in Chapter 3.

So far we have considered programs that were merely representatives of some distributions. There was no notion of a process. An interesting process, which has applications in physics, chemistry or biology, is the random walk and its many variations [63]. The simplest form of a random walk is the unbounded symmetrical walk on a line. Its description in pGCL is shown in Figure 1.3. Variations include introducing bounds and adding more dimensions producing random walks on grids or cubes. Despite its short and intuitive program text the analysis of such a process is far from trivial and requires advanced mathematics, cf. [62, Ch. 2.4] and [25, Ch. 14].

In the context of computer science we are mostly interested in modelling randomised algorithms or protocols. As an example of these, consider Zeroconf [15], which is a randomised

```
1  configured := false;
2  while(!configured) {
3          //choose random IP
4          (collision := true [q] collision := false);
5          //assume an unused IP was chosen
6          configured := true;
7          //query the network N times
8          i := 0;
9          while(i<N){
10                 {
11                         if(collision){
12                                 configured := false;
13                         }
14                 }
15                 [1-p]
16                 {
17                         skip;
18                 }
19                 i := i + 1;
20         }
21 }
```

Figure 1.4.: The Zeroconf protocol

protocol that allows to configure IP addresses within a network. It has been modelled and analysed before by Bohnenkamp et al. [8]. We adapt their model and obtain the program in Fig. 1.4. This program models the process of a new host connecting to a network and finding an unused IP address. Of course, the program abstracts from all implementation details of the internet protocol. Instead we focus on the probability q of guessing an unused IP and the probability p to miss a response from a host that indicates a collision. Depending on these parameters we can answer questions like: "what is the probability that a new host chooses an address which is already in use and therefore a collision will occur in the network?" This example nicely shows how the probabilistic behaviour of an actual protocol can be modelled

```
1  {x := 1 [0.5] x := 2}
2  []
3  {x := 1 [0.75] x := 2}
```

(a) This program offers a non-deterministic choice between two distributions over program variable x.

```
1  x := 1 [0.65] x := 2
```

(b) This is a possible refinement of the program in Fig. 1.5a. Note that an implementation does not have to be equal to either branch of the non-deterministic program but may be formed by a convex, i.e. probabilistic, combination of the choices.

Figure 1.5.: A non-deterministic probabilistic program and its refinement.

within pGCL. In Chapter 3 we will explain its analysis.

Finally, pGCL inherits non-deterministic choice from GCL. The benefit of that is twofold: First, it is possible to underspecify choices when no probabilistic information is available or it can be used in conjunction with probabilistic choice to specify probability ranges. Second, non-determinism allows for a notion of refinement between programs. Figure 1.5 illustrates both points. The program on the left sets x to 1 with probability *at least* $1/2$ and *at most* $3/4$. Conversely x is set to 2 with some probability between $1/4$ and $1/2$. The program on the right resolves the non-deterministic choice by a probabilistic choice where it takes the first option with probability 0.4 and the second option with probability 0.6. In this way a program is obtained where the probability to set x to 1 is $0.4 \cdot 0.5 + 0.6 \cdot 0.75 = 0.65$ and correspondingly the probability to set x to 2 is $0.4 \cdot 0.5 + 0.6 \cdot 0.25 = 0.35$. We may refer to the program in Figure 1.5a as a specification (or abstraction) and to the program in Figure 1.5b as its implementation (or refinement). Any claim we can prove for the specification will also hold for its implementation. For instance, a claim could be: "x is *at least* 1.5 on average". A further discussion of abstraction and refinement between probabilistic programs is beyond the scope of this thesis and we refer to e.g. [50].

1.3. Benefits and challenges of probabilistic programs

A formal language achieves two things: One is that we are given a formal yet intuitive way to describe a process. This eliminates ambiguity that we otherwise would have to face when describing a process in natural language. To illustrate this issue, consider the famous debate

about the solution to the Monty hall problem which may be formulated as follows [1]:

> Suppose you are on a game show, and you are given the choice of three doors: Behind one door is a car; behind the others, goats. You pick a door, say No. 1, and the host, who knows what is behind the doors, opens another door, say No. 3, which has a goat. He then says to you, "Do you want to pick door No. 2?" Is it to your advantage to switch your choice?

Would you describe the game as a pGCL program, then all assumptions become explicit and it can be rigorously proven that switching doors is the best strategy. This in fact was done for example in [18]. Moreover within the broad area of computer science, there is a number of fields that make use of probabilistic algorithms such as machine learning [7], artificial intelligence [60], security [6] or randomised algorithms design [52]. In all of these disciplines we are already used to write down programs in a programming language so a formalism that adds probabilistic behaviour to a programming language supports the straightforward description of randomised algorithms as advocated by Gordon et al. [27]. With probabilistic programs a programmer can use the well established constructs of sequential composition, conditional branching and loops to specify randomised algorithms. This, in fact, we consider as the main use case of probabilistic programs. So before we attempt any analysis we need to define rigorously the meaning of programs and make sure that our intuition about the meaning of a program matches its formal semantics. This issue is further addressed in Sections 1.4.1 and 1.4.2.

The second benefit that we gain from formalising processes inside a language like pGCL is that we are able to prove or disprove properties of the modelled process. In our introduction, model checking has been mentioned as an approach which is able to calculate probabilities of particular events in a given model. Unfortunately, many of the pGCL programs are not amenable to model checking. In Fig. 1.1 and Fig. 1.3 we have seen two examples of systems with an infinite underlying state space. In the first example, for every value of the counter x, there is a positive probability that the program will terminate with this value. In the second example, the walk can take arbitrarily many steps to the right before eventually returning to zero. Therefore if we want an exact analysis without further assumptions or restrictions on the systems, we have to deal with infinite state spaces. Another very useful feature of probabilistic programs is that they may be parameterised. In the examples above we did not specify numerical probabilities, e.g. $1/2$, but instead used a parameter p that stands for any number between zero and one. This is a great benefit. For example, the program in

Fig. 1.1 represents *all* geometric distributions and any property that we can verify for that program will hold for all instances of geometric distributions. The ability to reason with parameters furthermore facilitates parameter synthesis; a task in system design where one seeks to optimise the parameters of a system to meet given performance criteria. Again, we must pay a price for this generality as it precludes any numerical analysis technique. Our analysis tools must support symbolic computations if we deal with parameters. There have been efforts to model check infinite state spaces [22, 43, 37], and progress has been made to tackle parametric systems in model checking [41, 36]. As of today, runtime and the size of the system remain limiting factors for the applicability of the proposed methods. On the other hand verification by means of deductive reasoning with invariants can be carried out regardless of the underlying state space of a program or parameters in the program text. Furthermore finding a loop invariant achieves more than just verifying that a particular state can or cannot be reached with some probability. An invariant summarises the behaviour of the loop in just one expression. This is analogous to invariants found e.g. in physics that describe the behaviour of dynamical systems or reaction equations in chemistry. An example of invariants in physics are Newton's laws of motion. While physical laws are universally applicable in everyday life, we have to find new invariants for each and every written loop.

We have described the importance of probabilistic programs and we have given a list of challenges that occur when analysing these programs. In the following sections we go into more detail about which particular problems we have identified in our research and how we contributed to their solution.

1.4. Research questions and our contributions

In the scope of this thesis we have identified three topics, which we have studied in detail.

1.4.1. Linking operational and denotational semantics

Any formal language comprises two elements: syntax and semantics. While the former is simply given by a set of rules that tell us how to write programs in that language, the latter needs more attention. Semantics tell us what a given program text actually means. There are different ways to explain the meaning of a program. Probably the most popular is in terms of some transition system where a program defines a set of states and transitions between them. We call this the *operational semantics* of a program. Another possibility is to think

of the meaning of a program as a (partial) function. For example, Dijkstra [23] gave the meaning of a GCL program P in terms of a function $wp(P, \cdot)$ which maps a postcondition to a precondition such that when P is executed from a state that satisfies the precondition it is guaranteed to terminate in a state that satisfies the given postcondition. The pre- and postconditions are expressed as predicates (in first-order logic) and therefore this function is called a *predicate transformer*. The particular function $wp(P, \cdot)$ gives the most general precondition, i.e. a precondition that is satisfied by the largest possible set of initial states, and is therefore called *weakest precondition*. Whenever a meaning of a program text P is given by a function like $wp(P, \cdot)$ above, we call this the *denotational semantics* of a program. An important sanity check is that no matter which semantics are used to describe the meaning of a given program, they should all agree on the outcome of the program, i.e. they should assign the same "meaning" to the given program. Although transition systems have been used to describe the meaning of a program at least since the 1960s [26] and Dijkstra [23] introduced predicate transformer semantics in the 1970s, it was not until nearly 20 years later that Lukkien [46] has shown that these semantics agree. At the beginning of our research we have found a similar gap between semantics for probabilistic systems. McIver and Morgan [50] have given a denotational semantics for pGCL. In analogy to Dijkstra's approach they describe a $wp(P, \cdot)$ function that they call an *expectation* transformer. This is because for probabilistic programs we evaluate a random variable on the final states instead of a postcondition. And we are asking for the expected value of that random variable instead of a precondition. However a large part of the probabilistic verification community has been working with models that are presented as transition systems. For example, model checking algorithms operate on systems given as (among others) discrete time Markov chains (DTMCs) or Markov decision processes (MDPs). A straightforward question that comes to mind is: can pGCL programs be given an operational semantics in terms of MDPs and if so, what property of this MDP is captured by $wp(P, \cdot)$? We have addressed this question and in this thesis we give an operational semantics of pGCL using parametric MDPs with rewards (RMDPs). Subsequently we establish a link between McIver and Morgan's expectation transformer and the so called expected reward on the RMDP. This correspondence not only provides a good insight in how those two semantics are related but is also a nice tool because it allows to prove claims about pGCL programs using either semantics and then to transfer the result onto the other. This is why we refer to this theorem as the *transfer theorem*. For example, it is applied in the proof of Theorem 3 in Chapter 3. The transfer theorem is explained in Chapter 2. Our original work appeared in a

journal article [34] and in conference proceedings [32]. Both publication can be found in the appendix of this thesis.

1.4.2. Conditional probabilities and expectations

In probability theory, it is common to condition the probability of an event or the expectation of a random variable on the occurrence of some other event. In this way one obtains conditional probability distributions and conditional expectations. An application of conditional probabilities can, for instance, be found in medicine where one tries to estimate the likelihood of a particular disease after having observed some symptoms. In a similar way we may e.g. ask for the expected outcome of a probabilistic program *given* the fact that it has visited particular states during its execution. To allow for such specifications we follow Claret et al. [17] and add the *observe* keyword to the pGCL language. There, and in related work, e.g. [39, 54], they are concerned with purely probabilistic programs for which they try to find the probability of some outcome using simulation or symbolic program execution. All their programs are assumed to be terminating almost surely. Semantics are specified with these applications in mind and some questions are left open: How do we specify the semantics of a loop in general? What happens when we have non-terminating constructs? Can we retain non-determinism when reasoning about conditional measures? Can their semantics be phrased in terms of *wp* or a generalisation thereof? In our work we made an effort to answer all of these questions. We provide both denotational and operational semantics for pGCL with *observe* without making any assumptions about termination. In fact we discuss alternatives where non-termination can be considered favourable or unfavourable when conditioning. We then provide case studies that show how we can reason about those conditional measures over pGCL programs. The details are outlined in Chapter 3, which is based on our paper [30]. It can be found as Appendix C in this thesis.

1.4.3. Automated analysis

From the perspective of a computer scientist there is a large discrepancy between having a mathematical framework within which one can verify claims about a program and verifying those claims automatically. Just to name one example, it is common knowledge that the *halting problem* is undecidable, which means that there is no general and effective method that would correctly decide for every given program description whether it eventually terminates or not. Still there is no reason why a human could not (in principle) decide the halting

problem for each program presented to him – he "just" needs to come up with an original idea for every problem instance at hand. The formal semantics of pGCL consitutes a theory that can be mechanised [40, 12, 18]. This allows to use theorem provers like HOL [28] or Isabelle [53] to write down computer checkable proofs. However this mechanisation does not mean that proofs are carried out automatically. Rather an expert has to write the crucial parts of the proof and the theorem prover merely checks that these proofs are correct. A question that comes to mind is to what degree this process can be automated. Obviously a "push-button-technique" is not to be expected since pGCL is an extension of a Turing complete language. Therefore we focus on the problem on how to assist a human who tries to prove some property of a program. Verification of standard programs like GCL relies on loop invariants. McIver and Morgan [50] have generalised the idea of invariants to probabilistic programs and established some proof rules for total correctness of probabilistic programs. Later, Katoen et al. [42] suggested that candidate expressions can be checked for invariance automatically. Subsequently we have revised and implemented their method for invariant generation. Chapter 4 evaluates the tool on several case studies. It is based on our work [33], which can also be found in the appendix of this thesis.

In the next three chapters we discuss each of these contributions and present the results. For further details, e.g. proofs we refer the reader to the papers in the appendix.

2. Linking operational and denotational semantics

In Section 1.2 we have given some examples of pGCL programs and discussed their meaning intuitively. Now we formalise the syntax and semantics of pGCL. This leads to our first result on the equivalence of operational and denotational semantics.

2.1. The probabilistic Guarded Command Language

As is common for imperative languages, a program in pGCL will consist of a list of commands whose execution can modify or depend on the values of program variables. The only data type that we use for program variables are real numbers. The careful reader will object that it is impossible to use real valued numbers on a computer because in general a real number cannot be finitely represented. Since in the following we are not concerned with implementation details on actual hardware but rather focus on the mathematical properties of the language's semantics we follow the established terminology and speak about real numbers. All definitions and results used or given in this thesis can be rephrased using rational numbers instead.

The language, as introduced in [50], has eight commands. We start with the primitive commands. There is the *no-operation* command

$$\mathit{skip}$$

which has no effect and is simply used as a placeholder to explicitly indicate that nothing has to be done. Contrary to that the *improper termination* command

$$\mathit{abort}$$

is used to indicate that the program has reached a point from which nothing definite can be said about its behaviour. It might not terminate at all or it might stop in some arbitrary

state which we know nothing about. Finally there is the *assignment* command that assigns the result of some arithmetic expression to a variable:

$$x \coloneqq E \ .$$

The arithmetic expression E is built using the usual operations (addition, multiplication, subtraction and division) between program variables. The remaining commands are defined inductively. For this we assume to have some pGCL programs P and Q. The *conditional choice* allows to decide between two alternative subprograms based on the current truth value of a boolean predicate:

$$\mathit{if}(G)\,\{P\}\,\mathit{else}\,\{Q\} \ .$$

The predicate G is called the *guard* and is built using the boolean operators (conjunction, disjunction and negation) between predicates over program variables. Note that at the beginning of this section we defined all program variables to be real valued, however for better readability in some examples, we may assign boolean values to a variable or use a variable as a predicate whenever we know that it only takes boolean values. In such cases we use the convention $\mathit{true} = 1$ and $\mathit{false} = 0$. *Probabilistic choice* allows to choose between two alternative subprograms probabilistically:

$$\{P\}\ [a]\ \{Q\} \ .$$

Here, with probability a the left hand side program P is chosen to be executed next and with the remaining probability $1 - a$ the subprogram Q is chosen. The probability a may be explicitly given as a number in the interval $[0, 1]$ or it can remain as a symbol which denotes some *unknown but fixed* probability in that interval. This is the language construct that gives us access to Bernoulli trials as discussed in the introduction. When P and Q are primitive commands we may drop the curly braces for better readability. This language construct is in fact the only difference to Dijkstra's GCL [23]. As stated in the introduction a particular feature of pGCL is that it retains non-determinism. For the *non-deterministic choice* between subprogram P and Q we write:

$$\{P\}\ []\ \{Q\} \ .$$

Again, we may drop the curly braces for readability when both subprograms consist of a primitive instruction only. Like every other Turing complete language, pGCL has to have some repetition construct. At this point we could first introduce recursion in general and

then consider loops as special cases of that. However since we do not need recursion in the rest of this thesis we base our presentation on *while loops* directly. We write

$$\mathit{while}\,(G)\,\{P\}$$

to specify that the program P is repeated until the guard G becomes *false*. Finally a pGCL program is a sequence of one or more subprograms which is written as:

$$P; Q \ .$$

Here first P is executed and upon its termination Q is executed. Again for the sake of readability some simplification of notation applies: we usually use the semicolon only to either terminate or concatenate primitive commands, but we omit it between other constructs. For example, we may write

$$\mathit{if}(x > 1)\,\{x := x - 1;\}\,x := x + y;$$

instead of

$$\mathit{if}(x > 1)\,\{x := x - 1\};\ x := x + y \ ,$$

i.e. we use the semicolon to terminate the assignment statements but leave it out after the closing brace of the if-statement. This corresponds to the syntax used in everyday imperative programming languages like Java or C. This concludes the presentation of pGCL's syntax. In Chapter 3, we will introduce one more command called *observe*, which however is not relevant for us at this stage.

2.2. Operational semantics

We now formalise the meaning of the commands introduced above using an operational semantics. The purpose is to explain the meaning of a program by describing how its execution proceeds in a stepwise fashion. Using *structured operational semantics* (SOS) rules [55] we translate program text to a (possibly infinite) Markov decision process (MDP).

2.2.1. Markov decision processes

In this section we give a brief account on MDPs that suffices for the understanding of the following results. For our formal treatment of all subtleties see pages 5-9 of "Operational versus Weakest Pre-expectation Semantics for the Probabilistic Guarded Command Language"

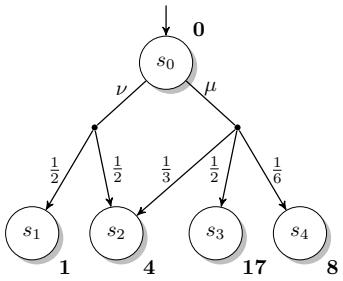


Figure 2.1.: An example MDP

in the appendix. For an introduction of MDPs in the context of software verification see [3, Ch. 10.6], for a more general treatment we refer to [56].

Discrete time Markov decision processes can be viewed as transition systems. Each state may have a finite number of enabled actions, each providing a probability distribution over successor states. The system is executed by choosing one such action non-deterministically and moving to a successor state probabilistically. To facilitate understanding, let us consider the example in Figure 2.1. There are two actions enabled in the state s_0 labelled with ν and μ respectively. We use these labels synonymously to refer to action names as well as to the distributions that these actions offer. For example, upon selecting distribution ν in state s_0 the system will move either to s_1 or to s_2 with $1/2$ probability each. The MDP itself does not provide any information on how non-deterministic choices should be resolved. These choices are made by an external entity called the *scheduler*[1]. A scheduler is a function that maps a sequence of states to actions. The sequence is called the history that leads up to the current state. A special case are schedulers that are memoryless[2], i.e. map individual states to actions.

An MDP can be extended in two ways. Firstly, we allow probabilistic transitions to be parameterised. This means that instead of giving numeric probabilities we can use parameters, which represent some number in $[0, 1]$. For example, we could define distribution ν such that it assigns probability p to s_1 and probability $1 - p$ to s_2. Another extension is the annotation of states with so called *rewards*. A reward function assigns values to states. In our example the reward of s_0 is 0. But upon reaching the state s_3 we earn reward 17. In the presence of probabilistic transitions it makes sense to speak of the *expected reward* which can be earned

[1] Also called environment, policy, adversary or strategy, depending on the context within which MDPs are studied.

[2] Also called positional.

starting in a given state. Let us examine our example MDP once again. Intuitively we see that distribution ν determines an expected reward calculated by $1/2 \cdot 1 + 1/2 \cdot 4 = 5/2$. Analogously we compute the expected reward under μ, which amounts to $1/3 \cdot 4 + 1/2 \cdot 17 + 1/6 \cdot 8 = 67/6$. But which one is the expected reward starting in s_0? In the presence of non-determinism it does not make sense to ask for *the* expected reward because depending on the scheduler's choice it will either be $5/2$ or $67/6$ and we have no information whatsoever to decide which one it will be. In order to solve this issue we consider high rewards desirable whereas we assume that a scheduler behaves adversely, i.e. tries to minimise the expected reward. Because of this adversarial behaviour we call the scheduler a *demon* and its choices *demonic*. In this way we can ask for the expected reward which is *guaranteed* regardless of how non-determinism is resolved. In our example, a demon would choose action ν to minimise the expected reward. Therefore we can say, that from state s_0 we can guarantee an expected reward of at least $2/5$.

This informal description of an MDP is reminiscent of the behaviour of pGCL and therefore provides a suitable setting for an operation semantics. A state of the MDP may represent the current valuation of the program variables and the position within the program text while the transitions allow for a direct encoding of probabilistic and non-deterministic choices. Formally, we give the operational semantics of a program P by the MDP $\mathcal{M}[\![P]\!] = (S, S_0, \to)$ where

- S is the set of pairs $\langle Q, \eta \rangle$ with Q a pGCL-program or $Q = \mathit{exit}$, and η is a variable valuation of the variables occurring in P,
- $S_0 = \{ \langle P, \eta \rangle \}$ where η is arbitrary, and
- $\to$ is the smallest relation that is induced by the following inference rules[3]:

  In a state where we have to execute the *skip* command there is only one transition which leads to an exit state without modifying the valuation

  $$\langle \mathit{skip}, \eta \rangle \to \langle \mathit{exit}, \eta \rangle \ .$$

  An *abort* state behaves like a trap to prevent the execution from reaching any proper exit states

  $$\langle \mathit{abort}, \eta \rangle \to \langle \mathit{abort}, \eta \rangle \ .$$

[3]For the sake of readability we simplify our transitions. Whenever there is only one action we drop its name from the transition and write the probabilities only. Whenever all enabled distributions are point distributions we drop the probabilities from the transition and thus whenever a step is taken deterministically, i.e. there is only one enabled action and one successor with probability 1, we simply write $s \to t$.

From a state where an assignment has to be performed we take a step to the exit state and update the valuation according to the assignment

$$\langle x \coloneqq E, \eta \rangle \to \langle \mathit{exit}, \eta[x \coloneqq [\![E]\!]_\eta] \rangle \ .$$

A conditional choice offers to choose between two branches. In the state where this choice needs to be made the guard is evaluated and thus a *single* successor is determined. Hence we have two inference rules. In case the current variable valuation satisfies the guard the first branch is taken

$$\frac{\eta \models G}{\langle \mathit{if}\,(G)\,\{P\}\,\mathit{else}\,\{Q\}, \eta \rangle \to \langle P, \eta \rangle}$$

otherwise the second branch is taken

$$\frac{\eta \not\models G}{\langle \mathit{if}\,(G)\,\{P\}\,\mathit{else}\,\{Q\}, \eta \rangle \to \langle Q, \eta \rangle} \ .$$

In a state where a while loop begins we evaluate the guard and similarly to the conditional choice decide between two alternatives: either the loop is executed or skipped. In the first case we have

$$\frac{\eta \models G}{\langle \mathit{while}\,(G)\,\{P\}, \eta \rangle \to \langle P; \mathit{while}\,(G)\,\{P\}, \eta \rangle}$$

and in the second

$$\frac{\eta \not\models G}{\langle \mathit{while}\,(G)\,\{P\}, \eta \rangle \to \langle \mathit{exit}, \eta \rangle} \ .$$

Now we consider the states that may have two successors. In a state where a probabilistic choice is made two transitions with the respective probabilities emanate:

$$\langle \{P\}\ [a]\ \{Q\}, \eta \rangle \xrightarrow{a} \langle P, \eta \rangle \qquad \langle \{P\}\ [a]\ \{Q\}, \eta \rangle \xrightarrow{1-a} \langle Q, \eta \rangle \ .$$

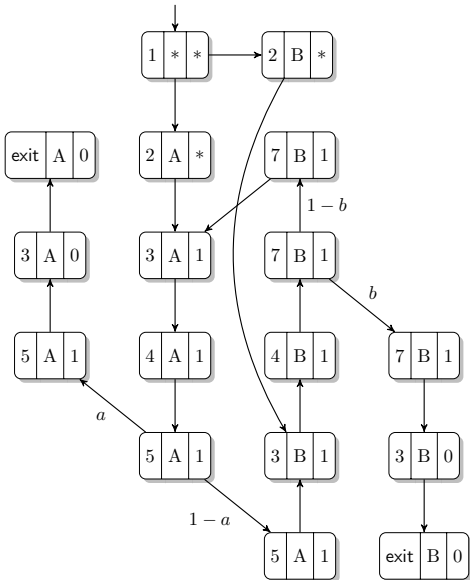
Analogously, for non-deterministic choice there are two successors as well. However they are annotated with actions instead of probabilities and it is up to the scheduler to determine which action will be selected during the execution:

$$\langle \{P\}\ []\ \{Q\}, \eta \rangle \xrightarrow{\nu} \langle P, \eta \rangle \qquad \langle \{P\}\ []\ \{Q\}, \eta \rangle \xrightarrow{\mu} \langle Q, \eta \rangle \ .$$

In order to define sequential composition we assume that from a state $\langle P, \eta \rangle$ a step to some distribution μ can be taken. Of course, there may be several possible successor distributions due to non-determinism. What we need to ensure now is that from the

```
1  (t := A [] t := B);
2  c := 1;
3  while (c = 1) {
4    if (t = A) {
5      (c := 0 [a] t := B);
6    } else {
7      (c := 0 [b] t := A);
8    }
9  }
```

(a) Program text



(b) Corresponding MDP where a state $\langle P, \eta \rangle$ is identified by the program line in which P starts and the individual values of t and c that are stored in η.

Figure 2.2.: Duelling cowboys program

state $\langle P; Q, \eta \rangle$ where the composition has to be executed the stepwise behaviour is the same until P terminates and only Q needs to be executed. This is expressed by the following rule

$$\frac{\langle P, \eta \rangle \to \mu}{\langle P; Q, \eta \rangle \to \nu} \text{ with } \nu(\langle P'; Q, \eta' \rangle) = \mu(\langle P', \eta' \rangle) \text{ where } exit; Q = Q.$$

The rule above states that if in a state $\langle P, \eta \rangle$ a distribution μ over successor states $\langle P', \eta' \rangle$ may be selected, then in a state modelling the sequential composition a distribution with the same probabilities over successor states exists. In this way it is ensured that first the program P is executed and when it terminates, i.e. the execution reaches a state of the form $\langle exit; Q, \eta'' \rangle$, the MDP proceeds with the execution of Q starting with valuation η''.

Example 1 (Operational semantics of programs)**.** We illustrate the program semantics using a simple program which has a finite underlying state space. Figure 2.2a shows the program

text. It models a duel between two cowboys A and B. We use the variable t to keep track of who's *turn* it is and we use c to keep track whether the duel *continues* after a shot has been fired or upon success of either contestant the program stops. Initially, one of the cowboys starts and then they alternate their shots. Each cowboy has a probability to hit his opponent which is given by a and b respectively. Figure 2.2b shows the corresponding MDP semantics. Formally, there are uncountably many possible initial states and we would need to fix a particular state to draw this system. However since the first two lines of the program basically lead us to only two possible states we make a simplification in our presentation and collapse all initial states into one. From there we apply the SOS rules given above and obtain the MDP. Each state contains a program line, i.e. the remaining program text and the valuation of the two variables t and c. This MDP induces two distributions over the outcomes $\langle \mathit{exit}, A, 0\rangle$ and $\langle \mathit{exit}, B, 0\rangle$ depending on the resolution of the choice in the initial state.

We might now, for example, ask for the least guaranteed probability that cowboy A wins the duel. For this we introduce a reward function that indicates if cowboy A has won. The reward function evaluates to zero for each state except $\langle \mathit{exit}, A, 0\rangle$ where it is one. We can then compute the expected rewards for all resolutions of non-determinism. Assume we take the transition $\langle 1, *, *\rangle \to \langle 2, A, *\rangle$ from the initial state then the expected reward is given by the sum of all terminating runs multiplied by the reward achieved. In this case this amounts to

$$\sum_{i=0}^{\infty}((1-a)(1-b))^i a = \frac{a}{a+b-ab} \ . \tag{2.1}$$

Analogously, if we assume that the initial choice is resolved by taking the step $\langle 1, *, *\rangle \to \langle 2, B, *\rangle$, then the calculation gives us

$$\sum_{i=0}^{\infty}((1-a)(1-b))^i a(1-b) = \frac{a(1-b)}{a+b-ab} \ . \tag{2.2}$$

We see that (2.2) is less than (2.1) so overall the minimal expected reward is

$$\frac{a(1-b)}{a+b-ab} \ .$$

What we learn from this is that no matter how the non-determinism in program 2.2a is resolved, we can guarantee that cowboy A will win the duel with probability at least $a(1-b)/a+b-ab$. □

In this section we have explained the meaning of pGCL programs by MDPs and we have shown how a random variable with respect to the program translates to a reward function over that MDP. Consequently we can measure expectations by determining the minimal expected reward of the RMDP which we generally define as:

Definition 1 (Minimal expected reward)**.** Let $(\mathcal{M}, r)$ be an RMDP with state space S, $T \subseteq S$ and $s \in S$. Further let $\mathfrak{C}$ denote the set of all cumulative reachability reward values that can be accumulated by paths from s to T in $(\mathcal{M}, r)$. The *minimal expected reward* until reaching T from s, denoted $\mathit{ExpRew}^{(\mathcal{M},r)}(s \models \Diamond T)$, is defined by:

$$\inf_{\mathfrak{S}} \sum_{c \in \mathfrak{C}} c \cdot \Pr^{\mathfrak{S}}\{\, \pi \in \mathit{Paths}^{\mathfrak{S}}(s, \Diamond T) \mid r_T(\pi) = c \,\} \ .$$

□

Alternatively, we may obtain the expected reward by a summation over all paths where for each summand we multiply the accumulated reward of that path with the probability of that path:

$$\mathit{ExpRew}^{(\mathcal{M},r)}(s \models \Diamond T) = \inf_{\mathfrak{S}} \sum_{\widehat{\pi} \in \mathit{Paths}^{\mathfrak{S}}_{\min}(s, \Diamond T)} \mathbf{P}^{\mathfrak{S}}(\widehat{\pi}) \cdot r_T(\widehat{\pi}) \ .$$

With Example 1 in mind this definition should be self-explanatory. For further details, we once again refer to our work [34].

2.3. Denotational semantics

In the previous section we have seen how a program translates to a probabilistic and nondeterministic transition system, an MDP. There we have seen that after resolving nondeterminism in some way this system defines a probability distribution on the outcomes of the program and we can ask for a minimal expected reward. We may say that the program implicitly describes a random experiment. The reward function that assigns values to final states of the MDP is nothing but a random variable which maps the program's outcomes to real values. Then what we determined with the minimal expected reward is just the expectation of this random variable that is distributed according to the given program. This terminology allows us to introduce the denotational semantics more intuitively and already hints at the result to be established in last section of this chapter.

2.3.1. Distribution based – forward

Instead of describing the meaning of the program by giving all possible state transitions as we did before, we may instead assume that we have some description of a probability distribution at hand and what we ask for are rules that tell us how each command of the language *transforms* this distribution. In fact, since the language features non-determinism the result of a transformation may be a set of distributions and subsequent transformations have to be applied to each of them. The description of the transformations to be applied to a distribution can be given for each of the eight commands individually in the same style as the previously described operational semantics. Assuming we would run through this exercise once more, we are able to define the meaning of a program as a function that maps a given initial distribution to a set of outcome distributions. It is possible to define some random variable with respect to these outcome distributions. What we can do then is *measure* the probability of events or any moment of that random variable with respect to each output distribution. Here we focus on determining the expectation. This is because events can be described by indicator random variables so that their expectation equals the probability of the described event. Furthermore determining variance and higher moments of a distribution goes beyond our research topic. So what we are left with in the end is a set of expectations. As before we can choose the least of these expectations to make claims about a lower bound on the expected value of our random variable.

2.3.2. Expectation based – backward

A crucial problem with the approach that keeps track of possible distributions is that we need to keep track of unboundedly many of them. We cannot throw away any of those until we know what the least expectation of a given random variable will be. On top of that, each distribution may have a support that is too large to fit in computer memory or even infinite. Therefore an alternative approach is to go backwards. Starting with an expression that describes how a random variable is evaluated over the final states of a program, we can proceed backwards through the program and arrive at an expression which is evaluated over the initial states, and that happens to be the minimal expectation of that given random variable. Any non-deterministic choice that is encountered on the way may be resolved immediately since we already know what is the function that we are minimising. To make this precise let us reconsider our language constructs one by one and explain how each of them determines an expectation of a given random variable. In the following let f be a random

variable that maps variable valuations to (non-negative) real values. We use $wp(P, f)$ to denote the minimal expectation of random variable f with respect to pGCL program P. The reason for this notation will become clear in a moment. The *skip* command does not alter the current distribution and conversely the expected value of f is whatever value f evaluates to in the initial state:

$$wp(skip, f) = f \ .$$

The *abort* command does not produce any distribution and hence the expectation of any random variable in any initial state is the least possible value which by definition is zero:

$$wp(abort, f) = 0 \ .$$

An assignment $x \coloneqq E$ will transform the random variable by substituting every occurrence of x in f by its new value E:

$$wp(x \coloneqq E, f) = f[x/E] \ .$$

Assuming we know how some subprograms P and Q produce the minimal expectation, we can give the rules for the inductively defined commands. Conditional choice between P and Q behaves as either one of them depending on the guard G, hence:

$$wp(if(G)\,\{P\}\,else\,\{Q\}, f) = [G] \cdot wp(P, f) + [\neg G] \cdot wp(Q, f) \ .$$

We use $[\cdot]$ to cast a boolean value to a real value assuming $[true] = 1$ and $[false] = 0$. In this way the whole expression remains a mapping from variable valuations to real values. Probabilistic choice is probably the most interesting one. It takes the weighted average between the expectation given by P and Q:

$$wp(\{P\}\ [a]\ \{Q\}, f) = a \cdot wp(P, f) + (1 - a) \cdot wp(Q, f) \ .$$

This of course agrees with our intuition from probability theory that for random variables Z, X and Y where $Z = a \cdot X + (1 - a) \cdot Y$ it holds that

$$\mathbb{E}(Z) = \mathbb{E}(a \cdot X + (1 - a) \cdot Y) = a \cdot \mathbb{E}(X) + (1 - a) \cdot \mathbb{E}(Y) \ .$$

As explained before we obtain the *minimal* expectation because non-deterministic choices are resolved demonically:

$$wp(\{P\}\ []\ \{Q\}, f) = \min\{wp(P, f), wp(Q, f)\} \ .$$

Thus $wp(\{P\}\,[]\,\{Q\}, f)$ is a function that agrees with the point-wise minimum between the expectations with respect to P and Q. As mentioned above this allows us to resolve choices directly and essentially turn them into deterministic choices with respect to the given random variable f. Sequential composition of two programs $P; Q$ will first determine the expectation of random variable f with respect to Q. The result will be regarded as a random variable again and its expectation with respect to P determines the expectation of f with respect to $P; Q$:

$$wp(P; Q, f) = wp(P, wp(Q, f)) \ .$$

The denotation of loops is defined using fixed point semantics:

$$wp(\mathit{while}\,(G)\,\{P\}, f) = \operatorname*{lfp}_x([G] \cdot wp(P, x) + [\neg G] \cdot f) \ .$$

In the terminology of McIver and Morgan [50], any expression that maps valuations of program variables to real values is called an *expectation*. In particular, our random variable f is called a *post-expectation* and what we have so far described as the minimal expectation $wp(P, f)$ is called *pre-expectation*. The motivation for this is that random variables may be regarded as expectation functions with a dirac distribution. The terms pre- and post-expectation are motivated by the fact that post-expectations are evaluated *after* the programs execution and pre-expectations are evaluated *before* the execution on the initial states. These terms also resemble the well established notions of pre- and postconditions known from Hoare logic. Finally, the mapping $wp(P, \cdot)$ requires an expectation and returns an expectation that is transformed according to the rules given before. Hence we call $wp(P, \cdot)$ an *expectation transformer* and we refer to denotational semantics which are defined using such transformations as *expectation transformer semantics* or *wp* semantics.

Example 2 (Evaluating *wp* semantics)**.** Using the rules above we may consider the program from Figure 2.2a once again and re-calculate the probability that cowboy A wins the duel which is given by

$$wp(\mathit{prog}, [t = A]) \ .$$

We start at the end of the program and work our way up as the rule for sequential composition suggests. Intermediate results of our calculation are given in between the program lines.

```
1  ⟨(1−b)a / a+b−ab⟩
2  ⟨min{ a / a+b−ab , (1−b)a / a+b−ab }⟩
3  (t := A [] t := B);
```

4 $\langle [t = A] \cdot \frac{a}{a+b-ab} + [t = B] \cdot \frac{(1-b)a}{a+b-ab} \rangle$
5 `c := 1;`
6 $\langle [t = A \wedge c = 0] \cdot 1 + [t = A \wedge c = 1] \cdot \frac{a}{a+b-ab} \; + [t = B \wedge c = 1] \cdot \frac{(1-b)a}{a+b-ab} \rangle$
7 `while (c = 1) {`
8 $\quad \langle [t = A \wedge c = 1] \cdot \frac{a}{a+b-ab} + [t = B \wedge c = 1] \cdot \frac{(1-b)a}{a+b-ab} \rangle$
9 $\quad \langle [t = A \wedge c \neq 1] \cdot a + [t = A \wedge c = 1] \cdot \frac{a}{a+b-ab}$
$\qquad +[t = B \wedge c = 0] \cdot (1-b) + [t = B \wedge c = 1] \cdot \frac{(1-b)a}{a+b-ab} \rangle$
10 `  if (t = A) {`
11 `    (c := 0 [a] t := B);`
12 `  } else {`
13 `    (c := 0 [b] t := A);`
14 `  }`
15 $\quad \langle [t = A \wedge c = 0] \cdot 1 + [t = A \wedge c = 1] \cdot \frac{a}{a+b-ab} \; + [t = B \wedge c = 1] \cdot \frac{(1-b)a}{a+b-ab} \rangle$
16 `}`
17 $\langle [t = A] \rangle$

We do not give the calculation of least fixed point that is determined in line 6. In Section 4.1.1 we return to this issue and show the calculations for a similar example. □

Not surprisingly, the expression in line 1 matches the probability we have found in Example 1. This observation is formalised in the next section and constitutes our first contribution.

2.4. Transfer theorem

As we have seen before the MDP semantics describes the possible executions of a program. However in order to reason about some expected reward, a reward function needs to be introduced in the first place. Similarly, the *wp* semantics requires a post-expectation to be given in order to tell what the pre-expectation will be. While the latter formalism takes the post-expectation as an argument, we need to incorporate it into the MDP as a reward function. This is straightforward as we have already suggested that a random variable is evaluated on the outcomes. Hence the reward function assigns non-zero values only to *exit* states by applying the given function to the variable valuation η of such a terminal state. Formally, for a given program P and random variable f, the reward-MDP $R_f[\![P]\!] = (\mathcal{M}, r)$ consists of an MDP $\mathcal{M}$ which is constructed according to the inference rules given in section 2.2.1 and the reward function r defined as $r(s) = f(\eta)$ if $s = \langle \mathit{exit}, \eta \rangle$ and $r(s) = 0$ otherwise.

Our terminology already indicated the similarity between the two presented semantics and the following theorem[4] makes this precise.

Theorem 1 (Transfer theorem)**.** For pGCL-program P, variable valuation η, and post-expectation f:

$$wp(P, f)(\eta) \;=\; \mathit{ExpRew}^{\mathcal{R}_f[\![P]\!]}(\langle P, \eta\rangle \models \Diamond P^{\surd}) \;,$$

where $P^{\surd}$ denotes the set of exit states of program P, i.e. states of the form $\langle \mathit{exit}, \eta'\rangle$. □

Theorem 1 says that for any initial variable valuation, any program and any given random variable the minimal expected reward in the RMDP corresponds to the weakest pre-expectation given by *wp*. The theorem asserts that the two semantics agree and serves as a sanity check. For the first time we provide a state based view on the meaning of pGCL programs. A question like "here is a DTMC, so what is the *wp* on that?" can now be answered precisely. Further more this correspondence theorem allows to carry over results that are known in the pGCL community to the community that studies MDPs and vice versa. In the following chapter we will see a result that has been proven using the operational view. It is due to Theorem 1 that we can carry out the proof using either semantics and transfer it then onto the other. This motivates the theorem's name.

[4] Theorem 1 appears as Theorem 23 in Appendix A and Appendix B where it is referred to as "correspondence theorem". Additionally, these papers contain a correspondence theorem for the notions of *wlp* and *LExpRew*.

3. Conditional probabilities and expectations

In the introduction, conditioning was identified as one of the important features of probabilistic programming. In this chapter we discuss both operational and denotational semantics of programs with conditioning. Our aim is to find semantics that coincide with our intuition, are as general as possible, e.g. do not require the programs to terminate almost surely, and generalise our previous definitions and the transfer theorem. We conclude the chapter by exploring various program transformations and case studies to showcase the analysis of conditional expectations in probabilistic programming.

3.1. Operational semantics for programs with conditioning

From probability theory we know that the conditional probability of an event A given an event B is determined by their joint probability which is scaled by the probability of B

$$\mathbf{P}(A|B) = \frac{\mathbf{P}(A \cap B)}{\mathbf{P}(B)} \ .$$

This can be phrased analogously with indicator random variables. Let $\mathbb{1}_A$ be the random variable which maps all outcomes in the event A to one and all outcomes outside A to zero and let $\mathbb{1}_B$ be defined analogously, then

$$\mathbf{P}(\mathbb{1}_A = 1|\mathbb{1}_B = 1) = \frac{\mathbf{P}(\mathbb{1}_A = 1, \mathbb{1}_B = 1)}{\mathbf{P}(\mathbb{1}_B = 1)} \ .$$

One can also generalise expectations to conditional expectations. The expected value of a discrete random variable X given an event B is defined as

$$\mathbb{E}(X|\mathbb{1}_B = 1) = \sum_{x \in range(X)} x \cdot \mathbf{P}(X = x|\mathbb{1}_B = 1) = \frac{\sum_{x \in range(X)} x \cdot \mathbf{P}(X = x, \mathbb{1}_B = 1)}{\mathbf{P}(\mathbb{1}_B = 1)} \ . \quad (3.1)$$

In the previous chapter we have seen that a program given as an MDP induces (a set of) distributions and a random variable with respect to such a distribution may be represented by a reward function, which gives rise to an RMDP. Using this connection we can naturally

```
1  (f1 := goldfish [0.5] f1 := piranha);
2  f2 := piranha;
3  (sample := f1 [0.5] sample := f2);
4  observe([sample = piranha]);
```

Figure 3.1.: The "fishbowl" program

define conditional probabilities and expectations over programs based on their underlying RMDPs. In the following we introduce the notion of *conditional minimal expected rewards* using a puzzle taken from [62, p. 216]. It goes as follows:

> One fish is contained within the confines of an opaque fishbowl. The fish is equally likely to be a piranha or a goldfish. A sushi lover throws a piranha into the fish bowl alongside the other fish. Then, immediately, before either fish can devour the other, one of the fish is blindly removed from the fishbowl. The fish that has been removed from the bowl turns out to be a piranha. What is the probability that the fish that was originally in the bowl by itself was a piranha?

Let us formalise this "story" in terms of a pGCL program. The result is displayed in Figure 3.1. We are looking for the probability that the fish, initially contained in the fishbowl, f_1 was a piranha. The *observation* that the fish removed has been a piranha is built into the program directly using the *observe* keyword. In order to understand this program's behaviour consider Figure 3.2. Each state of our transition system consists of the program line that the program is currently at and the valuations of the three program variables f_1, f_2 and *sample*. The program starts in line 1 with some undetermined variable valuation[1]. It proceeds to set f_1 probabilistically and f_2 deterministically. The variable *sample* is then assigned the value of either f_1 or f_2 probabilistically. Finally, the *observe* statement is checked. Let the event A be that initially there was a piranha in the fishbowl, i.e. $f_1 = \mathit{piranha}$. In order to measure this event on the RMDP we assign a reward of 1 to the state $\langle \mathit{exit}, p, p, p\rangle$ and 0 everywhere else. Let event B be that we *avoid program runs that terminate with sample* $\neq$ *piranha*. In order to measure this event we label $\langle \mathit{exit}, g, p, g\rangle$ with $\lightning$. Then the answer to the puzzle is

[1] As we did before in Figure 2.2b, we collapse infinitely many, initial states where the values have some particular values into one where we do not care about the variable valuation because the program will overwrite these values any way.

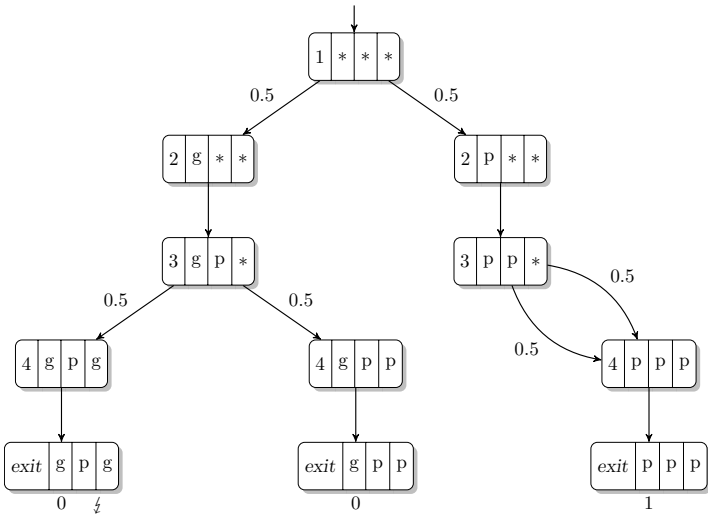


Figure 3.2.: Operational semantics of the fishbowl program

given by

$$\mathbf{P}(A|B) = \frac{1 \cdot 0.5 + 0 \cdot 0.25}{0.5 + 0.25} = \frac{1/2}{3/4} = \frac{2}{3} \ .$$

One may think B should be the event that we reach a state with *sample* = *piranha*. In this particular example both formulations are equivalent, however as we will see later they become crucially different for programs that do not terminate almost surely. This motivates the following definition of conditional minimal expected rewards in RMDPs.

Definition 2 (Conditional expected reward)**.** Let $(\mathcal{M}, r)$ be an RMDP with state space S, $T \subseteq S$ and $s \in S$. Further let $\mathfrak{C}$ denote the set of all cumulative reachability reward values that can be accumulated by paths from s to T in $(\mathcal{M}, r)$. The *conditional minimal expected reward* until reaching T from s avoiding $\lightning$ states, denoted $CExpRew^{(\mathcal{M},r)}(s \models \Diamond T | \neg \Diamond \lightning)$, is defined by:

$$\inf_{\mathfrak{S}} \frac{\sum_{c \in \mathfrak{C}} c \cdot \Pr^{\mathfrak{S}}\{\, \pi \in Paths^{\mathfrak{S}}(s, \Diamond T) \mid r_T(\pi) = c \,\}}{1 - \Pr^{\mathfrak{S}}\{\, \pi \in Paths^{\mathfrak{S}}(s, \Diamond \lightning) \,\}} \ .$$

□

Note that we are of course not the first to consider conditioning on Markov models, cf. [2, 4], for example. We condition on avoiding $\lightning$ states because the *observe* statement does not force a program to reach the observation – it could diverge before, or avoid it by resolving non-

deterministic choices accordingly. Consider the following example

$$\{abort\}\ [0.5]\ \{\{x := 0\}\ [0.5]\ \{x := 1\}\};$$
$$observe\,(x = 1)$$

With probability 0.5 the program diverges and does not reach the observation – and hence does not violate it. Accordingly, Definition 2 determines the conditional expectation of x to be

$$\frac{1 \cdot 0.25}{0.5 + 0.25} = \frac{1}{3}\ .$$

Intuitively the observation rules out $1/4$ of all runs that terminate with $x = 0$. The remaining $3/4$ of runs do not violate the observation, either because they do not reach it or they reach it with $x = 1$. One third of these runs have expectation 1 and two thirds (the aborting runs) have expectation 0, hence the conditional expectation of x is indeed $1/3$. In contrast to this, an alternative definition that conditions on runs that *must reach* the observation statement (instead of just avoiding its falsification) would alter the semantics of the probabilistic program and introduce inconsistencies. In the given example the conditional expectation would be 1 which contradicts our intuition about the operational behaviour as explained previously. Furthermore the statement $observe\,(true)$, which should not alter the behaviour of a program at all, could be used to enforce the program's termination. In particular the programs

$$\{abort\}\ [0.5]\ \{\{x := 0\}\ [0.5]\ \{x := 1\}\}$$

and

$$\{abort\}\ [0.5]\ \{\{x := 0\}\ [0.5]\ \{x := 1\}\};$$
$$observe\,(true)$$

would produce different expectations for x. In the Section 3.2.2 we revisit this issue on the level of *wp* semantics. Finally, note that our definition of conditional expectations corresponds also to our intuition about assert statements in standard programs. A run violates an assertion only if it reaches the assertion and the predicate in the assert statement evaluates to *false*. If a run diverges before, it does not violate the assertion.

3.2. Expectation transformer semantics for programs with conditioning

We extend the definition of the greatest pre-expectation by the rule

$$wp(observe\,(G), f) = [G] \cdot f \ \ ,$$

where G is a boolean predicate. Additionally we need the notion of the greatest *liberal* pre-expectation *wlp* [50, p. 184]. It behaves just like *wp* – in particular non-deterministic choices are resolved demonically as well – except that non-termination is considered desirable and generates the maximal expectation, 1 in this case. Thus *wlp* follows the same rules that were given in Section 2.3.2, page 24ff. except for

$$\begin{aligned} &wlp(abort, f) = 1 \ \text{, and} \\ &wlp(while\,(G)\,\{P\}, f) = \operatorname*{gfp}_x([G] \cdot wlp(P, x) + [\neg G] \cdot f) \ . \end{aligned}$$

In order to define the greatest fixed point, the cpo of expectations that $wlp(P, \cdot)$ ranges over has to be bounded. Since in the context of conditional pre-expectations we will use *wlp* to measure the *probability* to pass observations its range is naturally bounded in $[0, 1]$. In Chapter 4 we will revisit the notion of *wlp* in a broader context.

We extend *wlp* with a rule for *observe* in the same way we did for *wp* above and postulate

$$wlp(observe\,(G), f) = [G] \cdot f \ .$$

Using these expectation transformers, we are able to define conditional expectations on expectation transformer level.

Definition 3 (Conditional pre-expectation)**.** For fully probabilistic[2] pGCL programs P with *observe* statements we define the *conditional pre-expectation* $\underline{cw}p$ as

$$\underline{cw}p(P, f) = \frac{wp(P, f)}{wlp(P, 1)} \ .$$

□

Example 3 (Evaluation of $\underline{cw}p$)**.** Assume some pGCL program of the form $P; observe\,(G)$ and post-expectation f. Then

$$\underline{cw}p(P; observe\,(G), f) = \frac{wp(P; observe\,(G), f)}{wlp(P; observe\,(G), 1)} = \frac{wp(P, [G] \cdot f)}{wlp(P, [G])} \ .$$

[2]Programs without non-deterministic choice.

We see how the conditional weakest pre-expectation amounts to determining the expectation of f and the indicator random variable of G and dividing it by the probability that P does not terminate or establishes G. This reminds us of the definition of conditional expectations in (3.1). Note however that in general it is not required that the observation is the last statement of the program. In principle, as with any other command, an arbitrary number of observations may occur in the program text. □

For program P that does not contain any observations, it holds

$$\underline{cw}p(P, f) = wp(P, f) \ .$$

This means that our definition of $\underline{cw}p$ naturally generalises wp. A remark about notation: our new transformer is denoted $\underline{cw}p$ where the "c" stands for *conditional* and we keep the "wp" because it essentially behaves like wp. However one should note that the attribute "weakest" does not have much meaning here as we do not have any non-deterministic choices, which could make a difference between *weakest* and *strongest* pre-expectations.

We have to note that our pre-expectation definition of $observe\,(G)$ makes it a genuinely new language construct with respect to the wlp transformer. While for wp it holds

$$wp(observe\,(G), f) = wp(if(\neg G)\,\{abort\}\,else\,\{skip\}, f)$$

for all expectations f, there is no pGCL language construct that could mimic its behaviour with respect to wlp. This is because wlp treats abortion or divergence as an instance of unbounded non-determinism that is angelically resolved to achieve the highest possible expectation. If one could write down a malicious program *demon* that does terminate but has pre-expectation zero with respect to arbitrary post-expectations, then we would have

$$wlp(observe\,(G), f) = wlp(if(\neg G)\,\{demon\}\,else\,\{skip\}, f) \ .$$

However *demon* is not implementable. We state this fact without proof and refer to a result on a dual programming concept which is usually referred to as *magic* [51].

Furthermore we can extend our transfer theorem to conditional (fully probabilistic) programs. Remember that a program P induces an RMDP $\mathcal{R}_f[\![\, P \,]\!]$ on which we can measure the conditional minimal expected reward *CExpRew*. Together with Definition 3 this leads to the following result[3].

[3] Theorem 2 appears in Appendix C as Theorem 4.6. There it is called "correspondence theorem" to emphasise the relationship between the two semantics.

Theorem 2 (Transfer theorem for conditional programs)**.** For fully probabilistic pGCL-program P with *observe* statements, variable valuation η, and post-expectation f:

$$\underline{cw}p(P, f)(\eta) \;=\; CExpRew^{\mathcal{R}_f[\![P]\!]}(\langle P, \eta\rangle \models \Diamond P^{\checkmark} | \neg\Diamond\lightning) \;.$$

□

As before, we refer to Appendix C and the technical report [31] for technical details and proofs.

3.2.1. Infeasible programs

A peculiar corner case with conditional probabilities occurs when conditioning on an impossible event. In applications of *discrete* probability theory one may neglect this issue as "obviously there is no point in conditioning on an event with probability zero". However within the syntax of a programming language it is possible to write programs with impossible observations. We call such programs *infeasible*. There is no syntactical characterisation of feasible or infeasible programs as the study of the following three programs shows. In all three programs we are interested in the conditional expectation of x.

$$\begin{aligned} P_1 :&\quad \{x \coloneqq 0\}\ [0.5]\ \{x \coloneqq 1\}; \mathit{observe}\,(x = 1) \\ P_2 :&\quad \{x \coloneqq 0; \mathit{observe}\,(x = 1)\}\ [0.5]\ \{x \coloneqq 1; \mathit{observe}\,(x = 1)\} \\ P_3 :&\quad x \coloneqq 0; \mathit{observe}\,(x = 1) \end{aligned}$$

The situation for P_1 is straightforward, we have a uniform distribution over the outcomes zero and one, the condition ensures the outcome must be one and hence conditional expected value is one. In P_2 the observation has been pushed inside the probabilistic choice. If we were to consider the branches of the choice separately we see that the left branch contains an infeasible program which sets x to 0 but then ensures it is 1, which is impossible. So what is the meaning of the overall program P_2? It turns out it has the same semantics as P_1. The reader may check that both programs induce the same RMDP. Alternatively, we may evaluate the expectation transformer $\underline{cw}p$ and see that

$$\underline{cw}p(P_1, x) = \frac{wp(P_1, x)}{wlp(P_1, 1)} = \frac{0.5 \cdot 0 + 0.5 \cdot 1}{0.5 \cdot 0 + 0.5 \cdot 1} = \frac{wp(P_2, x)}{wlp(P_2, 1)} = \underline{cw}p(P_2, x) \;.$$

This example[4] shows that the conditional pre-expectation of probabilistic choice is not obtained as the weighted average between conditional pre-expectations of subprograms (as was

[4]Note, that we use "x" both as a variable identifier in program text and as an expectation which evaluates to the real value that is associated with x.

the case with wp).

$$\begin{aligned}\underline{cw}p(\{P\}\ [a]\ \{Q\}, f) &= \frac{wp(\{P\}\ [a]\ \{Q\}, f)}{wlp(\{P\}\ [a]\ \{Q\}, 1)} \\ &= \frac{a \cdot wp(P, f) + (1-a) \cdot wp(Q, f)}{a \cdot wlp(P, 1) + (1-a) \cdot wlp(Q, 1)} \\ &\neq a \cdot \frac{wp(P, f)}{wlp(P, 1)} + (1-a) \cdot \frac{wp(Q, f)}{wlp(Q, 1)} \\ &= a \cdot \underline{cw}p(P, f) + (1-a) \cdot \underline{cw}p(Q, f)\ .\end{aligned}$$

In fact, if we consider P_3, which consists of the left branch of P_2 only, we see that the conditional pre-expectation is undefined as we would need to divide by zero. We say P_3 is infeasible. Since there is no syntactical characterisation of feasible programs we need to check that $wlp(P, 1) \neq 0$, which, at least *theoretically*, may be intricate. As an illustration consider the following programs P and Q.

P :
$x \coloneqq 1;$
$\mathit{while}(x = 1)\,\{$
$\quad x \coloneqq 1$
$\}$

Q :
$x \coloneqq 1;$
$\mathit{while}(x = 1)\,\{$
$\quad \{x \coloneqq 1\}\ [0.5]\ \{x \coloneqq 2\};$
$\quad \mathit{observe}\,(x = 1);$
$\}$

Clearly the non-probabilistic program P does not terminate and thus its weakest liberal pre-expectation $wlp(P, 1)$ is always 1 and its conditional pre-expectation $\underline{cw}p(P, f)$ is 0 regardless of the given post-expectation f. Program Q is a slightly modified version of P where we first assign 1 or 2 to x and then ensure the variable was set to 1 using an observation. One may think that Q behaves the same way as P because it establishes $x = 1$ at the end of each iteration, but we claim that in fact $wlp(Q, 1) = 0$. We verify our claim by computing the wlp semantics, which involves finding the fixed point of the loop. The calculation below is shown for completeness only. A careful discussion of fixed point calculations seems distracting at this point and is deferred to Chapter 4.

$$\begin{aligned}&wlp(Q, 1) \\ &= wlp(x \coloneqq 1, wlp(\mathit{while}\,(x = 1)\,\{\{x \coloneqq 1;\}\ [0.5]\ \{x \coloneqq 2;\};\ \mathit{observe}\,(x = 1)\}, 1)) \\ &= wlp(x \coloneqq 1, \operatorname*{gfp}_{z}([x = 1] \cdot wlp(\{x \coloneqq 1;\}\ [0.5]\ \{x \coloneqq 2;\};\ \mathit{observe}\,(x = 1), z) + [x \neq 1] \cdot x)) \\ &= wlp(x \coloneqq 1, \operatorname*{inf}_{n}([x = 1] \cdot wlp(\{x \coloneqq 1;\}\ [0.5]\ \{x \coloneqq 2;\};\ \mathit{observe}\,(x = 1), 1) + [x \neq 1] \cdot x)^n)\end{aligned}$$

$$
\begin{aligned}
&= wlp(x \coloneqq 1, \inf_n([x=1] \cdot wlp(\{x \coloneqq 1;\}\ [0.5]\ \{x \coloneqq 2;\}; \mathit{observe}\,(x=1),\\
&\qquad [x=1] \cdot 0.5 + [x \neq 1] \cdot x) + [x \neq 1] \cdot x)^{n-1})\\
&= wlp(x \coloneqq 1, \inf_n([x=1] \cdot wlp(\{x \coloneqq 1;\}\ [0.5]\ \{x \coloneqq 2;\}; \mathit{observe}\,(x=1),\\
&\qquad [x=1] \cdot 0.5^2 + [x \neq 1] \cdot x) + [x \neq 1] \cdot x)^{n-2})\\
&\vdots\\
&= wlp(x \coloneqq 1, [x \neq 1] \cdot x)\\
&= 0
\end{aligned}
$$

The difference to the non-probabilistic program is that the observation admits one diverging run, but this run almost surely never happens. Intuitively, we are flipping a coin infinitely often and due to the observation force it to land on the same side every time. But the event that a fair coin will never land on tails (assuming 2 represents tails) in an infinite number of trials has probability zero. And since $wlp(Q, 1) = 0$, the program is infeasible and the conditional expected outcome cannot be measured.

3.2.2. Alternative definition

We defined $\underline{cwp}$ as wp scaled by wlp where the latter measures the probability to pass all observations or to avoid them. In principle we could have defined the conditional pre-expectation as

$$\frac{wp(P, f)}{wp(P, 1)} \ . \tag{3.2}$$

But as already discussed in Section 3.1, page 32ff., non-termination and violation of observations would be regarded as the same event in this case. As a consequence one would always implicitly condition on the fact that the program terminates almost surely and thus for observation free programs this definition does not generalise wp.

3.2.3. Expectation transformers and non-determinism

On inductive definitions of transformers and positional schedulers. The expectation transformers $wp(P, \cdot)$ and $wlp(P, \cdot)$ are defined *inductively* on the structure of the program P. This means in particular that in order to determine, say, $wlp(R; \{P\}\,[]\,\{Q\}, f)$, we first determine $f' = wlp(\{P\}\,[]\,\{Q\}, f)$ and subsequently $wlp(R, f')$. Here the intermediate expectation f' depends only on the transformation of f with respect to programs P and Q and is independent of R. Operationally this means that in all states which offer the choice between P

```
1  x := 0;
2  i := 0;
3  continue := true;
4  while(continue) {
5    (continue := false [] i := i + 1);
6  }
7  while(i > 0) {
8    (x := 1 [0.5] i := i - 1);
9  }
```

Figure 3.3.: This program can almost surely terminate with $x \neq 0$, but there exists no scheduler that implements this behaviour.

and Q, the scheduler can resolve it regardless of the path (i.e. history) that led to this state. This has been pointed out in [34] where they state that it is sufficient to consider positional[5] schedulers for calculating expectations. While this claim is undoubtedly true we would like to elaborate on this and stress why the "infimum" in the definition of e.g. the operational equivalent of *wlp*

$$\begin{aligned} &\mathit{LExpRew}^{(\mathcal{M},r)}(s \models \Diamond T) \\ &= \inf_{\mathfrak{S}} \left\{ \Pr^{\mathfrak{S}}(s \not\models \Diamond T) + \sum_{c \in \mathfrak{C}} c \cdot \Pr^{\mathfrak{S}}\{\, \pi \in \mathit{Paths}^{\mathfrak{S}}(s, \Diamond T) \mid r_T(\pi) = c \,\} \right\} \ , \end{aligned}$$

is crucial. The claim above merely states that to determine the infimum it suffices to consider only positional schedulers $\mathfrak{S}$, however this does not mean that there is a positional scheduler that actually achieves the lower bound, in fact it may be the case that such a scheduler does not exist at all. We demonstrate this by means of an example, which was inspired by [9]. Let P be the program in Figure 3.3. Obviously we have $\mathit{wlp}(P, [x = 1]) = 0$ because the demon would choose to terminate the first loop after the first iteration and thus never increase i. Consequently the second loop is not entered and the program terminates with $x = i = 0$. However $\mathit{wlp}(P, [x = 0])$ equals 0 as well, which is not that easy to see. A weakest liberal pre-expectation with value 0 means that the program terminates almost surely. Otherwise the non-termination probability would be included by *wlp* and the result would be strictly greater than 0. Further, upon termination $x \neq 0$ must hold. To avoid $x = 0$ almost surely,

[5] Also called memoryless.

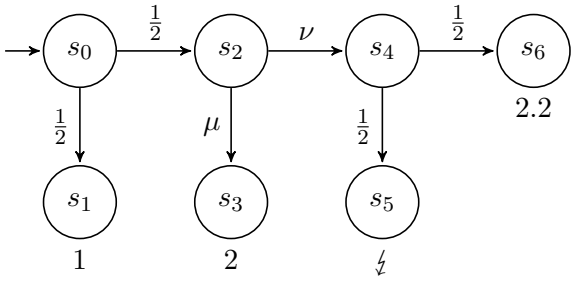


Figure 3.4.: Positional schedulers do not suffice in this RMDP.

we have to execute the second loop an unbounded number of times. For that the value of i must be arbitrarily large, which is only possible when the first loop is executed an unbounded number of times. Hence there is no particular value k of the counter i after which a scheduler may decide to stop incrementing i and proceed to the next loop. Instead our result $wlp(P, [x = 0]) = 0$ is obtained as the infimum over all schedulers that terminate the loop after some $k \in \mathbb{N}$ steps. This explains why the definitions of *ExpRew*, *LExpRew* and *CExpRew* use infima. However, for each k the underlying scheduler is positional. This insight is important as we are used to assuming finite state MDPs in the area of formal methods. For finite state MDPs one could obviously substitute the infimum by a minimum and consequently *implement* a positional scheduler that does achieve the minimal expected reward.

Schedulers are not compositional with respect to conditional minimal expected rewards. In the following we illustrate how schedulers that minimise the conditional expected reward in an RMDP depend on "context". As a consequence we do not have a $\underline{cwp}$ rule for non-deterministic choice that tells us how to compute the minimal conditional expectation from the current valuation and the $\underline{cwp}(P, \cdot)$ and $\underline{cwp}(Q, \cdot)$ of the two subprograms P and Q.

Consider the RMDP $\mathcal{R}$ in Figure 3.4. There are only two schedulers. Let $\mathfrak{S}_\mu$ be the scheduler that chooses to go from s_2 to s_3 and let $\mathfrak{S}_\nu$ be the scheduler that chooses s_4 as the successor of s_2. Further let $T = \{s_1, s_3, s_6\}$ and let $\mathcal{R}_{\mathfrak{S}}$ be the Markov chain obtained from $\mathcal{R}$ by resolving all choices according to $\mathfrak{S}$. Then we calculate $CExpRew^{\mathcal{R}_{\mathfrak{S}_\mu}}(\Diamond T \,|\, \neg\Diamond\lightning) = 1.5$ and $CExpRew^{\mathcal{R}_{\mathfrak{S}_\nu}}(\Diamond T \,|\, \neg\Diamond\lightning) = 1.4$. Hence $CExpRew^{\mathcal{R}}(\Diamond T \,|\, \neg\Diamond U) = 1.4$ and the minimising scheduler is $\mathfrak{S}_\nu$. However if we only consider the subsystem $\mathcal{R}$' that starts execution in state s_2 we obtain $CExpRew^{\mathcal{R}'_{\mathfrak{S}_\mu}}(\Diamond T \,|\, \neg\Diamond\lightning) = 2$ and $CExpRew^{\mathcal{R}'_{\mathfrak{S}_\nu}}(\Diamond T \,|\, \neg\Diamond\lightning) = 2.2$. So in that subsystem the minimising choice is given by $\mathfrak{S}_\mu$. This shows how choices are resolved

depending on the "context" within which the state occurs in the system.

Therefore any attempt to define any transformer $\mathcal{T}$ for non-deterministic choice as

$$\mathcal{T}(\{P\} [] \{Q\}, X) = \min_{\preccurlyeq}\{\mathcal{T}(P, X), \mathcal{T}(Q, X)\}$$

must fail for any representation of conditional expectations X and any order $\preccurlyeq$ between them because the decision is made at a point where the "context" information is missing. In this sense no inductive definition of a conditional expectation transformer is possible as was the case for *wp* and *wlp*.

To remedy this we could devise a transformer that does not resolve choices immediately but delays the decision between subprograms until the whole program has been evaluated. We would call such a transformer a *powerset expectation transformer* because it must keep track of all possible combinations of decisions. However it seems that this straightforward approach is impractical, particularly in the context of loops. At this stage the study of conditional expectation transformer semantics for non-deterministic choice remains a problem for future work.

3.3. Reasoning with conditioning

Note that our examples have just one observation at the end of the program. This is because it is quite natural to state a requirement about (parts of) the outcome of the program. However all results, unless otherwise stated, apply to programs with an arbitrary number of observations written at arbitrary positions within the program text.

3.3.1. Replacing observations by loops

In this section we show that conditional expectations over a program with observations can be cast as (unconditional) expectations over a transformed program without observations. In that sense, observations are "syntactic sugar" to the pGCL language. For conditional semantics that normalise with respect to the terminating behaviour of programs as in Section 3.2.2, we may use the following equivalence between program constructs

$$wp(observe\,(\neg G), f) = wp(if\,(G)\,\{abort\}\,else\,\{skip\}, f) = wp(while\,(\neg G)\,\{skip\}, f)\ .$$

Hence *observe* statements can readily be replaced by a non-terminating loop [17, 39]. The *conditional* expectations can be then computed on this transformed observation-free program.

Our semantics does not require that a run terminates but only that it does not violate any observation. We can establish an analogous result but the program transformation is more intricate. Briefly stated, the idea is to restart a violating run from the initial state until it satisfies all encountered observations. To achieve this we introduce a fresh boolean-valued variable *rerun* and transform a given program P into a new program P' as described below:

$$\begin{aligned} \mathit{observe}\,(G) &\quad\to\quad \mathit{if}(\neg G)\,\{\mathit{rerun} \coloneqq \mathit{true}\}\,\mathit{else}\,\{\mathit{skip}\} \\ x \coloneqq E &\quad\to\quad \mathit{if}(\neg \mathit{rerun})\,\{x \coloneqq E\}\,\mathit{else}\,\{\mathit{skip}\} \\ \mathit{abort} &\quad\to\quad \mathit{if}(\neg \mathit{rerun})\,\{\mathit{abort}\}\,\mathit{else}\,\{\mathit{skip}\} \\ \mathit{while}\,(G)\,\{\ldots\} &\quad\to\quad \mathit{while}\,(G \wedge \neg \mathit{rerun})\,\{\ldots\} \end{aligned} \tag{3.3}$$

Initially, the variable *rerun* is false. The first transformation replaces *observe* statements by if–then–else statements that use the variable *rerun* to indicate that some observation has been violated. The other transformations take account of commands that alter the program's state or divergence behaviour. Our aim is that once *rerun* is true, i.e. an observation has been violated, the execution skips over the rest of the program text to the end. If we do not skip over assignments this may lead to an undefined state as in the following example

$$\begin{aligned} &\{x \coloneqq 0\}\ [0.5]\ \{x \coloneqq 1\} \\ &\mathit{observe}\,(x \neq 0); \\ &y \coloneqq {}^{y}\!/_{x}; \end{aligned}$$

In this program the observation makes sure that the program does not divide by zero. Similarly, an observation may prevent a run from aborting or diverging as in the following example

$$\begin{aligned} &\{x \coloneqq 0\}\ [0.5]\ \{x \coloneqq 1\} \\ &\mathit{observe}\,(x \neq 0); \\ &\mathit{while}(x < 10)\,\{ \\ &\quad x \coloneqq x \cdot 2; \\ &\} \end{aligned}$$

In order to ensure that the termination behaviour of P and P' are the same, we encapsulate the *abort* statement as we did for assignment and we strengthen the guard of each loop.

The transformation from P to P' gives us an observation free-program such that for every post-expectation f, the conditional pre-expectation of f given that *rerun* remains false in P' equals the conditional pre-expectation of f in P. Now we can get rid of the conditioning

by repeatedly executing P' from the same initial state until *rerun* remains *false*, which corresponds to the event that a run in the original program P passes all observations.

This is implemented by program P'' below:

$$
\begin{aligned}
&s_1, \ldots, s_n \coloneqq x_1, \ldots, x_n;\\
&\mathit{rerun} \coloneqq \mathit{true};\\
&\mathit{while}(\mathit{rerun})\,\{\\
&\qquad x_1, \ldots, x_n \coloneqq s_1, \ldots, s_n;\\
&\qquad P';\\
&\}
\end{aligned}
\tag{3.4}
$$

Here, $s_1, \ldots, s_n$ are fresh variables and $x_1, \ldots, x_n$ are all program variables of P. The first assignment stores the initial state in the variables s_i and the first line of the loop body, ensures that the loop always starts with the same (initial) values. These transformations show that *observe* can be considered as syntactic sugar in the pGCL language, which is formally stated in our next theorem[6].

Theorem 3. Let programs P and P'' be as above. Then

$$\underline{cw}p[P](f) = wp[P''](f) \ .$$

□

Note that in the proof ([31], p. 23) we exploit our transfer theorem (Theorem 2) and argue on the level of operational semantics. Hence the result holds not only for fully probabilistic programs but for pGCL programs with *observe* in general. However $\underline{cw}p$ is restricted to fully probabilistic programs. We illustrate the two main steps of our transformation using our fishbowl example.

Example 4 (Replace observations by a loop)**.**
Original program P:

```
1  (f1 := goldfish [0.5] f1 := piranha);
2  f2 := piranha;
3  (sample := f1 [0.5] sample := f2);
4  observe([sample = piranha]);
```

Transformation to P' using transformations at 3.3:

[6] Theorem 3 appears as Theorem 6.2 in our extended technical report [31].

```
1  rerun := false;
2  {if(!rerun){f1 := goldfish;}} [0.5] {if(!rerun){f1 := piranha;}}
3  if(!rerun){f2 := piranha;}
4  {if(!rerun){sample := f1;}} [0.5] if(!rerun){{sample := f2;}}
5  if(sample != piranha) {rerun := true;}
```

Final result P'' using transformations at 3.4:

```
1  s1 := f1;
2  s2 := f2;
3  s3 := sample;
4  rerun := true;
5  while(rerun) {
6    f1 := s1;
7    f2 := s2;
8    sample := s3;
9    rerun := false;
10   {if(!rerun){f1 := goldfish;}} [0.5] {if(!rerun){f1 := piranha;}}
11   if(!rerun){f2 := piranha;}
12   {if(!rerun){sample := f1;}} [0.5] if(!rerun){{sample := f2;}}
13   if(sample != piranha) {rerun := true;}
14 }
```

□

3.3.2. Replacing loops by observations

Theorem 3 shows how to define and effectively calculate the conditional expectation using a straightforward program transformation and the well established notion of *wp*. However in practice it will often be infeasible to calculate the fixed point of the outer loop or to find a suitable loop invariant – even though it exists. This is because finding fixed points of loops is the major obstacle in automated program analysis as we will see in Chapter 4. Thus the loop introduced by this transformation increases the analysis effort. In particular a program with simple (i.e. non-nested) loops will be turned into a program with nested loops. While the result in the previous section is of theoretical interest it does not simplify the

analysis of programs. In practice, one would prefer to analyse the straight-line program P from Example 4 over the program with a loop. It seems beneficial to have a transformation that goes the other way around. However while the transformation in Theorem 3 works for any program, no transformation in the other direction can be expected that is applicable in general. Yet we can identify a subclass of programs that can easily be transformed to a loop-free program, albeit an additional *observe* statement. Reconsider the last program P'' in Example 4. We see that the variables $s_1, \ldots, s_3$ are obsolete because f_1, f_2 and *sample* are set independently of their values in lines 10-12. Moreover the decision whether the loop has to perform one more iteration is made at the very end in line 13. Hence we can push the predicate into the loop's header and replace line 4 by an arbitrary assignment that ensures $\mathit{sample} \neq \mathit{piranha}$. A simplified version of P'' can thus be written as:

```
1  sample := goldfish;
2  while(sample != piranha) {
3    (f1 := goldfish [0.5] f1 := piranha);
4    f2 := piranha;
5    (sample := f1 [0.5] sample := f2);
6  }
```

By the previous arguments we have seen that there is no dataflow between the iterations of the loop. Hence the iterations of the loop generate a sequence of program variable valuations that are *independent and identically distributed* (iid). We refer to such loops as *iid loops*, which can be formally defined as follows.

Definition 4 (iid loop)**.** A loop $\mathit{while}\,(G)\,\{P\}$ is called *iid*, if $\mathit{wp}(P, f) = \mathit{wp}(P^k, f)$ for all expectations f and $k \in \mathbb{N}\backslash\{\,0\,\}$, where $P^k = \underbrace{P; P; \ldots P}_{k}$ is the k-fold repetition of P. □

The aim of Definition 4 is to capture the absence of dataflow between loop iterations formally by requiring that the distribution generated by running the loop body P once is indistinguishable from the distribution generated after running it multiple times. Since we do not have access to distributions in our semantics, we require that the pre-expectations agree for any post-expectation f. Indeed two discrete distributions are equal if and only if the expectation of any random variable over these distributions is equal. In particular, if two discrete distributions differ, there must be an outcome ω, such that $\mathbb{E}(\mathbb{1}_\omega)$ differ between the two distributions.

Example 5 (iid loops)**.** Consider the programs P and Q:

$$
\begin{array}{lll}
P: & b \coloneqq 0 & \\
& \mathit{while}(c)\{ & \\
& \quad b \coloneqq 1 - b & \\
& \quad \{c \coloneqq 0\}\ [1/2]\ \{c \coloneqq 1\} & \\
& \} &
\end{array}
\qquad
\begin{array}{lll}
Q: & \mathit{while}(c)\{ & \\
& \quad b \coloneqq 1 & \\
& \quad \{c \coloneqq 0\}\ [1/2]\ \{c \coloneqq 1\} & \\
& \quad \mathit{if}(c)\{ & \\
& \qquad b \coloneqq 0 & \\
& \qquad \{c \coloneqq 0\}\ [1/2]\ \{c \coloneqq 1\} & \\
& \quad \} & \\
& \} &
\end{array}
$$

The loop in P inverts a bit b on every iteration. This inversion requires the knowledge of the value of b from the previous iteration, hence we have dataflow between iterations and the loop is not iid. Another way to see this is to check the distributions generated by the iterations of the loop body. Let *body* denote the loop body in P. We have

$$\mathit{wp}(\mathit{body}, [b = 1]) = [1 - b = 1] \neq [b = 1] = \mathit{wp}(\mathit{body}^2, [b = 1]) \ .$$

Thus running the loop once produces a distribution which is different from the distribution produced after two runs and that by Definition 4 violates the iid property.

By contrast, the loop in program Q is iid. This is easy to see, as both variables b and c are set regardless of their previous value inside the loop body of Q. More formally, let *body* denote the loop body of Q. Then we can show that for any post-expectation $f(b, c)$ and any number of repetitions of *body* the pre-expectations agree, i.e.:

$$\mathit{wp}(\mathit{body}, f(b,c)) = \frac{1}{4} \cdot f(0,0) + \frac{1}{4} \cdot f(0,1) + \frac{1}{2} \cdot f(1,0) = \mathit{wp}(\mathit{body}^k, f(b,c)) \quad \text{for all } k > 1 \ .$$

This example shows also a curiosity. Namely that sometimes a loop which is not iid can be rewritten into one which is iid. Here in fact program Q is obtained from P by merging two iterations of the loop together. Of course this does not always work. For example, a loop with a counter cannot be an iid loop. □

The iid property allows to replace a loop by its body and an observation.

Theorem 4 (Transformation of iid loops)**.** Let $\mathit{loop} = \mathit{while}\,(G)\,\{P\}$ be an iid loop and let $Q = \mathit{if}(G)\,\{P; \mathit{observe}\,(\neg G)\}\,\mathit{else}\,\{\mathit{skip}\}$. Then for any expectation f

$$\mathit{wp}(\mathit{loop}, f) = \underline{\mathit{cw}}p(Q, f) \ .$$

□

Proof. Apply Theorem 3 to program Q. Let the resulting program be *loop*'. Since *loop* is iid, from Definition 4 we have that $\mathit{wp}(P, f) = \mathit{wp}(P^k, f)$ for all f and k and therefore *loop*' is iid, too. Thus the same simplification steps as in Example 4 apply: there is only one *observe* statement at the end of *loop*' and furthermore there is no data flow between iterations of *loop*'. Hence by removing all if–then–else statements that are vacuously true and pushing the observation into the loop header we arrive at the desired program *loop*. □

Theorem 4 formalises a claim from our technical report [31], p. 10. There we tacitly assumed that the loop will be executed at least once and therefore left out the if–then–else statement in Q, which is now taken care of explicitly.

3.3.3. Observation hoisting

Here we present yet another program transformation that supports the removal of *observe* statements from programs. The idea is to "push" all *observe* statements upwards in the program text such that in the end we obtain a program with one initial observation followed by an observation free pGCL program text. For this we generalise observations to be functions in the $[0, 1]$ interval rather than just predicates. Intuitively such a quantitative observation gives us the probability that the program fragment that follows it will establish some condition. Figure 3.5 lists the transformation rules for each pGCL command. The single most important transformation rule is the one for probabilistic choice. Based on the valuation of the current state, it rescales the probabilistic choice proportional to the probability of the successor states to pass all observations.

With the transformation rules from Figure 3.5 we establish the following result[7].

Theorem 5 (Correctness of hoisting)**.** Let P admit at least one feasible run for every initial state and $\mathcal{T}(P, 1) = (\hat{P}, \hat{h})$. Then for any expectation f,

$$\mathit{wp}(\hat{P}, f) = \underline{\mathit{cw}}p(P, f) \ .$$

□

[7] Theorem 5 appears as Theorem 5.1 in Appendix C.

$$
\begin{array}{lcl}
\mathcal{T}(\mathit{observe}\,(G), f) & = & (\mathit{skip}, [G] \cdot f) \\
\mathcal{T}(\mathit{skip}, f) & = & (\mathit{skip}, f) \\
\mathcal{T}(\mathit{abort}, f) & = & (\mathit{abort}, 1) \\
\mathcal{T}(x \coloneqq E, f) & = & (x \coloneqq E, f[x/E]) \\
\mathcal{T}(\mathit{if}(G)\,\{P\}\,\mathit{else}\,\{Q\}, f) & = & (\mathit{if}(G)\,\{P'\}\,\mathit{else}\,\{Q'\}, [G] \cdot f_P + [\neg G] \cdot f_Q) \\
 & & \text{where } (P', f_P) = \mathcal{T}(P, f),\ (Q', f_Q) = \mathcal{T}(Q, f) \\
\mathcal{T}(\{P\}\ [a]\ \{Q\}, f) & = & (\{P'\}\ [a']\ \{Q'\}, a \cdot f_P + (1-a) \cdot f_Q) \\
 & & \text{where } (P', f_P) = \mathcal{T}(P, f),\ (Q', f_Q) = \mathcal{T}(Q, f), \text{ and} \\
 & & a' = \frac{a \cdot f_P}{a \cdot f_P + (1-a) \cdot f_Q} \\
\mathcal{T}(\mathit{while}\,(G)\,\{P\}, f) & = & (\mathit{while}\,(G)\,\{P'\}, f') \\
 & & \text{where } f' = \mathrm{gfp}_x\,([G] \cdot (\pi_2 \circ \mathcal{T})(P, x) + [\neg G] \cdot f)\,, \text{ and} \\
 & & (P', \_) = \mathcal{T}(P, f') \\
\mathcal{T}(P; Q, f) & = & (P'; Q', f'') \text{ where } (Q', f') = \mathcal{T}(Q, f),\ (P', f'') = \mathcal{T}(P, f')
\end{array}
$$

Figure 3.5.: Program transformation for hoisting *observe* statements. In the transformation of the while–loop the function π_2 is the projection to the second component (of $\mathcal{T}$).

Note that we apply the transformation rules from Figure 3.5 where f initially is the constant expectation 1. Hence none of the commands change it, except for the *observe* command. It will introduce the predicate G and all further hoisting steps are then carried out with respect to G (and any other observations found on the way up). We revisit our simple fishbowl example for a last time to illustrate a straightforward application of hoisting.

Example 6 (Hoisting *observe*)**.**
Original program

$$\begin{aligned}
&\{f_1 \coloneqq \mathit{goldfish}\}\ [0.5]\ \{f_1 \coloneqq \mathit{piranha}\};\\
&f_2 \coloneqq \mathit{piranha};\\
&\{\mathit{sample} \coloneqq f_1\}\ [0.5]\ \{\mathit{sample} \coloneqq f_2\};\\
&\mathit{observe}\,([\mathit{sample} = \mathit{piranha}])
\end{aligned}$$

First step of the hoisting transformation

$$\begin{aligned}
&\{f_1 \coloneqq \mathit{goldfish}\}\ [0.5]\ \{f_1 \coloneqq \mathit{piranha}\};\\
&f_2 \coloneqq \mathit{piranha};\\
&\mathit{observe}\,([f_1 = \mathit{piranha}] \cdot 0.5 + [f_2 = \mathit{piranha}] \cdot 0.5);\\
&\{\mathit{sample} \coloneqq f_1\}\ \left[\frac{[f_1 = \mathit{piranha}] \cdot 0.5}{[f_1 = \mathit{piranha}] \cdot 0.5 + [f_2 = \mathit{piranha}] \cdot 0.5}\right]\ \{\mathit{sample} \coloneqq f_2\}
\end{aligned}$$

Second step of the hoisting transformation

$$\begin{aligned}
&\{f_1 \coloneqq \mathit{goldfish}\}\ [0.5]\ \{f_1 \coloneqq \mathit{piranha}\};\\
&\mathit{observe}\,([f_1 = \mathit{piranha}] \cdot 0.5 + 0.5);\\
&f_2 \coloneqq \mathit{piranha};\\
&\{\mathit{sample} \coloneqq f_1\}\ \left[\frac{[f_1 = \mathit{piranha}] \cdot 0.5}{[f_1 = \mathit{piranha}] \cdot 0.5 + [f_2 = \mathit{piranha}] \cdot 0.5}\right]\ \{\mathit{sample} \coloneqq f_2\}
\end{aligned}$$

Last step of the hoisting transformation

$$\begin{aligned}
&\mathit{observe}\,(3/4);\\
&\{f_1 \coloneqq \mathit{goldfish}\}\ [1/3]\ \{f_1 \coloneqq \mathit{piranha}\};\\
&f_2 \coloneqq \mathit{piranha};\\
&\{\mathit{sample} \coloneqq f_1\}\ \left[\frac{[f_1 = \mathit{piranha}] \cdot 0.5}{[f_1 = \mathit{piranha}] \cdot 0.5 + [f_2 = \mathit{piranha}] \cdot 0.5}\right]\ \{\mathit{sample} \coloneqq f_2\}
\end{aligned}$$

□

Conceptually we have to pay the price of finding loop fixed points in order to hoist observations over loops. We gain a separation between the observations that we condition on and the rest of the program which can be analysed using the *wp* transformer. Such a hoisting can be applied e.g. in simulation approaches where one would like to terminate infeasible executions a soon as possible. Hoisting observations all the way through the program thus allows to generate only feasible runs and leads to a better performance of the simulation technique. This has been exploited in [54], but there instead of coin flips they introduce probability by sampling from distributions. Therefore their transformation rule is weaker in that it does not compute the weakest pre-expectation with respect to the probabilistic assignment. Instead their rule overapproximates probabilistic assignment by non-deterministic assignment and the observation is existentially quantified. In cases where the distribution is Bernoulli (or a distribution that can be build using Bernoulli trials as shown in the introduction) our technique is more accurate. However our hoisting transformation cannot be applied to programs that sample from e.g. continuous distributions.

The presented transformation could in principle be automated. All transformation rules except the rule for the loop are purely syntactical. In order to apply the transformation to a loop we first need to find a pre-expectation with respect to the original loop. In principle, we could cast the problem of finding a fixed point as an invariant generation problem (studied in Chapter 4) and apply the machinery there. The difference is just that instead of applying *wp*, we need to apply $\mathcal{T}$ to the loop body P and the given post-expectation f. Finally, given that pre-expectation, the loop body is transformed again in a syntactical manner. However at this point the transformation has not been yet implemented and evaluated but certainly is interesting future work for our tool development.

After independently discovering the rules in Figure 3.5 we found out that the same "trick" has recently been applied in [4] where a transformation of Markov chains is introduced to facilitate a fast computation of conditional probabilities of ω-regular objectives. The correspondence between our hoisting transformation of probabilistic programs and their transformation of Markov chains may once again be seen as a consequence of the transfer theorem and moreover serves as a good sanity check.

3.3.4. iid loops and hoisting – a case study

In what follows we apply the presented transformation techniques to determine conditional probabilities in a randomised network protocol.

Example 7 (Zeroconf)**.** When a device is connected to a network it needs to be assigned a unique IP address to enable communication with other devices on the network. The chosen address must be unique as the device would otherwise cause a collision in the network, which is highly undesirable. Zeroconf [15] is a protocol which allows the device to configure its own IP address automatically without the need of a centralised server that manages the IPs of all devices connected to the network. Such an IP configuration mechanism may be used in ad hoc networks, for example. We are interested in the high level, probabilistic behaviour of Zeroconf and follow [8] in its presentation.

Once connected, the device *randomly* generates its IP address. To verify that it is unique it broadcasts a probe to the network and waits a certain amount of time for a reply in case another device is already using the chosen address. It is possible that such an answer to the probe is lost or does not arrive before the timer expires. Therefore the device sends several probes and settles for the chosen IP address only if none of the probes receives an answer. We assume some probability q to pick an address which is already in use. Furthermore we assume that a fixed number N of probes is sent and each probe's answer may (independently of all the others) get lost with probability p. The protocol is modelled as a pGCL program in Figure 3.6. As said before it is crucial to avoid collisions. Therefore the goal is to find the probability that the protocol settles on an already used IP address, i.e. terminates with $collision = true$. First we simplify the program and remove the for-loop while updating the message loss probability to p^N. This can be done because in case there is no collision, the for-loop has no effect (and neither has the updated probabilistic choice) and in case a collision does happen, *configured* will be reset to *false* unless the left hand side of the choice is chosen on all N iterations which may happen with probability p^N. For future reference we call this program ZEROCONF_1.

```
1  configured := false;
2  while(!configured) {
3          (collision := true [q] collision := false);
4          configured := true;
5          {
6                  skip;
7          }
8          [p^N]
9          {
```

```
1  configured := false;
2  while(!configured) {
3    //choose random IP
4    (collision := true [q] collision := false);
5    //assume an unused IP was chosen
6    configured := true;
7    //query the network N times
8    for(1..N){
9      {
10       //no answer due to message loss
11       skip;
12     }
13     [p]
14     {
15       //if a used IP was chosen,
16       //the probe is answered accordingly
17       //and the protocol is restarted
18       if(collision){
19         configured := false;
20       }
21     }
22   }
23 }
```

Figure 3.6.: A pGCL program modelling the probabilistic behaviour of the Zeroconf protocol. We use $\mathit{for}\,(1..N)\,\{\ldots\}$ to abbreviate $i \coloneqq 0; \mathit{while}\,(i < N)\,\{\ldots; i \coloneqq i + 1\}$.

```
10                  if(collision){
11                          configured := false;
12                  }
13          }
14  }
```

Additionally, we see that there is no data flow between the iterations of the while-loop. The variables *collision* and *configured* are set on each iteration regardless of their previous values. Hence we have an iid loop to which Theorem 4 may be applied to replace the loop by an *observe* statement. Thus we obtain a program ZEROCONF$_2$.

```
1   configured := false;
2   (collision := true [q] collision := false);
3   configured := true;
4   {
5           skip;
6   }
7   [p^N]
8   {
9           if(collision){
10                  configured := false;
11          }
12  }
13  observe(configured = true);
```

We may hoist this observation all the way up through the program text by applying our program transformation from Figure 3.5 yielding the final program ZEROCONF$_3$.

```
1   observe(qp^N / 1-q(1-p^N));
2   configured := false;
3   (collision := true [qp^N / 1-q(1-p^N)] collision := false);
4   configured := true;
5   {
6           skip;
7   }
8   [[configured=true]p^N / [configured=true]p^N + (1-p^N)[\neg collision]]
```

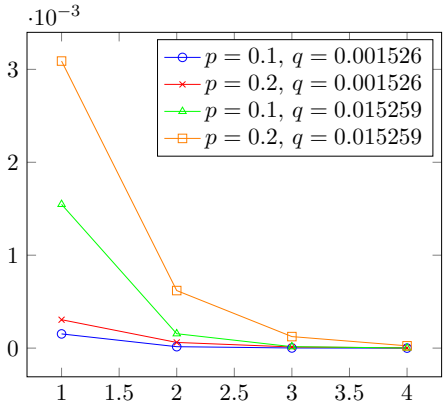


Figure 3.7.: Collision probabilities as functions of the number of probes sent.

```
9   {
10          if(collision){
11                  configured := false;
12          }
13  }
```

By Theorem 4, we have $wp(\text{ZEROCONF}_1, [collision = true]) = \underline{cwp}(\text{ZEROCONF}_2, [collision = true])$ and by Theorem 5, $\underline{cwp}(\text{ZEROCONF}_2, [collision = true]) = wp(\text{ZEROCONF}_3, [collision = true])$. The latter is now trivially given by the probabilistic choice in line 3. Our analysis shows that Zeroconf may cause a collision on the network with probability

$$\frac{qp^N}{1 - q(1 - p^N)} \ .$$

□

Figure 3.7 visualises how the collision probability depends on the number of probes sent. In order to reduce the dimension of the distributions we fix certain values for the probability parameters p and q. We assume the device picks an IP address within the 169.254/16 prefix uniformly at random, which amounts to 65536 possible addresses. Assuming further that 100 out of these are already in use gives $q = 0.001526$, or if 1000 are already in use $q = 0.015259$. We assume that a message is lost either with probability $p = 0.1$ or $p = 0.2$. This gives rise to four scenarios and consequently the four distributions in Figure 3.7. We see that already after three probes the collision probability is far below 10^{-3}. In fact for $q = 0.001526, p = 0.1, N = 3$ the probability amounts to $1.53 \cdot 10^{-6}$.

3.3.5. Conditional expectations in loopy programs – the Crowds protocol

We conclude this chapter by studying a variant of a network anonymity protocol. The example serves two purposes, one is that we demonstrate how conditional expectations are calculated for programs where our previously introduced simplification steps are not applicable. The other is that this example motivates the topic of the next chapter, which deals with the question how to compute fixed points of while-loops.

Example 8 (Crowds)**.** To demonstrate the applicability of the $\underline{cw}p$-semantics to a practical example, we consider the Crowds protocol [58]. A set of nodes forms a fully connected network called the *crowd*. Crowd members would like to exchange messages with a server without revealing their identity to the server. To achieve this, a node *initiates communication* by sending its message to a randomly chosen crowd member, possibly itself. Upon receiving a message, a node probabilistically decides to either *forward* the message once again to a randomly chosen node in the network or to relay it to the server directly. A commonly studied attack scenario is that some malicious nodes called *collaborators* join the crowd and participate in the protocol with the aim to reveal the identity of the sender. The pGCL-program Crowds in Figure 3.8 models this protocol where p is the forward probability and c is the fraction of collaborating nodes in the crowd. The initialisation corresponds to the communication initiation. Our goal is to determine the probability of a message not being intercepted by a collaborator. We condition this by the observation that a message is forwarded at most k times.

Note that the operational semantics of Crowds induce an *infinite parametric RMDP* since the value of k is fixed but arbitrary. Further note that due to the counter the loop is not iid and cannot be easily removed as in the previous examples. The hoisting transformation – though still applicable – requires to find a fixed point of the loop with respect to the observation, which is as hard as determining the $\underline{cw}p$ directly. The probability that a message is not intercepted given that it was rerouted no more than k times is given by

$$\underline{cw}p(\text{Crowds}, [\neg \mathit{intercepted}]) = \frac{wp(\text{Crowds}, [\neg \mathit{intercepted}])}{wlp(\text{Crowds}, 1)} \tag{3.5}$$

The computation of this quantity requires to find fixed points. For details we refer to Appendix A11 of our technical report [31]. As a result we obtain a closed form solution parametrized in p, c, and k:

$$(1-c)(1-p)\frac{1-(p(1-c))^k}{1-p(1-c)} \cdot \frac{1}{1-p^k}$$

```
1  // Let c be the fraction of corrupted nodes on the network
2  // Let p be the forward probability
3  intercepted := false;
4  delivered := false;
5  // Initiate communication path to server by sending the message
6  // to someone in the network
7  (intercepted := true [c] skip);
8  counter := 1;
9  while(delivered = false) {
10         {
11                 (intercepted := true [c] skip);
12                 counter := counter + 1;
13         }
14         [p]
15         {
16                 delivered := true;
17         }
18 }
19 observe(counter <= k)
```

Figure 3.8.: Model of the Crowds protocol where the length of the communication path through the network is bounded by some fixed number k.

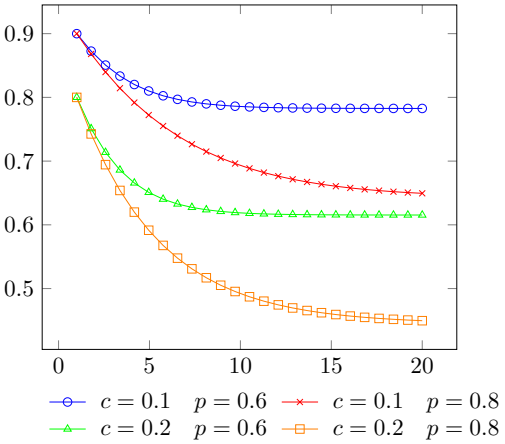


Figure 3.9.: The conditional probability that a message is intercepted as a function of k for fixed c and p.

One can visualise it as a function in k by fixing the parameters c and p. For example, Figure 3.9 shows the conditional probability plotted for various parameter settings. The automation of this analysis requires to find the fixed points in (3.5) automatically. This issue is addressed in the next chapter. □

4. Automated analysis

In the previous chapters we have studied the meaning of probabilistic programs. Now we focus on how we can actually calculate the weakest pre-expectation given a program and a post-expectation. In this chapter we consider pGCL programs as defined in Chapter 2 without observations. Recall that all language constructs other than the while-loop allow a purely syntactical calculation of the pre-expectation, which may easily be automated. However for loops, fixed points need to be found. In the following we use an example to explain how fixed points can be found "by hand". Then we summarise how invariants are used as an alternative means to reason about pre-expectation of loops. Subsequently we show one approach for the analysis of probabilistic programs using invariants and discuss to what extent our software tool Prinsys help the user in this process.

4.1. Proving properties of probabilistic programs

4.1.1. Computing fixed points

Our first example in Figure 1.1 was a program that generated samples according to a geometric distribution. While not very spectacular on its own, we have seen variations of it reappearing in other programs such as the duelling cowboys, cf. Figure 2.2a or the Crowds protocol, cf. Figure 3.8. These programs have a similar structure: there is a loop which may terminate with a fixed probability or update some variables with the complementary probability. The results of the analyses were thus given by some instance of the geometric series. Since we believe that the geometric distribution is at the heart of many other probabilistic programs, we choose its program as the example for the detailed *wp* calculation to come. In the following let P be that program, which for convenience is displayed in Figure 4.1 once again. We are interested to find the mean value of the random variable x. According to the

```
1  x := 0;
2  flip := 0;
3  while (flip = 0) {
4      ( flip := 1 [p] x := x + 1 );
5  }
```

Figure 4.1.: Program P generates a random sample x according to the geometric distribution with parameter p.

denotational semantics presented in Section 2.3.2, this quantity is given by

$$wp(P, x) = wp(x := 0, wp(\mathit{flip} := 0, wp(\mathsf{while}(\mathit{flip} = 0)\{\mathit{flip} := 1[p]x := x + 1\}, x))) \tag{4.1}$$

$$= wp(x := 0, wp(\mathit{flip} := 0, \operatorname*{lfp}_F \left(\underbrace{[\mathit{flip} = 0] \cdot wp(\mathit{flip} := 1[p]x := x + 1, F) + [\mathit{flip} \neq 0] \cdot x}_{\Phi(F)} \right))) \tag{4.2}$$

$$= wp(x := 0, wp(\mathit{flip} := 0, \sup_k \left(\underbrace{[\mathit{flip} = 0] \cdot wp(\mathit{flip} := 1[p]x := x + 1, 0) + [\mathit{flip} \neq 0] \cdot x}_{\Phi(0)} \right)^k)) \ . \tag{4.3}$$

Equation (4.1) is given directly by the semantics of sequential composition of pGCL commands. In the next line we apply the definition of wp for loops. The expectation transformer $\Phi(F) = [\mathit{flip} = 0] \cdot wp(\mathit{flip} := 1[p]x := x + 1, F) + [\mathit{flip} \neq 0] \cdot x$ takes some expectation F and returns its pre-expectation with respect to one loop iteration. The solution to the fixed point equation in (4.2) is obtained using the Kleene fixed point theorem. It tells us that the least fixed point of Φ can be found by taking the supremum over k of Φ^k which denotes the k-fold application of Φ. This results in (4.3). For a detailed account of fixed point theorems, we refer to [45]. There, Theorem 3 applies in our setting where the cpo is the set of expectations with point-wise ordering $(\{S \to \mathbb{R}^\infty_{\geq 0}\}, \leq)$. In order to find the supremum in (4.3) we consider the expression for several k and deduce a pattern:

$$\begin{aligned} \Phi(0) &= [\mathit{flip} = 0] \cdot wp(\mathit{flip} := 1[p]x := x + 1, 0) + [\mathit{flip} \neq 0] \cdot x \\ &= [\mathit{flip} \neq 0] \cdot x \end{aligned}$$

$$
\begin{aligned}
\Phi^2(0) &= \Phi([\mathit{flip} \neq 0] \cdot x) \\
&= [\mathit{flip} = 0] \cdot wp(\mathit{flip} := 1[p]x := x+1, [\mathit{flip} \neq 0] \cdot x) + [\mathit{flip} \neq 0] \cdot x \\
&= [\mathit{flip} = 0] \cdot (px + (1-p)[\mathit{flip} \neq 0](x+1)) + [\mathit{flip} \neq 0] \cdot x \\
&= [\mathit{flip} = 0] \cdot px + [\mathit{flip} \neq 0] \cdot x
\end{aligned}
$$

$$
\begin{aligned}
\Phi^3(0) &= \Phi([\mathit{flip} = 0] \cdot px + [\mathit{flip} \neq 0] \cdot x) \\
&= \dots \\
&= [\mathit{flip} = 0] \cdot (px + (1-p)p(x+1)) + [\mathit{flip} \neq 0] \cdot x
\end{aligned}
$$

$$
\begin{aligned}
\Phi^4(0) &= \Phi([\mathit{flip} = 0] \cdot (px + (1-p)p(x+1)) + [\mathit{flip} \neq 0] \cdot x) \\
&= \dots \\
&= [\mathit{flip} = 0] \cdot (px + (1-p)p(x+1) + (1-p)^2 p(x+2)) + [\mathit{flip} \neq 0] \cdot x \\
&\vdots \\
\Phi^{k+2}(0) &= [\mathit{flip} = 0] \cdot \sum_{i=0}^{k} (1-p)^i \cdot p \cdot (x+i) + [\mathit{flip} \neq 0] \cdot x \\
&\vdots \\
\sup_k \Phi^k(0) &= [\mathit{flip} = 0] \cdot \sum_{i=0}^{\infty} (1-p)^i \cdot p \cdot (x+i) + [\mathit{flip} \neq 0] \cdot x \\
&= [\mathit{flip} = 0] \cdot (\frac{1-p}{p} + x) + [\mathit{flip} \neq 0] \cdot x \ .
\end{aligned}
$$

The jump from $\Phi^4(0)$ to $\Phi^{k+2}(0)$ is justified "by inspection" rather than a formal argument. However, if we believe that this is correct, then we can easily find the supremum and hence the sought fixed point for equation (4.3). To verify that our guess was correct and that we have found the least fixed point, we first check that it indeed is a fixed point.

$$
\begin{aligned}
&\Phi([\mathit{flip} = 0] \cdot (\frac{1-p}{p} + x) + [\mathit{flip} \neq 0] \cdot x) \\
&= [\mathit{flip} = 0] \cdot (px + (1-p)([\mathit{flip} = 0] \cdot (\frac{1-p}{p} + x + 1) + [\mathit{flip} \neq 0] \cdot (x+1))) + [\mathit{flip} \neq 0] \cdot x \\
&= [\mathit{flip} = 0] \cdot (px + (1-p)(\frac{1-p}{p} + x + 1)) + [\mathit{flip} \neq 0] \cdot x \\
&= [\mathit{flip} = 0] \cdot (\frac{1-p}{p} + x) + [\mathit{flip} \neq 0] \cdot x \ .
\end{aligned} \tag{4.4}
$$

To convince ourselves that this fixed point is the least, we merely need to observe that the loop terminates almost surely and that for such loops the greatest and the least fixed points coincide [50, p. 182]. Using this result we can continue the calculation of the expected value of x in program 4.1 as follows:

$$\begin{aligned} wp(P, x) &= wp(x := 0, wp(\mathit{flip} := 0, \\ &\qquad \sup_k \left([\mathit{flip} = 0] \cdot wp(\mathit{flip} := 1[p]x := x + 1, 0) + [\mathit{flip} \neq 0] \cdot x\right)^k)) \\ &= wp(x := 0, wp(\mathit{flip} := 0, [\mathit{flip} = 0] \cdot (\frac{1-p}{p} + x) + [\mathit{flip} \neq 0] \cdot x)) \\ &= wp(x := 0, \frac{1-p}{p} + x) \\ &= \frac{1-p}{p} \end{aligned}$$

This agrees with the expectation of the geometric distribution.

The fixed point calculation can be automated to some degree. Barsotti and Wolovick [5] have suggested a method that iteratively approximates the fixed point. In their paper they study, among other examples, the geometric distribution program as well. The only difference is that they consider the variant where one is interested in the number of Bernoulli trials to get one success. Mathematically speaking, if in our example we consider a random variable X, they consider $X+1$. This difference is however irrelevant for the techniques discussed here and is only mentioned to avoid confusing the interested reader who studies our and their work at the same time. Their method relies on numerical calculations and therefore instead of a parameter p their probabilistic choice must use some numerical value, e.g. 0.5. Furthermore they require a user to provide a template, that is a set of predicates that partition the state space into disjoint areas. Assuming we found a good partition, then their method will iteratively approximate the expectation of the random variable for each of these partitions. In the given example, the template of course is rather straightforward since the execution of the program merely depends on the value of *flip*. Hence they start with an expression

$$[\mathit{flip} = 0] \cdot (a_2 \cdot \mathit{flip} + a_1 \cdot x + a_0) + [\mathit{flip} \neq 0] \cdot (b_2 \cdot \mathit{flip} + b_1 \cdot x + b_0) \tag{4.5}$$

and after some iterations converge to

$$a_2 = b_2 = b_0 = 0 \qquad \text{and} \qquad a_1 = b_1 = 1 \qquad \text{and} \qquad a_0 = 2 \ .$$

The *wp* calculation can be then carried out using the template with its parameters instantiated as above as the pre-expectation of the loop and finally arrive at the pre-expectation of 2 for

their program. This corresponds with the expectation of the modelled distribution which is $1/0.5 = 2$. While the implementation seems promising, it is hard to tell from the two examples discussed in [5] to what class of programs this fixed point approximation is applicable and where the practical limits of this approach are. Conceptually, they exclude reasoning with parameterised programs, i.e. programs where probabilities are not specified by a particular number but some parameter p.

Another approach to compute quantities in probabilistic programs with loops has been introduced by Claret et al. [17]. There again, they assume that all probabilities are given as numerical values and that all variables can be discretised to take values from a finite set. They then use abstract decision diagrams (ADDs) to represent joint probability distributions over the programs state space. Using a forward semantics (as discussed in section 2.3.1) they update an initial distribution until the program terminates and the output distribution can be used to determine any quantity of interest. In order to deal with loops they assume that one can find a threshold such that the distance (e.g. Kullback-Leibler divergence) between a distribution before and after one iteration of the loop becomes smaller than the chosen threshold. The loop is determined to stop then and the computation is carried on, possibly with a small numerical error. Based on the examples discussed in [17], their algorithm produces interesting results. However it is not clear to what degree it may suffer from numerical imprecision due to abstraction to a finite state space and the under-approximation of the loop behaviour. Further the result of their analysis is an ADD representation of the distribution over the program's outcomes. This allows to measure any events or moments, but the result is merely a number. It does not explain how the measured outcome depends on the variables of the program. On the contrary, fixed points—and invariants, which we are about to study in the next section—not only facilitate the calculation of an expectation but also provide a deep insight into the program's behaviour. For example, the fixed point that we found in (4.4) gives the expectation of x depending on the initial value of *flip* and probability parameter p. In the introduction we have compared this to scientific laws which describe relations between physical quantities by means of a mathematical formulas.

4.1.2. Invariants

In the verification of *standard*, i.e. non-probabilistic programs, we often find it convenient to separate the proofs of *termination* and *partial correctness*. For example, in Hoare calculus for standard programs variants are used to establish termination and invariants are required

to show partial correctness. Together they form the proof of *total correctness*. McIver and Morgan [50, Ch. 2] have introduced variants and invariants for probabilistic programs. If, for a probabilistic program P and expectations g and f, we can establish

$$g = wp(\mathit{while}\,(G)\,\{P\}, f) \ , \tag{4.6}$$

then we have shown the total correctness of $\mathit{while}\,(G)\,\{P\}$ with respect to post-expectation f. For an initial valuation η of program variables, $g(\eta)$ is the expectation of f computed over all terminating runs of $\mathit{while}\,(G)\,\{P\}$. In this sense, wp takes care of both, probabilistic termination and expectation calculation. In the following we divide the task of calculating pre-expectations into two separate subtasks: finding a liberal pre-expectation and proving almost sure termination. Stated in Hoare calculus terms we seek to find probabilistic invariants and variants. Probabilistic invariants are the topic in the remainder of this chapter. The theory of probabilistic variants is beyond the topic of our studies and we refer to e.g. [50] for further details.

Definition 5 (Probabilistic loop invariants)**.** An expectation $\mathcal{I}$ is called a probabilistic loop invariant for $\mathit{while}\,(G)\,\{P\}$ if

$$[G] \cdot \mathcal{I} \leq wlp(P, \mathcal{I})$$

□

Essentially, this definition is analogous to the standard definition of invariants for non-probabilistic loops except that our invariant is an expectation and not a predicate and therefore implication between predicates is lifted to an inequality between expectations. From now on we refer to probabilistic loop invariants simply as invariants for readability. The idea of an invariant is that it approximates the liberal pre-expectation of a loop from below since it cannot decrease after one iteration of the loop body P. In that sense, invariants may be seen as variants of martingales [64]. A submartingale is a stochastic process X_n with the property that the expected value of X_n given the knowledge of the value of the previous step X_{n-1} is no less than X_{n-1}. In Definition 5 above we can identify the current value of $\mathcal{I}$ with X_{n-1} and $wlp(P, \mathcal{I})$ with the expectation of X_n. Since non-determinism in P is resolved (demonically) in $wlp(P, \mathcal{I})$, we get a stochastic process and may apply results from martingale theory when reasoning about invariants of probabilistic programs. The link between program analysis of probabilistic programs and martingale theory has first been established by Chakarov and Sankaranarayanan [13].

We have introduced *wlp* in the previous chapter when defining conditional pre-expectations. The difference between *wp* and *wlp* is that the former gives the expectation which is measured over terminating runs whereas the latter gives the expectation over all runs. But what should be the reward of a non-terminating run? The intuition is that in a state where the program diverges the liberal transformer for *standard* programs returns *true*, which is the top element on the cpo of predicates. By analogy we would expect that the probabilistic variant of *wlp* returns the top element on the cpo of expectations. However since expectations are defined on non-negative real numbers, the top element would be an expectation that evaluates to infinity everywhere. But it is not possible to carry out any meaningful computations with infinity as the top element because any consistent definition of addition or multiplication with infinity evaluates to infinity again and any other definition of arithmetical operations immediately leads to contradictions. Therefore we need a top element that allows for reasonable calculations, hence must be an actual real number. For simplicity we fix the range of expectation that *wlp* transforms to $[0, 1]$ and thereby obtain a top element, which is the expectation that evaluates to *one* everywhere. When studying conditional pre-expectations the *wlp* transformer naturally was bound in $[0, 1]$ because it measured the *probability* to satisfy observations. In general this need not be the case. McIver and Morgan [50] merely require that the expectations are bounded by some real upper bound α which may depend on the program at hand. In order to justify the restriction to the $[0, 1]$ interval they argue that for almost surely terminating programs all calculations can be *rescaled* accordingly. Indeed all programs that we have studied in the context of this chapter do terminate almost surely.

A probabilistic invariant, as defined in (5) may be used to prove $g \leq \mathit{wlp}(\mathit{while}(G)\{P\}, f)$ if

$$g \leq \mathcal{I} \tag{4.7}$$

$$[\neg G] \cdot \mathcal{I} \leq f \ . \tag{4.8}$$

If we can additionally prove that the loop terminates from a set of states characterised by T, then we can use $\mathcal{I}$ for the stronger claim $[T] \cdot g \leq \mathit{wp}(\mathit{while}(G)\{P\}, f)$, i.e. we take termination into account.

A crucial difference between standard and probabilistic invariants is that the latter may *underestimate* the outcome. For example, an invariant for a standard program may be too specific and miss some of the initial states from which execution indeed would reach the postcondition. However a probabilistic invariant, not only could miss such states as well (i.e. assign the expectation 0 to them), but even in states where it is non-zero it may be far below

the actual expectation that could be achieved from that state. However in practice we want to find the pre-expectation of a loop precisely, not just some under-approximation. Indeed in all our examples throughout this chapter the invariants are chosen such that they establish the pre-expectation exactly.

Another approach to define invariants for probabilistic programs is due to Chakarov and Sankarnarayanan [14]. There an expression e over program variables is called an *inductive expectation invariant* (IEI) if

$$wp((\mathit{while}\,(G)\,\{P\})^n, e) \geq 0 \quad \forall n \in \mathbb{N} \tag{4.9}$$

where $(\mathit{while}\,(G)\,\{P\})^n$ is the n-fold unrolling of the loop and they assume $(\mathit{while}\,(G)\,\{P\})^0 = \mathit{skip}$. This definition tells us that the expectation of e has to be non-negative with respect to the initial distribution and with each further iteration of the loop the expectation of e remains non-negative with respect to the updated distribution. Their IEI expressions can be used to derive bounds on an unknown expected value. For example if $e = 2x + 1$ is an IEI, then from this we learn that

$$wp(\mathit{while}(G)\{P\}, 2x + 1) \geq 0 \ .$$

Since wp, which is nothing but an expectation, is linear, we can rewrite this as

$$2 \cdot wp(\mathit{while}(G)\{P\}, x) + 1 \geq 0 \tag{4.10}$$

$$wp(\mathit{while}(G)\{P\}, x) \geq -0.5 \tag{4.11}$$

Thus using e as a lower bound on the expectation of x can be derived. Interestingly, a method [14] that generates such an IEI e may do so without the need to find the expected value of x itself, which might be much harder.

An obvious question is how do invariants $\mathcal{I}$ in the sense of Definition 5 relate to IEI e in (4.9)? They are different in nature. An invariant $\mathcal{I}$ is defined to be a *quantity* that never decreases from one iteration of the loop to another. In contrast e is an expression such that the *predicate* in (4.9) is maintained for every iteration of the loop. However the value of e may fluctuate and is actually irrelevant (as long as its *expectation* is provably no less than 0). So in [14] they generate expressions e that in our framework satisfy

$$wp(\mathit{while}(G)\{P\}, e)(\eta_0) \geq 0$$

where η_0 is the initial state from which the loop starts its execution[1]. However if we choose to prove this inequality using a probabilistic invariant $\mathcal{I}$, it will not necessarily resemble the shape of e in any way.

In the next section we briefly survey approaches to computer aided invariant generation before we consider our software tool PRINSYS.

4.2. Feasible level of automation

Classical undecidability results in computer science show that we cannot expect to devise an algorithm to find an invariant for every given loop and every given postcondition (or post-expectation). Instead there are basically two orthogonal approaches for *computer aided* invariant generation. We say "computer aided" to emphasise that eventually it is a human user who actually finds the invariant but is assisted in various ways by a computer software. One approach, that we colloquially refer to as *abstract interpretation based*, starts with a representation of known facts about the initial state of the loop and updates these facts by iteratively unrolling the loop. To achieve convergence with a small number of unrollings a technique called *widening* is used, which approximates the further behaviour of the loop. The result of this generation technique is a set of invariants that a user inspects and amongst which hopefully finds a useful invariant for his proof goal. Due to widening, abstract interpretation based methods are doomed to be incomplete which means they cannot discover all facts (and thus all invariants) of a loop. Therefore it might be the case that the invariant that the user is actually searching for is "overlooked" by these methods. However in practice we see that they often do generate useful information, i.e. interesting loop invariants. For further details of this techniques and their implementations we refer to published results on standard [59] and probabilistic [14] programs.

Opposed to abstract interpretation based methods there exist so called *constraint based* methods. As we have seen from the previous section a loop $\mathit{while}\,(G)\,\{P\}$ and the post-expectation f yield constraints

$$[G] \cdot \mathcal{I} \leq \mathit{wlp}(P, \mathcal{I}) \tag{4.12}$$

$$[\neg G] \cdot \mathcal{I} \leq f \tag{4.13}$$

[1] In [14] they work with initial distributions, but we do not have a way to express an initial distribution. Luckily it does not really matter as for computing expectations they only need to know the average over the initial distribution which can be encoded in a state η_0.

where $\mathcal{I}$ is the invariant to be determined. In principle, $\mathcal{I}$ may be an arbitrary piece-wise defined function from variable valuation to real values. Without further information about $\mathcal{I}$ there are too many degrees of freedom for the choice of $\mathcal{I}$: How should $\mathcal{I}$ partition the state space and should be the values be described by a function that is linear or polynomial in the program variables? And if it is polynomial, what degree does it have for each variable? Thus we need to constrain the problem further. One way is to guess a candidate invariant precisely. Then the above inequalities may be checked and in case they are satisfied the guess was indeed successful. Of course, one would like to be less stringent and not require a correct guess of an invariant right away. So instead, the user may provide a so-called *template* for $\mathcal{I}$. This is an expectation, but some factors or additive constants may be parameterised. We have seen an example of a template earlier, cf. (4.5). For such templates we can automatically decide whether this template admits a solution and if so how those template parameters need to be instantiated. Further an important feature is that the constraint based approach is goal driven, i.e. the user has a post-expectation f in mind and needs to find a pre-expectation that satisfies (4.13). So he will shape his template accordingly and, when successful, is able to finish his proof. Finally this method is complete in a sense that if the user provides a template that has an invariant instantiation it is guaranteed to be included in the result of this approach. Of course the major drawback is that it is the user who has to provide a lot of information before the computer takes over. For standard programs, implementations exists, e.g. [35, 61, 19].

In the following section we discuss our implementation of a constraint based method due to [42] and evaluate its applicability on some examples. Its key benefit is that it checks (4.12) automatically and thereby saves the user the hardest calculations that are needed to establish pre- and post-expectations of probabilistic loops.

4.3. Prinsys

Prinsys implements an invariant generation method suggested by Katoen et al. [42]. It is available for download on `http://moves.rwth-aachen.de/research/tools/prinsys/` For a detailed explanation of Prinsys's methodology and implementation we refer to our paper in Appendix D. There we also revisit the program from Figure 4.1 and work out all steps which the tool performs to find the desired invariant. Here we only briefly mention the idea and then proceed to discuss other programs which we have studied.

4.3.1. Methodology

Prinsys requires the user to provide a program that contains a single, non-nested loop, such as the geometric distribution example from Figure 4.1. In fact all of the code before and after the loop is irrelevant to our tool so we may just as well focus on the loop only to which we refer as $\mathit{loop} = \mathit{while}\,(G)\,\{P\}$. Additionally, the user provides a so-called invariant template $\mathcal{T}$. For the loop above a suitable template might be chosen as

$$\mathcal{T} = [x \geq 0 \wedge \mathit{flip} = 0] \cdot (\alpha + x) + [x \geq 0 \wedge \mathit{flip} \neq 0] \cdot x \ ,$$

where α is the unknown template parameter the value of which we want to find. From *loop* and $\mathcal{T}$, Prinsys generates the necessary condition for invariance of $\mathcal{T}$

$$[G] \cdot \mathcal{T} \leq \mathit{wlp}(P, \mathcal{T}) \ . \tag{4.14}$$

Since P is free of loops $\mathit{wlp}(P, \mathcal{T})$ can be automatically evaluated and yields some expectation $\mathcal{T}$'. The inequality

$$[G] \cdot \mathcal{T} \leq \mathcal{T}'$$

is then translated to a first-order formula which is true if and only if the inequality holds. Subsequently we use Redlog [24] as an off-the-shelf solver to decide this formula. Since $\mathcal{T}$ contains free parameters the result of this decision procedure is not merely a yes-or-no answer but a constraint over those template parameters that characterises all invariant instances. For example, $\mathcal{T}$ above produces the following output

$$\alpha \cdot p - \alpha \leq 0 \wedge \alpha \cdot p + p - 1 \leq 0 \ .$$

This simplifies to

$$0 \leq \alpha \leq \frac{1-p}{p} \ ,$$

and in particular we have invariant instantiation

$$\mathcal{T}[\alpha/\frac{1-p}{p}] = [x \geq 0 \wedge \mathit{flip} = 0] \cdot (\frac{1-p}{p} + x) + [x \geq 0 \wedge \mathit{flip} \neq 0] \cdot x \ ,$$

which is a stronger expression then the fixed point in (4.4), but it suffices to show the desired pre-expectation. If a parameter-free template $\mathcal{T}$ is used, then Prinsys will simply report whether the inequality (4.14) is satisfied or not. For such invariance checks, we recently have added an option to use Z3 [21] as the back end. As there are no parameters to take care of, an SMT solving technique as implemented by Z3 suffices to decide the inequality (4.14). The

benefit is that SMT solving may outperform the quantifier elimination procedure of REDLOG. Furthermore in case the parameter-free $\mathcal{T}$ is not invariant, Z3 allows to extract a valuation of the variables that shows why inequality 4.14 does not hold. This may serve as valuable information to the user to refine his guess of what $\mathcal{T}$ should be.

4.3.2. Examples

Further examples in which we were able to successfully identify an invariant with the help of PRINSYS include the following.

Generating a biased coin from a fair one. In [42], Hurd's algorithm to generate a sample according to a biased coin flip using only fair coin flips has been analysed. This algorithm is given in terms of pGCL in Figure 4.2 With PRINSYS we have successfully verified that

$$\mathcal{I} = [x \geq 0 \land x - 1 \leq 0 \land (b - 1 = 0 \lor x = 0 \lor x - 1 = 0)] \cdot (x)$$

is invariant. Using $\mathcal{I}$ one may show that the probability to establish $x = 1$ is p and conversely $x = 0$ with probability $1 - p$. Thus the program generates a biased coin flip with a given bias p using a repetition of fair coin flips. For a detailed analysis we refer to our work in [29, p. 47ff.].

Generating a fair coin from a biased one. Here we consider an algorithm for the opposite problem. Using a coin with some arbitrary bias $0 < p < 1$, the algorithm in Figure 4.3 generates a sample according to a fair coin flip. The loop terminates when the biased coin was flipped twice and showed different outcomes. We remind the reader that this loop is an instance of an iid loop, which were introduced in Section 3.3.2, and can thus be elegantly analysed by introducing an *observe* statement. Here however we would like to show the applicability of invariants. Starting with the template

$$\mathcal{T} = [x = 0 \land y - 1 = 0] \cdot (\alpha) + [x - 1 = 0 \land y = 0] \cdot (\beta)$$

PRINSYS tells us that the template parameters α and β must satisfy

$$\alpha p^2 - \alpha p + \beta p^2 - \beta p \leq 0 \ .$$

This simplifies to $\alpha = -\beta$ and gives us an invariant that allows us to show that the probability of the outcomes $x = 0$ and $x = 1$ have probability $1/2$ each. For a detailed analysis we refer to our work in [33], cf. Appendix D.

```
1  x:= p;
2  b:= true;
3  while (b - 1 = 0) {
4    (b := false [0.5] b := true);
5    //if b is true
6    if (b - 1 = 0) {
7      x:= 2*x;
8      if (x - 1 >= 0) {
9        x:= x-1;
10     } else {
11       skip;
12     }
13   }
14   else if (x - 0.5 >= 0) {
15     x:= 1;
16   }
17   else {
18     x:= 0;
19   }
20 }
```

Figure 4.2.: Algorithm which generates a sample $x = 1$ with probability p and $x = 0$ with probability $1 - p$ by repeatedly flipping a fair coin.

```
1  x := 0;
2  y := 0;
3  while (x - y = 0) {
4    (x := 0 [p] x := 1);
5    (y := 0 [p] y := 1);
6  }
```

Figure 4.3.: Algorithm which generates $x = 0$ and $x = 1$ with equal probability by repeatedly flipping a coin with an arbitrary bias p.

```
1  x := 0;
2  n := 0;
3  while (n - M + 1 <= 0) {
4    (x := x + 1 [p] skip);
5    n := n + 1;
6  }
```

Figure 4.4.: Algorithm which generates a sample x distributed binomially with parameters p and M.

Binomial distribution. In this thesis we have seen many examples that in essence are a variant of the geometric distribution. Another important distribution underlying various interesting processes is the binomial distribution. Figure 4.4 gives a pGCL program that produces a sample x distributed according to the binomial distribution, i.e. the probability to terminate with $x = k$ is given by

$$\binom{M}{k} p^k (1-p)^{M-k} \ .$$

Using our expectation calculus we may show that the pre-expectation of x is pM, which agrees with the expectation of a binomial distribution with parameters p and M. For this we use the template

$$\mathcal{T} = [x \geq 0 \land x - n \leq 0 \land n - M \leq 0] \cdot (\alpha x + \beta n + \gamma) \ .$$

Interacting with Prinsys we arrive at an invariant instance which is given by

$$\mathcal{T}[\alpha/\frac{1}{M}, \beta/\frac{-p}{M}, \gamma/p] = [x \geq 0 \land x - n \leq 0 \land n - M \leq 0] \cdot (\frac{1}{M}x - \frac{p}{M}n + p) \ .$$

This invariant allows to show that the expectation of x/M is p. Since wp is linear (just as the mathematical expectation is) this result may be scaled by M to obtain the desired claim. For a detailed analysis we refer to our work in [29, p. 45ff.].

4.3.3. Problems

Technical. Essentially there are two technical problems with the Prinsys approach. A minor issue is that we are currently limited to the study of algorithms without nested loops. While nested loops maybe rewritten into an equivalent single loop [57], this transformation

seems impractical since part of the program's structure is lost and it will be harder to find an invariant for the new loop. Instead we believe our approach can be extended straightforwardly to support nested loops. For example consider the following program where G and H are boolean guards and P, Q and R are loop-free subprograms.

$$
\begin{aligned}
&\mathit{while}\,(G)\,\{\\
&\quad P;\\
&\quad \mathit{while}\,(H)\,\{\\
&\qquad R\\
&\quad \}\\
&\quad Q;\\
&\}
\end{aligned}
$$

Then we need two invariants: one invariant, say $\mathcal{I}$, for the outer loop and one, say $\mathcal{J}$, for the inner loop. Given some post-condition f, the conditions (4.12) and (4.13) generalise to

$$[G] \cdot \mathcal{I} \leq \mathit{wlp}(P, \mathcal{J}) \tag{4.15}$$

$$[H] \cdot \mathcal{J} \leq \mathit{wlp}(R, \mathcal{J}) \tag{4.16}$$

$$[\neg H] \cdot \mathcal{J} \leq \mathit{wlp}(Q, \mathcal{I}) \tag{4.17}$$

$$[\neg G] \cdot \mathcal{I} \leq f \ . \tag{4.18}$$

Condition (4.16) ensures that $\mathcal{J}$ is an invariant of the inner loop. Together with conditions (4.15) and (4.17) this further ensures that $\mathcal{I}$ is an invariant of the outer loop. Finally (4.18) establishes a lower bound on the given post-expectation f upon termination of the loops – in the same way as (4.13) did before. These constraints can be encoded in an first-order formula and solved. However this generalisation to nested loops has not been implemented as single-loop programs pose enough problems that need to be overcome before taking on further challenges.

The second technical problem is due to practical limitations of computer resources and insufficiencies in the current solvers. More precisely, the expressions that our tool handles grow very fast depending on the number of predicates in the template and the number of (conditional, probabilistic or non-deterministic) choices in the loop body. In the step where we translate (4.14) to a first-order formula, the size of the data structures blows

up exponentially. During our implementation we have learnt that it is crucial to simplify expressions at intermediate steps to maintain a manageable size. But even if Prinsys and the tools in the back end can successfully cope with the large data, the results that are returned to the user may be inconclusive. In the previous examples we have seen how well chosen templates lead to a small set of constraints that tell us what our invariant will be. However for other programs we may get results that are too large to be readable by a human user. A closer look however reveals that many simplifications, which seem obvious to a human user, are missed by the algorithms implemented in the solvers.

Conceptual. Conceptually we may identify two issues. The first is of a theoretical nature and that is we can only represent polynomial expectations. They do not suffice to express expectations of interesting random variables which are given by non-polynomial functions. For instance, our approach cannot be used to calculate the probability of an event such as $x = i$ for some fixed i, given the binomial or geometric programs (and in fact all algorithms that are based on those) because that would require reasoning with exponential functions and binomial coefficients, i.e. factorials. However an automated technique that can reason with such functions is not to be expected because currently it is not even known whether the theory of real numbers with exponential functions is decidable or not [49].

The greater conceptual issue is that the user has to provide a good template. This means essentially that if we know what the pre-expectation of the loop should look like, we can easily motivate the shape of a template and find an invariant instantiation with the help of Prinsys. However if a user has no clue what the sought pre-expectation of the loop is, it is not clear how to proceed. A constraint-based approach to invariant generation can only work with the input given by the user. A template that has no invariant instantiation will produce the answer *false*, but no hint will be given as to how to repair the template. The use of Z3 allowed us to find valuations that violate the invariance condition of Definition 5. However this does not give a direct hint at how to reshape the template. A possible remedy could be the use of patterns: If a software tool could automatically analyse the control flow structure of a loop and identify it to be an instance of an already known program – such as the geometric or binomial loop, which we have studied before – a good invariant candidate could be suggested to the user. However before the automation of such pattern detection algorithms can be approached we need to analyse a large number of case studies manually and identify the individual patterns that could be reused later.

5. Conclusion and future work

In this thesis we have studied the semantics and analysis techniques for probabilistic programs. We have given pGCL programs an operational semantics by means of reward Markov decision processes and linked expected rewards of such RMDPs to the weakest pre-expectation semantics. This correspondence result allows us to use either semantics to analyse or prove facts about pGCL programs. In a following step, we enriched pGCL by an observation statement which blocks all executions of a program that fail to satisfy a given predicate. Accordingly we extend both, the RMDP and wp, semantics to cater for this language extension. This gave rise to the notion of conditional minimal expected rewards over RMDPs and the conditional weakest pre-expectation over programs. Various examples have been studied to better understand the intricate behaviour of these conditional quantities as well as their practical application. As a result we were again able to establish a transfer theorem for the extended language, however restricting it to fully probabilistic programs only. We demonstrated by means of an example why a conditional expectation transformer cannot be given for non-deterministic probabilistic programs. The task to characterise the necessary and sufficient properties of programs that allow for a $\underline{cwp}$ style semantics have been left for future work.

The last chapter discussed our implementation of a constraint-based invariant generation technique for probabilistic programs. It saves the user a considerable amount of hard and error prone calculations and we have presented a set of programs that have been successfully analysed with our tool Prinsys. The major drawback in the application of Prinsys, turned out to be that it works well to check an educated guess of the user, but it fails to guide the user's search for an invariant in case the guess was not successful. We believe this may be remedied in the future by developing patterns for invariants. However, a larger set of case studies is needed to evaluate the applicability of such patterns. In the introduction we briefly mentioned that new abstraction techniques are evolving in the probabilistic model checking community which target infinite or parametric systems. Since our programs induce infinite, parametric Markov chains or Markov decision processes we hope to benefit from these recent

developments. Possibly they could be applied to learn invariants for interesting subclasses of pGCL programs.

Bibliography

[1] Monty hall problem. `http://en.wikipedia.org/wiki/Monty_Hall_problem`. Accessed: 15.12.2014.

[2] Miguel E. Andrés and Peter van Rossum. Conditional probabilities over probabilistic and nondeterministic systems. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 2008.

[3] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[4] Christel Baier, Joachim Klein, Sascha Klüppelholz, and Steffen Märcker. Computing conditional probabilities in markovian models efficiently. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 515–530. Springer, 2014.

[5] Damián Barsotti and Nicolás Wolovick. Automatic probabilistic program verification through random variable abstraction. In Alessandra Di Pierro and Gethin Norman, editors, *Proceedings Eighth Workshop on Quantitative Aspects of Programming Languages, QAPL 2010, Paphos, Cyprus, 27-28th March 2010.*, volume 28 of *EPTCS*, pages 34–47, 2010.

[6] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. Probabilistic relational reasoning for differential privacy. *ACM Trans. Program. Lang. Syst.*, 35(3):9, 2013.

[7] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[8] Henrik C. Bohnenkamp, Peter van der Stok, Holger Hermanns, and Frits W. Vaandrager. Cost-optimization of the ipv4 zeroconf protocol. In *2003 International Conference on Dependable Systems and Networks (DSN 2003), 22-25 June 2003, San Francisco, CA, USA, Proceedings*, pages 531–540. IEEE Computer Society, 2003.

[9] Tomás Brázdil, Václav Brozek, Kousha Etessami, Antonín Kucera, and Dominik Wojtczak. One-counter markov decision processes. In Moses Charikar, editor, *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 863–874. SIAM, 2010.

[10] Michael J. Brennan and Yihong Xia. Stock price volatility and equity premium. *Journal of Monetary Economics*, 47(2):249 – 283, 2001.

[11] Hauke Busch, Werner Sandmann, and Verena Wolf. A numerical aggregation algorithm for the enzyme-catalyzed substrate conversion. In Corrado Priami, editor, *Computational Methods in Systems Biology, International Conference, CMSB 2006, Trento, Italy, October 18-19, 2006, Proceedings*, volume 4210 of *Lecture Notes in Computer Science*, pages 298–311. Springer, 2006.

[12] Orieta Celiku and Annabelle McIver. Cost-based analysis of probabilistic programs mechanised in HOL. *Nord. J. Comput.*, 11(2):102–128, 2004.

[13] Aleksandar Chakarov and Sriram Sankaranarayanan. Probabilistic program analysis with martingales. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 511–526. Springer, 2013.

[14] Aleksandar Chakarov and Sriram Sankaranarayanan. Expectation invariants for probabilistic program loops as fixed points. In Markus Müller-Olm and Helmut Seidl, editors, *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings*, volume 8723 of *Lecture Notes in Computer Science*, pages 85–100. Springer, 2014.

[15] S. Cheshire, B. Aboba, and E. Guttman. Dynamic configuration of ipv4 link-local addresses, 2005.

[16] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137 – 167, 1959.

[17] Guillaume Claret, Sriram K. Rajamani, Aditya V. Nori, Andrew D. Gordon, and Johannes Borgström. Bayesian inference using data flow analysis. In Bertrand Meyer, Luciano Baresi, and Mira Mezini, editors, *ESEC/SIGSOFT FSE*, pages 92–102. ACM, 2013.

[18] David Cock. Verifying probabilistic correctness in isabelle with pgcl. In Franck Cassez, Ralf Huuck, Gerwin Klein, and Bastian Schlich, editors, *Proceedings Seventh Conference on Systems Software Verification, SSV 2012, Sydney, Australia, 28-30 November 2012.*, volume 102 of *EPTCS*, pages 167–178, 2012.

[19] Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. Linear invariant generation using non-linear constraint solving. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 2003.

[20] Adnan Darwiche. Bayesian networks. *Commun. ACM*, 53(12):80–90, 2010.

[21] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[22] Christian Dehnert, Joost-Pieter Katoen, and David Parker. Smt-based bisimulation minimisation of markov models. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*, volume 7737 of *Lecture Notes in Computer Science*, pages 28–47. Springer, 2013.

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

[24] Andreas Dolzmann and Thomas Sturm. Redlog computer algebra meets computer logic. *ACM SIGSAM Bulletin*, 31:2–9, 1996.

[25] William Feller. *An Introduction to Probability Theory and its Applications*, volume 1 of *Wiley mathematical statistics series*. Wiley, 1966.

[26] Robert W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.

[27] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. Probabilistic programming. In James D. Herbsleb and Matthew B. Dwyer, editors, *Proceedings of the on Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014*, pages 167–181. ACM, 2014.

[28] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, New York, NY, USA, 1993.

[29] Friedrich Gretz. Invariant Generation for Linear Probabilistic Programs. Master's thesis, RWTH Aachen, 2010. http://moves.rwth-aachen.de/people/fgretz/.

[30] Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Annabelle McIver, and Federico Olmedo. Conditioning in probabilistic programming. *To appear in MFPS 2015*.

[31] Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Annabelle McIver, and Federico Olmedo. Conditioning in probabilistic programming. *CoRR*, abs/1504.00198, 2015.

[32] Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. Operational versus weakest precondition semantics for the probabilistic guarded command language. In *Ninth International Conference on Quantitative Evaluation of Systems, QEST 2012, London, United Kingdom, September 17-20, 2012*, pages 168–177. IEEE Computer Society, 2012.

[33] Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. Prinsys - on a quest for probabilistic loop invariants. In Kaustubh R. Joshi, Markus Siegle, Mariëlle Stoelinga, and Pedro R. D'Argenio, editors, *Quantitative Evaluation of Systems - 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*, volume 8054 of *Lecture Notes in Computer Science*, pages 193–208. Springer, 2013.

[34] Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. Operational versus weakest pre-expectation semantics for the probabilistic guarded command language. *Perform. Eval.*, 73:110–132, 2014.

[35] Ashutosh Gupta and Andrey Rybalchenko. Invgen: An efficient invariant generator. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 634–640. Springer, 2009.

[36] Ernst Moritz Hahn, Holger Hermanns, Björn Wachter, and Lijun Zhang. PARAM: A model checker for parametric markov models. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 660–664. Springer, 2010.

[37] Holger Hermanns, Björn Wachter, and Lijun Zhang. Probabilistic CEGAR. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*, pages 162–175. Springer, 2008.

[38] Matthew D. Hoffman and Andrew Gelman. The no-U-turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, 15(Apr):1593–1623, 2014.

[39] Chung-Kil Hur, Aditya V. Nori, Sriram K. Rajamani, and Selva Samuel. Slicing probabilistic programs. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 16. ACM, 2014.

[40] Joe Hurd, Annabelle McIver, and Carroll Morgan. Probabilistic guarded commands mechanized in *HOL*. *Theor. Comput. Sci.*, 346(1):96–112, 2005.

[41] Nils Jansen, Florian Corzilius, Matthias Volk, Ralf Wimmer, Erika Ábrahám, Joost-Pieter Katoen, and Bernd Becker. Accelerating parametric probabilistic verification. In Gethin Norman and William H. Sanders, editors, *Quantitative Evaluation of Systems - 11th International Conference, QEST 2014, Florence, Italy, September 8-10, 2014. Proceedings*, volume 8657 of *Lecture Notes in Computer Science*, pages 404–420. Springer, 2014.

[42] Joost-Pieter Katoen, Annabelle McIver, Larissa Meinicke, and Carroll C. Morgan. Linear-invariant generation for probabilistic programs: - automated support for proof-based methods. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*, volume 6337 of *Lecture Notes in Computer Science*, pages 390–406. Springer, 2010.

[43] M. Kattenbelt, M. Kwiatkowska, G. Norman, and D. Parker. A game-based abstraction-refinement framework for Markov decision processes. *Formal Methods in System Design*, 36(3):246–280, 2010.

[44] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer, 2011.

[45] Jean-Louis Lassez, V. L. Nguyen, and Liz Sonenberg. Fixed Point Theorems and Semantics: A Folk Tale. *Inf. Process. Lett.*, 14(3):112–116, 1982.

[46] Johan J. Lukkien. Operational semantics and generalized weakest preconditions. *Sci. Comput. Program.*, 22(1-2):137–155, 1994.

[47] Jérémie Lumbroso. Optimal discrete uniform generation from coin flips, and applications. *CoRR*, abs/1304.1916, 2013.

[48] David Lunn, David Spiegelhalter, Andrew Thomas, and Nicky Best. The bugs project: Evolution, critique and future directions. *Statistics in Medicine*, 28(25):3049–3067, 2009.

[49] A. Macintyre and A.J. Wilkie. On the decidability of the real exponential field. In *Kreiseliana: About and around Georg Kreisel*, pages 441–467. A.K. Peters, 1996.

[50] Annabelle McIver and Carroll Morgan. *Abstraction, Refinement And Proof For Probabilistic Systems (Monographs in Computer Science)*. SpringerVerlag, 2004.

[51] Carroll Morgan and Annabelle McIver. Cost analysis of games, using program logic. In *Proceedings of the Eighth Asia-Pacific on Software Engineering Conference*, APSEC '01, pages 351–, Washington, DC, USA, 2001. IEEE Computer Society.

[52] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[53] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[54] Aditya V. Nori, Chung-Kil Hur, Sriram K. Rajamani, and Selva Samuel. R2: an efficient MCMC sampler for probabilistic programs. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pages 2476–2482. AAAI Press, 2014.

[55] Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[56] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley and Sons, 1994.

[57] T. M. Rabehaja and Jeff W. Sanders. Refinement algebra with explicit probabilism. In Wei-Ngan Chin and Shengchao Qin, editors, *TASE 2009, Third IEEE International Symposium on Theoretical Aspects of Software Engineering, 29-31 July 2009, Tianjin, China*, pages 63–70. IEEE Computer Society, 2009.

[58] Michael K. Reiter and Aviel D. Rubin. Crowds: Anonymity for web transactions. *ACM Trans. Inf. Syst. Secur.*, 1(1):66–92, 1998.

[59] Enric Rodríguez-Carbonell and Deepak Kapur. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Sci. Comput. Program.*, 64(1):54–75, 2007.

[60] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[61] Sriram Sankaranarayanan, Henny Sipma, and Zohar Manna. Non-linear loop invariant generation using gröbner bases. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 318–329. ACM, 2004.

[62] Henk C. Tijms. *Understanding Probability: Chance Rules in Everyday Life*. Cambridge University Press, 2004.

[63] George H. Weiss. Random walks and their applications. *American Scientist*, 71(1):pp. 65–71, 1983.

[64] David Williams. *Probability With Martingales*. Cambridge University Press, 1991.

[65] Mingsheng Ying, Nengkun Yu, Yuan Feng, and Runyao Duan. Verification of quantum programs. *Sci. Comput. Program.*, 78(9):1679–1700, 2013.

Appendix

A. Operational versus Weakest Precondition Semantics for the Probabilistic Guarded Command Language

Together with our paper in Appendix B, this paper forms the basis of Chapter 2.

Operational versus Weakest Precondition Semantics for the Probabilistic Guarded Command Language

Friedrich Gretz
RWTH Aachen University
Aachen, Germany
Email: fgretz@cs.rwth-aachen.de

Joost-Pieter Katoen
RWTH Aachen University
Aachen, Germany
Email: katoen@cs.rwth-aachen.de

Annabelle McIver
Macquarie University
Sydney, Australia
Email: annabelle.mciver@mq.edu.au

***Abstract*—This paper proposes a simple operational semantics of pGCL, Dijkstra's guarded command language extended with probabilistic choice, and relates this to pGCL's *wp*-semantics by McIver and Morgan. Parameterised Markov decision processes whose state rewards depend on the post-expectation at hand are used as operational model. We show that the *weakest pre-expectation* of a pGCL-program w.r.t. a post-expectation corresponds to the *expected cumulative reward* to reach a terminal state in the parameterised MDP associated to the program. In a similar way, we show a correspondence between weakest *liberal* pre-expectations and *liberal* expected cumulative rewards.**

I. Introduction

Formal semantics of programming languages has been the subject of intense research in computer science for several decades. Several approaches have been developed for the description of program semantics. Structured operational semantics defines the meaning of a program by means of an abstract machine where states correspond to program configurations (typically consisting of a program counter and a variable valuation) and transitions model the evolution of a program by executing statements. Program executions are then the possible runs of the abstract machine. Denotational semantics maps a program onto a mathematical object that describes for instance its input-output behaviour. Finally, axiomatic semantics provides the program semantics in an indirect manner by describing its properties. A prominent example of the latter are Hoare triples in which annotations, written in predicate logic, are associated to control points of the program.

The semantics of Dijkstra's seminal guarded command language [2] from the seventies is given in terms of weakest preconditions. It is in fact a predicate transformer semantics that is a total function between two predicates on the state of a program. The predicate transformer $E = \mathit{wp}(P, F)$ for program P and postcondition F yields the weakest precondition E on the initial state of P ensuring that the execution of P terminates in a final state satisfying F. There is a direct relation with axiomatic semantics: the Hoare triple $\{E\}P\{F\}$ holds for total correctness if and ony if $E \Rightarrow \mathit{wp}(P, F)$. The weakest *liberal* precondition $\mathit{wlp}(P, F)$ yields the weakest precondition for which P either does not terminate or establishes F. It does not ensure termination and corresponds to Hoare logic in partial correctness. Although providing an operational semantics for the guarded command language is rather straightforward, it lasted until the early nineties until Lukkien [8], [9] provided a formal connection between the predicate transformer semantics and the notion of a computation.

This research has been funded by the DFG Research Training Group 1298 (AlgoSyn), the EU FP7-Project CARP (Correct and Efficient Accelerator Programming), and the Australian Research Council DP1092464.

Qualitative annotations in predicate calculus are often insufficient for probabilistic programs as they cannot express quantities such as expectations over program variables. To that end, McIver and Morgan [10] generalised the methods of Dijkstra and Hoare to probabilistic programs by making the annotations real-valued expressions —referred to as expectations— in the program variables. Expectations are the quantitative analogue of predicates. This yields an expectation transformer semantics of the probabilistic guarded command language (pGCL, for short), an extension of Dijkstra's language with a probabilistic choice operator. An expectation transformer is a total function between two expectations on the state of a program. The expectation transformer $\mathit{wp}(P, f)$ for pGCL-program P and post-expectation f over final states yields the least expected value e on P's initial state ensuring that P's execution terminates with a value f. The annotation $\{e\}P\{f\}$ holds for total correctness if and only if $e \leq \mathit{wp}(P, f)$, where $\leq$ is to be interpreted in a point-wise manner. The weakest *liberal* pre-expectation $\mathit{wlp}(P, f)$ yields the least expectation for which P either does not terminate or establishes f. It does not ensure termination and corresponds to partial correctness.

This paper provides a simple operational semantics of pGCL using parametric Markov decision processes (pMDPs), a slight variant of MDPs in which probabilities may be parameterised [3]. Our main contribution in this paper is a formal connection between the *wp*- and *wlp*-semantics of pGCL by McIver and Morgan and the operational semantics. This provides a clean and insightful relationship between the abstract expectation transformer semantics that has been proven useful for formal reasoning about probabilistic programs, and the notion of a computation in terms of the operational model, a pMDP. In order to establish this connection we equip pMDPs with state rewards that depend on the post-expectation at hand. Intuitively speaking, we decorate a terminal state in the operational model of a program with a reward that corresponds to the value of the post-expectation. All other states are assigned reward zero. We then show that the *weakest pre-*

expectation of a pGCL-program P w.r.t. a post-expectation corresponds to the *expected cumulative reward* to reach a terminal state in the pMDP associated to P. In a similar way, we show that weakest *liberal* pre-expectations correspond to *liberal* expected cumulative rewards. The proofs are by induction on the structure of our probabilistic programs. This paper thus yields a computational view on the expectation transformer semantics of probabilistic programs using first principles of Markov decision processes.

A. Structure of this paper.

The rest of the paper is divided as follows. In Sect. II we introduce the probabilistic programming language pGCL. Parametric Markov decision processes with rewards are introduced in Sect. III. Section IV recaps the denotational semantics of pGCL [10] and introduces operational semantics for this language. Then the main result is established, namely that the two semantics are equivalent. Finally, Sect. V provides an example of reasoning over pGCL programs.

II. Probabilistic Programs

Our input language pGCL [10] is an extension of Dijkstra's guarded command language [2]. Besides a non-deterministic choice operator, denoted $[]$, and a conditional choice, it incorporates a probabilistic choice operator, denoted $[p]$, where p is a real parameter (or constant) whose values lies in the range $[0,1]$. pGCL is a language to model sequential programs containing randomized assignments. For instance, the assignment $(x := 2 \cdot x \ [0.75] \ x := x{+}1)$ doubles the value of x with probability $\frac{3}{4}$ and increments it by one with the remaining probability $\frac{1}{4}$.

Definition 1. (Syntax of pGCL) Let P, P_1, P_2 be pGCL-programs, p a probability variable, x a program variable, E an expression, and G a Boolean expression. The syntax of a pGCL program P adheres to the following grammar:

$$\mathsf{skip} \mid \mathsf{abort} \mid x := E \mid P_1; P_2 \mid P_1 \,[]\, P_2 \mid P_1 \,[p]\, P_2 \mid$$
$$\mathsf{if}(G)\{P_1\}\,\mathsf{else}\,\{P_2\} \mid \mathsf{while}(G)\{P\}.$$

■

skip stands for the empty statement, abort for abortion, and $x := E$ for an assignment of the value of expression E (over the program variables) to variable x. The sequentially composed program $P_1; P_2$ behaves like P_1 and subsequently like P_2 on the successful termination of P_1. The statement $P_1 \,[]\, P_2$ denotes a non-deterministic choice; it behaves like either P_1 or P_2. The statement $P_1 \,[p]\, P_2$ denotes a probabilistic choice. It behaves like P_1 with probability p and like P_2 with probability $1{-}p$. The remaining two statements are standard: conditional choice and while-loop. Throughout this paper, we assume that pGCL-programs are well-typed. This entails that for assignments of the form $x := E$ we assume that x and E are of the same type. In a similar way, we assume G to denote a Boolean expression and variable p to denote a probability in the real interval $[0,1]$.

Listing 1. The duelling cowboys, cf. [10].

```
1  int cowboyDuel(a, b) {
       // 0 < a, b < 1
2    (t := A [] t := B);
         // decide who starts
3    c := 1;
4    while (c = 1) {
5      if (t = A) {
6        (c := 0 [a] t := B);
7      } else {
8        (c := 0 [b] t := A);
9      }
10   }
11   return t; // the survivor
12 }
```

Example 2. (Duelling cowboys [10]) The pGCL program in Lst. 1 models the following situation: There are two cowboys, A and B, who are fighting a classical duel. They take turns, shooting at each other until one of them is hit. If A (resp. B) shoots then he hits B (resp. A) with probability a (resp. b). We assume that either cowboy A or B is allowed to start; the choice of who will start is resolved nondeterministically. Variable t keeps track of the turns, while c determines whether the duel continues or someone is hit. Note that it is a distinctive feature that we do not have to specify exact probabilities and instead allow arbitrary parameters. ■

III. Markov decision processes

This section introduces the basics of MDPs enriched with state rewards. We first recall the definition of an MDP with a countable state space and define elementary notions such as paths and policies. Subsequently, we introduce reward-MDPs in which states are equipped with an integer reward and focus on reachability objectives, in particular (liberal) expected cumulative rewards to reach a set of states. These measures are later shown to closely correspond to weakest pre-condition semantics of pGCL-programs.

A. Preliminaries

Let X be a finite set of real-valued variables, and $V(X)$ denote the set of expressions over X.

Definition 3. (Parametric distribution) A parametric distribution μ is a function that maps states to probabilities. The probabilities are real values in $[0,1]$ or expressions over X:

$$\mu : S \to V(X) \quad \text{with} \quad \sum_{s \in S} \mu(s) = 1.$$

■

Example 4. (Parametric distribution) Consider $S = \{s_0, s_1, s_2\}$. Then a parametric distribution μ might be: $\mu(s_0) = p$, $\mu(s_1) = 1 - p$ and $\mu(s_2) = 0$ where $p \in [0,1]$. Just note that p is a symbol and not an explicit number like 0.4. ■

Definition 5. (Markov decision process) An MDP $\mathcal{M}$ is a tuple $(S, S_0, \longrightarrow)$ where S is a countable set of states with

initial state-set $S_0 \subseteq S$ where $S_0 \neq \emptyset$, and $\longrightarrow \subseteq S \times \mathit{Dist}(S)$ is a transition relation from a state to a set of distributions over states. ■

Let $s \to \mu$ denote $(s, \mu) \in \longrightarrow$ and $s \to t$ denote $s \to \mu$ with $\mu(t) = 1$. We define $\mathit{Dist}(s) = \{\, \mu \mid s \to \mu \,\}$ to be the set of enabled distributions in state s. The intuitive operational behavior of an MDP $\mathcal{M}$ is as follows. First, non-deterministically select some initial state $s_0 \in S_0$. In state s with $\mathit{Dist}(s) \neq \emptyset$, non-deterministically select $\mu \in \mathit{Dist}(s)$. The next state t is randomly chosen with probability $\mu(t)$. If $\mathit{Dist}(t) = \emptyset$, exit; otherwise continue as for state s.

Our MDPs are called parametric because the underlying distributions are parametric.

Remark 6. (Finite support) In the context of this paper we are only interested in finitely branching Markov decision processes. This means that every state has finitely many successor states. Therefore $|Dist(s)| < \infty$ for all $s \in S$ and all distributions are assumed to have finite support. ■

A path of MDP $\mathcal{M}$ is a maximal alternating sequence $\pi = s_0 \xrightarrow{\mu_0} s_1 \xrightarrow{\mu_1} \ldots$ such that $\mu_i(s_{i+1}) > 0$ for all $i \geq 0$. As any path is a maximal sequence, it is either infinite or ends in state s with $\mathit{Dist}(s) = \emptyset$. Reasoning about probabilities on sets of paths of an MDP relies on the resolution of non-determinism. This resolution is performed by a policy[1] that selects one of the enabled distributions in a state. Whereas in general a policy may base its decision in state s on the path fragment from $s_0 \in S_0$ to s, it suffices in the context of this paper to consider positional policies.

Definition 7. (Positional policy) Function $\mathfrak{P} : S \to \mathit{Dist}(S)$ is a positional policy for MDP $\mathcal{M} = (S, S_0, \longrightarrow)$ with $\mathfrak{P}(s) \in \mathit{Dist}(s)$ for all $s \in S$. ■

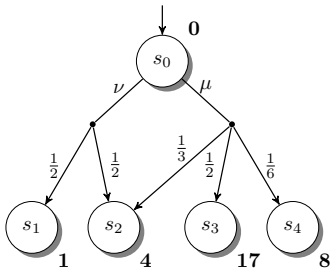
A positional policy thus selects an enabled distribution based on the current state s only. As in the rest of this paper, we only consider positional policies, we call them simply policies. The path fragment leading to s does not play any role. The path $\pi = s_0 \xrightarrow{\mu_0} s_1 \xrightarrow{\mu_1} \ldots$ is called a $\mathfrak{P}$-path if it is induced by the policy $\mathfrak{P}$, that is, $\mathfrak{P}(s_i) = \mu_i$ for all $i \geq 0$. Let $\mathit{Paths}^{\mathfrak{P}}(s)$ denote the set of $\mathfrak{P}$-paths starting from state s. A policy of an MDP $\mathcal{M}$ induces a Markov chain $\mathcal{M}^{\mathfrak{P}}$ with the same state space as $\mathcal{M}$ and transition probabilities $\mathfrak{P}(s)(t)$ for states s and t. For finite path fragment $\widehat{\pi} = s_0 \xrightarrow{\mu_0} s_1 \xrightarrow{\mu_1} \ldots \xrightarrow{\mu_{k-1}} s_k$ of a $\mathfrak{P}$-path, let $\mathbf{P}(\widehat{\pi})$ denote the probability of $\widehat{\pi}$ which is defined by $\mu_0(s_1) \times \ldots \times \mu_{k-1}(s_k) = \prod_{i=1}^{k} \mu_{i-1}(s_i)$. Let $\mathit{Pr}^{\mathfrak{P}}(\Pi)$ denote the probability of the set of paths Π under policy $\mathfrak{P}$. This probability measure is defined in the standard way using a cylinder set construction on the induced Markov chain $\mathcal{M}^{\mathfrak{P}}$ [1].

To compare our operational semantics of pGCL with its *wp*- and *wlp*-semantics, we use rewards (or, dually costs).

Definition 8. (MDP with rewards) An MDP with *rewards* (also called reward-MDP, or shortly RMDP) is a pair $(\mathcal{M}, r)$ with $\mathcal{M}$ an MDP with state space S and $r : S \to \mathbb{N}$ a function assigning a natural reward to each state. ■

Intuitively, the reward $r(s)$ stands for the reward earned on entering state s. The cumulative reward of a finite path fragment $s_0 \xrightarrow{\mu_0} s_1 \xrightarrow{\mu_1} \ldots s_k$ is the sum of the rewards in all states that have been visited, i.e., $r(s_0) + \ldots + r(s_k)$ provided $k > 0$, and 0 otherwise.

Example 9. (RMDP, cumulative reward of a path)



Assume a policy $\mathfrak{P}$ with $\mathfrak{P}(s_0) = \mu$. Then $\pi = s_0 \xrightarrow{\mu} s_3$ is a possible path that is taken with probability 0.5 and has cumulative reward $r(\pi) = 17$. ■

B. Reachability objectives

We are interested in reachability events in reward-MDPs. Let $T \subseteq S$ be a set of target states. The event $\Diamond T$ stands for the reachability of some state in T, i.e., $\Diamond T$ is the set of paths in MDP $\mathcal{M}$ that hit some state $s \in T$. Formally $\Diamond T = \{\, \pi \in \mathit{Paths} \mid \exists i \geq 0. \pi[i] \in T \,\}$ where $\pi[i]$ denotes the i-th state visited allong π. We write $\pi \models \Diamond T$ whenever π belongs to $\Diamond T$. It follows by standard arguments that $\Diamond T$ is a measurable event. The cumulative cost for this event is defined as follows.

Definition 10. (Cumulative cost for reachability) Let $\pi = s_0 \xrightarrow{\mu_0} s_1 \xrightarrow{\mu_1} \ldots$ be a maximal path in reward-MDP $(\mathcal{M}, r)$ and $T \subseteq S$ a set of target states. If $\pi \models \Diamond T$, the *cumulative cost* along π before reaching T is defined by: $r_T(\pi) = r(s_0) + \ldots + r(s_k)$ where $s_i \notin T$ for all $i < k$ and $s_k \in T$. If $\pi \not\models \Diamond T$, then $r_T(\pi) = 0$. ■

Stated in words, the cumulative costs for a path π to reach T is the cumulative cost of the minimal prefix of π satisfying $\Diamond T$. In case π never reaches a state in T, the cumulative cost is defined to be zero. We denote by $\mathit{Paths}(s, \Diamond T)$ the set of paths starting in s that eventually reach T.

Definition 11. (Expected reward for reachability) Let $(\mathcal{M}, r)$ be an RMDP with state space S and $T \subseteq S$ and $s \in S$. The *minimal expected reward* until reaching $T \subseteq S$ from $s \in S$, denoted $\mathit{ExpRew}^{(\mathcal{M},r)}(s \models \Diamond T)$, is defined by:

$$\min_{\mathfrak{P}} \sum_{c=0}^{\infty} c \cdot \mathit{Pr}^{\mathfrak{P}}\{\, \pi \in \mathit{Paths}^{\mathfrak{P}}(s, \Diamond T) \mid r_T(\pi) = c \,\} \ .$$

The minimal *liberal* expected reward until reaching T from s,

[1] Also called scheduler, strategy or adversary.

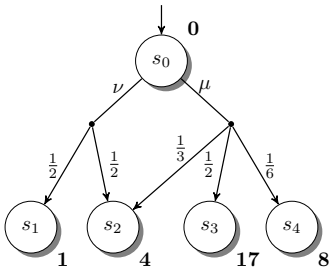
denoted $\mathit{LExpRew}^{(\mathcal{M},r)}(s \models \Diamond T)$, is defined by:

$$\min_{\mathfrak{P}} \left\{ \sum_{c=0}^{\infty} c \cdot \mathit{Pr}^{\mathfrak{P}}\{\, \pi \in \mathit{Paths}^{\mathfrak{P}}(s, \Diamond T) \mid r_T(\pi) = c \,\} + \mathit{Pr}^{\mathfrak{P}}(s \not\models \Diamond T) \right\} .$$

We leave away the superscript when the underlying model is clear from context. ■

The expected reward in s to reach some state in T is the expected cumulative cost over all paths (reaching T) induced under a demonic policy. The motivation to consider a demonic and not an angelic policy becomes clear further on in this paper, and has a direct relation with the notion of *weakest* pre-expectation. Note that in case T is not reachable from s under a demonic policy, $\mathit{ExpRew}(s \models \Diamond T) = 0$. $\mathit{LExpRew}(s \models \Diamond T)$ is the expected reward to reach T or never reach it from s. In case there is no policy under which T can be reached from s, we have that $\mathit{LExpRew}(s \models \Diamond T) = 1$. Note that *ExpRew* and *LExpRew* coincide if T is reached with probability 1. For finite MDPs without parameters, expected and liberal expected rewards for reachability objectives can be obtained by solving a linear programming problem. A detailed description is outside the scope of this paper; its analogue for Markov chains is fully described in [1, Ch. 10.5].

Example 12. (Expected rewards)



Let $T = \{s_2, s_3\}$. Then $\mathit{ExpRew}(s_0 \models \Diamond T) = \min\{2, \frac{59}{6}\} = 2$. And $\mathit{LExpRew}(s_0 \models \Diamond T) = \min\{2.5, 10\} = 2.5$. ■

IV. pGCL SEMANTICS

This section describes an expectation transformer semantics of pGCL, as well as an operational semantics using MDPs. The main result of this section is a formal connection between these two semantics.

A. Denotational Semantics

When probabilistic programs are executed they determine a probability distribution over final values of program variables. For instance, on termination of

$$(x := 1 \ [0.75] \ x := 2);$$

the final value of x is 1 with probability $\frac{3}{4}$ or 2 with probability $\frac{1}{4}$. An alternative way to characterise that probabilistic behaviour is to consider the expected values over random variables with respect to that distribution. For example, to determine the probability that x is set to 1, we can compute the expected value of the random variable "x is 1" which is $\frac{3}{4} \cdot 1 + \frac{1}{4} \cdot 0 = \frac{3}{4}$. Similarly, to determine the average value of x, we compute the expected value of the random variable "x" which is $\frac{3}{4} \cdot 1 + \frac{1}{4} \cdot 2 = \frac{5}{4}$.

More generally, rather than a distribution-centred approach, we take an "expectation transformer" [10] approach. We annotate probabilistic programs with *expectations*, cf. [10]. Expectations are functions which map program states to real values. They are the quantitative analogue to Hoare's predicates for non-probabilistic programs. An expectation transformer is a total function between two expectations on the state of a program. The transformer $\mathit{wp}(P, f)$ for program P and post-expectation f yields the least expected value e on P's initial state ensuring that P's execution terminates with a value f. Annotation $\{e\}\, P\, \{f\}$ holds for total correctness if and only if $e \ \leq \ \mathit{wp}(P, f)$ where $\leq$ is to be interpreted in a point-wise manner. Intuitively, implication between predicates is generalised to pointwise inequality between expectations. For convenience we use square brackets to link boolean truth values to numbers and by convention $[\text{true}] = 1$ and $[\text{false}] = 0$.

Definition 13. (*wp*-semantics of pGCL) Let P and Q be pGCL-programs, f a post-expectation, x a program variable, E an expression, and G a Boolean expression. The *wp*-semantics of a program is defined by structural induction follows:

- $\mathit{wp}(\mathsf{skip}, f) = f$
- $\mathit{wp}(\mathsf{abort}, f) = 0$
- $\mathit{wp}(x := E, f) = f[x := E]$
- $\mathit{wp}(P; Q, f) = \mathit{wp}(P, \mathit{wp}(Q, f))$
- $\mathit{wp}(\mathsf{if}(G)\{P\}\mathsf{else}\{Q\}, f) = [G] \cdot \mathit{wp}(P, f) + [\neg G] \cdot \mathit{wp}(Q, f)$
- $\mathit{wp}(P \,[]\, Q, f) = \min(\mathit{wp}(P, f), \mathit{wp}(Q, f))$
- $\mathit{wp}(P \,[p]\, Q, f) = p \cdot \mathit{wp}(P, f) + (1{-}p) \cdot \mathit{wp}(Q, f)$
- $\mathit{wp}(\mathsf{while}(G)\{P\}, f) = \mu X.\, ([G]{\cdot}\mathit{wp}(P, X){+}[\neg G]{\cdot} f)$

Here μ is the least fixed point operator w.r.t. the ordering $\leq$ on expectations. ■

If program P does not contain a probabilistic choice, then this *wp* is isomorphic to Dijkstra's *wp* [10]. A weakest liberal pre-expectation $\mathit{wlp}(P, f)$ yields the least expectation for which P either does not terminate or establishes f.

Definition 14. (*wlp*-semantics of pGCL) *wlp*-semantics differs from *wp*-semantics only for while and abort:

- $\mathit{wlp}(\mathsf{abort}, f) = 1$
- $\mathit{wlp}(\mathsf{while}(G)\{P\}, f) = \nu X.\, ([G]{\cdot}\mathit{wlp}(P, X){+}[\neg G]{\cdot} f)$

Here ν is the greatest fixed point operator w.r.t. the ordering $\leq$ on expectations. ■

So the difference between *wp* and *wlp* is lies in the handling non-termination. As for *ExpRew* and *LExpRew* the expectation transformers *wp* and *wlp* coincide for programs that terminate with probability 1.

Example 15. (Application of *wlp*-semantics) Consider again the duelling cowboys example. Assume we are given

the post-expectation:

$$f = [t = A \wedge c = 0] + [t = A \wedge c = 1] \cdot \frac{a}{a+b-ab} + [t = B \wedge c = 1] \cdot \frac{(1-b)a}{a+b-ab} \ .$$

Let us compute the weakest liberal pre-expectation of the loop body from Lst. 1 w.r.t. the post-expectation f. This yields:

$$\begin{aligned}
&\mathit{wlp}(\mathsf{if}(t = A)\{(c := 0\,[a]\,t := B);\}\\
&\qquad \mathsf{else}\{(c := 0\,[b]\,t := A);\}, f)\\
&= [t = A] \cdot \mathit{wlp}((c := 0\,[a]\,t := B), f)\\
&\quad + [t \neq A] \cdot \mathit{wlp}((c := 0\,[b]\,t := A), f)\\
&= [t = A] \cdot (a \cdot \mathit{wlp}((c := 0), f) + (1-a) \cdot \mathit{wlp}(t := B, f))\\
&\quad + [t \neq A] \cdot (b \cdot \mathit{wlp}((c := 0, f) + (1-b) \cdot \mathit{wlp}(t := A, f))\\
&= [t = A \wedge c \neq 1] \cdot a + [t = A \wedge c = 1] \cdot \frac{a}{a+b-ab}\\
&\quad + [t \neq A \wedge c = 0] \cdot (1-b) + [t \neq A \wedge c = 1] \cdot \frac{(1-b)a}{a+b-ab}
\end{aligned}$$

The result of this example will be used in Sect. V. ■

Remark 16. (Expectations are bounded) Reasoning within denotational semantics requires a lower and upper bound on expectations. In [10] expectations are defined to be non-negative with 0 as the least element and 1 as the maximum. We just note that these bounds can be altered or even given up provided that the program at hand has certain properties - the discussion of details is beyond this work. In the following we stick to the original definitions with bounds 0 and 1. ■

B. Operational Semantics

Our aim is to model the stepwise behaviour of a pGCL-program P by an MDP denoted $\mathcal{M}[\![\, P \,]\!]$. This MDP represents the operational interpretation of the program P and intuitively acts as an abstract machine for P. This is done as follows. Let η be a variable valuation of the program variables. That is, η is a mapping from the program variables onto their (possibly infinite) domains. For variable x, $\eta(x)$ denotes the value of x under η. For expression E, let $[\![\, E \,]\!]_\eta$ denote the valuation of E under valuation η. This is defined in the standard way, e.g., for $E = 2 * x{+}y$ with $\eta(x) = 3$ and $\eta(y) = 7$, we have $[\![\, E \,]\!]_\eta = 2 * \eta(x){+}\eta(y) = 13$. We use the distinguished semantic construct exit to denote the successful termination of a program. States in the MDP are of the form $\langle Q, \eta \rangle$ with Q a pGCL-statement or $Q = \mathsf{exit}$ and η a variable valuation. For instance, the execution of the assignment $x := 2 * x{+}y$ under evaluation η with $\eta(x) = 3$ and $\eta(y) = 7$ results in the state $\langle \mathsf{exit}, \eta' \rangle$ where η' is the same as η except that $\eta'(x) = 13$. Initial states of program P are tuples $\langle P, \eta \rangle$ where η maps any variable onto an arbitrary value.

Definition 17. (Operational semantics of pGCL) The operational semantics of pGCL-program P, denoted $\mathcal{M}[\![\, P \,]\!]$, is the MDP $(S, S_0, \to)$ where:

- S is the set of pairs $\langle Q, \eta \rangle$ with Q a pGCL-program or $Q = \mathsf{exit}$, and η is a variable valuation of the variables occurring in P,
- $S_0 = \{\, \langle P, \eta \rangle \,\}$ where η maps every variabe in P to an arbitrary value, and
- $\to$ is the smallest relation that is induced by the inference rules in Table I.

■

TABLE I
INFERENCE RULES FOR pGCL PROGRAMS

$$\langle \mathsf{skip}, \eta \rangle \to \langle \mathsf{exit}, \eta \rangle \qquad \langle \mathsf{abort}, \eta \rangle \to \langle \mathsf{abort}, \eta \rangle$$

$$\langle x := \mathsf{expr}, \eta \rangle \to \langle \mathsf{exit}, \eta[x := [\![\, \mathsf{expr} \,]\!]_\eta] \rangle$$

$$\frac{\langle P, \eta \rangle \to \mu}{\langle P; Q, \eta \rangle \to \nu} \text{ with } \nu(\langle P'; Q, \eta' \rangle) = \mu(\langle P', \eta' \rangle) \text{ where } \mathsf{exit}; Q = Q.$$

$$\langle P \ [] \ Q, \eta \rangle \to \langle P, \eta \rangle \qquad \langle P \ [] \ Q, \eta \rangle \to \langle Q, \eta \rangle$$

$$\langle P \ [p] \ Q, \eta \rangle \to \mu \text{ with } \mu(\langle P, \eta \rangle) = p \text{ and } \mu(\langle Q, \eta \rangle) = 1{-}p$$

$$\frac{\eta \models G}{\langle \mathsf{if}(G)\{P\}\,\mathsf{else}\,\{Q\}, \eta \rangle \to \langle P, \eta \rangle}$$

$$\frac{\eta \not\models G}{\langle \mathsf{if}(G)\{P\}\,\mathsf{else}\,\{Q\}, \eta \rangle \to \langle Q, \eta \rangle}$$

$$\frac{\eta \models G}{\langle \mathsf{while}(G)\{P\}, \eta \rangle \to \langle P; \mathsf{while}(G)\{P\}, \eta \rangle}$$

$$\frac{\eta \not\models G}{\langle \mathsf{while}(G)\{P\}, \eta \rangle \to \langle \mathsf{exit}, \eta \rangle}$$

Example 18. (Operational semantics) Figure 1 depicts the MDP underlying the cowboy example. This MDP is parameterized with parameters a and b but has a finite state space. A slight adaptation of our example program in which we keep track of the number of shots before one of the cowboys dies, yields an MDP with infinitely many states. The support of any distribution in this MDP is finite however. ■

Let $P^{\checkmark}$ denote the set of states in MDP $\mathcal{M}[\![\, P \,]\!]$ of the form $\langle \mathsf{exit}, \eta \rangle$ for arbitrary variable valuation η. Note that states in $P^{\checkmark}$ represent the successful termination of P. If $P^{\checkmark} = \emptyset$, program P diverges under all possible policies.

Definition 19. (Reward-MDP of a pGCL-program) Let P be a pGCL-program and f a post-expectation for P. The reward-MDP associated to P and f is defined as $\mathcal{R}_f[\![\, P \,]\!] =$

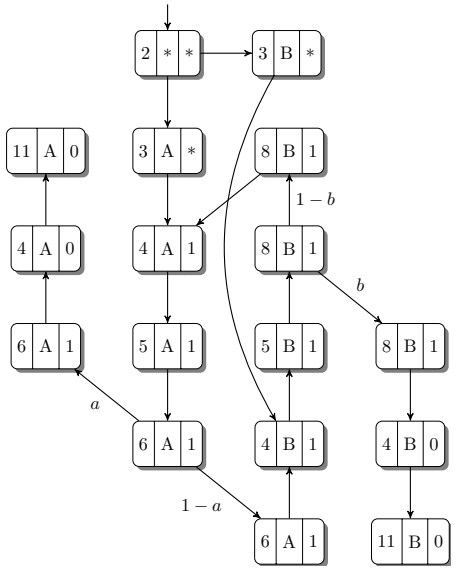


Fig. 1. MDP $\mathcal{M}$ for the duelling cowboys example. Each state is determined by a 3-tuple: (program location, value of t, value of c) where $*$ denotes an arbitrary value.

$(\mathcal{M}[\![\, P \,]\!], r)$ with $\mathcal{M}[\![\, P \,]\!]$ the MDP of P as defined before and reward function r defined by $r(s) = f(\eta)$ if $s = \langle \mathsf{exit}, \eta \rangle \in P^{\surd}$ and $r(s) = 0$ otherwise. ■

Note that we use a special reward structure: only terminal states are assigned a reward which is not necessarily 0. All other states have a zero reward. This property allows us to rewrite the definition of expected rewards as follows.

Lemma 20. (Characterizing expected rewards)

For pGCL program P and variable valuation η, we have:

$$\begin{aligned} &\mathit{ExpRew}^{\mathcal{R}_f[\![\, P \,]\!]}(\langle P, \eta \rangle \models \Diamond P^{\surd}) \\ &= \min_{\mathfrak{P}} \sum_{\widehat{\pi} \in \mathit{Paths}^{\mathfrak{P}}_{\min}(s, \Diamond P^{\surd})} \mathbf{P}(\widehat{\pi}) \cdot r_{P^{\surd}}(\widehat{\pi}) \ , \end{aligned}$$

where $\mathit{Paths}^{\mathfrak{P}}_{\min}(s, \Diamond T)$ is the set containing all finite paths of the form $s_0 \dots s_k$ with $s_0 = s$, $s_k \in T$ and $s_i \notin T$ for all $0 \leq i < k$ that adhere to the policy $\mathfrak{P}$. ■

Proof: Let $T = P^{\surd}$ for pGCL program P. The proof has two ingredients. First, we observe that a path which fails to reach a final state has reward 0 according to the *wp* semantics of abort. Secondly, in a finitely-branching MDP with countably many states there are "only" countably many paths that reach any given set. Consider the definition of expected reward:

$$\min_{\mathfrak{P}} \sum_{c=0}^{\infty} c \cdot \mathit{Pr}^{\mathfrak{P}}\{\, \pi \in \mathit{Paths}^{\mathfrak{P}}(s, \Diamond T) \mid r_T(\pi) = c \,\}.$$

Given that $Pr(\pi \models \Diamond T) = \mathbf{P}(\widehat{\pi})$ where prefix $\widehat{\pi}$ of π is minimal and ends in T, the above term equals:

$$\min_{\mathfrak{P}} \sum_{c=0}^{\infty} c \cdot \mathbf{P}\{\widehat{\pi} \in \mathit{Paths}^{\mathfrak{P}}_{\min}(s, T) \mid r_T(\widehat{\pi}) = c\}.$$

As $\mathcal{M}[\![\, P \,]\!]$ is a finitely branching MC, there are countably many $\widehat{\pi}$ for each reward c. This yields:

$$\min_{\mathfrak{P}} \sum_{\pi \in \mathit{Paths}^{\mathfrak{P}}_{\min}(s, T)} \mathbf{P}(\widehat{\pi}) \cdot r_T(\widehat{\pi})$$

■

We use this fact in our proofs later on.

Remark 21. (Real valued rewards) Lemma 20 provides a straight-forward way to calculate expected rewards when the rewards are real valued instead of just integer. This is because the summation runs not over the possible cumulative rewards (of which there are uncountably many in the case of real valued rewards) but over the possible paths that reach an *exit* state. In the following we stay with integer rewards as introduced earlier but bear in mind that Theorems 23 and 24 also hold for real valued post-expectations. ■

Analogously we obtain:

Lemma 22. (Characterizing liberal expected rewards)

For pGCL program P and variable valuation η, we have:

$$\begin{aligned} &\mathit{LExpRew}^{\mathcal{R}_f[\![\, P \,]\!]}(\langle P, \eta \rangle \models \Diamond P^{\surd}) \\ &= \min_{\mathfrak{P}} \sum_{\pi \in \mathit{Paths}^{\mathfrak{P}}_{\min}(s, P^{\surd})} \mathbf{P}(\widehat{\pi}) \cdot r_{P^{\surd}}(\widehat{\pi}) + \mathit{Pr}^{\mathfrak{P}}(\langle P, \eta \rangle \not\models \Diamond P^{\surd}) \ . \end{aligned}$$

■

Proof: Follows immediately from Lemma 20. ■

C. Main Results

This brings us at a position to present our main results of this paper: a formal relationship between the *wp*-semantics of pGCL-program P and its operational semantics in terms of a reward-MDP, and similary for the *wlp*-semantics. We first consider the *wp*-semantics.

Theorem 23. (Operational vs. *wp*-semantics) For pGCL-program P, variable valuation η, and post-expectation f:

$$\mathit{wp}(P, f)(\eta) \ = \ \mathit{ExpRew}^{\mathcal{R}_f[\![\, P \,]\!]}(\langle P, \eta \rangle \models \Diamond P^{\surd}).$$ ■

Proof: By structural induction over the pGCL program P. For the sake of convenience, let $\mathit{Paths}(P^{\surd}, \eta, c)$ denote the set

$$\{\, \pi \in \mathit{Paths}(\langle P, \eta \rangle, \Diamond P^{\surd}) \mid r_{P^{\surd}}(\pi) = c \,\}.$$

Furthermore we write paths as sequences of states and leave out the distribution in between each pair of states because it is obvious. Induction base:

- For $P = \mathsf{skip}$ we derive:

$$\begin{aligned} &\mathit{ExpRew}^{\mathcal{R}_f[\![\, \mathsf{skip} \,]\!]}(\langle \mathsf{skip}, \eta \rangle \models \Diamond \mathsf{skip}^{\surd}) \\ &= \min_{\mathfrak{P}} \sum_{c=0}^{\infty} c \cdot \mathit{Pr}^{\mathfrak{P}}\left(\mathit{Paths}^{\mathfrak{P}}(\mathsf{skip}^{\surd}, \eta, c)\right) \\ &= f(\eta) \cdot \mathit{Pr}\{\pi = \langle \mathsf{skip}, \eta \rangle \langle \mathsf{exit}, \eta \rangle \mid r_{\mathsf{skip}^{\surd}}(\pi) = f(\eta)\} \\ &= f(\eta) \cdot 1 \\ &= f(\eta) \\ &= \mathit{wp}(\mathsf{skip}, f)(\eta). \end{aligned}$$

- For $P = \mathsf{abort}$ we derive:

$$\begin{aligned}
&\mathit{ExpRew}^{\mathcal{R}_f[\![\,\mathsf{abort}\,]\!]}(\langle \mathsf{abort}, \eta\rangle \models \Diamond \mathsf{abort}^{\surd}) \\
&= \min_{\mathfrak{P}} \sum_{c=0}^{\infty} c \cdot \mathit{Pr}^{\mathfrak{P}}\left(\mathit{Paths}^{\mathfrak{P}}(\mathsf{abort}^{\surd}, \eta, c)\right) \\
&= 0 \\
&= \mathit{wp}(\mathsf{abort}, f)(\eta)
\end{aligned}$$

  as there is no path starting from $\langle \mathsf{abort}, \eta\rangle$ that reaches an exit-state.
- Let P be the assigment $x := E$. For this case, we have:

$$\begin{aligned}
&\mathit{ExpRew}^{\mathcal{R}_f[\![\,x:=E\,]\!]}(\langle x := E, \eta\rangle \models \Diamond x := E^{\surd}) \\
&= \min_{\mathfrak{P}} \sum_{c=0}^{\infty} c \cdot \mathit{Pr}^{\mathfrak{P}}\left(\mathit{Paths}^{\mathfrak{P}}(x := E^{\surd}, \eta, c)\right) \\
&= f(\eta[x/E]) \cdot \mathit{Pr}\{\, \pi = \langle x := E, \eta\rangle\langle \mathsf{exit}, \eta[x/E]\rangle \\
&\qquad\qquad \mid r_{x:=E^{\surd}}(\pi) = f(\eta[x/E])\,\} \\
&= f(\eta[x/E]) \cdot 1 \\
&= f(\eta[x/E]) \\
&= \mathit{wp}(x := E, f)(\eta).
\end{aligned}$$

Induction hypothesis: assume

$$\mathit{wp}(P, f)(\eta) = \mathit{ExpRew}^{\mathcal{R}_f[\![\,P\,]\!]}(\langle P, \eta\rangle \models \Diamond P^{\surd}).$$

Induction step:

- Consider the probabilistic choice $P\,[p]\,Q$ (this also covers conditional choice since it can be written as $P\,[G]\,Q$):

$$\begin{aligned}
&\mathit{ExpRew}^{\mathcal{R}_f[\![\,P\,[p]\,Q\,]\!]}(\langle P\,[p]\,Q, \eta\rangle \models \Diamond (P\,[p]\,Q)^{\surd}) \\
&= \min_{\mathfrak{P}} \sum_{c=0}^{\infty} c \cdot \mathit{Pr}^{\mathfrak{P}}\left(\mathit{Paths}^{\mathfrak{P}}((P\,[p]\,Q)^{\surd}, \eta, c)\right) \\
&= \min_{\mathfrak{P}} \sum_{c=0}^{\infty} c \cdot p \cdot \mathit{Pr}^{\mathfrak{P}}\left(\mathit{Paths}^{\mathfrak{P}}(P^{\surd}, \eta, c)\right) \\
&\quad + \sum_{c=0}^{\infty} c \cdot (1-p) \cdot \mathit{Pr}^{\mathfrak{P}}\left(\mathit{Paths}^{\mathfrak{P}}(Q^{\surd}, \eta, c)\right) \\
&\overset{*}{=} p \cdot \min_{\mathfrak{P}_1} \sum_{c=0}^{\infty} c \cdot \mathit{Pr}^{\mathfrak{P}_1}\left(\mathit{Paths}^{\mathfrak{P}_1}(P^{\surd}, \eta, c)\right) \\
&\quad + (1-p) \cdot \min_{\mathfrak{P}_2} \sum_{c=0}^{\infty} c \cdot \mathit{Pr}^{\mathfrak{P}_2}\left(\mathit{Paths}^{\mathfrak{P}_2}(Q^{\surd}, \eta, c)\right) \\
&= p \cdot \mathit{ExpRew}^{\mathcal{R}_f[\![\,P\,]\!]}(\langle P, \eta\rangle \models \Diamond P^{\surd}) \\
&\quad + (1-p) \cdot \mathit{ExpRew}^{\mathcal{R}_f[\![\,Q\,]\!]}(\langle Q, \eta\rangle \models \Diamond Q^{\surd}) \\
&\overset{I.H.}{=} p \cdot \mathit{wp}(P, f)(\eta) + (1-p) \cdot \mathit{wp}(Q, f)(\eta) \\
&= \mathit{wp}(P\,[p]\,Q, f)(\eta)
\end{aligned}$$

  In $*$ we use the fact that the policy for paths starting in $\langle P, \eta\rangle$ is independent of the policy for paths starting in $\langle Q, \eta\rangle$.
- Consider the non-deterministic choice $P\,[]\,Q$:

$$\begin{aligned}
&\mathit{ExpRew}^{\mathcal{R}_f[\![\,P\,[]\,Q\,]\!]}(\langle P\,[]\,Q, \eta\rangle \models \Diamond (P\,[]\,Q)^{\surd}) \\
&= \min_{\mathfrak{P}} \sum_{c=0}^{\infty} c \cdot \mathit{Pr}^{\mathfrak{P}}\left(\mathit{Paths}^{\mathfrak{P}}((P\,[]\,Q)^{\surd}, \eta, c)\right) \\
&= \min\left\{ \min_{\mathfrak{P}} \sum_{c=0}^{\infty} c \cdot \mathit{Pr}^{\mathfrak{P}}\left(\mathit{Paths}^{\mathfrak{P}}(P^{\surd}, \eta, c)\right), \right. \\
&\qquad \left. \min_{\mathfrak{P}} \sum_{c=0}^{\infty} c \cdot \mathit{Pr}^{\mathfrak{P}}\left(\mathit{Paths}^{\mathfrak{P}}(Q^{\surd}, \eta, c)\right) \right\} \\
&= \min\{\mathit{ExpRew}^{\mathcal{R}_f[\![\,P\,]\!]}(\langle P, \eta\rangle \models \Diamond P^{\surd}), \\
&\qquad \mathit{ExpRew}^{\mathcal{R}_f[\![\,Q\,]\!]}(\langle Q, \eta\rangle \models \Diamond Q^{\surd})\} \\
&\overset{I.H.}{=} \min\{\mathit{wp}(P, f), \mathit{wp}(Q, f)\} \\
&= \mathit{wp}(P[]Q, f)(\eta)
\end{aligned}$$

- Consider the sequential composition $P; Q$:

$$\begin{aligned}
&\mathit{ExpRew}^{\mathcal{R}_f[\![\,P;Q\,]\!]}(\langle P; Q, \eta\rangle \models \Diamond (P; Q)^{\surd}) \\
&= \min_{\mathfrak{P}} \sum_{c=0}^{\infty} c \cdot \mathit{Pr}^{\mathfrak{P}}\left(\mathit{Paths}^{\mathfrak{P}}((P; Q)^{\surd}, \eta, c)\right) \\
&\overset{La20}{=} \min_{\mathfrak{P}} \sum_{\widehat{\pi} \in \mathit{Paths}^{\mathfrak{P}}_{\min}(s, P^{\surd})} \mathbf{P}(\widehat{\pi}) \cdot r_{P;Q^{\surd}}(\widehat{\pi}) \\
&\overset{*}{=} \min_{\mathfrak{P}} \sum_{\widehat{\pi} \in \mathit{Paths}^{\mathfrak{P}}_{\min}(s, P^{\surd})} \mathbf{P}(\widehat{\pi}) \cdot r^{q}_{P^{\surd}}(\widehat{\pi})
\end{aligned}$$

  where $r^{q}_{P^{\surd}}(\widehat{\pi})$ is the sum of rewards r_q along $\widehat{\pi}$ with

$$r_q(s) = \min_{\mathfrak{P}'} \left(\sum_{\widehat{\pi}' \in \mathit{Paths}^{\mathfrak{P}'}_{\min}(s, Q^{\surd})} \mathbf{P}(\widehat{\pi}') \cdot r_{Q^{\surd}}(\widehat{\pi}') \right)$$

  if $s = \langle \mathsf{exit}, \eta'\rangle \in P^{\surd}$ and $r_q(s) = 0$ otherwise

$$\begin{aligned}
&= \mathit{ExpRew}^{\mathcal{R}_g[\![\,P\,]\!]}(\langle P, \eta\rangle \models \Diamond P^{\surd}) \\
&\quad \text{where } g(\eta) = \mathit{ExpRew}^{\mathcal{R}_f[\![\,Q\,]\!]}(\langle Q, \eta\rangle \models \Diamond Q^{\surd}) \\
&\overset{I.H.}{=} \mathit{wp}(P; \mathit{wp}(Q, f))(\eta) \\
&= \mathit{wp}(P; Q, f)(\eta)\ .
\end{aligned}$$

  In $*$ we rewrite each single path into a prefix which corresponds to the execution of P and all possible continuations according to Q. Then we can compute the expected reward r_q of Q and use this as an intermediate result to compute the expected reward of the sequential composition.
- Consider the loop $\mathsf{while}(G)\{P\}$. This case is proven by induction on the number of iterations that a while-loop performs. Let the bounded while-loop for $k > 0$ be:

$$\begin{aligned}
&(\mathsf{while}(G)\{P\})^{k+1} \\
&= \mathsf{if}(G)\{P; (\mathsf{while}(G)\{P\})^{k}\}\ \mathsf{else}\ \{\mathsf{skip}\}\ .
\end{aligned}$$

  where the base case is $(\mathsf{while}(G)\{P\})^{0} = \mathsf{abort}$. We will show for every k that

$$\begin{aligned}
&\mathit{wp}((\mathsf{while}(G)\{P\})^{k}, f)(\eta) \\
&= \mathit{ExpRew}^{\mathcal{R}_f[\![\,(\mathsf{while}(G)\{P\})^{k}\,]\!]}(\eta)\ .
\end{aligned} \tag{1}$$

Observe that

$$\begin{aligned} &wp((\mathsf{while}(G)\{P\})^{k+1}, f)(\eta) \\ \geq\ &wp((\mathsf{while}(G)\{P\})^{k}, f)(\eta)\ . \end{aligned}$$

From the fixpoint theorem 3 in [7] we know that the more iterations the bounded while loop is allowed to perform the closer it approximates the fixpoint given in Def. 13. Formally this means

$$\begin{aligned} &\lim_{k\to\infty} wp((\mathsf{while}(G)\{P\})^{k}, f)(\eta) \\ =\ &wp(\mathsf{while}(G)\{P\}, f)(\eta)\ . \end{aligned} \tag{2}$$

From (1) it follows that for every k, *ExpRew* behaves identically to *wp*. Thus with (2) it follows that

$$wp((\mathsf{while}(G)\{P\}), f)(\eta) = ExpRew^{\mathcal{R}_f[\![\,(\mathsf{while}(G)\{P\})\,]\!]}(\eta).$$

It remains to prove (1). This is done by induction on k. Base case ($k = 0$):

$$\begin{aligned} &wp((\mathsf{while}(G)\{P\})^{0}, f)(\eta) \\ =\ &wp(\mathsf{abort}, f)(\eta) \\ \overset{*}{=}\ &ExpRew^{\mathcal{R}_f[\![\,\mathsf{abort}\,]\!]}(\eta) \\ =\ &ExpRew^{\mathcal{R}_f[\![\,(\mathsf{while}(G)\{P\})^{0}\,]\!]}(\eta) \end{aligned}$$

$(*)$ was already shown earlier in the case abort.
Induction hypothesis: equation (1) holds for some unspecified but fixed value of k.
Induction step:

$$\begin{aligned} &wp((\mathsf{while}(G)\{P\})^{k+1}, f)(\eta) \\ =\ &wp(\mathsf{if}(G)\{P; (\mathsf{while}(G)\{P\})^{k}\}\mathsf{else}\{\mathsf{skip}\}, f)(\eta) \\ =\ &[G]\cdot wp(P; (\mathsf{while}(G)\{P\})^{k}) + [\neg G]\cdot wp(\mathsf{skip}, f)(\eta) \\ \overset{*}{=}\ &[G]\cdot ExpRew^{\mathcal{R}_f[\![\,P;(\mathsf{while}(G)\{P\})^{k}\,]\!]}(\eta) \\ &+ [\neg G]\cdot ExpRew^{\mathcal{R}_f[\![\,\mathsf{skip}\,]\!]}(\eta) \\ =\ &ExpRew^{\mathcal{R}_f[\![\,\mathsf{if}(G)\{P;(\mathsf{while}(G)\{P\})^{k}\}\mathsf{else}\{\mathsf{skip}\}\,]\!]}(\eta) \\ =\ &ExpRew^{\mathcal{R}_f[\![\,(\mathsf{while}(G)\{P\})^{k+1}\,]\!]}(\eta) \end{aligned}$$

$(*)$ follows from the induction hypothesis and the previously shown cases for skip and sequential composition. ∎

Thus, $wp(P, f)$ evaluated at η is the least expected value of f over any of the result distributions of P.

Theorem 24. (Operational vs. *wlp*-semantics) For pGCL-program P, variable valuation η, and post-expectation f:

$$wlp(P, f)(\eta)\ =\ LExpRew^{\mathcal{R}_f[\![\,P\,]\!]}(\langle P, \eta\rangle \models \Diamond P^{\surd}).$$ ∎

Proof: By structural induction over the pGCL program P (analogously to the proof of Theorem 23). Due to space limitations we skip the base cases which are rather simple.
Induction hypothesis: assume

$$wlp(P, f)(\eta) = LExpRew^{\mathcal{R}_f[\![\,P\,]\!]}(\langle P, \eta\rangle \models \Diamond P^{\surd}).$$

Induction step:

- Consider the probabilistic choice $P\,[p]\,Q$ (again, this covers conditional choice):

$$\begin{aligned} &LExpRew^{\mathcal{R}_f[\![\,P\,[p]\,Q\,]\!]}(\langle P\,[p]\,Q, \eta\rangle \models \Diamond (P\,[p]\,Q)^{\surd}) \\ =\ &\min_{\mathfrak{P}} \Bigg(\sum_{c=0}^{\infty} c\cdot Pr^{\mathfrak{P}}\left(Paths^{\mathfrak{P}}((P\,[p]\,Q)^{\surd}, \eta, c)\right) \\ &\quad + Pr^{\mathfrak{P}}(s \not\models \Diamond (P\,[p]\,Q)^{\surd})\Bigg) \\ =\ &\min_{\mathfrak{P}} \Bigg(\sum_{c=0}^{\infty} c\cdot p\cdot Pr^{\mathfrak{P}}\left(Paths^{\mathfrak{P}}(P^{\surd}, \eta, c)\right) \\ &\quad + p\cdot Pr^{\mathfrak{P}}(s \not\models \Diamond P^{\surd}) \\ &\quad + \sum_{c=0}^{\infty} c\cdot (1-p)\cdot Pr^{\mathfrak{P}}\left(Paths^{\mathfrak{P}}(Q^{\surd}, \eta, c)\right) \\ &\quad + (1-p)\cdot Pr^{\mathfrak{P}}(s \not\models \Diamond Q^{\surd})\Bigg) \\ =\ &p\cdot \min_{\mathfrak{P}_1} \Bigg(\sum_{c=0}^{\infty} c\cdot Pr^{\mathfrak{P}_1}\left(Paths^{\mathfrak{P}_1}(P^{\surd}, \eta, c)\right) \\ &\quad + Pr^{\mathfrak{P}_1}(s \not\models \Diamond P^{\surd})\Bigg) \\ &+ (1-p)\cdot \min_{\mathfrak{P}_2} \Bigg(\sum_{c=0}^{\infty} c\cdot Pr^{\mathfrak{P}_2}\left(Paths^{\mathfrak{P}_2}(Q^{\surd}, \eta, c)\right) \\ &\quad + Pr^{\mathfrak{P}_2}(s \not\models \Diamond Q^{\surd})\Bigg) \\ =\ &p\cdot LExpRew^{\mathcal{R}_f[\![\,P\,]\!]}(\langle P, \eta\rangle \models \Diamond P^{\surd}) \\ &+ (1-p)\cdot LExpRew^{\mathcal{R}_f[\![\,Q\,]\!]}(\langle Q, \eta\rangle \models \Diamond Q^{\surd}) \\ \overset{I.H.}{=}\ &p\cdot wlp(P, f)(\eta) + (1-p)\cdot wlp(Q, f)(\eta) \\ =\ &wlp(P\,[p]\,Q, f)(\eta)\ . \end{aligned}$$

- Consider the non-deterministic choice $P[]Q$:

$$\begin{aligned} &LExpRew^{\mathcal{R}_f[\![\,P\,[]\,Q\,]\!]}(\langle P\,[]\,Q, \eta\rangle \models \Diamond (P\,[]\,Q)^{\surd}) \\ =\ &\min_{\mathfrak{P}} \Bigg(\sum_{c=0}^{\infty} c\cdot Pr^{\mathfrak{P}}\left(Paths^{\mathfrak{P}}((P\,[]\,Q)^{\surd}, \eta, c)\right) \\ &\quad + Pr^{\mathfrak{P}}(s \not\models \Diamond (P\,[]\,Q)^{\surd})\Bigg) \\ =\ &\min\Bigg\{ \min_{\mathfrak{P}} \Bigg(\sum_{c=0}^{\infty} c\cdot Pr^{\mathfrak{P}}\left(Paths^{\mathfrak{P}}(P^{\surd}, \eta, c)\right) \\ &\qquad + Pr^{\mathfrak{P}}(s \not\models \Diamond P^{\surd})\Bigg), \\ &\qquad \min_{\mathfrak{P}} \Bigg(\sum_{c=0}^{\infty} c\cdot Pr^{\mathfrak{P}}\left(Paths^{\mathfrak{P}}(Q^{\surd}, \eta, c)\right) \\ &\qquad + Pr^{\mathfrak{P}}(s \not\models \Diamond Q^{\surd})\Bigg)\Bigg\} \\ =\ &\min\{LExpRew^{\mathcal{R}_f[\![\,P\,]\!]}(\langle P, \eta\rangle \models \Diamond P^{\surd}), \\ &\qquad LExpRew^{\mathcal{R}_f[\![\,Q\,]\!]}(\langle Q, \eta\rangle \models \Diamond Q^{\surd})\} \\ \overset{I.H.}{=}\ &\min\{wlp(P, f)(\eta), wlp(Q, f)(\eta)\} \\ =\ &wlp(P[]Q, f)(\eta)\ . \end{aligned}$$

- Consider the sequential composition $P;Q$:

$$
\begin{aligned}
&\mathit{LExpRew}^{\mathcal{R}_f[\![P;Q]\!]}(\langle P;Q,\eta\rangle \models \Diamond(P;Q)^{\surd}) \\
&= \min_{\mathfrak{P}} \left(\sum_{c=0}^{\infty} c \cdot \mathit{Pr}^{\mathfrak{P}}\left(\mathit{Paths}^{\mathfrak{P}}((P;Q)^{\surd},\eta,c)\right) \right. \\
&\quad \left. + \mathit{Pr}^{\mathfrak{P}}(s \not\models \Diamond(P;Q)^{\surd}) \right) \\
&\overset{La22}{=} \min_{\mathfrak{P}} \left(\sum_{\pi \in \mathit{Paths}^{\mathfrak{P}}_{\min}(s,P^{\surd})} \mathbf{P}(\widehat{\pi}) \cdot r_{P;Q^{\surd}}(\widehat{\pi}) \right. \\
&\quad \left. + \mathit{Pr}^{\mathfrak{P}}\{\langle P;Q,\eta\rangle \not\models \Diamond P;Q^{\surd}\} \right) \\
&= \min_{\mathfrak{P}} \left(\sum_{\pi \in \mathit{Paths}^{\mathfrak{P}}_{\min}(s,P^{\surd})} \mathbf{P}(\widehat{\pi}) \cdot r^{q}_{P^{\surd}}(\widehat{\pi}) \right. \\
&\quad \left. + \mathit{Pr}^{\mathfrak{P}}\{\langle P,\eta\rangle \not\models \Diamond P^{\surd}\} \right)
\end{aligned}
$$

  where $r^{q}_{P^{\surd}}(\widehat{\pi})$ is the sum of rewards r_q along $\widehat{\pi}$ with

$$
\begin{aligned}
r_q(s) = \min_{\mathfrak{P}'} \Bigg(& \sum_{\widehat{\pi}' \in \mathit{Paths}^{\mathfrak{P}'}_{\min}(s,Q^{\surd})} \mathbf{P}(\widehat{\pi}') \cdot r_{Q^{\surd}}(\widehat{\pi}') \\
& + \mathit{Pr}^{\mathfrak{P}}\{\langle Q,\eta'\rangle \not\models \Diamond Q^{\surd}\} \Bigg)
\end{aligned}
$$

  if $s = \langle \mathit{exit},\eta'\rangle \in P^{\surd}$ and $r_q(s) = 0$ otherwise

$$
\begin{aligned}
&= \mathit{LExpRew}^{\mathcal{R}_g[\![P]\!]}(\langle P,\eta\rangle \models \Diamond P^{\surd}) \\
&\quad \text{where } g(\eta) = \mathit{LExpRew}^{\mathcal{R}_f[\![Q]\!]}(\langle Q,\eta\rangle \models \Diamond Q^{\surd}) \\
&\overset{I.H.}{=} \mathit{wlp}(P;\mathit{wlp}(Q,f))(\eta) \\
&= \mathit{wlp}(P;Q,f)(\eta) \ .
\end{aligned}
$$

- Consider the while loop $\mathsf{while}(G)\{P\}$. Again we prove this case by induction on the number of iterations that a while-loop performs. Let $(\mathsf{while}(G)\{P\})^k$ be defined as in the proof of the previous theorem. We show for every k that

$$
\begin{aligned}
&\mathit{wlp}((\mathsf{while}(G)\{P\})^k, f)(\eta) \\
&= \mathit{LExpRew}^{\mathcal{R}_f[\![(\mathsf{while}(G)\{P\})^k]\!]}(\eta) \ .
\end{aligned} \tag{3}
$$

  The only difference is now that

$$
\begin{aligned}
&\mathit{wlp}((\mathsf{while}(G)\{P\})^{k+1}, f)(\eta) \\
&\leq \mathit{wlp}((\mathsf{while}(G)\{P\})^k, f)(\eta) \ .
\end{aligned}
$$

  Using this we again know that the bounded while loop approximates the fixpoint given in Def. 14 (only this time from above). Formally this means

$$
\begin{aligned}
&\lim_{k\to\infty} \mathit{wlp}((\mathsf{while}(G)\{P\})^k, f)(\eta) \\
&= \mathit{wlp}(\mathsf{while}(G)\{P\}, f)(\eta) \ .
\end{aligned} \tag{4}
$$

  From (3) we know that for every k *LExpRew* behaves identically to *wlp*. Thus with (4) it follows that

$$
\begin{aligned}
&\mathit{wlp}((\mathsf{while}(G)\{P\}), f)(\eta) \\
&= \mathit{LExpRew}^{\mathcal{R}_f[\![(\mathsf{while}(G)\{P\})]\!]}(\eta) \ .
\end{aligned}
$$

  It remains to prove (3). This is done by induction on k. Base case ($k = 0$):

$$
\begin{aligned}
&\mathit{wlp}((\mathsf{while}(G)\{P\})^0, f)(\eta) \\
&= \mathit{wlp}(\mathsf{abort}, f)(\eta) \\
&\overset{*}{=} \mathit{LExpRew}^{\mathcal{R}_f[\![\mathsf{abort}]\!]}(\eta) \\
&= \mathit{LExpRew}^{\mathcal{R}_f[\![(\mathsf{while}(G)\{P\})^0]\!]}(\eta)
\end{aligned}
$$

  ($*$) was already shown earlier in the case abort.

  Induction hypothesis: equation (3) holds for some unspecified but fixed value of k.

  Induction step:

$$
\begin{aligned}
&\mathit{wlp}((\mathsf{while}(G)\{P\})^{k+1}, f)(\eta) \\
&= \mathit{wlp}(\mathsf{if}(G)\{P;(\mathsf{while}(G)\{P\})^k\}\mathsf{else}\{\mathsf{skip}\}, f)(\eta) \\
&= [G] \cdot \mathit{wlp}(P;(\mathsf{while}(G)\{P\})^k) + [\neg G] \cdot \mathit{wlp}(\mathsf{skip}, f)(\eta) \\
&\overset{*}{=} [G] \cdot \mathit{LExpRew}^{\mathcal{R}_f[\![P;(\mathsf{while}(G)\{P\})^k]\!]}(\eta) \\
&\quad + [\neg G] \cdot \mathit{LExpRew}^{\mathcal{R}_f[\![\mathsf{skip}]\!]}(\eta) \\
&= \mathit{LExpRew}^{\mathcal{R}_f[\![\mathsf{if}(G)\{P;(\mathsf{while}(G)\{P\})^k\}\mathsf{else}\{\mathsf{skip}\}]\!]}(\eta) \\
&= \mathit{LExpRew}^{\mathcal{R}_f[\![(\mathsf{while}(G)\{P\})^{k+1}]\!]}(\eta)
\end{aligned}
$$

  ($*$) follows from the induction hypothesis and the previously shown cases for skip and sequential composition.

■

The weakest liberal pre-expectation $\mathit{wlp}(P,f)$ is thus the least expected value of f over any of the result distributions of P plus the probability that P does not terminate.

Example 25. (Duelling cowboys.) Consider again the duelling cowboys example from Lst. 1. Assume we are interested in the probability that cowboy A wins the duel. In terms of the MDP semantics this means we are interested in

$$
\mathit{LExpRew}^{(\mathcal{M},r)}(\langle 2,*,*\rangle \models \Diamond(\mathcal{M},r)^{\surd})
$$

where $\mathcal{M}$ is the MDP from Fig. 1 and r is the reward function that indicates whether cowboy A has won or not, i.e.

$$
r(s) = \begin{cases} 1 & \text{if } s = \langle 11,A,0\rangle \\ 0 & \text{otherwise} \end{cases}
$$

In this example the MDP is finite and this allows us to compute the desired expected cumulative reward easily. That is, cowboy A wins with probability at least

$$
\frac{(1-b)a}{a+b-ab} \ .
$$

Figure 2 visualises this result. ■

V. Analysis

Although the computation of (liberal) expected rewards on MDPs may be numerically involved, it is intuitive in principle. However, pGCL programs will usually have an infinite state space due to the infinite domain of the program variables. It is then not possible to compute the expected reward on the reward model in general. In contrast to this, the denotational

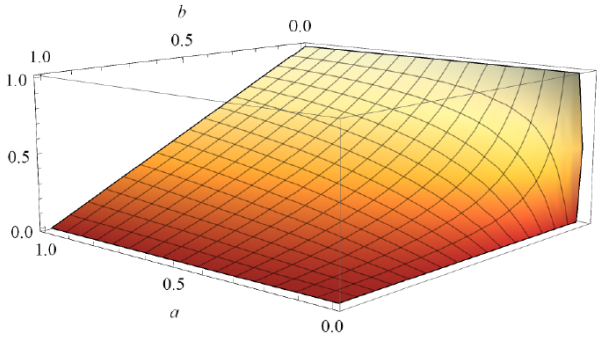


Fig. 2. Probability that A wins the duel, depending on a and b. Bear in mind that this is the least guaranteed probability that A wins. In the worst case (for A) cowboy B will shoot first and therefore as b tends to 1 the plot goes to 0, i.e. cowboy A has no chances. However for smaller values of b the influence of a increases.

semantics do not depend on the underlying state space but on the structure of the program. In this section we show how to determine a pre-expectation using *wlp*-semantics.

Again let us determine the probability that cowboy A wins the duel. Therefore we choose $[t = A]$ as the post-expectation and want to find *wlp*(cowboyDuel, $[t = A]$). Listing 2 shows the cowboy duelling program with annotations.

Listing 2. The duelling cowboys, annotated with expectations

```
1  int cowboyDuel(a, b) {
2    ⟨(1−b)a / a+b−ab⟩
3    ⟨min{ a/a+b−ab , (1−b)a/a+b−ab }⟩
4    (t := A [] t := B);
5    ⟨[t = A] · a/a+b−ab + [t = B] · (1−b)a/a+b−ab⟩
6    c := 1;
7    ⟨[t = A ∧ c = 0] · 1 + [t = A ∧ c = 1] · a/a+b−ab
           +[t = B ∧ c = 1] · (1−b)a/a+b−ab⟩
8    while (c = 1) {
9      ⟨[t = A ∧ c = 1] · a/a+b−ab + [t = B ∧ c = 1] · (1−b)a/a+b−ab⟩
10     ⟨[t = A ∧ c ≠ 1] · a + [t = A ∧ c = 1] · a/a+b−ab
             +[t = B ∧ c = 0] · (1 − b) + [t = B ∧ c = 1] · (1−b)a/a+b−ab⟩
11     if (t = A) {
12       (c := 0 [a] t := B);
13     } else {
14       (c := 0 [b] t := A);
15     }
16     ⟨[t = A ∧ c = 0] · 1 + [t = A ∧ c = 1] · a/a+b−ab
           +[t = B ∧ c = 1] · (1−b)a/a+b−ab⟩
17   }
18   ⟨[c ≠ 1] · ( [t = A ∧ c = 0] · 1 + [t = A ∧ c = 1] · a/a+b−ab
           +[t = B ∧ c = 1] · (1−b)a/a+b−ab )⟩
19   ⟨[t = A]⟩
20   return t; // the survivor
21 }
```

The program is annotated backwards according to the rules from Def. 13 (and 14). In line 19 we start with the post-expectation that we are interested in. We finish with the sought probability in line 2. The only non-trivial step is to discover the so-called invariant which appears in line 7 and 16. But let us assume for the moment that it is given. Then all other annotations are obtained by applying the syntactic rules from Def. 13. In particular the calculation from line 16 to line 10 was already shown in Example 15. This means that the analysis can be automatically carried out by a computer once we have found the aforementioned invariant - irrespective of the underlying state space size.

The annotation in line 7 and 16 which we call invariant is an expectation that over-approximates the fixed point solution in Def. 14. More precisely, an annotation f is called invariant if

$$f \cdot [G] \leq \mathit{wlp}(\mathit{loop\_body}, f) \ . \tag{5}$$

In our example, f is the expectation in line 7, G is the loop guard $c = 1$ and *loop_body* is the code in lines 11–15. In line 9 the expectation represents $f \cdot [G]$ and line 10 is $\mathit{wlp}(\mathit{loop\_body}, f)$. Clearly, (5) is satisfied in our example.

The difficulty in reasoning with denotational semantics is to find suitable invariants. The invariant generation process is a topic on its own and beyond the scope of this paper. We refer to [4], [10] for this matter. Our recently developed tool PRINSYS[2] helps the user to find certain kinds of invariants semi-automatically.

VI. CONCLUSION

This paper provided a formal connection between the expectation transformer semantics of pGCL by McIver and Morgan [10] and a simple operational semantics using (parametric) MDPs. This yields an insightful relationship between semantics used for formal reasoning for probabilistic programs and the notion of a computation in terms of an MDP. Our approach assigns rewards to terminal states (only), and establishes that expected cumulative rewards correspond to wp-semantics. A slight variant of expected rewards yields a connection to the wlp-semantics.

Possible future work is to establish a relation to a denotational semantics in terms of metric spaces, like in [6] or to link our semantics to the seminal work by Kozen [5] where probabilistic programs are interpreted as partial measurable functions on a measurable space.

REFERENCES

[1] Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
[2] Dijkstra, E.W.: A Discipline of Programming. Prentice Hall (1976)
[3] Howard, R.A.: Dynamic Programming and Markov Processes. MIT Press (1960)
[4] Katoen, J.P., McIver, A., Meinicke, L., Morgan, C.: Linear-Invariant Generation for Probabilistic Programs. In: SAS. LNCS, Springer (2010)
[5] Kozen, D.: Semantics of probabilistic programs. J. Comput. Syst. Sci. 22(3), 328–350 (1981)
[6] Kwiatkowska, M.Z., Norman, G.: Probabilistic Metric Semantics for a Simple Language with Recursion. In: MFCS. LNCS, Springer (1996)
[7] Lassez, J.L., Nguyen, V.L., Sonenberg, L.: Fixed point theorems and semantics: A folk tale. Inf. Process. Lett. 14(3), 112–116 (1982)
[8] Lukkien, J.J.: An Operational Semantics for the Guarded Command Language. In: MPC. LNCS, vol. 669, pp. 233–249. Springer (1992)
[9] Lukkien, J.J.: Operational Semantics and Generalized Weakest Preconditions. Sci. Comput. Program. 22(1-2), 137–155 (1994)
[10] McIver, A., Morgan, C.: Abstraction, Refinement And Proof For Probabilistic Systems (Monographs in Computer Science). Springer (2004)

[2]Available at: http://www-i2.informatik.rwth-aachen.de/prinsys/.

B. Operational versus weakest pre-expectation semantics for the probabilistic guarded command language

Together with our paper in Appendix A, this paper forms the basis of Chapter 2.

Operational versus Weakest Pre-expectation Semantics for the Probabilistic Guarded Command Language

Friedrich Gretz[a,b], Joost-Pieter Katoen[a], Annabelle McIver[b]

[a] *RWTH Aachen University, Aachen, Germany*
[b] *Macquarie University, Sydney, Australia*

Abstract

This paper proposes a simple operational semantics of `pGCL`, Dijkstra's guarded command language extended with probabilistic choice, and relates this to `pGCL`'s *wp*-semantics by McIver and Morgan. Parametric Markov decision processes whose state rewards depend on the post-expectation at hand are used as the operational model. We show that the *weakest pre-expectation* of a `pGCL`-program w.r.t. a post-expectation corresponds to the *expected cumulative reward* to reach a terminal state in the parametric MDP associated to the program. In a similar way, we show a correspondence between weakest *liberal* pre-expectations and *liberal* expected cumulative rewards. The verification of probabilistic programs using *wp*-semantics and operational semantics is illustrated using a simple running example.

Keywords: expectation transformer semantics, operational semantics, Markov decision process, expected rewards

1. Introduction

Formal semantics of programming languages has been the subject of intense research in computer science for several decades. Various approaches have been developed for the description of program semantics. Structured operational semantics defines the meaning of a program by means of an abstract machine where states correspond to program configurations (typically consisting of a program counter and a variable valuation) and transitions model the evolution of a program by executing statements. Program executions are then the possible runs of the abstract machine. Denotational semantics maps a program onto a mathematical object that describes for instance its input-output behaviour. Finally, axiomatic semantics provides the program semantics in an indirect manner by describing its properties. A prominent example of the latter are Hoare

Email addresses: `fgretz@cs.rwth-aachen.de` (Friedrich Gretz), `katoen@cs.rwth-aachen.de` (Joost-Pieter Katoen), `annabelle.mciver@mq.edu.au` (Annabelle McIver)

triples in which annotations, written in predicate logic, are associated to control points of the program.

The semantics of Dijkstra's seminal guarded command language [1] from the seventies is given in terms of weakest preconditions. It is in fact a predicate transformer semantics, i.e. a total function between two predicates on the state of a program. The predicate transformer $E = wp(P, F)$ for program P and postcondition F yields the weakest precondition E on the initial state of P ensuring that the execution of P terminates in a final state satisfying F. There is a direct relation with axiomatic semantics: the Hoare triple $\langle E \rangle\, P\, \langle F \rangle$ holds for total correctness if and ony if $E \Rightarrow wp(P, F)$. The weakest *liberal* precondition $wlp(P, F)$ yields the weakest precondition for which P either does not terminate or establishes F. It does not ensure termination and corresponds to Hoare logic in partial correctness. Although providing an operational semantics for the guarded command language is rather straightforward, it was not until the early nineties that Lukkien [2, 3] provided a formal connection between the predicate transformer semantics and the notion of a computation.

Qualitative annotations in predicate calculus are often insufficient for probabilistic programs as they cannot express quantities such as expectations over program variables. To that end, McIver and Morgan [4] generalised the methods of Dijkstra and Hoare to probabilistic programs by making the annotations real-valued expressions —referred to as expectations— in the program variables. Expectations are the quantitative analogue of predicates. This yields an expectation transformer semantics of the probabilistic guarded command language (`pGCL`, for short), an extension of Dijkstra's language with a probabilistic choice operator. An expectation transformer is a total function between two expectations on the state of a program. The expectation transformer $e = wp(P, f)$ for `pGCL`-program P and post-expectation f over final states yields the least expected value e on P's initial state ensuring that P's execution terminates with a value f. The annotation $\langle e \rangle\, P\, \langle f \rangle$ holds for total correctness if and only if $e \leq wp(P, f)$, where $\leq$ is to be interpreted in a point-wise manner. The weakest *liberal* pre-expectation $wlp(P, f)$ yields the least expectation for which P either does not terminate or establishes f. It does not ensure termination and corresponds to partial correctness.

This paper provides a simple operational semantics of `pGCL` using parametric Markov decision processes (pMDPs), a slight variant of MDPs in which probabilities may be parameterised [5, 6]. Our main contribution in this paper is a formal connection between the *wp*- and *wlp*-semantics of `pGCL` by McIver and Morgan and the operational semantics of `pGCL`. This provides a clean and insightful relationship between the abstract expectation transformer semantics that has been proven useful for formal reasoning about probabilistic programs, and the notion of a computation in terms of the operational model, a pMDP. In order to establish this connection we equip pMDPs with state rewards that depend on the post-expectation at hand. Intuitively speaking, we decorate a terminal state in the operational model of a program with a reward that corresponds to the value of the post-expectation. All other states are assigned reward zero. We then show that the *weakest pre-expectation* of a `pGCL`-program P w.r.t. a post-

expectation corresponds to the *expected cumulative reward* to reach a terminal state in the pMDP associated to P. In a similar way, we show that weakest *liberal* pre-expectations correspond to *liberal* expected cumulative rewards. The proofs are by induction on the structure of our probabilistic programs using standard results from fixed point theory. This paper thus yields a correspondence theorem that enables us to understand the mathematically involved expectation transformers intuitively using only first principles of Markov decision processes with rewards. In addition, for finite-state programs (or program fragments), our result implies that algorithms for computing expected accumulated rewards in MDPs – for which efficient algorithms and tools based on linear programming exist – can be employed for computing weakest pre-expectations. Finally we recall the notion of probabilistic invariants [4] and apply our correspondence theorem to find an operational characterisation of invariants (which originally are defined in terms of expectation transformers).

1.1. Related Work

The MDP semantics of `pGCL` in this paper bears strong resemblance to the operational semantics of similar languages. To mention a few, Baier *et al.* [7] provide an MDP semantics of a probabilistic version of Promela, the modeling language of the SPIN model checker. Di Pierro *et al.* [8] give a semantics to a very similar programming language without non-determinism. The seminal work by Kozen [9] provides two semantics of a deterministic variant of `pGCL` and shows their correspondence. Kozen interprets probabilistic programs as partial measurable functions on a measurable space, and as continuous linear operators on a Banach space of measures. He *et al.* [10] provide a mapping from a semantics based on a probabilistic complete partial order which contains non-determinism à la Jones [11] to a semantics which is a mapping from initial states to sets of probability distributions over final states. To our knowledge, our results on relating weakest pre-expectations of pGCL and an operational semantics are novel. Our set-up and results can be considered as a probabilistic analogue of the work by Lukkien [2, 3] who provided a formal connection between the predicate transformer semantics of Dijktra's guarded command language and the operational notion of a computation.

More examples of how to discover and apply invariants when reasoning about probabilistic programs can be found at [12]. There we also describe Prinsys, a tool for semi-automatic invariant generation.

1.2. Structure of this paper.

The rest of the paper is divided as follows. In Sect. 2 we introduce the probabilistic programming language `pGCL`. Parametric Markov decision processes with rewards are introduced in Sect. 3. Section 4 recaps the denotational semantics of `pGCL` [4] and introduces operational semantics for this language. Then the main result is established, namely that the two semantics are equivalent. Section 5 provides an example of reasoning over `pGCL` programs. Finally, Sect. 6 introduces invariants and uses our main result to give an operational characterisation for them.

This paper is an extended version of the conference paper [13]. This version contains a generalised version of the proofs of Theorems 23 and 24, a new section on invariants and an appendix with a new proof for continuity of $wp(P, \cdot)$ and $wlp(P, \cdot)$.

2. Probabilistic Programs

Our programming language pGCL [4] is an extension of Dijkstra's guarded command language [1]. Besides a non-deterministic choice operator, denoted [], and a conditional choice, it incorporates a probabilistic choice operator, denoted $[p]$, where p is a real parameter (or constant) whose value lies in the range $[0, 1]$. pGCL is a language to model sequential programs containing randomized assignments. For instance, the assignment $(x := 2{\cdot}x\ [0.75]\ x := x{+}1)$ doubles the value of x with probability $\frac{3}{4}$ and increments it by one with the remaining probability $\frac{1}{4}$.

Definition 1. (Syntax of pGCL) Let P, P_1, P_2 be pGCL-programs, p a probability variable, x a program variable, E an expression, and G a Boolean expression. The syntax of a pGCL program P adheres to the following grammar:

$$\mathsf{skip} \mid \mathsf{abort} \mid x := E \mid P_1; P_2 \mid P_1 \,[]\, P_2 \mid P_1 \,[p]\, P_2 \mid$$
$$\mathsf{if}(G)\{P_1\}\,\mathsf{else}\,\{P_2\} \mid \mathsf{while}(G)\{P\}.$$

■

skip stands for the empty statement, abort for abortion, and $x := E$ for an assignment of the value of expression E (over the program variables) to variable x. The sequentially composed program $P_1; P_2$ behaves like P_1 and subsequently like P_2 on the successful termination of P_1. The statement $P_1 \,[]\, P_2$ denotes a non-deterministic choice; it behaves like either P_1 or P_2. The statement $P_1 \,[p]\, P_2$ denotes a probabilistic choice. It behaves like P_1 with probability p and like P_2 with probability $1{-}p$. The remaining two statements are standard: conditional choice and while-loop. Throughout this paper, we assume that pGCL-programs are well-typed. This entails that for assignments of the form $x := E$ we assume that x and E are of the same type. We assume G to denote a Boolean expression and variable p to denote a probability in the real interval $[0, 1]$.

Example 2. (Duelling cowboys [4]) The pGCL program in Lst. 1 models the following situation: There are two cowboys, A and B, who are fighting a classical duel. They take turns, shooting at each other until one of them is hit. If A (resp. B) shoots then he hits B (resp. A) with probability a (resp. b). We assume that either cowboy A or B is allowed to start; the choice of who will start is resolved nondeterministically. Variable t keeps track of the turns, while c determines whether the duel continues or someone is hit. Note that it is a distinctive feature that we do not have to specify exact probabilities and instead allow arbitrary parameters such as a and b. ■

Listing 1: The duelling cowboys, cf. [4].

```
(t := A [] t := B);
c := 1;
while (c = 1) {
  if (t = A) {
    (c := 0 [a] t := B);
  } else {
    (c := 0 [b] t := A);
  }
}
```

3. Markov decision processes

This section introduces the basics of Markov decision processes (MDPs) [5, 6] enriched with state rewards. We first recall the definition of an MDP with a countable state space and define elementary notions such as paths and policies. Subsequently, we introduce reward-MDPs in which states are equipped with a real valued reward and focus on reachability objectives, in particular (liberal) expected cumulative rewards to reach a set of states. These measures are later shown to closely correspond to the weakest (liberal) pre-condition semantics of pGCL-programs.

3.1. Preliminaries

Let Var be a set of variables. In the following we consider a countable state space S where each state s is a valuation which maps variables to real values

$$S : \mathit{Var} \to \mathbb{R} \ .$$

This approach later allows us to nicely connect states of a program to states of a transition system like an MDP (as defined below). Additionally we fix a set of parameters Par which are independent of states. These parameters will represent probabilities which can be left unspecified. So each parameter represents a value from the interval $[0, 1]$. Let $V(\mathit{Par})$ denote the set of expressions over Par.

Definition 3. (Parametric distribution) A parametric distribution μ is a function that maps states to probabilities. The probabilities are real values in $[0, 1]$ or expressions over Par:

$$\mu : S \to V(\mathit{Par}) \cup [0, 1] \quad \text{with} \quad \sum_{s \in S} \mu(s) = 1.$$

The set of all parametric distributions over state space S is denoted $\mathit{Dist}(S)$. ■

Example 4. (Parametric distribution) Consider $S = \{s_0, s_1, s_2\}$. Then a parametric distribution μ might be: $\mu(s_0) = p$, $\mu(s_1) = 1 - p$ and $\mu(s_2) = 0$ where $p \in V(\mathit{Par})$ is an expression that consists of just a single parameter. The expression's value is fixed but unknown. The parameter p can be refined by any value from $[0, 1]$. ■

It is important to stress that Def. 3 requires that for any state s the resulting expression $\mu(s)$ is a probability, i.e. an expression that evaluates to a value in $[0,1]$ for all possible states s and all possible parameter valuations.

Definition 5. (Parametric Markov decision process) A pMDP $\mathcal{M}$ is a tuple $(S, S_0, \longrightarrow)$ where S is a countable set of states with initial state-set $S_0 \subseteq S$ where $S_0 \neq \emptyset$, and $\longrightarrow \subseteq S \times \mathit{Dist}(S)$ is a transition relation from a state to a set of parametric distributions over states. ■

Let $s \to \mu$ denote $(s,\mu) \in \longrightarrow$ and $s \to t$ denote $s \to \mu$ with $\mu(t) = 1$. We define $\mathit{Dist}(s) = \{\, \mu \mid s \to \mu \,\}$ to be the set of *enabled distributions* in state s. In the following we do not emphasise the parametric nature of our structures and use the term "Markov decision process" (MDP) synonymously. The intuitive operational behaviour of an MDP $\mathcal{M}$ is as follows. First, non-deterministically select some initial state $s_0 \in S_0$. In state s with $\mathit{Dist}(s) \neq \emptyset$, non-deterministically select $\mu \in \mathit{Dist}(s)$. The next state t is randomly chosen with probability $\mu(t)$. If $\mathit{Dist}(t) = \emptyset$, exit; otherwise continue as for state s.

Remark 6. (Countability of paths) In the context of this paper we are only interested in finitely branching Markov decision processes with bounded non-determinism. Therefore $|\mathit{Dist}(s)| < \infty$ for all $s \in S$ and all distributions are assumed to have finite support. Consequently every state has finitely many successor states. Hence there are countably many (finite) paths between any two states. This property is crucial for Def. 12 and Lem. 21 later on. ■

A *path* π of MDP $\mathcal{M}$ is a maximal alternating sequence of states and distributions, written $\pi = s_0 \xrightarrow{\mu_0} s_1 \xrightarrow{\mu_1} \ldots$ such that $\mu_i \in \mathit{Dist}(s_i)$ and $\mu_i(s_{i+1}) > 0$ for all $i \geq 0$. As any path is a maximal sequence, it is either infinite or ends in a state s with $\mathit{Dist}(s) = \emptyset$. The set of all paths in $\mathcal{M}$ is denoted $\mathit{Paths}(\mathcal{M})$. Reasoning about probabilities on sets of paths of an MDP relies on the resolution of non-determinism. This resolution is performed by a policy[2] that selects one of the enabled distributions in a state. In general a policy may base its decision in state s on the path fragment from $s_0 \in S_0$ to s. However in the context of this paper we are interested in computing expectations in MDPs and so it suffices to consider positional policies as shown in [6].

Definition 7. (Positional policy) A function $\mathfrak{P} : S \to \mathit{Dist}(S)$ with $\mathfrak{P}(s) \in \mathit{Dist}(s)$ for all $s \in S$ is called a positional policy for MDP $\mathcal{M} = (S, S_0, \longrightarrow)$. ■

A positional policy deterministically selects an enabled distribution based on the current state s only. As in the rest of this paper we only consider positional policies, we call them simply policies. The path fragment leading to s does not play any role. The path $\pi = s_0 \xrightarrow{\mu_0} s_1 \xrightarrow{\mu_1} \ldots$ is called a $\mathfrak{P}$-path if it is induced by the policy $\mathfrak{P}$, that is, $\mathfrak{P}(s_i) = \mu_i$ for all $i \geq 0$. Let $\mathit{Paths}^{\mathfrak{P}}(s)$ denote the set of $\mathfrak{P}$-paths starting from state s. A policy of an MDP $\mathcal{M}$ induces a Markov chain $\mathcal{M}^{\mathfrak{P}}$ with the same state space as $\mathcal{M}$

[2] Also called scheduler, strategy or adversary.

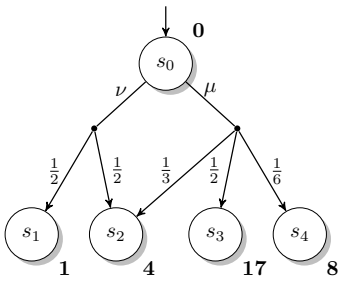
and transition probabilities $\mathfrak{P}(s)(t)$ for states s and t. For finite path fragment $\widehat{\pi} = s_0 \xrightarrow{\mu_0} s_1 \xrightarrow{\mu_1} \ldots \xrightarrow{\mu_{k-1}} s_k$ of a $\mathfrak{P}$-path, let $\mathbf{P}^{\mathfrak{P}}(\widehat{\pi})$ denote the probability of $\widehat{\pi}$ which is defined by $\mu_0(s_1) \times \ldots \times \mu_{k-1}(s_k) = \prod_{i=1}^{k} \mu_{i-1}(s_i)$. Let $Pr_s^{\mathfrak{P}}(\Pi)$ denote the probability of the set of paths Π all starting in s under policy $\mathfrak{P}$. This probability measure is defined in the standard way using a cylinder set construction on the induced Markov chain $\mathcal{M}^{\mathfrak{P}}$ [14]. In the following we drop the subscript s whenever it is clear from the context. Note that the measure $\mathbf{P}^{\mathfrak{P}}(\pi)$ of a path π starting in state s is 0 if $\pi \notin \mathit{Paths}^{\mathfrak{P}}(s)$.

To compare our operational semantics of pGCL with its *wp*- and *wlp*-semantics, we use rewards (or, dually costs).

Definition 8. (MDP with rewards) An MDP with *rewards* (also called reward-MDP, or shortly RMDP) is a pair $(\mathcal{M}, r)$ with $\mathcal{M}$ an MDP with state space S and $r : S \to \mathbb{R}_{\geq 0}$ a function assigning a real reward to each state. ■

Intuitively, the reward $r(s)$ stands for the reward earned on entering state s. The cumulative reward of a finite path fragment $s_0 \xrightarrow{\mu_0} s_1 \xrightarrow{\mu_1} \ldots s_k$ is the sum of the rewards in all states that have been visited, i.e., $r(s_0) + \ldots + r(s_k)$.

Example 9. (RMDP, cumulative reward of a path)



Assume a policy $\mathfrak{P}$ with $\mathfrak{P}(s_0) = \mu$. Then $\pi = s_0 \xrightarrow{\mu} s_3 \in \mathit{Paths}^{\mathfrak{P}}(s_0)$ is a possible path that is taken under policy $\mathfrak{P}$ with probability $\mathbf{P}^{\mathfrak{P}}(\pi) = 0.5$ and has cumulative reward $r(\pi) = 17$. ■

3.2. Reachability objectives

We are interested in reachability events in RMDPs. Let $T \subseteq S$ be a set of target states. The event $\Diamond T$ stands for the reachability of some state in T, i.e., $\Diamond T$ is the set of paths in MDP $\mathcal{M}$ that hit some state $s \in T$. Formally $\Diamond T = \{\, \pi \in \mathit{Paths}(\mathcal{M}) \mid \exists i \geq 0. \pi[i] \in T \,\}$ where $\pi[i]$ denotes the i-th state visited along π. We write $\pi \models \Diamond T$ whenever π belongs to $\Diamond T$. It follows by standard arguments that $\Diamond T$ is a measurable event. Its measure depends on the chosen initial state and policy. In order to define the cumulative reward for this event we need to introduce the cumulative reward along a path.

Definition 10. (Cumulative reachability reward) Let $\pi = s_0 \xrightarrow{\mu_0} s_1 \xrightarrow{\mu_1} \ldots$ be a maximal path in RMDP $(\mathcal{M}, r)$ and $T \subseteq S$ a set of target states. If $\pi \models \Diamond T$, the *cumulative reward* along π before reaching T is defined by:

$r_T(\pi) = r(s_0) + \ldots + r(s_k)$ where $s_i \notin T$ for all $i < k$ and $s_k \in T$. If $\pi \not\models \Diamond T$, then $r_T(\pi) = 0$. ■

Stated in words, the cumulative reward for a path π to reach T is the cumulative reward of the minimal prefix of π satisfying $\Diamond T$. In case π never reaches a state in T, the cumulative reward is defined to be zero. We denote by $Paths^{\mathfrak{P}}(s, \Diamond T)$ the set of paths starting in s that eventually reach T under policy $\mathfrak{P}$.

Remark 11. (Reward for paths that fail to reach an objective) One can argue that the choice of zero as the reward for never reaching T is arbitrary and that this reward could alternatively be defined as e.g., any constant or even infinity. This depends on the purpose of rewards. Later we will reward states that correspond to the terminal states of a program. If an execution fails to reach a terminal state, then we treat this as "undesired" behaviour that has reward zero. This agrees with the previous definition. ■

We can now define the *expected* reward for reachability. Usually an expectation of a real valued random variable X with a density function $p(x)$ is defined as

$$E(X) = \int_x x \cdot p(x) dx \ .$$

Our random variable is the reward but it is not continuous. Even though a reward function maps states to non-negative real values, there are only countably many different rewards that can be accumulated on the way from a state s to a target set T. This is because there are only countably many finite prefixes of paths that lead from a state s to states in T. Hence the random variable – the reward – can assume only countably many distinct values. We can therefore define a discrete probability distribution, which assigns each given reward c the probability of all finite path prefixes that run from s to T and have the cumulative reachability reward c.

Definition 12. (Expected reward for reachability) Let $(\mathcal{M}, r)$ be an RMDP with state space S and $T \subseteq S$ and $s \in S$. Further let $\mathfrak{C}$ denote the set of all cumulative reachability reward values that can be accumulated by paths from s to T in $(\mathcal{M}, r)$. The *minimal expected reward* until reaching $T \subseteq S$ from $s \in S$, denoted $ExpRew^{(\mathcal{M},r)}(s \models \Diamond T)$, is defined by:

$$\inf_{\mathfrak{P}} \sum_{c \in \mathfrak{C}} c \cdot Pr^{\mathfrak{P}}\{\, \pi \in Paths^{\mathfrak{P}}(s, \Diamond T) \mid r_T(\pi) = c \,\} \ .$$

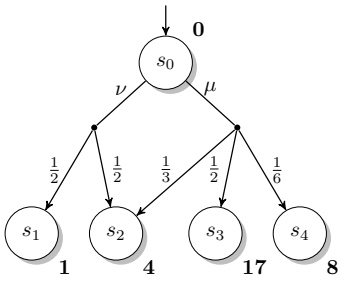
The minimal *liberal* expected reward until reaching some state in T from s, denoted $LExpRew^{(\mathcal{M},r)}(s \models \Diamond T)$, is defined by:

$$\inf_{\mathfrak{P}} \left\{ Pr^{\mathfrak{P}}(s \not\models \Diamond T) + \sum_{c \in \mathfrak{C}} c \cdot Pr^{\mathfrak{P}}\{\, \pi \in Paths^{\mathfrak{P}}(s, \Diamond T) \mid r_T(\pi) = c \,\} \right\} \ .$$

We omit the superscript $(\mathcal{M}, r)$ when the underlying model is clear from the context. ■

The expected reward in s to reach some state in T is the expected cumulative reward over all paths (reaching T) induced under a demonic policy. A demonic policy resolves non-determinism such that the expected reward is minimised. The motivation to consider a demonic and not an angelic (maximising) policy lies in the relationship between demonic choice and the notion of *weakest* pre-expectation of `pGCL` programs. As we will see later, a non-deterministic choice in `pGCL` will be resolved in such a way that the pre-expectation is minimised. Note that in case T is not reachable with positive probability from s under a demonic policy, $\mathit{ExpRew}(s \models \Diamond T) = 0$. $\mathit{LExpRew}(s \models \Diamond T)$ is the expected reward to reach T or never reach it from s. In case there is no policy under which T can be reached from s with positive probability, we have that $\mathit{LExpRew}(s \models \Diamond T) = 1$. This measure is motivated by reasoning about partial correctness where termination is not guaranteed and we will later see the relationship to the weakest *liberal* pre-expectation in Thm. 24. Note that *ExpRew* and *LExpRew* coincide if T is reached with probability one under all policies. For finite MDPs without parameters, expected and liberal expected rewards for reachability objectives can be obtained by solving a linear programming problem, cf. [6].

Example 13. (Expected rewards)



Let $T = \{s_2, s_3\}$. Then $\mathit{ExpRew}(s_0 \models \Diamond T) = \min\{4 \cdot \frac{1}{2}, 4 \cdot \frac{1}{3} + 17 \cdot \frac{1}{2}\} = \min\{2, \frac{59}{6}\} = 2$. And $\mathit{LExpRew}(s_0 \models \Diamond T) = \min\{\frac{1}{2} + 4 \cdot \frac{1}{2}, \frac{1}{6} + 4 \cdot \frac{1}{3} + 17 \cdot \frac{1}{2}\} = \min\{2.5, 10\} = 2.5$. ■

4. `pGCL` semantics

This section describes an expectation transformer semantics of `pGCL`, as well as an operational semantics using MDPs. The main result of this section is a formal connection between these two semantics.

4.1. Denotational Semantics

When probabilistic programs are executed they determine a probability distribution over final values of program variables. For instance, on termination of

$$(x := 1 \ [0.75] \ x := 2);$$

the final value of x is 1 with probability $\frac{3}{4}$ and 2 with probability $\frac{1}{4}$. An alternative way to characterise that probabilistic behaviour is to consider the expected

values over random variables with respect to that distribution. For example, to determine the probability that x is set to 1, we can compute the expected value of the random variable "x is 1" which is $\frac{3}{4} \cdot 1 + \frac{1}{4} \cdot 0 = \frac{3}{4}$. Similarly, to determine the average value of x, we compute the expected value of the random variable "x" which is $\frac{3}{4} \cdot 1 + \frac{1}{4} \cdot 2 = \frac{5}{4}$.

More generally, rather than a distribution-centred approach [15, 16], we take an "expectation transformer" [4] approach. We annotate probabilistic programs with *expectations*. As before we assume a state space S, a set of parameters Par and a set of expressions $V(Par)$ over it.

Definition 14. (Expectation) Expectations are functions which map program states to (non-negative) real values or expressions over parameters. The set of expectations over state space S is then

$$\mathbf{E} = \{f \mid f : S \to \mathbb{R}_{\geq 0} \cup V(Par)\} \ .$$

■

Note that every expectation f maps to a non-negative real value or an expression that is non-negative for all possible evaluations of its parameters. Expectations are the quantitative analogue to Hoare's predicates for non-probabilistic programs. An expectation transformer is a total function between two expectations. The expectation transformer $wp(P, f)$ for program P and post-expectation f yields the least expected value e on P's initial state ensuring that P's execution terminates with a value f. Annotation $\langle e \rangle\, P\, \langle f \rangle$ holds for total correctness if and only if $e \ \leq \ wp(P, f)$ where $\leq$ is to be interpreted in a point-wise manner. Intuitively, implication between predicates is generalised to pointwise inequality between expectations. For convenience we use square brackets to cast Boolean truth values to numbers and by convention $[\text{true}] = 1$ and $[\text{false}] = 0$.

Definition 15. (wp-semantics of `pGCL` [4]) Let P and Q be `pGCL`-programs, f a post-expectation, x a program variable, E an expression, and G a Boolean expression. The wp-semantics of a program is defined by structural induction over the program as follows:

- $wp(\mathsf{skip}, f) = f$
- $wp(\mathsf{abort}, f) = 0$
- $wp(x := E, f) = f[x/E]$
- $wp(P; Q, f) = wp(P, wp(Q, f))$
- $wp(\mathsf{if}(G)\{P\}\mathsf{else}\{Q\}, f) = [G] \cdot wp(P, f) + [\neg G] \cdot wp(Q, f)$
- $wp(P \,[]\, Q, f) = \min\left(wp(P, f), wp(Q, f)\right)$
- $wp(P \,[p]\, Q, f) = p \cdot wp(P, f) + (1{-}p) \cdot wp(Q, f)$
- $wp(\mathsf{while}(G)\{P\}, f) = \mu X.\, ([G] \cdot wp(P, X) + [\neg G] \cdot f)$

Here $f[x/E]$ denotes a function that is obtained from f by replacing every occurrence of x by E. The least fixed point operator μ is used w.r.t. the ordering $\leq$ on expectations. The existence of the fixed point can be seen from Thm. 30 in the appendix. We refer to [4] for more details. ■

If program P does not contain a probabilistic choice, then this *wp* is isomorphic to Dijkstra's *wp* [4]. A weakest liberal pre-expectation $wlp(P, f)$ yields the least expectation for which P either does not terminate or establishes f.

Definition 16. **(*wlp*-semantics of pGCL)** *wlp*-semantics differs from *wp*-semantics only for while and abort:

- $wlp(\mathsf{abort}, f) = 1$
- $wlp(\mathsf{while}(G)\{P\}, f) = \nu X.\,([G]\cdot wlp(P, X) + [\neg G]\cdot f)$

Here ν is the greatest fixed point operator w.r.t. the ordering $\leq$ on expectations. ■

While expectations do not need to be bounded from above in general, an upper bound is required for the definition of *wlp*. This is because non-terminating programs produce the maximal pre-expectation which has to be well-defined. In this paper we set this upper bound of wlp to one. In [17] a more general approach is discussed.

As the previous definitions indicate, the difference between *wp* and *wlp* lies in the handling of non-termination. The expectation transformers *wp* and *wlp* coincide for programs that terminate with probability one as is the case for *ExpRew* and *LExpRew*.

Example 17. **(Application of *wp*-semantics)** Consider again the duelling cowboys example. Assume we are given the post-expectation:

$$f(c,t) = [t = A \wedge c = 0] + [t = A \wedge c = 1]\cdot\frac{a}{a+b-ab} + [t = B \wedge c = 1]\cdot\frac{(1-b)a}{a+b-ab}\ .$$

The post-expectation f gives the probability that A wins, depending on the state that the program is in. The states are characterised by the predicates in square brackets. Let us compute the weakest pre-expectation of the loop body from Lst. 1 w.r.t. the post-expectation f. This yields:

$$\begin{aligned}
&wp(\mathsf{if}(t = A)\{(c := 0\,[a]\,t := B);\}\mathsf{else}\{(c := 0\,[b]\,t := A);\}, f)\\
&= [t = A]\cdot wp((c := 0\,[a]\,t := B), f)\\
&\quad + [t \neq A]\cdot wp((c := 0\,[b]\,t := A), f)\\
&= [t = A]\cdot(a\cdot wp(c := 0, f) + (1{-}a)\cdot wp(t := B, f))\\
&\quad + [t \neq A]\cdot(b\cdot wp(c := 0, f) + (1{-}b)\cdot wp(t := A, f))\\
&= [t = A]\cdot\Bigg(a\cdot\Bigg([t = A \wedge 0 = 0] + [t = A \wedge 0 = 1]\cdot\frac{a}{a+b-ab}\\
&\qquad\qquad + [t = B \wedge 0 = 1]\cdot\frac{(1-b)a}{a+b-ab}\Bigg)
\end{aligned}$$

$$
\begin{aligned}
&+(1-a)\cdot\Bigg([B=A\wedge c=0]+[B=A\wedge c=1]\cdot\frac{a}{a+b-ab}\\
&\qquad+[B=B\wedge c=1]\cdot\frac{(1-b)a}{a+b-ab}\Bigg)\Bigg)\\
&+[t\neq A]\cdot\Bigg(b\cdot\Bigg([t=A\wedge 0=0]+[t=A\wedge 0=1]\cdot\frac{a}{a+b-ab}\\
&\qquad+[t=B\wedge 0=1]\cdot\frac{(1-b)a}{a+b-ab}\Bigg)\\
&\qquad+(1-b)\cdot\Bigg([A=A\wedge c=0]+[A=A\wedge c=1]\cdot\frac{a}{a+b-ab}\\
&\qquad\qquad+[A=B\wedge c=1]\cdot\frac{(1-b)a}{a+b-ab}\Bigg)\Bigg)\\
&=[t=A]\cdot\left(a\cdot[t=A]+(1-a)\cdot[c=1]\cdot\frac{(1-b)a}{a+b-ab}\right)\\
&\quad+[t\neq A]\cdot\left(b\cdot[t=A]+(1-b)\left([c=0]+[c=1]\cdot\frac{a}{a+b-ab}\right)\right)\\
&=[t=A]\cdot a+[t=A\wedge c=1]\cdot\frac{(1-a)(1-b)a}{a+b-ab}\\
&\quad+[t\neq A\wedge c=0]\cdot(1-b)+[t\neq A\wedge c=1]\cdot\frac{(1-b)a}{a+b-ab}\\
&=[t=A\wedge c\neq 1]\cdot a+[t=A\wedge c=1]\cdot\frac{a}{a+b-ab}\\
&\quad+[t\neq A\wedge c=0]\cdot(1-b)+[t\neq A\wedge c=1]\cdot\frac{(1-b)a}{a+b-ab}
\end{aligned}
$$

In the last step we use the fact that $[t=A]\cdot a$ can be split into

$$[t=A\wedge c=1]\cdot a+[t=A\wedge c\neq 1]\cdot a\ .$$

This computation tells us that if, say f describes the probability after one iteration of the loop that cowboy A wins, then the computed expression gives the probability that A wins before that iteration of the loop. The result of this example will be used in Sect. 5. ■

4.2. Operational Semantics

Our aim is to model the stepwise behaviour of a pGCL-program P by an MDP denoted $\mathcal{M}[\![P]\!]$. This MDP represents the operational interpretation of the program P and intuitively acts as an abstract machine for P. This is done as follows. Let η be a variable valuation of the program variables. That is, η is a mapping from the program variables onto their (possibly infinite) domains. For variable x, $\eta(x)$ denotes the value of x under η. For expression E, let $[\![E]\!]_\eta$ denote the value of E under valuation η. This is defined in the standard way, e.g., for $E = 2{\cdot}x{+}y$ with $\eta(x) = 3$ and $\eta(y) = 7$, we have

$[\![E]\!]_\eta = 2{\cdot}\eta(x){+}\eta(y) = 13$. We use the distinguished semantic construct exit to denote the successful termination of a program. States in the MDP are of the form $\langle Q, \eta \rangle$ with Q a pGCL-statement or $Q = \mathsf{exit}$ and η a variable valuation. For instance, the execution of the assignment $x := 2{\cdot}x{+}y$ under valuation η with $\eta(x) = 3$ and $\eta(y) = 7$ results in the state $\langle \mathsf{exit}, \eta' \rangle$ where η' is the same as η except that $\eta'(x) = 13$. Initial states of program P are tuples $\langle P, \eta \rangle$ where η is arbitrary.

Definition 18. (Operational semantics of pGCL) The operational semantics of pGCL-program P, denoted $\mathcal{M}[\![P]\!]$, is the MDP $(S, S_0, \to)$ where:

- S is the set of pairs $\langle Q, \eta \rangle$ with Q a pGCL-program or $Q = \mathsf{exit}$, and η is a variable valuation of the variables occurring in P,
- $S_0 = \{ \langle P, \eta \rangle \}$ where η is arbitrary, and
- $\to$ is the smallest relation that is induced by the inference rules in Table 1.

■

Each rule tells us how to obtain the successors from a state. For example, in a state with probabilistic choice the MDP will make a transition to the parametric distribution μ. Then a successor state is chosen according to μ. We omit the distribution when it is obvious and write an arrow to the successor instead. For instance the rules for non-deterministic choice are a shorthand for

$$\langle P \,[]\, Q, \eta \rangle \to \{\mu, \nu\}$$
$$\text{with } \mu(\langle P, \eta \rangle) = 1 \text{ and } \nu(\langle Q, \eta \rangle) = 1 \ .$$

A premise is used to enable or disable transitions depending on the variable valuation of the current state. Consider the last two rules: if the system evolves from a state that represents a loop, it will proceed to a state where the loop body has to be executed once before going back to the loop header *provided* that the current variable valuation η satisfies the loop's guard G. If it does not, the last rule dictates that the loop is terminated, i.e. the system moves to an exit state.

Example 19. (Operational semantics) Figure 1 depicts the MDP underlying the cowboy example. This MDP is parameterized with parameters a and b. Technically, for every possible initial variable evaluation there should be an initial state. However, a programmer usually has to initialise the program variables before they may be used in a computation. This is also the case in our example program from Lst. 1. Therefore it does not matter in which initial state we start as the initialisation steps will always take us to the states $(4, A, 1)$ or $(4, B, 1)$. This observation allows us to represent the program by an MDP with a finite state space where all initial states have been merged into one. A slight adaptation of our example program in which we keep track of the number of shots before one of the cowboys dies, yields an MDP with infinitely many states. The support of any distribution in this MDP is finite however. ■

Table 1: Inference rules for pGCL programs

$$\langle \mathsf{skip}, \eta \rangle \to \langle \mathsf{exit}, \eta \rangle \qquad \langle \mathsf{abort}, \eta \rangle \to \langle \mathsf{abort}, \eta \rangle$$

$$\langle x := E, \eta \rangle \to \langle \mathsf{exit}, \eta[x := [\![E]\!]_\eta] \rangle$$

$$\frac{\langle P, \eta \rangle \to \mu}{\langle P; Q, \eta \rangle \to \nu} \text{ with } \nu(\langle P'; Q, \eta' \rangle) = \mu(\langle P', \eta' \rangle) \text{ where } \mathsf{exit}; Q = Q.$$

$$\langle P \,[]\, Q, \eta \rangle \to \langle P, \eta \rangle \qquad \langle P \,[]\, Q, \eta \rangle \to \langle Q, \eta \rangle$$

$$\langle P \;[p]\; Q, \eta \rangle \to \mu \quad \text{with } \mu(\langle P, \eta \rangle) = p \text{ and } \mu(\langle Q, \eta \rangle) = 1{-}p$$

$$\frac{\eta \models G}{\langle \mathsf{if}(G)\{P\}\,\mathsf{else}\,\{Q\}, \eta \rangle \to \langle P, \eta \rangle}$$

$$\frac{\eta \not\models G}{\langle \mathsf{if}(G)\{P\}\,\mathsf{else}\,\{Q\}, \eta \rangle \to \langle Q, \eta \rangle}$$

$$\frac{\eta \models G}{\langle \mathsf{while}(G)\{P\}, \eta \rangle \to \langle P; \mathsf{while}(G)\{P\}, \eta \rangle}$$

$$\frac{\eta \not\models G}{\langle \mathsf{while}(G)\{P\}, \eta \rangle \to \langle \mathsf{exit}, \eta \rangle}$$

Let $P^{\surd}$ denote the set of states in MDP $\mathcal{M}[\![P]\!]$ of the form $\langle \mathsf{exit}, \eta \rangle$ for arbitrary variable valuation η. Note that states in $P^{\surd}$ represent the successful termination of P. If $P^{\surd}$ is not reachable, program P diverges under all possible policies.

Definition 20. (RMDP of a pGCL-program) Let P be a pGCL-program and f a post-expectation for P. The reward-MDP associated with P and f is defined as $\mathcal{R}_f[\![P]\!] = (\mathcal{M}[\![P]\!], r)$ with $\mathcal{M}[\![P]\!]$ the MDP of P as defined before and reward function r defined by $r(s) = f(\eta)$ if $s = \langle \mathsf{exit}, \eta \rangle \in P^{\surd}$ and $r(s) = 0$ otherwise. ■

Note that we use a special reward structure: only terminal states are assigned a reward which is not necessarily zero. All other states have a zero reward. The following lemma explains how expected rewards can be computed in $\mathcal{R}_f[\![P]\!]$.

Lemma 21. (Characterizing expected rewards)

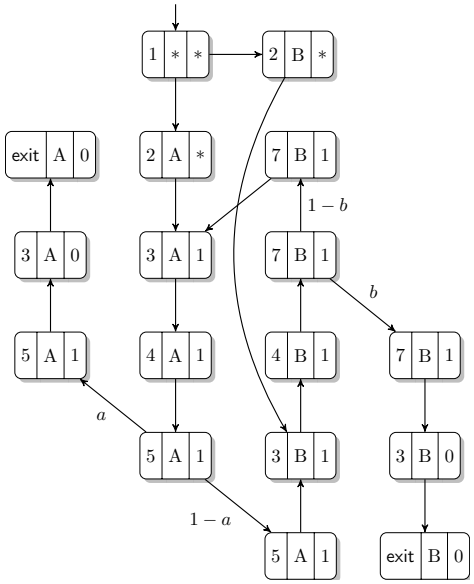


Figure 1: MDP $\mathcal{M}$ for the duelling cowboys example. Each state is determined by a triple: (program location, value of t, value of c) where $*$ denotes an arbitrary value.

For pGCL program P and a state $s = \langle P, \eta \rangle$, we have:

$$ExpRew^{\mathcal{R}_f[\![P]\!]}(s \models \Diamond P^{\surd}) = \inf_{\mathfrak{P}} \sum_{\widehat{\pi} \in Paths^{\mathfrak{P}}_{\min}(s, \Diamond P^{\surd})} \mathbf{P}^{\mathfrak{P}}(\widehat{\pi}) \cdot r_{P^{\surd}}(\widehat{\pi}) \ ,$$

where $Paths^{\mathfrak{P}}_{\min}(s, \Diamond P^{\surd})$ is the set containing all (finite) paths of the form $s_0 \dots s_k$ with $s_0 = s$, $s_k \in P^{\surd}$ and $s_i \notin P^{\surd}$ for all $0 \leq i < k$ that adhere to the policy $\mathfrak{P}$. ■

Proof: Let $T = P^{\surd}$ for pGCL program P. The proof requires a property stated in Remark 6 namely that in an MDP there are only countably many finite paths that lead from one state to another. Consider the definition of expected reward:

$$\inf_{\mathfrak{P}} \sum_{c \in \mathfrak{C}} c \cdot Pr^{\mathfrak{P}}\{\, \pi \in Paths^{\mathfrak{P}}(s, \Diamond T) \mid r_T(\pi) = c \,\} \ .$$

Given that $Pr^{\mathfrak{P}}(\pi \models \Diamond T) = \mathbf{P}^{\mathfrak{P}}(\widehat{\pi})$ where prefix $\widehat{\pi}$ of π is minimal and ends in T, the above term equals:

$$\inf_{\mathfrak{P}} \sum_{c \in \mathfrak{C}} c \cdot \mathbf{P}^{\mathfrak{P}}\{\widehat{\pi} \in Paths^{\mathfrak{P}}_{\min}(s, \Diamond T) \mid r_T(\widehat{\pi}) = c\} \ .$$

As in $\mathcal{R}_f[\![P]\!]$ the number of finite path prefixes $\widehat{\pi}$ that reach T and accumulate

a reward c is countable we can rewrite the sum into:

$$\inf_{\mathfrak{P}} \sum_{\widehat{\pi} \in Paths^{\mathfrak{P}}_{\min}(s, \Diamond T)} \mathbf{P}^{\mathfrak{P}}(\widehat{\pi}) \cdot r_T(\widehat{\pi}) \ .$$

■

Lemma 21 expresses the expected reward in terms of paths that reach an exit state and their cumulative rewards. This provides a straightforward way to calculate expected rewards (for finite systems). In the next subsection Lem. 21 will be helpful in the proofs of our main results.

Analogously we obtain:

Lemma 22. (Characterizing liberal expected rewards)

For pGCL program P and variable valuation η, we have:

$$\begin{aligned} & LExpRew^{\mathcal{R}_f[\![P]\!]}(\langle P, \eta \rangle \models \Diamond P^{\checkmark}) \\ = \inf_{\mathfrak{P}} & \left\{ Pr^{\mathfrak{P}}(\langle P, \eta \rangle \not\models \Diamond P^{\checkmark}) + \sum_{\widehat{\pi} \in Paths^{\mathfrak{P}}_{\min}(s, \Diamond P^{\checkmark})} \mathbf{P}^{\mathfrak{P}}(\widehat{\pi}) \cdot r_{P^{\checkmark}}(\widehat{\pi}) \right\} \ . \end{aligned}$$

■

Proof: Follows immediately from Def. 12 and Lem. 21. ■

4.3. Correspondence between operational and expectation transformer semantics

We now present the main results of this paper: a formal relationship between the *wp*-semantics of pGCL-program P and its operational semantics in terms of a RMDP, and similary for the *wlp*-semantics. We first consider the *wp*-semantics.

Theorem 23. (Correspondence theorem) For pGCL-program P, variable valuation η, and post-expectation f:

$$wp(P, f)(\eta) \ = \ ExpRew^{\mathcal{R}_f[\![P]\!]}(\langle P, \eta \rangle \models \Diamond P^{\checkmark}) \ .$$

■

Proof: By structural induction over the pGCL program P. We write paths as sequences of states and leave out the distribution in between each pair of states for the ease of presentation. In this proof we use the alternative definition for expected rewards given in Lem. 21.
Induction base:

- For $P = \mathsf{skip}$ we use the fact that skip does not change the post-expectation.

We derive:

$$\begin{aligned}
&ExpRew^{\mathcal{R}_f[\![\,\mathsf{skip}\,]\!]}(\langle \mathsf{skip}, \eta\rangle \models \Diamond \mathsf{skip}^{\surd}) \\
&= \inf_{\mathfrak{P}} \sum_{\widehat{\pi} \in Paths^{\mathfrak{P}}_{\min}(\langle \mathsf{skip},\eta\rangle, \Diamond \mathsf{skip}^{\surd})} \mathbf{P}^{\mathfrak{P}}(\widehat{\pi}) \cdot r_{\mathsf{skip}^{\surd}}(\widehat{\pi}) \\
&= \inf_{\mathfrak{P}} \ \mathbf{P}^{\mathfrak{P}}(\langle \mathsf{skip}, \eta\rangle\langle \mathsf{exit}, \eta\rangle) \cdot f(\eta) \\
&= 1{\cdot}f(\eta) \\
&= f(\eta) \\
&= wp(\mathsf{skip}, f)(\eta).
\end{aligned}$$

- For $P = \mathsf{abort}$ we use the fact that it fails to terminate and has a pre-expectation of zero. We derive:

$$\begin{aligned}
&ExpRew^{\mathcal{R}_f[\![\,\mathsf{abort}\,]\!]}(\langle \mathsf{abort}, \eta\rangle \models \Diamond \mathsf{abort}^{\surd}) \\
&= \inf_{\mathfrak{P}} \sum_{\widehat{\pi} \in Paths^{\mathfrak{P}}_{\min}(\langle \mathsf{abort},\eta\rangle, \Diamond \mathsf{abort}^{\surd})} \mathbf{P}^{\mathfrak{P}}(\widehat{\pi}) \cdot r_{\mathsf{abort}^{\surd}}(\widehat{\pi}) \\
&= 0 \\
&= wp(\mathsf{abort}, f)(\eta)
\end{aligned}$$

  as there is no path starting from $\langle \mathsf{abort}, \eta\rangle$ that reaches an exit-state.

- Let P be the assignment $x := E$. For this case we apply the substitution:

$$\begin{aligned}
&ExpRew^{\mathcal{R}_f[\![\,x:=E\,]\!]}(\langle x := E, \eta\rangle \models \Diamond (x := E)^{\surd}) \\
&= \inf_{\mathfrak{P}} \sum_{\widehat{\pi} \in Paths^{\mathfrak{P}}_{\min}(\langle x:=E,\eta\rangle, \Diamond (x:=E)^{\surd})} \mathbf{P}^{\mathfrak{P}}(\widehat{\pi}) \cdot r_{(x:=E)^{\surd}}(\widehat{\pi}) \\
&= \inf_{\mathfrak{P}} \ \mathbf{P}^{\mathfrak{P}}(\langle x := E, \eta\rangle\langle \mathsf{exit}, \eta[x/E]\rangle) \cdot f(\eta[x/E]) \\
&= 1{\cdot}f(\eta[x/E]) \\
&= f(\eta[x/E]) \\
&= f[x/E](\eta) \\
&= wp(x := E, f)(\eta).
\end{aligned}$$

Induction hypothesis: assume that for program P (and analogously for Q)

$$wp(P, f)(\eta) = ExpRew^{\mathcal{R}_f[\![\,P\,]\!]}(\langle P, \eta\rangle \models \Diamond P^{\surd}) \ .$$

Induction step:

- Consider the probabilistic choice $P\,[p]\,Q$ (this also covers conditional choice since it can be written as $P\,[[G]]\,Q$)[3]. The idea is to represent the expected reward as a weighted sum of the expected rewards computed from

[3]The guard is enclosed in square brackets twice: the inner brackets cast the boolean formula to a $\{0, 1\}$-valued function, the outer brackets are part of the probabilistic choice statement.

successor states. This corresponds to the weighted sum for the weakest pre-expectation:

$$
\begin{aligned}
& \mathit{ExpRew}^{\mathcal{R}_f[\![P\,[p]\,Q]\!]}(\langle P\,[p]\,Q, \eta\rangle \models \Diamond (P\,[p]\,Q)^{\surd}) \\
= \; & \inf_{\mathfrak{P}} \sum_{\widehat{\pi} \in \mathit{Paths}^{\mathfrak{P}}_{\min}(\langle P\,[p]\,Q, \eta\rangle, \Diamond (P\,[p]\,Q)^{\surd})} \mathbf{P}^{\mathfrak{P}}(\widehat{\pi}) \cdot r_{(P\,[p]\,Q)^{\surd}}(\widehat{\pi}) \\
= \; & \inf_{\mathfrak{P}} \sum_{\widehat{\pi} \in \mathit{Paths}^{\mathfrak{P}}_{\min}(\langle P, \eta\rangle, \Diamond P^{\surd})} p \cdot \mathbf{P}^{\mathfrak{P}}(\widehat{\pi}) \cdot r_{P^{\surd}}(\widehat{\pi}) \\
& \quad + \sum_{\widehat{\pi} \in \mathit{Paths}^{\mathfrak{P}}_{\min}(\langle Q, \eta\rangle, \Diamond Q^{\surd})} (1-p) \cdot \mathbf{P}^{\mathfrak{P}}(\widehat{\pi}) \cdot r_{Q^{\surd}}(\widehat{\pi}) \\
\overset{*}{=} \; & p \cdot \inf_{\mathfrak{P}_1} \sum_{\widehat{\pi} \in \mathit{Paths}^{\mathfrak{P}_1}_{\min}(\langle P, \eta\rangle, \Diamond P^{\surd})} \mathbf{P}^{\mathfrak{P}}(\widehat{\pi}) \cdot r_{P^{\surd}}(\widehat{\pi}) \\
& \quad + (1-p) \cdot \inf_{\mathfrak{P}_2} \sum_{\widehat{\pi} \in \mathit{Paths}^{\mathfrak{P}_2}_{\min}(\langle Q, \eta\rangle, \Diamond Q^{\surd})} \mathbf{P}^{\mathfrak{P}}(\widehat{\pi}) \cdot r_{Q^{\surd}}(\widehat{\pi}) \\
= \; & p \cdot \mathit{ExpRew}^{\mathcal{R}_f[\![P]\!]}(\langle P, \eta\rangle \models \Diamond P^{\surd}) \\
& \quad + (1-p) \cdot \mathit{ExpRew}^{\mathcal{R}_f[\![Q]\!]}(\langle Q, \eta\rangle \models \Diamond Q^{\surd}) \\
\overset{I.H.}{=} \; & p \cdot \mathit{wp}(P, f)(\eta) + (1-p) \cdot \mathit{wp}(Q, f)(\eta) \\
= \; & \mathit{wp}(P\,[p]\,Q, f)(\eta)
\end{aligned}
$$

In $*$ we use the fact that the policy for paths starting in $\langle P, \eta\rangle$ is independent of the policy for paths starting in $\langle Q, \eta\rangle$ because they are positional policies.

- Consider the non-deterministic choice $P \,[\!]\, Q$ which is analogous to probabilistic choice, except that min replaces the weighted sum:

$$
\begin{aligned}
& \mathit{ExpRew}^{\mathcal{R}_f[\![P \,[\!]\, Q]\!]}(\langle P \,[\!]\, Q, \eta\rangle \models \Diamond (P \,[\!]\, Q)^{\surd}) \\
= \; & \inf_{\mathfrak{P}} \sum_{\widehat{\pi} \in \mathit{Paths}^{\mathfrak{P}}_{\min}(\langle P \,[\!]\, Q, \eta\rangle, \Diamond (P \,[\!]\, Q)^{\surd})} \mathbf{P}^{\mathfrak{P}}(\widehat{\pi}) \cdot r_{(P \,[\!]\, Q)^{\surd}}(\widehat{\pi}) \\
= \; & \min \left\{ \inf_{\mathfrak{P}} \sum_{\widehat{\pi} \in \mathit{Paths}^{\mathfrak{P}}_{\min}(\langle P, \eta\rangle, \Diamond P^{\surd})} \mathbf{P}^{\mathfrak{P}}(\widehat{\pi}) \cdot r_{P^{\surd}}(\widehat{\pi}) \;, \right. \\
& \qquad \left. \inf_{\mathfrak{P}} \sum_{\widehat{\pi} \in \mathit{Paths}^{\mathfrak{P}}_{\min}(\langle Q, \eta\rangle, \Diamond Q^{\surd})} \mathbf{P}^{\mathfrak{P}}(\widehat{\pi}) \cdot r_{Q^{\surd}}(\widehat{\pi}) \right\} \\
= \; & \min\{ \mathit{ExpRew}^{\mathcal{R}_f[\![P]\!]}(\langle P, \eta\rangle \models \Diamond P^{\surd}), \\
& \qquad \mathit{ExpRew}^{\mathcal{R}_f[\![Q]\!]}(\langle Q, \eta\rangle \models \Diamond Q^{\surd}) \} \\
\overset{I.H.}{=} \; & \min\{ \mathit{wp}(P, f), \mathit{wp}(Q, f) \} \\
= \; & \mathit{wp}(P [\!] Q, f)(\eta)
\end{aligned}
$$

- Consider the sequential composition $P;Q$. The idea is break up each path into a prefix that corresponds to the execution of P and a suffix that corresponds to the execution of Q. We can then compute the expected reward over the suffixes and use this intermediate result to compute the expected reward over the prefixes which corresponds to the nesting of weakest pre-expectations:

$$
\begin{aligned}
& \mathit{ExpRew}^{\mathcal{R}_f[\![\, P;Q \,]\!]}(\langle P;Q,\eta\rangle \models \Diamond (P;Q)^{\checkmark}) \\
&= \inf_{\mathfrak{P}} \sum_{\widehat{\pi}\in \mathit{Paths}^{\mathfrak{P}}_{\min}(\langle P;Q,\eta\rangle, \Diamond (P;Q)^{\checkmark})} \mathbf{P}^{\mathfrak{P}}(\widehat{\pi}) \cdot r_{(P;Q)^{\checkmark}}(\widehat{\pi}) \\
&\overset{*}{=} \inf_{\mathfrak{P}} \sum_{\widehat{\pi}\in \mathit{Paths}^{\mathfrak{P}}_{\min}(\langle P;Q,\eta\rangle, \Diamond P^{\checkmark})} \mathbf{P}^{\mathfrak{P}}(\widehat{\pi}) \cdot r^{q}_{P^{\checkmark}}(\widehat{\pi}) \\
&\quad \text{where } r^{q}_{P^{\checkmark}}(\widehat{\pi}) \text{ is the sum of rewards } r_q \text{ along } \widehat{\pi} \text{ with} \\
&\quad r_q(s) = \inf_{\mathfrak{P}'} \left(\sum_{\widehat{\pi}'\in \mathit{Paths}^{\mathfrak{P}'}_{\min}(s, \Diamond Q^{\checkmark})} \mathbf{P}^{\mathfrak{P}'}(\widehat{\pi}') \cdot r_{Q^{\checkmark}}(\widehat{\pi}') \right) \\
&\quad \text{if } s \in P^{\checkmark} \text{and } r_q(s) = 0 \text{ otherwise} \\
&= \mathit{ExpRew}^{\mathcal{R}_g[\![\, P \,]\!]}(\langle P,\eta\rangle \models \Diamond P^{\checkmark}) \\
&\quad \text{where } g(\eta) = \mathit{ExpRew}^{\mathcal{R}_f[\![\, Q \,]\!]}(\langle Q,\eta\rangle \models \Diamond Q^{\checkmark}) \\
&\overset{I.H.}{=} \mathit{wp}(P; \mathit{wp}(Q, f))(\eta) \\
&= \mathit{wp}(P; Q, f)(\eta) \ .
\end{aligned}
$$

  In $*$ we divide each path into the aforementioned pre- and suffixes. We use the positionality of policies as the policies according to which the suffixes are constructed are independent of the history, i.e. the prefix of those paths.

- Consider the loop $\mathsf{while}(G)\{P\}$. For this case we show by induction that the two semantics correspond for every iteration that the loop performs. We rely on the previously shown cases for abort, skip and sequential composition. Let the bounded while-loop for $k > 0$ be

$$
(\mathsf{while}(G)\{P\})^{k+1} = \mathsf{if}(G)\{P; (\mathsf{while}(G)\{P\})^{k}\} \ \mathsf{else}\ \{\mathsf{skip}\}
$$

  where the base case is $(\mathsf{while}(G)\{P\})^{0} = \mathsf{abort}$. We will show for every k that

$$
\mathit{wp}((\mathsf{while}(G)\{P\})^{k}, f)(\eta) = \mathit{ExpRew}^{\mathcal{R}_f[\![\, (\mathsf{while}(G)\{P\})^{k} \,]\!]}(\eta) \ . \tag{1}
$$

  Observe that

$$
\mathit{wp}((\mathsf{while}(G)\{P\})^{k+1}, f)(\eta) \geq \mathit{wp}((\mathsf{while}(G)\{P\})^{k}, f)(\eta) \ .
$$

From the fixpoint theorem 3 in [18] we know that the more iterations the bounded while loop is allowed to perform, the closer it approximates the fixpoint given in Def. 15. Formally this means

$$\lim_{k\to\infty} wp((\mathsf{while}(G)\{P\})^k, f)(\eta) = wp(\mathsf{while}(G)\{P\}, f)(\eta) \ . \qquad (2)$$

A thorough justification for (2) is given in Appendix A.

From (1) it follows that for every k, *ExpRew* behaves identically to *wp*. Thus with (2) it follows that

$$wp(\mathsf{while}(G)\{P\}, f)(\eta) = ExpRew^{\mathcal{R}_f[\![\,\mathsf{while}(G)\{P\}\,]\!]}(\eta).$$

It remains to prove (1). This is done by induction on k. Base case ($k = 0$):

$$\begin{aligned} & wp((\mathsf{while}(G)\{P\})^0, f)(\eta) \\ =\ & wp(\mathsf{abort}, f)(\eta) \\ \overset{*}{=}\ & ExpRew^{\mathcal{R}_f[\![\,\mathsf{abort}\,]\!]}(\eta) \\ =\ & ExpRew^{\mathcal{R}_f[\![\,(\mathsf{while}(G)\{P\})^0\,]\!]}(\eta) \end{aligned}$$

$(*)$ was already shown earlier in the case abort.

Induction hypothesis: equation (1) holds for some unspecified but fixed value of k.

Induction step:

$$\begin{aligned} & wp((\mathsf{while}(G)\{P\})^{k+1}, f)(\eta) \\ =\ & wp(\mathsf{if}(G)\{P; (\mathsf{while}(G)\{P\})^k\}\mathsf{else}\{\mathsf{skip}\}, f)(\eta) \\ =\ & \big([G]\cdot wp(P; (\mathsf{while}(G)\{P\})^k, f) + [\neg G]\cdot wp(\mathsf{skip}, f)\big)(\eta) \\ \overset{*}{=}\ & \Big([G]\cdot ExpRew^{\mathcal{R}_f[\![\,P;(\mathsf{while}(G)\{P\})^k\,]\!]} + [\neg G]\cdot ExpRew^{\mathcal{R}_f[\![\,\mathsf{skip}\,]\!]}\Big)(\eta) \\ =\ & ExpRew^{\mathcal{R}_f[\![\,\mathsf{if}(G)\{P;(\mathsf{while}(G)\{P\})^k\}\mathsf{else}\{\mathsf{skip}\}\,]\!]}(\eta) \\ =\ & ExpRew^{\mathcal{R}_f[\![\,(\mathsf{while}(G)\{P\})^{k+1}\,]\!]}(\eta) \end{aligned}$$

$(*)$ follows from the induction hypothesis and the previously shown cases for skip and sequential composition.

■

Thus, $wp(P, f)$ evaluated at η is the least expected value of f over any of the result distributions of P.

Theorem 24. (Correspondence theorem for liberal semantics) For pGCL-program P, variable valuation η, and post-expectation f:

$$wlp(P, f)(\eta) \ = \ LExpRew^{\mathcal{R}_f[\![\,P\,]\!]}(\langle P, \eta\rangle \models \Diamond P^{\checkmark}) \ .$$

■

Proof: By structural induction over the pGCL program P (analogously to the proof of Theorem 23). Similarly we apply Lem. 22 here. To avoid repetition we skip the base cases which are rather simple.
Induction hypothesis: assume that for program P (and analogously for Q)

$$wlp(P,f)(\eta) = LExpRew^{\mathcal{R}_f[\![\,P\,]\!]}(\langle P,\eta\rangle \models \Diamond P^{\surd}) \ .$$

Induction step:

- Consider the probabilistic choice $P\,[p]\,Q$ (again, this covers conditional choice):

$$\begin{aligned}
&LExpRew^{\mathcal{R}_f[\![\,P\,[p]\,Q\,]\!]}(\langle P\,[p]\,Q,\eta\rangle \models \Diamond(P\,[p]\,Q)^{\surd}) \\
&= \inf_{\mathfrak{P}} \ Pr^{\mathfrak{P}}(\langle P\,[p]\,Q,\eta\rangle \not\models \Diamond(P\,[p]\,Q)^{\surd}) \\
&\quad + \sum_{\widehat{\pi}\in Paths^{\mathfrak{P}}_{\min}(\langle P\,[p]\,Q,\eta\rangle,\Diamond(P\,[p]\,Q)^{\surd})} \mathbf{P}^{\mathfrak{P}}(\widehat{\pi})\cdot r_{(P\,[p]\,Q)^{\surd}}(\widehat{\pi}) \\
&= \inf_{\mathfrak{P}} \Big(+p\cdot Pr^{\mathfrak{P}}(\langle P,\eta\rangle \not\models \Diamond P^{\surd}) \\
&\quad + \sum_{\widehat{\pi}\in Paths^{\mathfrak{P}}_{\min}(\langle P,\eta\rangle,\Diamond P^{\surd})} p\cdot \mathbf{P}^{\mathfrak{P}}(\widehat{\pi})\cdot r_{P^{\surd}}(\widehat{\pi}) \\
&\quad + \ (1-p)\cdot Pr^{\mathfrak{P}}(\langle Q,\eta\rangle \not\models \Diamond Q^{\surd}) \\
&\quad + \sum_{\widehat{\pi}\in Paths^{\mathfrak{P}}_{\min}(\langle Q,\eta\rangle,\Diamond Q^{\surd})} (1-p)\cdot \mathbf{P}^{\mathfrak{P}}(\widehat{\pi})\cdot r_{Q^{\surd}}(\widehat{\pi}) \Big) \\
&= p\cdot \inf_{\mathfrak{P}_1} \ Pr^{\mathfrak{P}_1}(\langle P,\eta\rangle \not\models \Diamond P^{\surd}) \\
&\quad + \sum_{\widehat{\pi}\in Paths^{\mathfrak{P}_1}_{\min}(\langle P,\eta\rangle,\Diamond P^{\surd})} \mathbf{P}^{\mathfrak{P}}(\widehat{\pi})\cdot r_{P^{\surd}}(\widehat{\pi}) \\
&\quad + (1-p)\cdot \inf_{\mathfrak{P}_2} \ Pr^{\mathfrak{P}_2}(\langle Q,\eta\rangle \not\models \Diamond Q^{\surd}) \\
&\quad + \sum_{\widehat{\pi}\in Paths^{\mathfrak{P}_2}_{\min}(\langle Q,\eta\rangle,\Diamond Q^{\surd})} \mathbf{P}^{\mathfrak{P}}(\widehat{\pi})\cdot r_{Q^{\surd}}(\widehat{\pi}) \\
&= \quad p\cdot LExpRew^{\mathcal{R}_f[\![\,P\,]\!]}(\langle P,\eta\rangle \models \Diamond P^{\surd}) \\
&\quad + (1-p)\cdot LExpRew^{\mathcal{R}_f[\![\,Q\,]\!]}(\langle Q,\eta\rangle \models \Diamond Q^{\surd}) \\
&\overset{I.H.}{=} p\cdot wlp(P,f)(\eta) + (1-p)\cdot wlp(Q,f)(\eta) \\
&= wlp(P\,[p]\,Q,f)(\eta) \ .
\end{aligned}$$

- Consider the non-deterministic choice $P \,[]\, Q$:

$$
\begin{aligned}
& \mathit{LExpRew}^{\mathcal{R}_f[\![P \,[]\, Q]\!]}(\langle P \,[]\, Q, \eta\rangle \models \Diamond (P \,[]\, Q)^{\surd}) \\
= \; & \inf_{\mathfrak{P}} \; \mathit{Pr}^{\mathfrak{P}}(\langle P \,[]\, Q, \eta\rangle \not\models \Diamond (P \,[]\, Q)^{\surd}) \\
& \quad + \sum_{\widehat{\pi} \in \mathit{Paths}^{\mathfrak{P}}_{\min}(\langle P \,[]\, Q, \eta\rangle, \Diamond (P \,[]\, Q)^{\surd})} \mathbf{P}^{\mathfrak{P}}(\widehat{\pi}) \cdot r_{(P \,[]\, Q)^{\surd}}(\widehat{\pi}) \\
= \; & \min \left\{ \inf_{\mathfrak{P}} \left(\mathit{Pr}^{\mathfrak{P}}(\langle P, \eta\rangle \not\models \Diamond P^{\surd}) + \sum_{\widehat{\pi} \in \mathit{Paths}^{\mathfrak{P}}_{\min}(\langle P, \eta\rangle, \Diamond P^{\surd})} \mathbf{P}^{\mathfrak{P}}(\widehat{\pi}) \cdot r_{P^{\surd}}(\widehat{\pi}) \right), \right. \\
& \qquad \left. \inf_{\mathfrak{P}} \left(\mathit{Pr}^{\mathfrak{P}}(\langle Q, \eta\rangle \not\models \Diamond Q^{\surd}) + \sum_{\widehat{\pi} \in \mathit{Paths}^{\mathfrak{P}}_{\min}(\langle Q, \eta\rangle, \Diamond Q^{\surd})} \mathbf{P}^{\mathfrak{P}}(\widehat{\pi}) \cdot r_{Q^{\surd}}(\widehat{\pi}) \right) \right\} \\
= \; & \min\{ \mathit{LExpRew}^{\mathcal{R}_f[\![P]\!]}(\langle P, \eta\rangle \models \Diamond P^{\surd}), \\
& \qquad \mathit{LExpRew}^{\mathcal{R}_f[\![Q]\!]}(\langle Q, \eta\rangle \models \Diamond Q^{\surd}) \} \\
\overset{I.H.}{=} \; & \min\{ \mathit{wlp}(P, f)(\eta), \mathit{wlp}(Q, f)(\eta) \} \\
= \; & \mathit{wlp}(P [] Q, f)(\eta) \;\; .
\end{aligned}
$$

- Consider the sequential composition $P; Q$:

$$
\begin{aligned}
& \mathit{LExpRew}^{\mathcal{R}_f[\![P;Q]\!]}(\langle P; Q, \eta\rangle \models \Diamond (P; Q)^{\surd}) \\
= \; & \inf_{\mathfrak{P}} \Big(\mathit{Pr}^{\mathfrak{P}}\{\langle P; Q, \eta\rangle \not\models \Diamond (P; Q)^{\surd}\} \\
& \quad + \sum_{\widehat{\pi} \in \mathit{Paths}^{\mathfrak{P}}_{\min}(\langle P;Q, \eta\rangle, \Diamond (P;Q)^{\surd})} \mathbf{P}^{\mathfrak{P}}(\widehat{\pi}) \cdot r_{(P;Q)^{\surd}}(\widehat{\pi}) \Big) \\
\overset{*}{=} \; & \inf_{\mathfrak{P}} \Big(\mathit{Pr}^{\mathfrak{P}}\{\langle P, \eta\rangle \not\models \Diamond P^{\surd}\} \\
& \quad + \sum_{\widehat{\pi} \in \mathit{Paths}^{\mathfrak{P}}_{\min}(\langle P;Q, \eta\rangle, \Diamond P^{\surd})} \mathbf{P}^{\mathfrak{P}}(\widehat{\pi}) \cdot r^{q}_{P^{\surd}}(\widehat{\pi}) \Big) \\
& \text{where } r^{q}_{P^{\surd}}(\widehat{\pi}) \text{ is the sum of rewards } r_q \text{ along } \widehat{\pi} \text{ with} \\
& r_q(s) = \inf_{\mathfrak{P}'} \Big(\mathit{Pr}^{\mathfrak{P}'}\{\langle Q, \eta'\rangle \not\models \Diamond Q^{\surd}\} \\
& \qquad + \sum_{\widehat{\pi}' \in \mathit{Paths}^{\mathfrak{P}'}_{\min}(s, \Diamond Q^{\surd})} \mathbf{P}^{\mathfrak{P}'}(\widehat{\pi}') \cdot r_{Q^{\surd}}(\widehat{\pi}') \Big) \\
& \text{if } s \in P^{\surd} \text{and } r_q(s) = 0 \text{ otherwise} \\
= \; & \mathit{LExpRew}^{\mathcal{R}_g[\![P]\!]}(\langle P, \eta\rangle \models \Diamond P^{\surd}) \\
& \text{where } g(\eta) = \mathit{LExpRew}^{\mathcal{R}_f[\![Q]\!]}(\langle Q, \eta\rangle \models \Diamond Q^{\surd})
\end{aligned}
$$

$$\stackrel{I.H.}{=} wlp(P; wlp(Q, f))(\eta)$$
$$= wlp(P; Q, f)(\eta) \ .$$

In $*$ we again rewrite each path into a prefix and a suffix and use positionality of policies. Additionally, observe that diverging paths are also split up into paths that already diverge before reaching an exit state of P and paths that do reach the end of P but diverge before reaching an exit state of Q. The probability of the former is captured by $Pr^{\mathfrak{P}}\{\langle P, \eta\rangle \not\models \Diamond P^{\checkmark}\}$ and the probability of the latter is the product of the probability of the prefix and the suffix whose probability is captured by $r^{q}_{P^{\checkmark}}$.

- Consider the loop $\mathsf{while}(G)\{P\}$. Again we prove this case by induction on the number of iterations that a while-loop performs. Let $(\mathsf{while}(G)\{P\})^k$ be defined as in the proof of the previous theorem. We show for every k that

$$wlp((\mathsf{while}(G)\{P\})^k, f)(\eta) = LExpRew^{\mathcal{R}_f[\![\,(\mathsf{while}(G)\{P\})^k\,]\!]}(\eta) \ . \quad (3)$$

The only difference is now that

$$wlp((\mathsf{while}(G)\{P\})^{k+1}, f)(\eta) \leq wlp((\mathsf{while}(G)\{P\})^k, f)(\eta) \ .$$

Using this we again know that the bounded while loop approximates the fixpoint given in Def. 16 (only this time from above). Formally this means

$$\lim_{k\to\infty} wlp((\mathsf{while}(G)\{P\})^k, f)(\eta) = wlp(\mathsf{while}(G)\{P\}, f)(\eta) \ . \quad (4)$$

From (3) we know that for every k *LExpRew* behaves identically to *wlp*. Thus with (4) it follows that

$$wlp(\mathsf{while}(G)\{P\}, f)(\eta) = LExpRew^{\mathcal{R}_f[\![\,\mathsf{while}(G)\{P\}\,]\!]}(\eta) \ .$$

It remains to prove (3). This is done by induction on k. Base case ($k = 0$):

$$wlp((\mathsf{while}(G)\{P\})^0, f)(\eta)$$
$$= wlp(\mathsf{abort}, f)(\eta)$$
$$\stackrel{*}{=} LExpRew^{\mathcal{R}_f[\![\,\mathsf{abort}\,]\!]}(\eta)$$
$$= LExpRew^{\mathcal{R}_f[\![\,(\mathsf{while}(G)\{P\})^0\,]\!]}(\eta)$$

$(*)$ was already shown earlier in the case abort.

Induction hypothesis: equation (3) holds for some unspecified but fixed value of k.

Induction step:

$$
\begin{aligned}
& wlp((\mathsf{while}(G)\{P\})^{k+1}, f)(\eta) \\
= \; & wlp(\mathsf{if}(G)\{P; (\mathsf{while}(G)\{P\})^k\}\mathsf{else}\{\mathsf{skip}\}, f)(\eta) \\
= \; & \left([G] \cdot wlp(P; (\mathsf{while}(G)\{P\})^k, f) + [\neg G] \cdot wlp(\mathsf{skip}, f)\right)(\eta) \\
\overset{*}{=} \; & \left([G] \cdot LExpRew^{\mathcal{R}_f[\![P;(\mathsf{while}(G)\{P\})^k]\!]} + [\neg G] \cdot LExpRew^{\mathcal{R}_f[\![\mathsf{skip}]\!]}\right)(\eta) \\
= \; & LExpRew^{\mathcal{R}_f[\![\mathsf{if}(G)\{P;(\mathsf{while}(G)\{P\})^k\}\mathsf{else}\{\mathsf{skip}\}]\!]}(\eta) \\
= \; & LExpRew^{\mathcal{R}_f[\![(\mathsf{while}(G)\{P\})^{k+1}]\!]}(\eta)
\end{aligned}
$$

$(*)$ follows from the induction hypothesis and the previously shown cases for skip and sequential composition.

■

The weakest liberal pre-expectation $wlp(P, f)$ is thus the least expected value of f over any of the result distributions of P plus the probability that P does not terminate.

Example 25. (Duelling cowboys.) Consider again the duelling cowboys example from Lst. 1. Assume we are interested in the probability that cowboy A wins the duel. In terms of the MDP semantics this means we are interested in

$$ExpRew^{(\mathcal{M},r)}(\langle 2, *, * \rangle \models \Diamond(\mathcal{M}, r)^{\checkmark})$$

where $\mathcal{M}$ is the MDP from Fig. 1 and r is the reward function that indicates whether cowboy A has won or not, i.e.

$$r(s) = \begin{cases} 1 & \text{if } s = \langle 11, A, 0 \rangle \\ 0 & \text{otherwise} \end{cases}$$

In this example the MDP is finite and this allows us to compute the desired expected cumulative reward easily. That is, cowboy A wins with probability at least

$$\frac{(1-b)a}{a+b-ab} \; .$$

Figure 2 visualises this result. We nicely see how the expected winning chance depends on a and b. Parametrising our model allows us to carry out a calculation only once and make statements about all possible refinements of the system. According to Theorem 23 we can obtain the same result when applying the expectation transformer mechanism. The following section illustrates how this works. ■

5. Analysis

Although the computation of (liberal) expected rewards on MDPs may be numerically involved, its basic idea is intuitive in principle. However, `pGCL`

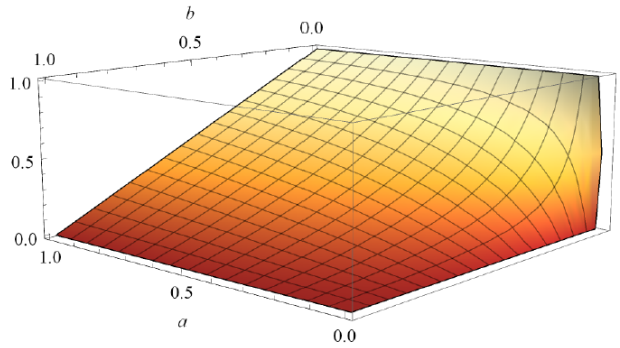


Figure 2: Probability that A wins the duel, depending on a and b. Bear in mind that this is the least guaranteed probability that A wins. In the worst case (for A) cowboy B will shoot first and therefore as b tends to 1 the plot goes to 0, i.e. cowboy A has no chances. However for smaller values of b the influence of a increases.

programs will often have an infinite state space in particular due to the infinite domain of the program variables. In that case it is not possible to compute the expected reward on the reward model in general. In contrast to this, the denotational semantics does not depend on the underlying state space but on the structure of the program. In this section we show how to determine a pre-expectation using *wp*-semantics.

Again let us determine the probability that cowboy A wins the duel. Therefore we choose $[t = A]$ as the post-expectation and determine

$$wp(\text{cowboyDuel}, [t = A]) \ .$$

Listing 2 shows the program cowboyDuel with annotations.

Listing 2: The duelling cowboys, annotated with expectations

```
1  ⟨(1−b)a / a+b−ab⟩
2  ⟨min{a / a+b−ab, (1−b)a / a+b−ab}⟩
3  (t := A [] t := B);
4  ⟨[t = A] · a / a+b−ab + [t = B] · (1−b)a / a+b−ab⟩
5  c := 1;
6  ⟨[t = A ∧ c = 0] · 1 + [t = A ∧ c = 1] · a / a+b−ab  +[t = B ∧ c = 1] · (1−b)a / a+b−ab⟩
7  while (c = 1) {
8    ⟨[t = A ∧ c = 1] · a / a+b−ab + [t = B ∧ c = 1] · (1−b)a / a+b−ab⟩
9    ⟨[t = A ∧ c ≠ 1] · a + [t = A ∧ c = 1] · a / a+b−ab
         +[t = B ∧ c = 0] · (1 − b) + [t = B ∧ c = 1] · (1−b)a / a+b−ab⟩
10   if (t = A) {
11     (c := 0 [a] t := B);
12   } else {
13     (c := 0 [b] t := A);
14   }
15   ⟨[t = A ∧ c = 0] · 1 + [t = A ∧ c = 1] · a / a+b−ab  +[t = B ∧ c = 1] · (1−b)a / a+b−ab⟩
16 }
```

17 $\langle [c \neq 1] \cdot \Big([t = A \wedge c = 0] \cdot 1 + [t = A \wedge c = 1] \cdot \frac{a}{a+b-ab}$
$+ [t = B \wedge c = 1] \cdot \frac{(1-b)a}{a+b-ab} \Big) \rangle$

18 $\langle [t = A] \rangle$

The program is annotated backwards according to the rules from Def. 15. In line 18 we start with the post-expectation that we are interested in. We finish with the sought probability in line 1. The only non-trivial step is to discover the so-called *invariant* which appears in lines 6 and 15. But let us assume for the moment that it is given. Then all other annotations are obtained by applying the syntactic rules from Def. 15 and rewriting. We can rewrite the expectation in line 9 into the expectation in line 8 because at this point the program must be in a state where $c = 1$ (the loop guard) holds and the expectations are equivalent for all these states. The same applies to expectations in lines 17 and 18 because at that point $c \neq 1$ holds. The calculation from line 15 to line 9 was already shown in Example 17. This means that once we have found the aforementioned invariant, the analysis can be automatically carried out by a computer - irrespective of the underlying state space size.

6. Invariants

The annotation in lines 6 and 15 of Lst. 2 which we call an invariant is an expectation that over-approximates the fixed point solution in Def. 15. In the following we use $\mathcal{I}$ for invariant expectations.

Definition 26. (Probabilistic invariant) An expectation $\mathcal{I}$ is called invariant for a loop $\mathsf{while}(G)\{P\}$ if

$$\mathcal{I} \cdot [G] \leq wlp(P, \mathcal{I}) \ . \tag{5}$$

■

In our example, $\mathcal{I}$ is the expectation in line 6, G is the loop guard $c = 1$ and loop body P is the code in lines 10–14. In line 8, the expectation represents $\mathcal{I} \cdot [G]$ and line 9 is $wlp(P, \mathcal{I})$. Clearly, (5) is satisfied in our example.

6.1. Standard Invariants

Before we elaborate more on invariants for probabilistic programs, let us consider invariants for traditional, non-probabilistic programs and how they are used. For this we remind the reader of the non-probabilistic version of Def. 26 along the lines of [1].

Definition 27. (Standard invariant) A predicate $\mathcal{I}$ is called invariant for a loop $\mathsf{while}(G)\{P\}$ if

$$\mathcal{I} \wedge G \Rightarrow wlp(P, \mathcal{I}) \ . \tag{6}$$

■

In this definition a single iteration of the loop body P is considered. The implication ensures that an execution of the loop body preserves the validity of

$\mathcal{I}$. Note that G appears in the premise because we restrict our attention to states from which the loop will perform (at least) one iteration. States characterised by $\mathcal{I} \wedge \neg G$ are irrelevant because the loop will be skipped and one can trivially conclude that at the end of the loop's execution $\mathcal{I}$ is still true. Since (6) has to be satisfied on every iteration of the loop it follows that any execution beginning in a state that satisfies the invariant will terminate in a state that again satisfies the invariant (or the execution of the loop does not terminate). This property is colloquially summarised as "the set of states characterised by $\mathcal{I}$ is not left by the execution of the loop".

The key motivation for invariant annotations is that they establish the following relationship:

$$\mathcal{I} \Rightarrow wlp(\mathsf{while}(G)\{P\}, \mathcal{I} \wedge \neg G) \ .$$

This relationship is called *partial correctness*. It means that every execution of the loop from a state satisfying the invariant can only terminate in a state that also satisfies the invariant and violates the guard G. The correctness is partial because it is possible that there are some executions which never terminate. In a separate proof, e.g. using a loop variant, one can establish that the loop terminates when started in some state in $\mathcal{I}$. This gives us *total correctness*:

$$\mathcal{I} \Rightarrow wp(\mathsf{while}(G)\{P\}, \mathcal{I} \wedge \neg G) \ .$$

In practice one usually wants to prove that given some precondition *pre* before the beginning of the loop, the postcondition *post* will hold after the loop's execution. The straightforward way is to show this by directly applying *wp* semantics, i.e. proving

$$pre \Rightarrow wp(\mathsf{while}(G)\{P\}, post) \ .$$

But it turns out to be hard because this requires to find the least fixed point of the loop with respect to *post*. Although that fixed point is mathematically well-defined it often is difficult to compute it in practice. Instead it usually[4] is easier to

1. find a predicate $\mathcal{I}$ such that

$$pre \Rightarrow \mathcal{I} \quad \text{and} \quad \mathcal{I} \wedge \neg G \Rightarrow post \ ,$$

2. show $\mathcal{I}$ is invariant for the loop $\mathsf{while}(G)\{P\}$, cf. Def. 27 and
3. prove that the loop terminates from any state in $\mathcal{I} \wedge G$.

Via this detour the same relation between *pre* and *post* is established as

$$pre \Rightarrow \mathcal{I} \Rightarrow wp(\mathsf{while}(G)\{P\}, \mathcal{I} \wedge \neg G) \quad \text{and} \quad \mathcal{I} \wedge \neg G \Rightarrow post \ .$$

[4]One can construct a loop and pre- and postconditions such that the alternative approach turns out to be as hard as finding the fixed point. In practice however there is a big difference.

6.2. Probabilistic Invariants

Let us now return to probabilistic programs. In the probabilistic setting, we use an invariant $\mathcal{I}$ in the same way but this time it is an expectation. The post-expectation $\mathcal{I} \cdot [\neg G]$ has the pre-expectation $\mathcal{I}$.

However there is a crucial difference between non-probabilistic and probabilistic programs. Once we have shown that the loop in a non-probabilistic program terminates we have at the same time established that the set of reachable states is finite. This is because in a non-probabilistic program there may be several different executions from a given state due to non-determinism but the proof of termination shows that there are only finitely many emanating executions and each of them has finite length. For probabilistic programs the situation is somewhat different. Consider the example in Lst. 3 below.

Listing 3: Symmetric random walk over $\mathbb{N}$ with absorbing barrier at zero [4].

```
1 n := 1;
2 while(n != 0){
3   (n := n - 1 [0.5] n := n + 1);
4 }
```

This loop will terminate with probability one[5] because the probability of an infinite walk is zero. However there exist infinitely many different walks of finite length from the initial state. Each of these walks has a positive probability. To convince ourselves that this indeed makes a difference we choose the invariant $\mathcal{I} = n$. It does satisfy Def. 26. Hence,

$$n \leq \mathit{wlp}(\mathsf{while}(n \neq 0)\{n := n - 1\,[0.5]\,n := n + 1\}, n{\cdot}[n = 0]) \ .$$

And we have already shown that the loop terminates almost surely. We could therefore falsely conclude that

$$n \leq \mathit{wp}(\mathsf{while}(n \neq 0)\{n := n - 1\,[0.5]\,n := n + 1\}, n{\cdot}[n = 0]) \ .$$

This would "prove" that the expected value of $n{\cdot}[n = 0]$ depends on the initial value of n and in the given example it is one. This of course is wrong as $n{\cdot}[n = 0]$ is zero everywhere. It is a nice exercise to compute the fixed point of this loop w.r.t. n. It gives a function that evaluates to zero for every n which coincides with the intuition that the expected outcome, in fact the only possible one, is zero. This also nicely shows that establishing termination is not the same as establishing termination with probability one.

In conclusion, given a probabilistic loop $\mathsf{while}(G)\{P\}$ and a post-expectation *post*, we can establish an upper bound for pre-expectation *pre* if we

1. find an expectation $\mathcal{I}$ such that

$$\mathit{pre} \leq \mathcal{I} \quad \text{and} \quad \mathcal{I} \wedge \neg G \leq \mathit{post} \ ,$$

[5] Also referred to as *almost* sure termination.

2. show $\mathcal{I}$ is invariant for the loop $\mathsf{while}(G)\{P\}$, cf. Def. 26,
3. prove that the loop terminates from any state in G with probability one and
4. either show that from every initial state of the loop only a finite state space is reachable
   or make sure that $\mathcal{I}$ is bounded from above by some fixed constant
   or show that $wp(P, \mathcal{I} \cdot [G])$ tends to zero as the number of iterations tends to infinity.

The last item gives sufficient conditions to make reasoning with invariants sound for probabilistic programs [4, pp. 71-72].

Even though we argue that finding invariants and proving termination separately is easier than computing the least fixed point of the loop directly, it still remains a hard task to find a non-trivial invariant. Trivial invariants are constant functions such as 0 or 1 which are invariant for every loop but will hardly be useful for calculating an expectation. The invariant generation process is a topic on its own and beyond the scope of this paper. For probabilistic programs we have developed a constraint-based approach to generate invariants [19]. These ideas were implemented in our recently developed prototype tool Prinsys [12]. It helps the user to find invariants for probabilistic programs semi-automatically. For instance, it was applied to our running example, the duelling cowboys, to calculate cowboy A's winning probability in Sect. 5.

In the rest of this section we apply our theoretical set-up to obtain an operational view of invariants for probabilistic systems.

6.3. Characterising invariants operationally

Recall that invariants for non-probabilistic programs are predicates, and they characterise the set of states that is never left by a loop as discussed before. This yields a straightforward operational characterisation of invariants: in a transition system that represents the loop we can identify a set of states with the property that one iteration of the loop started in such a state will end in this set again. If we can describe this set by a predicate this predicate will satisfy Def. 27.

For probabilistic programs the situation is more complicated. Invariants are expectations and not predicates and therefore do not necessarily characterise states. Instead they assign values to states such that an execution of the loop body started from a state with value e will *on average* end in a state with a value no less than e (if the execution terminates at all). Thus invariants establish lower bounds on pre-expectations. In fact, this idea generalises standard invariants which can be represented as $\{0, 1\}$-valued expectations.

Theorem 24 allows us to give an operational interpretation to invariants for probabilistic programs in terms of MDPs. Given an MDP for a while loop that is constructed according to the rules in Table 1 we can characterise an invariant in the following way:

Corollary 28. (Operational interpretation of quantitative loop invariants.) An expectation $\mathcal{I}$ is invariant for the loop $\mathsf{while}(G)\{P\}$ if in

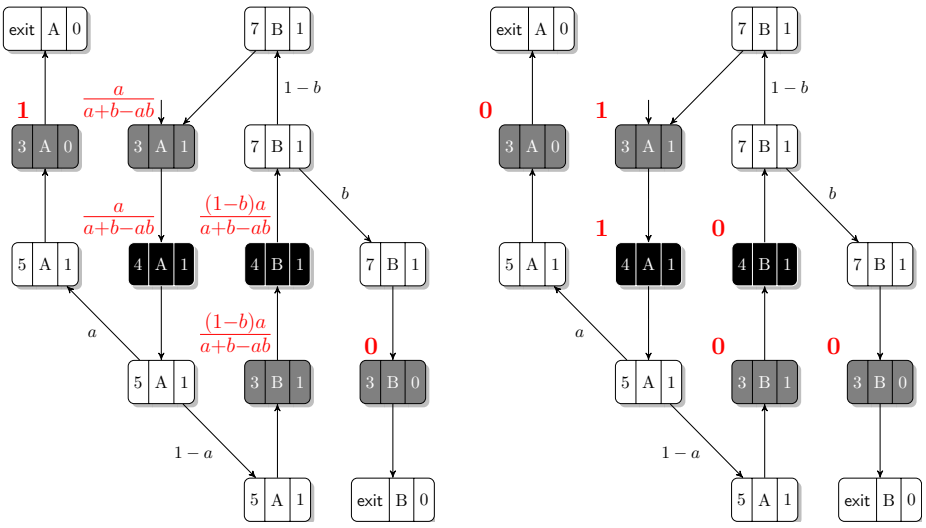
$\mathcal{M}[\![\,\mathsf{while}(G)\{P\}\,]\!]$ for any state s of the form $\langle P; \mathsf{while}(G)\{P\}, \eta\rangle$:

$$\mathcal{I}(\eta) \leq LExpRew^{\mathcal{R}_{\mathcal{I}}[\![\,P\,]\!]}(\langle P, \eta\rangle \models \Diamond P^{\checkmark}) \ .$$

Intuitively this formula requires that the liberal expected reward w.r.t. the execution of the loop body P and the post-expectation $\mathcal{I}$ is bounded from below by $\mathcal{I}$. Corollary 28 is an immediate consequence of Def. 26 and Thm. 24. On the left hand side we evaluate $\mathcal{I}$ on η in those states where the loop has just been entered. This corresponds to the left hand side of the inequality in Def. 26. For the right hand side we have applied Thm. 24 to the right hand side of Def. 26. Thus the operational characterisation of invariants requires an expectation to meet a set of inequality constraints. Any expectation that is a solution to this set of constraints will satisfy Def. 26 as well. The difference between Def. 26 and Cor. 28 is that the former gives one inequality between two functions and this inequality is obtained by applying the *wlp* calculus while the latter gives one inequality for every possible initial state of the loop and each inequality is obtained by calculating a reachability objective in an MDP.

Below is an example for two different expectation functions, one of which is invariant and the other is not.

Example 29. (Invariants)



Both MDPs represent the loop from our running example in Lst. 1. Here the solid black states are of the form $\langle P; \mathsf{while}(G)\{P\}, \eta\rangle$ and the grey shaded states are the ones where one iteration of the loop has finished.

Let us once again consider the expectation

$$f = [t = A \wedge c = 0] \cdot 1 + [t = A \wedge c = 1] \cdot \frac{a}{a+b-ab} + [t = B \wedge c = 1] \cdot \frac{(1-b)a}{a+b-ab} \ .$$

We evaluate this expectation in the black and grey states of the left MDP. According to Cor. 28 the following two inequalities have to be checked:

$$\begin{aligned} f(A,1) = \frac{a}{a+b-ab} &\leq \frac{a}{a+b-ab} \\ &= a \cdot 1 + (1-a) \cdot \frac{(1-b)a}{a+b-ab} \\ &= a \cdot f(A,0) + (1-a) \cdot f(B,1) \quad \checkmark \\ f(B,1) = \frac{(1-b)a}{a+b-ab} &\leq \frac{(1-b)a}{a+b-ab} \\ &= b \cdot 0 + (1-b) \cdot \frac{a}{a+b-ab} \\ &= b \cdot f(B,0) + (1-b) \cdot f(A,1) \quad \checkmark \end{aligned}$$

Both are fulfilled and hence f is invariant.

Now let us choose a different expectation, for instance

$$g = [t = A] \cdot c \ .$$

The expectation g is evaluated on the black and grey states of the right MDP and the inequalities given by Cor. 28 become:

$$\begin{aligned} g(A,1) = 1 &\not\leq 0 \\ &= a \cdot 0 + (1-a) \cdot 0 \\ &= a \cdot g(A,0) + (1-a) \cdot g(B,1) \quad \times \\ g(B,1) = 0 &\leq 1 - b \\ &= b \cdot 0 + (1-b) \cdot 1 \\ &= b \cdot g(B,0) + (1-b) \cdot g(A,1) \quad \checkmark \end{aligned}$$

We find out that g is not invariant because the inequality corresponding to state $\langle 4, A, 1\rangle$ does not hold. ■

In this section we have discussed the purpose of invariants and their properties. Furthermore we have shown an operational characterisation for invariants for probabilistic programs. As mentioned above, in other work [19] we have developed a technique to support the semi-automatic discovery of loop invariants which can be used to verify probabilistic properties of while loops.

7. Conclusion

This paper provides a formal connection between the expectation transformer semantics of pGCL by McIver and Morgan [4] and a simple operational semantics using (parametric) MDPs. This yields an insightful relationship between semantics used for deductive reasoning for probabilistic programs and the notion of a computation in terms of an MDP. Our approach assigns rewards to

terminal states (only), and establishes that expected cumulative rewards correspond to wp-semantics. A slight variant of expected rewards yields a connection to the wlp-semantics.

The presented operational semantics provides a connection to model-checking algorithms for MDPs. In case a pGCL program yields a finite (parameterised) MDP, expected cumulative rewards (and thus weakest pre-expectations) can be computed by solving rational functions [20]. Future research will focus on exploiting results on the analysis of infinite MDPs (such as one-counter MDPs) to the verification of pGCL programs.

8. Acknowledgements

This work is supported by the DFG Research Training Group 1298 "AlgoSyn", the iMQRES scholarship, the EU FP7 project CARP (Correct and Efficient Accelerator Programming), the DFG-NWO biliteral ROCKS project and by Australian Research Council Discovery Grant DP1092464. The authors would like to thank Daniel Stan (ENS Cachan) who during his internship at the RWTH University contributed to the proof that appears in the appendix. Additionally, we thank Tahiry Rabehaja and the anonymous reviewers for their valuable feedback.

References

[1] E. W. Dijkstra, A Discipline of Programming, Prentice Hall, 1976.

[2] J. J. Lukkien, An Operational Semantics for the Guarded Command Language, in: MPC, Vol. 669 of LNCS, Springer, 1992, pp. 233–249.

[3] J. J. Lukkien, Operational Semantics and Generalized Weakest Preconditions, Sci. Comput. Program. 22 (1-2) (1994) 137–155.

[4] A. McIver, C. Morgan, Abstraction, Refinement And Proof For Probabilistic Systems (Monographs in Computer Science), Springer, 2004.

[5] R. A. Howard, Dynamic Programming and Markov Processes, MIT Press, 1960.

[6] M. L. Puterman, Markov Decision Processes: Discrete Stochastic Dynamic Programming, John Wiley and Sons, 1994.

[7] C. Baier, F. Ciesinski, M. Größer, Probmela: a modeling language for communicating probabilistic processes, in: Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04, IEEE, 2004, pp. 57–66.

[8] A. Di Pierro, C. Hankin, H. Wiklicky, Program Analysis Probably Counts, Comput. J. 53 (6) (2010) 871–880.

[9] D. Kozen, Semantics of Probabilistic Programs, J. Comput. Syst. Sci. 22 (3) (1981) 328–350.

[10] J. He, K. Seidel, A. McIver, Probabilistic Models for the Guarded Command Language, Sci. Comput. Program. 28 (2-3) (1997) 171–192.

[11] C. Jones, Probabilistic non-determinism, Ph.D. thesis, University of Edinburgh (1989).

[12] F. Gretz, J.-P. Katoen, A. McIver, Prinsys - On a Quest for Probabilistic Loop Invariants, in: K. R. Joshi, M. Siegle, M. Stoelinga, P. R. D'Argenio (Eds.), QEST, Vol. 8054 of Lecture Notes in Computer Science, Springer, 2013, pp. 193–208.

[13] F. Gretz, J.-P. Katoen, A. McIver, Operational Versus Weakest Precondition Semantics for the Probabilistic Guarded Command Language, in: QEST, IEEE Computer Society, 2012, pp. 168–177.

[14] C. Baier, J.-P. Katoen, Principles of Model Checking, MIT Press, 2008.

[15] J. den Hartog, E. P. de Vink, Verifying Probabilistic Programs Using a Hoare Like Logic, Int. J. Found. Comput. Sci. 13 (3) (2002) 315–340.

[16] G. Barthe, B. Köpf, F. Olmedo, S. Z. Béguelin, Probabilistic Relational Reasoning for Differential Privacy, in: J. Field, M. Hicks (Eds.), POPL, ACM, 2012, pp. 97–110.

[17] O. Celiku, A. McIver, Cost-Based Analysis of Probabilistic Programs Mechanised in HOL, Nord. J. Comput. 11 (2) (2004) 102–128.

[18] J.-L. Lassez, V. L. Nguyen, L. Sonenberg, Fixed Point Theorems and Semantics: A Folk Tale, Inf. Process. Lett. 14 (3) (1982) 112–116.

[19] J.-P. Katoen, A. McIver, L. Meinicke, C. C. Morgan, Linear-Invariant Generation for Probabilistic Programs: - Automated Support for Proof-Based Methods, in: R. Cousot, M. Martel (Eds.), SAS, Vol. 6337 of Lecture Notes in Computer Science, Springer, 2010, pp. 390–406.

[20] E. M. Hahn, H. Hermanns, L. Zhang, Probabilistic Reachability for Parametric Markov Models, STTT 13 (1) (2011) 3–19.

Appendix A. Proofs

In the proofs of Theorems 23 and 24 we rely on the fact that the *wp* and *wlp* semantics of a loop can be seen as the limit of a loop that may perform only n steps where n tends to infinity. This is a consequence of one of the renowned fixed point theorems. In [18] the authors discuss several similar fixed point theorems due to Kleene, Knaster and Tarski. We justify our proof by what they propose as the "folk theorem":

Theorem 30. (Theorem 3 in [18]) Every continuous function F over a cpo has a least fixed point which is $\text{lub}_{n\in\mathbb{N}}\{F^n(\bot)\}$. ∎

Note that "continuous" means Scott-continuous here. In the following we define the necessary notions to understand the theorem and subsequently show that our expectation transformer $wp(P,\cdot)$ is indeed a Scott-continuous function. Hence we can characterise the behaviour of a loop by the supremum[6] over the behaviours of bounded loops. Finally, since $wp((\mathsf{while}(G)\{P\})^k,\cdot)$ is monotonically increasing we can exchange supremum for limit. Analogously, $wlp((\mathsf{while}(G)\{P\})^k,\cdot)$ is monotonically decreasing and the infimum can be replaced by the limit as well.

The first necessary notion is that of a directed set.

Definition 31. (Directed set) A non-empty set D is directed if for all $x, y \in D$, there exists $z \in D$ such that $z \geq x$ and $z \geq y$. ∎

The expectation space $(\mathbf{E}, \leq)$ (cf. Def. 14, page 10) is directed because for any two expectations we can find an expectation which is (pointwise) greater than both, for example by taking their pointwise supremum.

Definition 32. (Complete partial order) A set C is a (directed) complete partial order (cpo) if for every directed subset $D \subseteq C$ the supremum of D exists and lies in C. ∎

$(\mathbf{E}, \leq)$ is not a cpo in general because there is no top element. Hence infinitely ascending chains are possible in $(\mathbf{E}, \leq)$. So for technical reasons we assume there is a top element ∞ in $\mathbf{E}$ such that for any directed subset D of expectations we can take the pointwise supremum which again is in $\mathbf{E}$. Alternatively we may restrict ourselves to pGCL programs for which we can prove that a bounded post-expectation will always be transformed to a bounded pre-expectation. This alternative approach is taken in [4].

Note that Thm. 30 above requires a bottom element $\bot$ in the cpo. In the domain of expectations this is the constant 0 function.

Definition 33. (Scott-Continuous Function) Let C, C' be cpo's. A function $F : C \to C'$ is Scott-continuous if

- for all directed subsets $D \subseteq C$, the image $F(D)$ is directed and

[6] Least upper bound (lub) and supremum (sup) mean the same.

- $F(\sup D) = \sup F(D)$

■

This concludes our introduction of necessary definitions and we can now prove the following lemma that justifies the application of Thm. 30 in the proof of Thm. 23 (and similarly Thm. 24).

Lemma 34. (Continuity of expectation transformers)

For every pGCL program P the expectation transformer $wp(P, \cdot)$ is a Scott-continuous function over $(\mathbf{E}, \leq)$. The same holds for $wlp(P, \cdot)$. ■

Proof: The first point of Def. 33 follows immediately from monotonicity of $wp(P, \cdot)$.

It remains to show that for any directed subset D of expectations

$$wp(P, \sup_{f \in D} f) = \sup_{f \in D} wp(P, f) \ .$$

The proof is carried out by induction on the structure of the program P.
Induction base:

- $P = \mathsf{skip}$:
$$wp(\mathsf{skip}, \sup_{f \in D} f) = \sup_{f \in D} f = \sup_{f \in D} wp(\mathsf{skip}, f) \ .$$
- $P = \mathsf{abort}$:
$$wp(\mathsf{abort}, \sup_{f \in D} f) = 0 = \sup_{f \in D} wp(\mathsf{abort}, f) \ .$$
- Let P be an assignment $x := E$:
$$\begin{aligned} wp(x := E, \sup_{f \in D} f) &= \left(\sup_{f \in D} f \right)[x/E] \\ &\overset{*}{=} \sup_{f \in D} (f[x/E]) \\ &= \sup_{f \in D} wp(x := E, f) \ . \end{aligned}$$

  For $(*)$ observe that one can construct the supremum expectation and then substitute x for the expression E or first do the substitution and then construct (the same) pointwise supremum.

Induction hypothesis: assume that for program P (and analogously for Q)

$$wp(P, \sup_{f \in D} f) = \sup_{f \in D} wp(P, f) \ .$$

Induction step:

- Consider the sequential composition $P; Q$:

$$\begin{aligned} wp(P; Q, \sup_{f \in D} f) &= wp(P, wp(Q, \sup_{f \in D} f)) \\ &\overset{I.H.}{=} wp(P, \sup_{f \in D} wp(Q, f)) \\ &\overset{I.H.}{=} \sup_{f \in D} wp(P, wp(Q, f)) \end{aligned}$$

- Consider the probabilistic choice $P \,[p]\, Q$ (this also covers conditional choice because it can be rewritten as $P \,[[G]]\, Q$):

  A property of sup immediately provides an inequality:

$$\begin{aligned} wp(P \,[p]\, Q, \sup_{f \in D} f) &= p \cdot wp(P, \sup_{f \in D} f) + (1-p) \cdot wp(Q, \sup_{f \in D} f) \\ &\overset{I.H.}{=} \sup_{f \in D} (p \cdot wp(P, f)) + \sup_{f \in D} ((1-p) \cdot wp(Q, f)) \\ &\geq \sup_{f \in D} (p \cdot wp(P, f) + (1-p) \cdot wp(Q, f)) \\ &= \sup_{f \in D} wp(P \,[p]\, Q, f) \ . \end{aligned}$$

  We can strengthen the inequality to equality. Assume for the purpose of contradiction that

$$\begin{aligned} &\sup_{f \in D} (p \cdot wp(P, f) + (1-p) \cdot wp(Q, f)) \\ <\ &\sup_{f \in D} (p \cdot wp(P, f)) + \sup_{f \in D} ((1-p) \cdot wp(Q, f)) \ . \end{aligned}$$

  Then by definition of sup there must exist $g_1, g_2 \in D$ for which this strict inequality holds.

$$\begin{aligned} &\sup_{f \in D} (p \cdot wp(P, f) + (1-p) \cdot wp(Q, f)) \\ <\ &p \cdot wp(P, g_1) + (1-p) \cdot wp(Q, g_2) \\ \overset{*}{\leq}\ &p \cdot wp(P, h) + (1-p) \cdot wp(Q, h) \\ \leq\ &\sup_{f \in D} (p \cdot wp(P, f) + (1-p) \cdot wp(Q, f)) \ \text{Contradiction!} \end{aligned}$$

  Where in $(*)$ we use that D is directed and therefore there is an $h \in D$ with $g_1 \leq h$ and $g_2 \leq h$. And since $wp(P, \cdot)$ is monotonous for any P, the summands cannot decrease and hence the sum cannot decrease which gives the (non-strict) inequality.

- Consider the non-deterministic choice $P \,[]\, Q$. The proof for this case goes along the same lines as for probabilistic choice: A property of sup imme-

diately provides an inequality:

$$\begin{aligned} wp(P \,[]\, Q, \sup_{f \in D} f) &= \min \left\{ wp(P, \sup_{f \in D} f), wp(Q, \sup_{f \in D} f) \right\} \\ &\overset{I.H.}{=} \min \left\{ \sup_{f \in D} wp(P, f), \sup_{f \in D} wp(Q, f) \right\} \\ &\geq \sup_{f \in D} \min \{ wp(P, f), wp(Q, f) \} \\ &= \sup_{f \in D} wp(P \,[]\, Q, f) \ . \end{aligned}$$

We can strengthen the inequality to equality. Assume for the purpose of contradiction that

$$\begin{aligned} &\sup_{f \in D} \min \{ wp(P, f), wp(Q, f) \} \\ < \ &\min \left\{ \sup_{f \in D} wp(P, f), \sup_{f \in D} wp(Q, f) \right\} \ . \end{aligned}$$

Then by definition of sup there must exist $g_1, g_2 \in D$ for which this strict inequality holds.

$$\begin{aligned} &\sup_{f \in D} \min \{ wp(P, f), wp(Q, f) \} \\ < \ &\min \{ wp(P, g_1), wp(Q, g_2) \} \\ \leq \ &\min \{ wp(P, h), wp(Q, h) \} \\ \leq \ &\sup_{f \in D} \min \{ wp(P, f), wp(Q, f) \} \ \text{Contradiction!} \end{aligned}$$

- Consider the loop $\mathsf{while}(G)\{P\}$.

  In Def. 15, the semantics of the loop (w.r.t. a post-expectation f) were defined as the least fixed point of a function

  $$F(X) = [G] \cdot wp(P, X) + [\neg G] \cdot f$$

  For the sake of notation we additionally define:

  $$F_{\sup}(X) = [G] \cdot wp(P, X) + [\neg G] \cdot \sup_{f \in D} f \ .$$

  By the induction hypothesis $F(X)$ is Scott-continuous in X. Therefore we can apply Thm. 30:

  $$\mu X.F(X) = \sup_{n \in \mathbb{N}} F^n(0)$$

Using this we deduce:

$$\begin{aligned} wp(\mathsf{while}(G)\{P\}, \sup_{f\in D} f) &= \sup_{n\in\mathbb{N}} F^n_{\sup}(0) \\ &= \sup_{n\in\mathbb{N}} (\sup_{f\in D} F)^n(0) \\ &\overset{*}{=} \sup_{n\in\mathbb{N}} \sup_{f\in D} F^n(0) \\ &\overset{**}{=} \sup_{f\in D} \sup_{n\in\mathbb{N}} F^n(0) \\ &= \sup_{f\in D} wp(\mathsf{while}(G)\{P\}, f) \end{aligned}$$

For $(*)$ one can show that $(\sup_{f\in D} F)^n(0) = \sup_{f\in D} F^n(0)$ by induction on n using the definition of F, directedness of D and continuity of $wp(P,\cdot)$ as per induction hypothesis.

The equality in $(**)$ can be established by applying the suprema to $F^n(0)$ one by one and showing inequality in both directions. The existence of these suprema is a consequence of the induction hypothesis which gives us the continuity of F and Thm. 30. ■

The *wlp*-semantics for a loop is defined as the greatest fixed point solution of an equation, cf. Def. 16. We can adapt the proof above to $wlp(P,\cdot)$, if we reverse the direction of the directed set of expectations and bound expectation functions from above by 1 (as in the Def. 16). The semantics of the loop will be the limit of $F^n(1)$ (where the constant expectation function 1 is the top element in the reversed cpo).

Remark 35. (Continuity proofs) Continuity was proven for regular transformers of expectations over finite and countable state spaces [4]. Our result generalises to uncountable state spaces but restricts to `pGCL` programs which induce a subset of regular transformers. ■

C. Conditioning in Probabilistic Programming

To appear in MFPS 2015. This paper forms the basis of Chapter 3. An extended technical report can be found on arXiv [31].

Conditioning in Probabilistic Programming

Nils Jansen, Benjamin Lucien Kaminski,
Joost-Pieter Katoen, Federico Olmedo

RWTH Aachen University, Aachen, Germany

Friedrich Gretz, Annabelle McIver

Macquarie University, Sydney, Australia

Abstract

In this paper, we investigate the semantic intricacies of conditioning in probabilistic programming, a major feature, e.g., in machine learning. We provide a quantitative weakest pre–condition semantics. In contrast to all other approaches, non–termination is taken into account by our semantics. We also present an operational semantics in terms of Markov models and show that expected rewards coincide with quantitative pre–conditions. A program transformation that entirely eliminates conditioning from programs is given; the correctness is shown using our semantics. Finally, we show that an inductive semantics for conditioning in non–deterministic probabilistic programs cannot exist.

Keywords: Probabilistic Programming, Semantics, Conditional Probabilities, Program Transformation

1 Introduction

In recent years, interest in probabilistic programming has rapidly grown [9,11]. This is due to its wide applicability, for example in machine learning for describing distribution functions; Bayesian inference is pivotal in their analysis. It is used in security for describing both cryptographic constructions such as randomized encryption and experiments defining security properties [4]. Probabilistic programs, being extensions of familiar notions, render these fields accessible to programming communities. A rich palette of probabilistic programming languages exists including `Church` [8] as well as modern approaches like probabilistic `C` [23], `Tabular` [10] and `R2` [22].

Probabilistic programs are sequential programs having two main features: (1) the ability to *draw values at random* from probability distributions, and (2) the ability to *condition the value of variables* in a program through so–called *observations*. The semantics of languages without conditioning is well–understood: In his seminal work, Kozen [19] considered denotational semantics for probabilistic

⋆ This work was supported by the Excellence Initiative of the German federal and state government.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

programs without non–determinism or observations. One of these semantics—the expectation transformer semantics—was adopted by McIver and Morgan [21], who added support for non–determinism; a corresponding operational semantics is given in [13]. Other relevant works include probabilistic power–domains [17], semantics of constraint probabilistic programming languages [15,14], and semantics for stochastic λ–calculi [26].

Semantic intricacies. The difficulties that arise when program variables are conditioned through observations is less well–understood. This gap is filled in this paper. Previous work on semantics for programs with `observe` statements [22,16] do neither consider the possibility of *non–termination* nor the powerful feature of *non–determinism*. In contrast, we thoroughly study a more general setting which accounts for non–termination by means of a very simple yet powerful probabilistic programming language supporting non–determinism and observations. Let us first analyze a few examples illustrating the different problems. We start with the problem of non–termination; consider the two program snippets

$$x \coloneqq 2 \qquad \text{and} \qquad \{x \coloneqq 2\}\ [1/2]\ \{\texttt{abort}\}\ .$$

The program on the left just assigns the value 2 to the program variable x, while the program on the right tosses a fair coin—which is modeled through a *probabilistic choice*—and depending on the outcome either performs the same variable assignment or diverges due to the `abort` instruction. The semantics given in [22,16] does not distinguish these two programs and is only sensible in the context of terminating programs. A programmer writing only terminating programs is already unrealistic in the non–probabilistic setting. Our semantics does not rely on the assumption that programs always terminate and is able to distinguish these two programs.

To discuss *observations*, consider the program snippet P_{obs_1}

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

which assigns zero to the variable x with probability $1/2$ while x is assigned one with the same likelihood, after which we condition to the outcome of x being one. The `observe` statement blocks all invalid runs violating its condition and renormalizes the probabilities of the remaining valid runs. This differs, e.g., from program annotations like (probabilistic) *assertions* [25] as we will see later. The interpretation of the program is the expected outcome conditioned on the valid runs. For P_{obs_1}, this yields the outcome $1 \cdot 1$—there is one valid run that happens with probability one, with x being one.

More involved problems arise when programs are *infeasible* meaning all runs are blocked. Consider a slight variant of the program above, called P_{obs_2}:

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

The left branch of the probabilistic choice is infeasible. Is this program equivalent to the sample program P_{obs_1}? It will turn out that this is the case, meaning that setting an infeasible program into context can render it feasible.

The situation becomes more complicated when considering loopy programs that

may diverge. Consider the following two programs:

$$P_{div}: \quad x \coloneqq 1;\ \texttt{while}\,(x{=}1)\,\{x \coloneqq 1\}$$
$$P_{andiv}: \quad x \coloneqq 1;\ \texttt{while}\,(x{=}1)\,\{\{x \coloneqq 1\}\ [1/2]\ \{x \coloneqq 0\};\ \texttt{observe}\ (x{=}1)\}$$

Program P_{div} diverges as x is set to one in every iteration. This yields a null *expected outcome*. Due to the conditioning on $x{=}1$, P_{andiv} has just a single (valid)—non–terminating—run, but this run *almost surely* never happens, i.e. it happens with probability zero. The conditional expected outcome of P_{andiv} can thus not be measured. Our semantics can distinguish these programs while programs with (probabilistic) assertions must be loop–free to avoid similar problems [25]. Other approaches insist on the absence of diverging loops [5]. Neither of these assumptions are realistic.

Non–determinism is a powerful means to deal with unknown information, as well as to specify abstractions in situations where implementation details are unimportant. This feature turns out to be intricate in combination with conditioning. [1] Consider the program P_{nondet}

$$\{\{x \coloneqq 5\} \,\square\, \{x \coloneqq 2\}\}\ [1/4]\ \{x \coloneqq 2\};\ \texttt{observe}\ (x{>}3),$$

where with probability $1/4$, x is set either to 5 or to 2 non–deterministically (denoted $\{x \coloneqq 5\} \,\square\, \{x \coloneqq 2\}$), while x is set to 2 with likelihood $3/4$. Resolving the non–deterministic choice in favor of setting x to five yields a conditional expectation of 5 for x, obtained as $5 \cdot 1/4$ rescaled over the single valid run of P_{nondet}. Taking the right branch however induces two invalid runs due to the violation of the condition $x{>}3$, yielding a non–measurable conditional outcome.

Contributions. The above issues—non–termination, loops, and non–determinism—indicate that conditioning in probabilistic programs is far from trivial. This paper presents a thorough semantic treatment of conditioning in a probabilistic extension of Dijkstra's guarded command language (known as pGCL [21]), an elementary though foundational language that includes (amongst others) parametric probabilistic choice. We take several semantic viewpoints.

We first provide a conditional version of a *weakest pre–condition* (wp) semantics à la [21]. This is typically defined inductively over the structure of the program. We show that combining both non–determinism and conditioning *cannot* be treated in this manner. To treat possibly non–terminating programs, due to e.g., diverging loops or abortion, this is complemented by a weakest *liberal* pre–condition (wlp) semantics. Moreover, our w(l)p semantics is backward compatible with the original pGCL semantics for programs without conditioning; this *does not* apply to alternative approaches such as R2 [22].

Furthermore, Markov Decision Processes (MDPs) [24] are used as the basis for an *operational* semantics. This semantics is simple and elegant while covering *all* aforementioned phenomena, including non–determinism. We show that *conditional*

[1] As stated in [11], "representing and inferring sets of distributions is more complicated than dealing with a single distribution, and hence there are several technical challenges in adding non–determinism to probabilistic programs".

expected rewards in the MDP–semantics correspond to (conditional) wp in the denotational semantics, extending a similar result for pGCL [13].

Finally, we present a program transformation which entirely eliminates conditioning from any program and prove its correctness using our semantics.

Summarized, after introducing pGCL (Section 2), we give a denotational semantics for fully probabilistic programs (Section 3). We provide the first operational semantics for imperative probabilistic programming languages with conditioning and both probabilistic and non–deterministic choice (Section 4). Our semantics enables us to prove the correctness of a program transformation that eliminates `observe` statements (Section 5). Finally, we show that it is not possible to provide an inductive semantics for programs that include both conditioning and non–determinism (Section 6).

An extended version of this paper including all proofs and further program transformations for eliminating `observe` statements is available in [12].

2 The Programming Language

In this section we briefly present the probabilistic programming language used for our development. The language is an extension of the *probabilistic guarded command language* (pGCL) of McIver and Morgan [21]. The original pGCL is given by syntax

$$\begin{aligned}\mathcal{P} \;::=\; & \texttt{skip} \mid \texttt{abort} \mid x \coloneqq E \mid \mathcal{P}; \mathcal{P} \mid \texttt{ite}\,(G)\,\{\mathcal{P}\}\,\{\mathcal{P}\} \\ & \mid \{\mathcal{P}\}\ [p]\ \{\mathcal{P}\} \mid \{\mathcal{P}\} \,\square\, \{\mathcal{P}\} \mid \texttt{while}\,(G)\,\{\mathcal{P}\}\end{aligned}$$

and constitutes a plain extension of Dijkstra's guarded command language (GCL) [7] with a binary probabilistic choice operator. Here, x belongs to $\mathcal{V}$, the set of program variables; E is an arithmetical expression over $\mathcal{V}$; G a Boolean expression over $\mathcal{V}$; and p a real–valued parameter with domain $[0, 1]$. Most of the pGCL instructions are self–explanatory; we elaborate only on the following: $\{P\}\ [p]\ \{Q\}$ is a *probabilistic choice* where program P is executed with probability p and program Q with probability $1{-}p$; $\{P\} \,\square\, \{Q\}$ is a *non–deterministic choice* between P and Q; finally `abort` is syntactic sugar for the diverging program `while` (`true`) {`skip`}.

To model probabilistic programs with conditioning we extend pGCL with observations, leading to the *conditional* pGCL (cpGCL). At the syntactic level, an observation is introduced with the instruction `observe` (G), G being a Boolean expression over $\mathcal{V}$. The effect of such an instruction is to block all invalid program executions violating G and rescale the probability of the remaining executions so that they sum up to one.

As an illustrative example consider the following pair of programs:

$$\begin{aligned} P_1\colon &\quad \{x \coloneqq 0\}\ [p]\ \{x \coloneqq 1\};\ \{y \coloneqq 0\}\ [q]\ \{y \coloneqq -1\} \\ P_2\colon &\quad \{x \coloneqq 0\}\ [p]\ \{x \coloneqq 1\};\ \{y \coloneqq 0\}\ [q]\ \{y \coloneqq -1\};\ \texttt{observe}\ (x{+}y{=}0) \end{aligned}$$

Program P_1 admits all (four) runs, two of which satisfy $x{=}0$; for this program the probability of $x{=}0$ is p. Program P_2—due to the `observe` statement requiring $x{+}y{=}0$—admits only two runs, only one of them satisfying $x{=}0$; for this program

the probability of $x{=}0$ is $\frac{pq}{pq+(1-p)(1-q)}$.

Note that there exists a connection between the `observe` statement used in our work and the well–known assert statement. Both statements `observe` (G) and assert (G) block all runs violating G. The crucial difference, however, is that `observe` (G) normalizes the probability of the unblocked runs while assert (G) does not, yielding then a sub–distribution of total mass possibly less than one [20,4].

3 Denotational Semantics for Conditional pGCL

In this section we recall the expectation transformer semantics of pGCL and extend it to conditional programs in the fully probabilistic fragment of cpGCL.

3.1 Expectation Transformers in pGCL

Expectation transformers are a quantitative version of predicate transformers [7] used to endow probabilistic pGCL programs a formal semantics. Loosely speaking, they capture the average or expected outcome of a program, measured w.r.t. a utility or reward function over the set of final states. To make this more precise, let $\mathbb{S}$ be the set of program states, where a *program state* is a variable valuation. Now assume that P is a *fully probabilistic* program, i.e. a program without non–deterministic choices. Intuitively, we can think of P as a mapping from an initial state $\sigma \in \mathbb{S}$ to a distribution of final states $[\![P]\!](\sigma)$; its formal semantics is captured by a transformer $\mathsf{wp}[P]$, which acts as follows: Given a random variable $f\colon \mathbb{S} \to \mathbb{R}_{\geq 0}$, $\mathsf{wp}[P](f)$ maps every initial state σ to the expected value $\mathbf{E}_{[\![P]\!](\sigma)}(f)$ of f with respect to the distribution of final states $[\![P]\!](\sigma)$. Symbolically,

$$\mathsf{wp}[P](f)(\sigma) \;=\; \mathbf{E}_{[\![P]\!](\sigma)}(f)\ .$$

In particular, if $f = \chi_A$ is the characteristic function of some event A, $\mathsf{wp}[P](f)$ retrieves the probability that the event occurred after the execution of P. (Moreover, if P is a deterministic program in GCL, $\mathbf{E}_{[\![P]\!](\sigma)}(\chi_A)$ is $\{0,1\}$–valued and we recover the ordinary notion of predicate transformers introduced by Dijkstra [7].)

For a program P including non–deterministic choices, the execution of P yields a *set* of final distributions. To account for this, we assume that $\mathsf{wp}[P](f)(\sigma)$ gives the tightest lower bound $\inf_{\mu \in [\![P]\!](\sigma)} \mathbf{E}_\mu(f)$ for the expected value of f. This corresponds with the notion of a *demonic* adversary resolving the non–deterministic choices.

We follow McIver and Morgan [21] and use the term *expectation* to refer to a random variable mapping program states to real values. The expectation transformer wp then transforms a post–expectation f into a pre–expectation $\mathsf{wp}[P](f)$ and can be defined by induction on the structure of P, following the rules in Figure 1. The transformer wp also admits a liberal variant wlp, which differs from wp in the way in which non–termination is treated.

Formally, the transformer wp operates on *unbounded expectations* in $\mathbb{E} = \mathbb{S} \to \mathbb{R}^\infty_{\geq 0}$ and wlp operates on *bounded expectations* in $\mathbb{E}_{\leq 1} = \mathbb{S} \to [0,1]$. Here $\mathbb{R}^\infty_{\geq 0}$ denotes the set of non–negative real values with the adjoined ∞ value. In order to guarantee the well–definedness of wp and wlp we need to provide $\mathbb{E}$ and $\mathbb{E}_{\leq 1}$ the structure of a directed–complete partial order. Expectations are ordered pointwise,

i.e. $f \sqsubseteq g$ iff $f(\sigma) \leq g(\sigma)$ for every state $\sigma \in \mathbb{S}$. The least upper bound of directed subsets is also defined pointwise.

In the remainder we make use of the following notation related to expectations. We use bold fonts for constant expectations, e.g. $\mathbf{1}$ denotes the constant expectation 1. Given an arithmetical expression E over program variables we simply write E for the expectation that in state σ returns $\sigma(E)$. Given a Boolean expression G over program variables we use χ_G to denote the $\{0,1\}$–valued expectation that returns 1 if $\sigma \models G$ and 0 otherwise.

3.2 Conditional Expectation Transformers

We now study how to extend the notion of expectation transformers to conditioned probabilistic programs without non–determinism in cpGCL. To illustrate the intuition behind our solution, consider the following scenario: Assume we want to measure the probability that some event A occurs after the execution of a conditioned program P. Since P contains observations, its execution leads to a conditional distribution $\mu|_O$ of final states. Now the conditional probability that A occurs (given that O occurs) is given as the quotient of the probabilities $\Pr[\mu \in A \wedge O]$ and $\Pr[\mu \in O]$. Motivated by this observation, we introduce an expectation transformer $\mathsf{cwp}[\cdot] \colon \mathbb{E} \times \mathbb{E}_{\leq 1} \to \mathbb{E} \times \mathbb{E}_{\leq 1}$, whose application $\mathsf{cwp}[P](\chi_A, \mathbf{1})$ will yield the desired pair of probabilities $(\Pr[\mu \in A \wedge O], \Pr[\mu \in O])$. We are only left to define a transformer $\underline{\mathsf{cwp}}[P]$ that computes the corresponding quotient. Formally, we let

$$\underline{\mathsf{cwp}}[P](f) \;\triangleq\; \frac{\mathsf{cwp}_1[P](f, \mathbf{1})}{\mathsf{cwp}_2[P](f, \mathbf{1})} \,,$$

where $\mathsf{cwp}_1[P](f,g)$ (resp. $\mathsf{cwp}_2[P](f,g)$) denotes the first (resp. second) component of $\mathsf{cwp}[P](f,g)$. If $\mathsf{cwp}_2[P](f,\mathbf{1})(\sigma) = 0$, then $\underline{\mathsf{cwp}}[P](f)$ is not well–defined in σ (in the same way as the conditional probability $\Pr(A \mid B)$ is not well–defined[2] when $\Pr(B) = 0$) and we say that program P is *infeasible* from state σ, meaning that *all* its executions are blocked by observations.

As so defined, $\underline{\mathsf{cwp}}[P](f)$ represents the weakest *conditional* pre–expectation of P with respect to post–expectation f and $\underline{\mathsf{cwp}}[\cdot]$ generalizes the transformer $\mathsf{wp}[\cdot]$ to conditioned programs. The weakest liberal conditional pre–expectation $\underline{\mathsf{cwlp}}[P](f)$ is defined analogously, in terms of the transformer $\mathsf{cwlp}[P] \colon \mathbb{E}_{\leq 1} \times \mathbb{E}_{\leq 1} \to \mathbb{E}_{\leq 1} \times \mathbb{E}_{\leq 1}$.

We are only left to provide definitions for $\mathsf{cwp}[P]$ and $\mathsf{cwlp}[P]$. Both transformers are defined by induction on the structure of P, following the rules in Figure 1. Let us briefly explain these rules. $\mathsf{cwp}[\texttt{skip}]$ behaves as the identity since `skip` has no effect. $\mathsf{cwp}[\texttt{abort}]$ maps any pair of post–expectations to the pair of constant pre–expectations $(\mathbf{0}, \mathbf{1})$. Assignments induce a substitution on expectations, i.e. $\mathsf{cwp}[x \coloneqq E]$ maps (f,g) to pre–expectation $(f[x/E], g[x/E])$, where $h[x/E](\sigma) = h(\sigma[x/E])$ and $\sigma[x/E]$ denotes the usual variable update on states. $\mathsf{cwp}[P_1; P_2]$ is obtained as the functional composition (denoted $\circ$) of $\mathsf{cwp}[P_1]$ and $\mathsf{cwp}[P_2]$. $\mathsf{cwp}[\texttt{observe}\ (G)]$ restricts post–expectations to those states that satisfy G; states that do not satisfy G are mapped to 0. $\mathsf{cwp}[\texttt{ite}\,(G)\,\{P_1\}\,\{P_2\}]$ behaves

[2] In the *continuous* setting we could define a conditional density even when conditioning on events with 0 measure using the Radon–Nikodym theorem. However, our programs generate *discrete* distributions only.

P	$\mathsf{wp}[P](f)$	$\mathsf{cwp}[P](f,\, g)$
`skip`	f	$(f,\, g)$
`abort`	$\mathbf{0}$	$(\mathbf{0},\, \mathbf{1})$
$x \coloneqq E$	$f[x/E]$	$(f[x/E],\, g[x/E])$
`observe` (G)	— not defined —	$\chi_G \cdot (f,\, g)$
$P_1;\ P_2$	$(\mathsf{wp}[P_1] \circ \mathsf{wp}[P_2])(f)$	$(\mathsf{cwp}[P_1] \circ \mathsf{cwp}[P_2])(f, g)$
`ite` $(G)\,\{P_1\}\,\{P_2\}$	$\chi_G \cdot \mathsf{wp}[P_1](f) + \chi_{\neg G} \cdot \mathsf{wp}[P_2](f)$	$\chi_G \cdot \mathsf{cwp}[P_1](f,\, g) + \chi_{\neg G} \cdot \mathsf{cwp}[P_2](f,\, g)$
$\{P_1\}\ [p]\ \{P_2\}$	$p \cdot \mathsf{wp}[P_1](f) + (1{-}p) \cdot \mathsf{wp}[P_2](f)$	$p \cdot \mathsf{cwp}[P_1](f,\, g) + (1{-}p) \cdot \mathsf{cwp}[P_2](f,\, g)$
$\{P_1\} \,\square\, \{P_2\}$	$\lambda\sigma.\ \min\{\mathsf{wp}[P_1](f)(\sigma),\, \mathsf{wp}[P_2](f)(\sigma)\}$	— not defined —
`while` $(G)\,\{P'\}$	$\boldsymbol{\mu}\, \hat{f}.\ \Big(\chi_G \cdot \mathsf{wp}[P'](\hat{f}) + \chi_{\neg G} \cdot f\Big)$	$\boldsymbol{\mu}_{\sqsubseteq,\sqsupseteq}(\hat{f},\, \hat{g}).\ \Big(\chi_G \cdot \mathsf{cwp}[P'](\hat{f},\, \hat{g}) + \chi_{\neg G} \cdot (f,\, g)\Big)$
P	$\mathsf{wlp}[P](f)$	$\mathsf{cwlp}[P](f,\, g)$
`abort`	$\mathbf{1}$	$(\mathbf{1},\, \mathbf{1})$
`while` $(G)\,\{P'\}$	$\boldsymbol{\nu}\, \hat{f}.\ \Big(\chi_G \cdot \mathsf{wp}[P'](\hat{f}) + \chi_{\neg G} \cdot f\Big)$	$\boldsymbol{\nu}_{\sqsubseteq,\sqsubseteq}(\hat{f},\, \hat{g}).\ \Big(\chi_G \cdot \mathsf{cwp}[P'](\hat{f},\, \hat{g}) + \chi_{\neg G} \cdot (f,\, g)\Big)$

Fig. 1. Definitions for the wp/wlp and cwp/cwlp operators. The wlp (cwlp) operator differs from wp (cwp) only for `abort` and the `while`–loop. Multiplication $h \cdot (f,\, g)$ is meant componentwise yielding $(h \cdot f,\, h \cdot g)$. Likewise, addition $(f,\, g) + (f',\, g')$ is meant componentwise yielding $(f + f',\, g + g')$.

either as $\mathsf{cwp}[P_1]$ or $\mathsf{cwp}[P_2]$ according to the evaluation of G. $\mathsf{cwp}[\{P_1\}\ [p]\ \{P_2\}]$ is obtained as a convex combination of $\mathsf{cwp}[P_1]$ and $\mathsf{cwp}[P_2]$, weighted according to p. $\mathsf{cwp}[\texttt{while}\,(G)\,\{P'\}]$ is defined using standard fixed point techniques.[3] The cwlp transformer follows the same rules as cwp, except for the `abort` and `while` statements. $\mathsf{cwlp}[\texttt{abort}]$ takes any post–expectation to pre–expectation $(\mathbf{1}, \mathbf{1})$; $\mathsf{cwlp}[\texttt{while}\,(G)\,\{P\}]$ is defined in terms of a greatest rather than a least fixed point.

Observe that Figure 1 presents no rule for the non–deterministic choice operator. Therefore our conditional expectation transformers $\underline{\mathsf{cwp}}$/$\underline{\mathsf{cwlp}}$ can only handle fully probabilistic cpGCL programs. In Section 6 we elaborate on this limitation.

Example 3.1 Assume we want to compute the expected value of the expression $10{+}x$ after executing program P' given as:

1: $\{x \coloneqq 0\}\ [1/2]\ \{x \coloneqq 1\};$
2: $\texttt{ite}\,(x = 1)\,\big\{\{y \coloneqq 0\}\ [1/2]\ \{y \coloneqq 2\}\big\}\big\{\{y \coloneqq 0\}\ [4/5]\ \{y \coloneqq 3\}\big\};$
3: $\texttt{observe}\ (y{=}0)$

The computation of $\mathsf{cwp}[P'](10{+}x, \mathbf{1})$ goes as follows:

$$
\begin{aligned}
\mathsf{cwp}[P'](10{+}x, \mathbf{1}) &= \mathsf{cwp}[P'_{1\text{-}2}](\mathsf{cwp}[\texttt{observe}\ (y{=}0)](10{+}x, \mathbf{1})) \\
&= \mathsf{cwp}[P'_{1\text{-}2}](f, g) \text{ where } (f, g) = \chi_{y=0} \cdot (10{+}x, \mathbf{1}) \\
&= \mathsf{cwp}[P'_{1\text{-}1}](\mathsf{cwp}[\texttt{ite}\,(x{=}1)\,\{\ldots\}\,\{\ldots\}](f, g)) \\
&= \mathsf{cwp}[P'_{1\text{-}1}](\chi_{x=1} \cdot (h, i) + \chi_{x\neq 1} \cdot (h', i')) \quad \text{where} \\
&\qquad (h, i) = \mathsf{cwp}[\{y{:=}0\}\ [1/2]\ \{y{:=}2\}](f, g) = \tfrac{1}{2} \cdot (10 + x,\, \mathbf{1})\ , \text{ and} \\
&\qquad (h', i') = \mathsf{cwp}[\{y{:=}0\}\ [4/5]\ \{y{:=}3\}](f, g) = \tfrac{4}{5} \cdot (10 + x,\, \mathbf{1}) \\
&= \tfrac{1}{2} \cdot \tfrac{4}{5} \cdot (\mathbf{10 + 0},\, \mathbf{1}) + \tfrac{1}{2} \cdot \tfrac{1}{2} \cdot (\mathbf{10 + 1},\, \mathbf{1}) = \big(\tfrac{\mathbf{27}}{\mathbf{4}}, \tfrac{\mathbf{13}}{\mathbf{20}}\big)\ .
\end{aligned}
$$

The expected value of $10{+}x$ is then given by $\underline{\mathsf{cwp}}[P'](10{+}x) = \frac{\mathbf{27}}{\mathbf{4}} / \frac{\mathbf{13}}{\mathbf{20}} = \frac{\mathbf{135}}{\mathbf{13}} \approx 10.38$.

[3] We define $\mathsf{cwp}[\texttt{while}\,(G)\,\{P\}]$ as the least fixed point w.r.t. the order $(\sqsubseteq, \sqsupseteq)$ in $\mathbb{E} \times \mathbb{E}_{\leq 1}$. This way we encode the greatest fixed point in the second component w.r.t. the order $\sqsubseteq$ over $\mathbb{E}_{\leq 1}$ as the least fixed point w.r.t. the dual order $\sqsupseteq$.

In the rest of this section we investigate some properties of the expectation transformer semantics (of the fully probabilistic fragment) of cpGCL. As every fully probabilistic pGCL program is contained in cpGCL, we begin by studying the relation between the w(l)p–semantics of pGCL and the cw(l)p–semantics of cpGCL. To that end, we extend the w(l)p operator to cpGCL by the clauses $\mathsf{wp}[\texttt{observe}\ (G)](f) = \chi_G \cdot f$ and $\mathsf{wlp}[\texttt{observe}\ (G)](f) = \chi_G \cdot f$. Our first result says that cwp (resp. cwlp) can be decoupled as the product wp × wlp (resp. wlp × wlp).

Lemma 3.2 (Decoupling of cw(l)p) *Let P be a fully probabilistic* cpGCL *program, $f \in \mathbb{E}$ and $f', g \in \mathbb{E}_{\leq 1}$. Then* $\mathsf{cwp}[P](f,\, g) = \big(\mathsf{wp}[P](f),\, \mathsf{wlp}[P](g)\big)$ *and* $\mathsf{cwlp}[P](f',\, g) = \big(\mathsf{wlp}[P](f'),\, \mathsf{wlp}[P](g)\big)$.

Our next result shows that the $\underline{\mathsf{cwp}}$–semantics is a conservative extension of the wp–semantics for the fully probabilistic fragment of pGCL. The same applies to the weakest liberal pre–expectation semantics.

Theorem 3.3 (Compatibility with the w(l)p–semantics) *Let P be a fully probabilistic* pGCL *program, $f \in \mathbb{E}$, and $g \in \mathbb{E}_{\leq 1}$. Then* $\mathsf{wp}[P](f) = \underline{\mathsf{cwp}}[P](f)$ *and* $\mathsf{wlp}[P](g) = \underline{\mathsf{cwlp}}[P](g)$.

Proof. By Lemma 3.2 and the fact that $\mathsf{wlp}[P](\mathbf{1}) = \mathbf{1}$ (see Lemma 3.4). □

We now show that $\underline{\mathsf{cwp}}$ and $\underline{\mathsf{cwlp}}$ preserve the so–called healthiness conditions of wp and wlp.

Lemma 3.4 (Healthiness conditions of $\underline{\mathsf{cwp}}$ and $\underline{\mathsf{cwlp}}$) *For every fully probabilistic* cpGCL *program P with at least one feasible execution (from every initial state), every $f, g \in \mathbb{E}$ and non–negative real constants α, β:*

i) $f \sqsubseteq g$ *implies* $\underline{\mathsf{cwp}}[P](f) \sqsubseteq \underline{\mathsf{cwp}}[P](g)$ *and likewise for* $\underline{\mathsf{cwlp}}$ *(monotonicity).*

ii) $\underline{\mathsf{cwp}}[P](\alpha \cdot f + \beta \cdot g) = \alpha \cdot \underline{\mathsf{cwp}}[P](f) + \beta \cdot \underline{\mathsf{cwp}}[P](g)$ *(linearity).*

iii) $\underline{\mathsf{cwp}}[P](\mathbf{0}) = \mathbf{0}$ *and* $\underline{\mathsf{cwlp}}[P](\mathbf{1}) = \mathbf{1}$.

Proof. Using Lemma 3.2 one can show that the transformers $\underline{\mathsf{cwp}}$ and $\underline{\mathsf{cwlp}}$ inherit these properties from wp and wlp. For details see [12, p. 15]. □

We conclude this section by discussing alternative approaches for providing an expectation transformer semantics for $P \in$ cpGCL. By Lemma 3.2, the transformers $\underline{\mathsf{cwp}}[P]$ and $\underline{\mathsf{cwlp}}[P]$ can be recast as

$$f \mapsto \frac{\mathsf{wp}[P](f)}{\mathsf{wlp}[P](\mathbf{1})} \qquad \text{and} \qquad f \mapsto \frac{\mathsf{wlp}[P](f)}{\mathsf{wlp}[P](\mathbf{1})}\ ,$$

respectively. An alternative is to normalize using wp instead of wlp in the denominator, yielding the two transformers

$$\textit{i)}\ \ f \mapsto \frac{\mathsf{wp}[P](f)}{\mathsf{wp}[P](\mathbf{1})} \qquad \text{and} \qquad \textit{ii)}\ \ f \mapsto \frac{\mathsf{wlp}[P](f)}{\mathsf{wp}[P](\mathbf{1})}\ .$$

Transformer *ii)* is not meaningful, as the denominator $\mathsf{wp}[P](\mathbf{1})(\sigma)$ may be smaller than the numerator $\mathsf{wlp}[P](f)(\sigma)$ for some state $\sigma \in \mathbb{S}$. This might lead to probabilities exceeding one. Transformer *i)* normalizes w.r.t. the terminating executions.

This interpretation corresponds to the semantics of the probabilistic programming language R2 [22,16] and is *only meaningful if programs terminate almost surely* (i.e. with probability one). A noteworthy consequence of adopting transformer *i)* is that `observe` (G) is equivalent to `while` $(\neg G)$ $\{$`skip`$\}$ [16], see the discussion in Section 5.

Let us briefly compare the four alternatives by means of a concrete program P:

$$\big\{\texttt{abort}\big\}\,[1/2]\big\{\{x \coloneqq 0\}\,[1/2]\,\{x \coloneqq 1\}; \{y \coloneqq 0\}\,[1/2]\,\{y \coloneqq 1\};\, \texttt{observe}\,(x{=}0 \vee y{=}0)\big\}$$

P tosses a fair coin and according to the outcome either diverges or tosses a fair coin twice and observes at least once heads $(y{=}0 \vee x{=}0)$. We measure the probability that the outcome of the last coin toss was heads according to each transformer:

$$\frac{\mathsf{wp}[P](\chi_{y=0})}{\mathsf{wlp}[P](\mathbf{1})} = \frac{2}{7} \qquad \frac{\mathsf{wlp}[P](\chi_{y=0})}{\mathsf{wlp}[P](\mathbf{1})} = \frac{6}{7} \qquad \frac{\mathsf{wp}[P](\chi_{y=0})}{\mathsf{wp}[P](\mathbf{1})} = \frac{2}{3} \qquad \frac{\mathsf{wlp}[P](\chi_{y=0})}{\mathsf{wp}[P](\mathbf{1})} = 2$$

As mentioned before, the transformer *ii)* is not significant as it yields a "probability" exceeding one. Note that our cwp–semantics yields that the probability of $y{=}0$ after the execution of P while passing all `observe`–statements is $\frac{2}{7}$. As shown before, this is a conservative and natural extension of the wp–semantics. This does not apply to the R2–semantics, as this would require an adaptation of rules for `abort` and `while`.

4 Operational Semantics for Conditional pGCL

This section presents an operational semantics for cpGCL using Markov decision processes (MDPs) as underlying model. We begin by recalling the notion of MDPs. For that, let $\mathit{Distr}(S)$ denote the set of distributions $\mu\colon S \to \mathbb{R}$ over S with $\sum_{s\in S} \mu(s) = 1$.

Definition 4.1 An *MDP* is a tuple $\mathfrak{R} = (S,\, s_I,\, \mathit{Act},\, \mathcal{P},\, L)$ with a countable set of states S, an initial state $s_I \in S$, a finite set of actions Act, a transition probability function $\mathcal{P}\colon S \times \mathit{Act} \to \mathit{Distr}(S)$ with $\sum_{s'\in S} \mathcal{P}(s,\,\alpha)(s') = 1$ for all $(s,\,\alpha) \in S \times \mathit{Act}$ and a labeling function $L\colon S \to 2^{AP}$ for a set of atomic propositions AP.

A function $r\colon S \to \mathbb{R}_{\geq 0}$ is used to add *rewards* to an MDP. A *path* of $\mathfrak{R}$ is a finite or infinite sequence $\pi = s_0\alpha_0 s_1\alpha_1 \ldots$ such that $s_i \in S$, $\alpha_i \in \mathit{Act}$, $s_0 = s_I$, and $\mathcal{P}(s_i,\,\alpha_i)(s_{i+1}) > 0$ for all $i \geq 0$. The i-th state s_i of π is denoted $\pi(i)$. The set of all paths of $\mathfrak{R}$ is denoted by $\mathsf{Paths}^{\mathfrak{R}}$. $\mathsf{Paths}^{\mathfrak{R}}(s)$ is the set of paths starting in s and $\mathsf{Paths}^{\mathfrak{R}}(s, s')$ is the set of all finite paths starting in s and ending in s'. This is also lifted to sets of states. We sometimes omit superscript $\mathfrak{R}$ in $\mathsf{Paths}^{\mathfrak{R}}$.

MDPs operate by a non–deterministic choice of an action $\alpha \in \mathit{Act}$ that is *enabled* at state s and a subsequent probabilistic determination of a successor state according to $\mathcal{P}(s, \alpha)$. For resolving the non–deterministic choices, so–called *schedulers* are used. Here, *deterministic* and *memoryless* schedulers suffice which are functions $\mathfrak{S}\colon S \to \mathit{Act}$. Let $\mathit{Sched}^{\mathfrak{R}}$ denote the class of all such schedulers for $\mathfrak{R}$.

For MDP $\mathfrak{R}$, the fully probabilistic system ${}^{\mathfrak{S}}\mathfrak{R}$ induced by a scheduler $\mathfrak{S} \in \mathit{Sched}^{\mathfrak{R}}$ is called the *induced Markov Chain (MC)* on which a *probability measure* over paths is defined. The measure $\Pr^{\mathcal{R}}$ for MC $\mathcal{R}$ is given by $\Pr^{\mathcal{R}}\colon \mathsf{Paths}^{\mathcal{R}} \to [0,\,1] \subseteq \mathbb{R}$ with $\Pr^{\mathcal{R}}(\hat{\pi}) = \prod_{i=0}^{n-1} \mathcal{P}(s_i, s_{i+1})$, for a finite path $\hat{\pi} = s_0 \ldots s_n$. This is

lifted to infinite paths using the standard cylinder set construction, see [2, Ch. 10]. The *cumulated reward* of a finite path $\hat{\pi} = s_0 \dots s_n$ is given by $r(\hat{\pi}) = \sum_{i=0}^{n-1} r(s_i)$. Note that in our special setting the cumulated reward will not be infinite.

We consider *reachability properties* $\Diamond T$ for a set of target states $T \subseteq S$ where $\Diamond T$ also denotes all paths that reach T from the initial state s_I. Analogously, the set $\neg\Diamond T$ contains all paths that never reach a state in T.

First, consider reward objectives for MCs. The *expected reward* for a countable set of paths $\Diamond T$ is given by $\mathsf{ExpRew}^{\mathcal{R}}(\Diamond T) = \sum_{\hat{\pi}\in\Diamond T} \Pr^{\mathcal{R}}(\hat{\pi}) \cdot r(\hat{\pi})$. For a reward bounded by one, the notion of the *liberal* expected reward also takes the mere probability of *not* reaching the target states into account: $\mathsf{LExpRew}^{\mathcal{R}}(\Diamond T) = \mathsf{ExpRew}^{\mathcal{R}}(\Diamond T) + \Pr^{\mathcal{R}}(\neg\Diamond T)$. To exclude the probability of paths that reach "undesired" states, we let $U = \{s \in S \mid \lightning \in L(s)\}$ and define the *conditional expected reward* for the condition $\neg\Diamond U$ by [4]

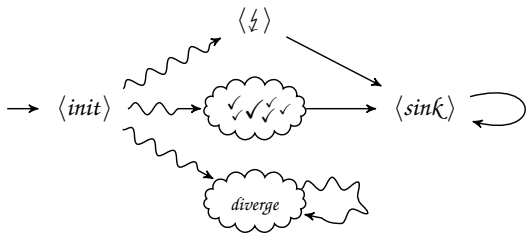
$$\mathsf{CExpRew}^{\mathcal{R}}(\Diamond T \mid \neg\Diamond U) \triangleq \frac{\mathsf{ExpRew}^{\mathcal{R}}(\Diamond T \cap \neg\Diamond U)}{\Pr^{\mathcal{R}}(\neg\Diamond U)} .$$

Reward objectives for MDPs are now defined using a *demonic* scheduler $\mathfrak{S} \in \mathit{Sched}^{\mathfrak{R}}$ minimizing probabilities and expected rewards for the induced MC ${}^{\mathfrak{S}}\mathfrak{R}$. For the expected reward this yields $\mathsf{ExpRew}^{\mathfrak{R}}(\Diamond T) = \inf_{\mathfrak{S}\in\mathit{Sched}^{\mathfrak{R}}} \mathsf{ExpRew}^{{}^{\mathfrak{S}}\mathfrak{R}}(\Diamond T)$. For conditional expected reward properties, the value of the quotient is minimized:

$$\mathsf{CExpRew}^{\mathfrak{R}}(\Diamond T \mid \neg\Diamond U) \triangleq \inf_{\mathfrak{S}\in\mathit{Sched}^{\mathfrak{R}}} \frac{\mathsf{ExpRew}^{{}^{\mathfrak{S}}\mathfrak{R}}(\Diamond T \cap \neg\Diamond U)}{\Pr^{{}^{\mathfrak{S}}\mathfrak{R}}(\neg\Diamond U)} .$$

The liberal reward notions for MDPs are analogous. Regarding the quotient minimization we assume "$\frac{0}{0} < 0$" as we see $\frac{0}{0}$—being undefined—to be less favorable than 0. For details about conditional probabilities and expected rewards see [3].

The structure of the operational MDP of a cpGCL program is depicted on the right. Terminating runs eventually end up in the $\langle \mathit{sink} \rangle$ state; other runs are diverging (never reach $\langle \mathit{sink} \rangle$). A program terminates either successfully, i.e. a run passes a ✓–labelled state, or terminates due to a violation of an observation, i.e. a run passes $\langle \lightning \rangle$. Squiggly arrows indicate reaching certain states via possibly multiple paths and states; the clouds indicate that there might be several states of the particular kind. The ✓–labelled states are the *only ones* with positive reward. Note that the sets of paths that eventually reach $\langle \lightning \rangle$, or ✓, or diverge are pairwise disjoint.



Definition 4.2 [Operational cpGCL semantics] The *operational semantics* of $P \in \mathsf{cpGCL}$ for $\sigma \in \mathbb{S}$ and $f \in \mathbb{E}$ is the MDP $\mathfrak{R}^f_\sigma[\![P]\!] = (S, \langle P, \sigma\rangle, \mathit{Act}, \mathcal{P}, L, r)$, such that S is the smallest set of states with $\langle \lightning \rangle \in S$, $\langle \mathit{sink} \rangle \in S$, and $\langle Q, \tau\rangle, \langle \downarrow, \tau\rangle \in S$

[4] Note that strictly formal one would have to define the intersection of sets of finite and possibly infinite paths by means of a cylinder set construction considering all infinite extensions of finite paths.

for $Q \in \mathsf{pGCL}$ and $\tau \in \mathbb{S}$. $\langle P, \sigma\rangle \in S$ is the initial state. $Act = \{\mathit{left}, \mathit{right}\}$ is the set of actions. A state of the form $\langle\downarrow, \tau\rangle$ denotes a terminal state in which no program is left to be executed. $\mathcal{P}$ is formed according to SOS rules given in [12, p. 5].

For some $\tau \in \mathbb{S}$, the labelling and the reward function is given by:

$$L(s) \triangleq \begin{cases} \{\checkmark\}, & \text{if } s = \langle\downarrow, \tau\rangle \\ \{\mathit{sink}\}, & \text{if } s = \langle \mathit{sink}\rangle \\ \{\lightning\}, & \text{if } s = \langle\lightning\rangle \\ \emptyset, & \text{otherwise,} \end{cases} \qquad r(s) \triangleq \begin{cases} f(\tau), & \text{if } s = \langle\downarrow, \tau\rangle \\ 0, & \text{otherwise .} \end{cases}$$

To determine the *conditional expected outcome of program* P given that all observations are true, we need to determine the *expected reward to reach* $\langle \mathit{sink}\rangle$ *from the initial state conditioned on not reaching* $\langle\lightning\rangle$ under a demonic scheduler. For $\mathfrak{R}^f_\sigma[\![P]\!]$ this is given by $\mathsf{CExpRew}^{\mathfrak{R}^f_\sigma[\![P]\!]}(\Diamond \mathit{sink} \,|\, \neg\Diamond\lightning)$. Recall for the condition $\neg\Diamond\lightning$ that all paths not eventually reaching $\langle\lightning\rangle$ either diverge (thus collect reward 0) or pass by a $\checkmark$–labelled state and eventually reach $\langle\mathit{sink}\rangle$. This gives us:

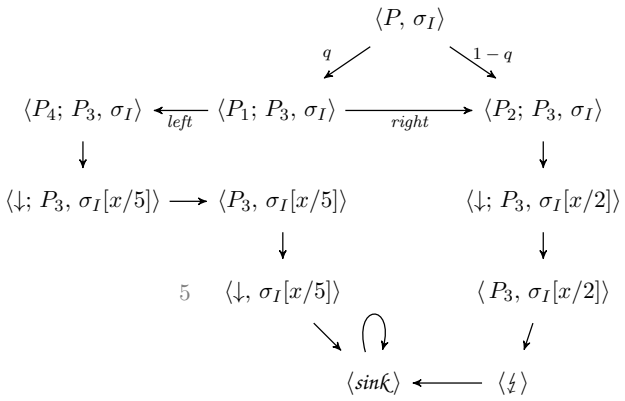
$$\begin{aligned} \mathsf{CExpRew}^{\mathfrak{R}^f_\sigma[\![P]\!]}(\Diamond \mathit{sink} \,|\, \neg\Diamond\lightning) &= \inf_{\mathfrak{S} \in \mathit{Sched}^{\mathfrak{R}^f_\sigma[\![P]\!]}} \frac{\mathsf{ExpRew}^{\mathfrak{S}\mathfrak{R}^f_\sigma[\![P]\!]}(\Diamond \mathit{sink} \cap \neg\Diamond\lightning)}{\Pr^{\mathfrak{S}\mathfrak{R}^f_\sigma[\![P]\!]}(\neg\Diamond\lightning)} \\ &= \inf_{\mathfrak{S} \in \mathit{Sched}^{\mathfrak{R}^f_\sigma[\![P]\!]}} \frac{\mathsf{ExpRew}^{\mathfrak{S}\mathfrak{R}^f_\sigma[\![P]\!]}(\Diamond \mathit{sink})}{\Pr^{\mathfrak{S}\mathfrak{R}^f_\sigma[\![P]\!]}(\neg\Diamond\lightning)} \,. \end{aligned}$$

The liberal version $\mathsf{CLExpRew}^{\mathcal{R}^f_\sigma[\![P]\!]}(\Diamond \mathit{sink} \,|\, \neg\Diamond\lightning)$ is defined analogously.

Example 4.3 Consider the program $P \in \mathsf{cpGCL}$:

$$\{\{x \coloneqq 5\} \,\square\, \{x \coloneqq 2\}\} \;[q]\; \{x \coloneqq 2\}; \texttt{observe}\ (x{>}3)$$

where with parametrized probability q a non–deterministic choice between x being assigned 2 or 5 is executed, and with probability $1{-}q$, x is directly assigned 2. Let for readability $P_1 = \{x \coloneqq 5\} \,\square\, \{x \coloneqq 2\}$, $P_2 = x \coloneqq 2$, $P_3 = \texttt{observe}\ (x{>}3)$, and $P_4 = x \coloneqq 5$. The operational MDP $\mathfrak{R}^x_{\sigma_I}[\![P]\!]$ for an arbitrary initial variable valuation σ_I and post–expectation x is depicted below:



The only state with positive reward is $s' := \langle \downarrow, \sigma_I[x/5] \rangle$ and its reward is indicated by number 5. Assume first a scheduler choosing action *left* in state $\langle P_1; P_3, \sigma_I \rangle$. In the induced MC the only path accumulating positive reward is the path π going from $\langle P, \sigma_I \rangle$ via s' to $\langle \mathit{sink} \rangle$ with $r(\pi) = 5$ and $\Pr(\pi) = q$. This gives an expected reward of $5 \cdot q$. The overall probability of not reaching $\langle \lightning \rangle$ is also q. The conditional expected reward of eventually reaching $\langle \mathit{sink} \rangle$ given that $\langle \lightning \rangle$ is not reached is hence $\frac{5 \cdot q}{q} = 5$. Assume now the *demonic* scheduler choosing *right* at state $\langle P_1; P_3, \sigma_I \rangle$. In this case there is no path having positive accumulated reward in the induced MC, yielding an expected reward of 0. The probability of not reaching $\langle \lightning \rangle$ is also 0. The conditional expected reward in this case is undefined ($0/0$) and thus the *right* branch is preferred over the *left* branch. In general, the operational MDP need not be finite, even if the program terminates almost–surely (i.e. with probability 1).

We now investigate the connection to the denotational semantics of Section 3, starting with some auxiliary results. First, we establish a relation between (liberal) expected rewards and weakest (liberal) pre–expectations.

Lemma 4.4 *For any fully probabilistic $P \in \mathsf{cpGCL}$, $f \in \mathbb{E}, g \in \mathbb{E}_{\leq 1}$, and $\sigma \in \mathbb{S}$:*

$$\mathsf{ExpRew}^{\mathcal{R}_\sigma^f[\![P]\!]}\left(\Diamond \langle \mathit{sink} \rangle\right) = \mathsf{wp}[P](f)(\sigma) \tag{i}$$

$$\mathsf{LExpRew}^{\mathcal{R}_\sigma^g[\![P]\!]}\left(\Diamond \langle \mathit{sink} \rangle\right) = \mathsf{wlp}[P](g)(\sigma) \tag{ii}$$

Moreover, the probability of never reaching $\langle \lightning \rangle$ in the MC of program P coincides with the weakest liberal pre–expectation of P w.r.t. post–expectation $\mathbf{1}$:

Lemma 4.5 *For any fully probabilistic $P \in \mathsf{cpGCL}$, $g \in \mathbb{E}_{\leq 1}$, and $\sigma \in \mathbb{S}$ we have* $\Pr^{\mathcal{R}_\sigma^g[\![P]\!]}(\neg \Diamond \lightning) = \mathsf{wlp}[P](\mathbf{1})(\sigma)$.

We now have all prerequisites in order to present the main result of this section: the correspondence between the operational and expectation transformer semantics of fully probabilistic cpGCL programs. It turns out that the weakest (liberal) pre–expectation $\underline{\mathsf{cwlp}}[P](f)(\sigma)$ (resp. $\underline{\mathsf{cwlp}}[P](f)(\sigma)$) coincides with the conditional (liberal) expected reward in the RMC $\mathcal{R}_\sigma^f[\![P]\!]$ of terminating while never violating an `observe`–statement, i.e., avoiding the $\langle \lightning \rangle$ states.

Theorem 4.6 (Correspondence theorem) *For any fully probabilistic $P \in \mathsf{cpGCL}$, $f \in \mathbb{E}$, $g \in \mathbb{E}_{\leq 1}$ and $\sigma \in \mathbb{S}$,*

$$\begin{aligned} \mathsf{CExpRew}^{\mathcal{R}_\sigma^f[\![P]\!]}\left(\Diamond \mathit{sink} \,|\, \neg \Diamond \lightning\right) &= \underline{\mathsf{cwp}}[P](f)(\sigma) \\ \mathsf{CLExpRew}^{\mathcal{R}_\sigma^g[\![P]\!]}\left(\Diamond \mathit{sink} \,|\, \neg \Diamond \lightning\right) &= \underline{\mathsf{cwlp}}[P](g)(\sigma)\ . \end{aligned}$$

Proof. The proof makes use of Lemmas 4.4, 4.5, and Lemma 3.2 which are themselves proven by induction on the structure of P. For details see [12, p. 13-14, 16-21]. □

5 Program Transformation

In this section we present a program transformation for removing observations from fully probabilistic cpGCL programs and use the expectation transformer semantics

$$
\begin{array}{lcl}
\mathcal{T}(\texttt{skip}, f) & = & (\texttt{skip}, f) \\
\mathcal{T}(\texttt{abort}, f) & = & (\texttt{abort}, \mathbf{1}) \\
\mathcal{T}(x \coloneqq E, f) & = & (x \coloneqq E, f[E/x]) \\
\mathcal{T}(\texttt{observe}\ (G), f) & = & (\texttt{skip}, \chi_G \cdot f) \\
\mathcal{T}(\texttt{ite}\,(G)\,\{P\}\,\{Q\}, f) & = & (\texttt{ite}\,(G)\,\{P'\}\,\{Q'\}, \chi_G \cdot f_P + \chi_{\neg G} \cdot f_Q) \text{ where} \\
 & & \quad (P', f_P) = \mathcal{T}(P, f),\ (Q', f_Q) = \mathcal{T}(Q, f) \\
\mathcal{T}(\{P\}\ [p]\ \{Q\}, f) & = & (\{P'\}\ [p']\ \{Q'\}, p \cdot f_P + (\mathbf{1}{-}p) \cdot f_Q) \text{ where} \\
 & & \quad (P', f_P) = \mathcal{T}(P, f),\ (Q', f_Q) = \mathcal{T}(Q, f),\ p' = \frac{p \cdot f_P}{p \cdot f_P + (\mathbf{1}-p) \cdot f_Q} \\
\mathcal{T}(\texttt{while}\,(G)\,\{P\}, f) & = & (\texttt{while}\,(G)\,\{P'\}, f') \text{ where} \\
 & & \quad f' = \boldsymbol{\nu}\, X_\bullet\ (\chi_G \cdot (\pi_2 \circ \mathcal{T})(P, X) + \chi_{\neg G} \cdot f)\,,\ (P', \_) = \mathcal{T}(P, f') \\
\mathcal{T}(P; Q, f) & = & (P'; Q', f'') \text{ where } (Q', f') = \mathcal{T}(Q, f),\ (P', f'') = \mathcal{T}(P, f')
\end{array}
$$

Fig. 2. Program transformation for eliminating `observe` statements in fully probabilistic cpGCL programs.

from Section 3 to prove the transformation correct. Intuitively, the presented program transformation "hoists" the `observe` statements while updating the probabilities of the probabilistic choices. Given a fully probabilistic program $P \in \mathsf{cpGCL}$, the transformation delivers a semantically equivalent `observe`–free program $\hat{P} \in \mathsf{pGCL}$ and—as a side product—an expectation $\hat{h} \in \mathbb{E}_{\leq 1}$ that captures the probability that the original program establishes all `observe` statements. For an intuition, reconsider the program from Example 3.1. The transformation yields program

$$\{x \coloneqq 0\}\ [8/13]\ \{x \coloneqq 1\}; \texttt{ite}\ (x{=}1)\ \{\{y \coloneqq 0\}\ [1]\ \{y \coloneqq 2\}\}\{\{y \coloneqq 0\}\ [1]\ \{y \coloneqq 3\}\}$$

and expectation $\hat{h} = \frac{\mathbf{13}}{\mathbf{20}}$. By eliminating dead code in both probabilistic choices and coalescing the branches in the conditional, we can simplify the program to

$$\{x \coloneqq 0\}\ [8/13]\ \{x \coloneqq 1\};\ y \coloneqq 0$$

As a sanity check note that the expected value of $10{+}x$ in this program is equal to $10 \cdot \frac{8}{13} + 11 \cdot \frac{5}{13} = \frac{135}{13}$, which agrees with the result obtained by analyzing the original program. Formally, the program transformation is given by a function

$$\mathcal{T} : \mathsf{cpGCL} \times \mathbb{E}_{\leq 1} \to \mathsf{pGCL} \times \mathbb{E}_{\leq 1}\ .$$

To apply the transformation to a program P we need to determine $\mathcal{T}(P, \mathbf{1})$, which gives the semantically equivalent program $\hat{P}$ and the expectation $\hat{h}$.

The transformation is defined in Figure 2 and works by inductively computing the weakest pre–expectation that guarantees the establishment of all `observe`–statements and updating the probability parameter of probabilistic choices so that the pre–expectations of their branches are established in accordance with the original probability parameter. The computation of these pre–expectations is performed following the same rules as the wlp operator. The correctness of the transformation is established by the following Theorem, which states that a program and its transformed version share the same terminating and non–terminating behavior.

Theorem 5.1 (Program Transformation Correctness) *Let P be a fully probabilistic* cpGCL *program that admits at least one valid run for every initial state and let $\mathcal{T}(P, \mathbf{1}) = (\hat{P}, \hat{h})$. Then for any $f \in \mathbb{E}$ and $g \in \mathbb{E}_{\leq 1}$, we have $\mathsf{wp}[\hat{P}](f) = \underline{\mathsf{cwp}}[P](f)$ and $\mathsf{wlp}[\hat{P}](g) = \underline{\mathsf{cwlp}}[P](g)$.*

Proof. See [12, p. 21]. □

A similar program transformation has been given by Nori *et al.* [22]. Whereas they use random assignments to introduce randomization in their programming model, we use probabilistic choices. Consequently, they can hoist `observe`–statements only until the occurrence of a random assignment, while we are able to hoist `observe`–statements over probabilistic choices and completely remove them from programs. Another difference is that the semantics of Nori *et al.* only accounts for terminating program behaviors and thus they can guarantee the correctness of the program transformation for almost–surely terminating programs only. Our semantics is more expressive and enables establishing the correctness for non–terminating program behavior, too.

6 Denotational Semantics for Full cpGCL

In this section we argue why (under mild assumptions) it is not possible to provide a denotational semantics in the style of conditional pre–expectation transformers (CPETs for short) for full cpGCL, i.e. including non–determinism. To show this, it suffices to consider a simple fragment of cpGCL containing only assignments, observations, probabilistic and non–deterministic choices. Let x be the only program variable that can be written or read in this fragment. We denote this fragment by cpGCL^-. Assume D is some appropriate domain for *representing* conditional expectations of the program variable x with respect to some *fixed* initial state σ_0 and let $[\![\cdot]\!] \colon D \to \mathbb{R} \cup \{\bot\}$ be an interpretation function such that for any $d \in D$ we have that $[\![d]\!]$ *is equal to* the (possibly undefined) conditional expected value of x.

Definition 6.1 [Inductive CPETs] A *CPET* is a function $\mathsf{cwp}^* \colon \mathsf{cpGCL}^- \to D$ such that for any $P \in \mathsf{cpGCL}^-$, $[\![\mathsf{cwp}[P]]\!] = \mathsf{CExpRew}^{\mathfrak{R}^x_{\sigma_0}[\![P]\!]}(\Diamond \mathit{sink} \mid \neg \Diamond \lightning)$. cwp^* is called *inductive*, if there exist two functions $\mathcal{K} \colon \mathsf{cpGCL}^- \times [0,1] \times \mathsf{cpGCL}^- \to D$ and $\mathcal{N} \colon \mathsf{cpGCL}^- \times \mathsf{cpGCL}^- \to D$, such that for any $P_1, P_2 \in \mathsf{cpGCL}^-$ we have $\mathsf{cwp}^*[\{P_1\}\ [p]\ \{P_2\}] = \mathcal{K}(\mathsf{cwp}^*[P_1], p, \mathsf{cwp}^*[P_2])$ and $\mathsf{cwp}^*[\{P_1\} \,\square\, \{P_2\}] = \mathcal{N}(\mathsf{cwp}^*[P_1], \mathsf{cwp}^*[P_2])$, where $\forall d_1, d_2 \in D_\bullet\ \mathcal{N}(d_1, d_2) \in \{d_1, d_2\}$.

This definition suggests that the conditional pre–expectation of $\{P_1\}\ [p]\ \{P_2\}$ is determined only by the conditional pre–expectation of P_1 and P_2, and the probability p. Furthermore the above definition suggests that the conditional pre–expectation of $\{P_1\} \,\square\, \{P_2\}$ is also determined by the conditional pre–expectation of P_1 and P_2 only. Consequently, the non–deterministic choice can be resolved by replacing it either by P_1 or P_2. While this might seem like a strong limitation, the above definition is compatible with the interpretation of non–deterministic choice as demonic choice: The choice is deterministically driven towards the worst option. The requirement $N(d_1, d_2) \in \{d_1, d_2\}$ is also necessary for interpreting non–deterministic choice as an abstraction where implementation details are not important.

As we assume a fixed initial state and a fixed post–expectation, the non–deterministic choice turns out to be deterministic once the pre–expectations of P_1 and P_2 are known. Under the above assumptions (which do apply to the wp and wlp transformers) we claim:

Theorem 6.2 *There exists no inductive CPET.*

Proof Sketch (for details, see [12, p. 11]). By contradiction: Consider the program $P = \{P_1\}\ [1/2]\ \{P_5\}$ with

$$
\begin{aligned}
P_1 &: \quad x \coloneqq 1 \\
P_5 &: \quad \{P_2\} \,\square\, \{P_4\} \\
P_2 &: \quad x \coloneqq 2 \\
P_4 &: \quad \{\texttt{observe false}\}\ [1/2]\ \{P_3\} \\
P_3 &: \quad x \coloneqq 2.2
\end{aligned}
$$

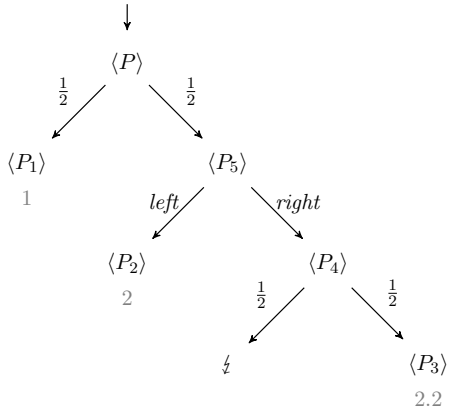


Fig. 3: Schematic depiction of the RMDP $\mathfrak{R}^{x}_{\sigma_0}[\![P]\!]$

A schematic depiction of the $\mathfrak{R}^{x}_{\sigma_0}[\![P]\!]$ is given in Figure 3. Assume there exists an inductive CPET cwp^* over some appropriate domain D. With the program given above, one can get to the contradiction $[\![\mathsf{cwp}^*[P_5]]\!] = [\![\mathsf{cwp}^*[P_4]]\!] > [\![\mathsf{cwp}^*[P_2]]\!] = [\![\mathsf{cwp}^*[P_5]]\!]$. □

As an immediate corollary of Theorem 6.2 we obtain the following result:

Corollary 6.3 *We cannot extend the* cwp *or* cwlp *rules in Figure 1 for non–deterministic programs such that Theorem 4.6 extends to full* cpGCL.

This result is related to Varacca and Winskel's work [27], who have already noticed the difficulties that arise when trying to integrate non–determinism and probabilities, even in the absence of conditioning. When conditioning is taken into account, Andrés and van Rossum [1] have also observed that positional schedulers—i.e. the kind of schedulers implicitly considered in the expectation transformer semantics—are not sufficient for minimizing probabilities. In contrast to our work, their development is done in the context of temporal logics.

7 Conclusion and Future Work

This paper presented an extensive treatment of semantic issues in probabilistic programs with conditioning. Major contributions are the treatment of non–terminating programs (both operationally and for weakest liberal pre–expectations), our results on combining non–determinism with conditioning, as well as the presented program transformation. We firmly believe that a thorough understanding of these semantic issues provides a main cornerstone for enabling automated analysis techniques such as loop invariant synthesis [5,18], program analysis [6] and model checking [3] to the class of probabilistic programs with conditioning. Future work consists of investigating conditional invariants and a further investigation of non–determinism in combination with conditioning.

Acknowledgments. We would like to thank Pedro d'Argenio and Tahiry Rabehaja for the valuable discussions preceding this paper.

References

[1] Andrés, M. E. and P. van Rossum, *Conditional probabilities over probabilistic and nondeterministic systems*, in: *Proc. of TACAS*, LNCS **4963** (2008), pp. 157–172.

[2] Baier, C. and J. Katoen, "Principles of Model Checking," MIT Press, 2008.

[3] Baier, C., J. Klein, S. Klüppelholz and S. Märcker, *Computing conditional probabilities in Markovian models efficiently*, in: *Proc. of TACAS*, LNCS **8413** (2014), pp. 515–530.

[4] Barthe, G., B. Köpf, F. Olmedo and S. Z. Béguelin, *Probabilistic relational reasoning for differential privacy*, ACM Trans. Program. Lang. Syst. **35** (2013), p. 9.

[5] Chakarov, A. and S. Sankaranarayanan, *Expectation invariants for probabilistic program loops as fixed points*, in: *Proc. of SAS*, LNCS **8723** (2014), pp. 85–100.

[6] Cousot, P. and M. Monerau, *Probabilistic abstract interpretation*, in: H. Seidl, editor, *Proc. of ESOP*, LNCS **7211** (2012), pp. 169–193.

[7] Dijkstra, E. W., "A Discipline of Programming," Prentice Hall, 1976.

[8] Goodman, N. D., V. K. Mansinghka, D. M. Roy, K. Bonawitz and J. B. Tenenbaum, *Church: a language for generative models*, in: *Proc. of UAI* (2008), pp. 220–229.

[9] Goodman, N. D. and A. Stuhlmüller, "The Design and Implementation of Probabilistic Programming Languages." (electronic), 2014, http://dippl.org.

[10] Gordon, A. D., T. Graepel, N. Rolland, C. V. Russo, J. Borgström and J. Guiver, *Tabular: a schema-driven probabilistic programming language*, in: *Proc. of POPL* (2014), pp. 321–334.

[11] Gordon, A. D., T. A. Henzinger, A. V. Nori and S. K. Rajamani, *Probabilistic programming*, in: *Proc. of FOSE* (2014), pp. 167–181.

[12] Gretz, F., N. Jansen, B. L. Kaminski, J.-P. Katoen, A. McIver and F. Olmedo, *Conditioning in probabilistic programming*, CoRR (2015).

[13] Gretz, F., J.-P. Katoen and A. McIver, *Operational versus weakest pre-expectation semantics for the probabilistic guarded command language*, Perform. Eval. **73** (2014), pp. 110–132.

[14] Gupta, V., R. Jagadeesan and P. Panangaden, *Stochastic processes as concurrent constraint programs*, in: *Proc. of POPL* (1999), pp. 189–202.

[15] Gupta, V., R. Jagadeesan and V. A. Saraswat, *Probabilistic concurrent constraint programming*, in: *Concurrency Theory*, LNCS **1243** (1997), pp. 243–257.

[16] Hur, C.-K., A. V. Nori, S. K. Rajamani and S. Samuel, *Slicing probabilistic programs*, in: *Proc. of PLDI* (2014), pp. 133–144.

[17] Jones, C. and G. D. Plotkin, *A probabilistic powerdomain of evaluations*, in: *Logic in Computer Science* (1989), pp. 186–195.

[18] Katoen, J.-P., A. McIver, L. Meinicke and C. C. Morgan, *Linear-invariant generation for probabilistic programs*, in: *Proc. of SAS*, LNCS **6337** (2010), pp. 390–406.

[19] Kozen, D., *Semantics of probabilistic programs*, J. Comput. Syst. Sci. **22** (1981), pp. 328–350.

[20] Kozen, D., *A probabilistic {PDL}*, Journal of Computer and System Sciences **30** (1985), pp. 162 – 178.

[21] McIver, A. and C. Morgan, "Abstraction, Refinement And Proof For Probabilistic Systems," Springer, 2004.

[22] Nori, A. V., C.-K. Hur, S. K. Rajamani and S. Samuel, *R2: An efficient MCMC sampler for probabilistic programs*, in: *Proc. of AAAI* (2014).

[23] Paige, B. and F. Wood, *A compilation target for probabilistic programming languages*, in: *Proc. of ICML*, JMLR Proceedings **32** (2014), pp. 1935–1943.

[24] Puterman, M., "Markov Decision Processes: Discrete Stochastic Dynamic Programming," John Wiley and Sons, 1994.

[25] Sampson, A., P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman and L. Ceze, *Expressing and verifying probabilistic assertions*, in: *Proc. of PLDI* (2014), p. 14.

[26] Scott, D. S., *Stochastic λ-calculi: An extended abstract*, J. Applied Logic **12** (2014), pp. 369–376.

[27] Varacca, D. and G. Winskel, *Distributing probability over non-determinism*, Mathematical. Structures in Comp. Sci. **16** (2006), pp. 87–113.

D. Prinsys – On a Quest for Probabilistic Loop Invariants

This paper forms the basis of Chapter 4.

PRINSYS—on a Quest for Probabilistic Loop Invariants *

Friedrich Gretz[1,2], Joost-Pieter Katoen[1], and Annabelle McIver[2]

[1] RWTH Aachen University, Germany
lastname@cs.rwth-aachen.de,
[2] Macquarie University, Australia
firstname.lastname@mq.edu.au

Abstract. PRINSYS (pronounced "princess") is a new software-tool for probabilistic invariant synthesis. In this paper we discuss its implementation and improvements of the methodology which was set out in previous work. In particular we have substantially simplified the method and generalised it to non-linear programs and invariants. PRINSYS follows a constraint-based approach. A given parameterised loop annotation is speculatively placed in the program. The tool returns a formula that captures precisely the invariant instances of the given candidate. Our approach is sound and complete. PRINSYS's applicability is evaluated on several examples. We believe the tool contributes to the successful analysis of sequential probabilistic programs with infinite-domain variables and parameters.

Keywords: invariant generation, probabilistic programs, non-linear constraint solving

1 Introduction

Motivation. Probabilistic programs are pivotal in different application fields like security, privacy [2]—several probabilistic protocols (e.g. onion-routing) aim to ensure privacy, and there is an increasing interest in the topic, partly driven by the social-media world—and cryptography [1] as well as quantum computing [13]. Such programs are single threaded and typically consist of a small number of code lines, but are hard to understand and analyse. The two major reasons for their complexity are the occurrence of program variables with unbounded domains, and parameters. Such parameters can be either loop bounds, number of participants (in a protocol), or probabilistic choices where the parameters range over concrete probabilities. For example, the following simple program generates a sample x according to a geometric distribution with parameter p. In every loop iteration, the variable x is increased by one with probability $1-p$

* This work is partially funded by the DFG Research Training Group Algosyn, the EU FP7 Project CARP (Correct and Efficient Accelerator Programming), and the EU MEALS exchange project with Latin America.

Listing 1. $x \sim \text{geom}(p)$

```
x := 0;
flip := 0;
while (flip = 0) {
    ( flip := 1 [p] x := x+1 );
}
```

and *flip* is set to one with probability p, where p is an unknown real value from the range $(0, 1)$. The occurrence of unbounded variables and parameters comes at a price, namely that probabilistic programs in general cannot be analysed automatically by model-checking tools such as PRISM [10], PARAM [6], PASS [5] or APEX [9].

Approach. Instead we resort to deductive techniques. Recall that one of the main approaches to the verification of sequential programs rests on the pioneering work of Floyd, Hoare, and Dijkstra in which annotations are associated with control points in the program. Whereas the annotations for sequential programs are qualitative and can be expressed in predicate logic, quantitative annotations are needed to reason about probabilistic program correctness. McIver and Morgan [11] have extended the method of Floyd, Hoare, and Dijkstra to probabilistic programs by making the annotations real- rather than Boolean-valued expressions in the program variables. Using these methods we can prove that in the above program the average value of x is $\frac{1-p}{p}$. Annotating a probabilistic program with such expressions is non-trivial and undecidable in general. The main reason is the occurrence of loops. This all boils down to the question on how to establish a loop invariant. It is known that this is a notorious hard problem for traditional programs. For probabilistic programs it is even more difficult as loop invariants are quantitative—so-called probabilistic loop invariants. Variables do no longer have a value, but have a certain value with a given likelihood. Finding an invariant is hard and requires both ingenuity as well as involved computations to check that a given expression is indeed invariant. Recently, Katoen *et al.* [7] have proposed a technique for finding linear invariants for linear probabilistic programs. Linearity refers to the fact that right-hand sides of assignments and guards are linear expressions in the program variables (and parameters). This technique is based on speculatively annotating a loop with a template (in fact a linear inequality) and using constraint solving techniques to distill all parameters for which the template is indeed a loop invariant.

Contributions of this paper. The contributions of this paper are manifold. First and foremost, this paper presents PRINSYS (pronounce "princess"), a novel tool for supporting the semi-automated generation of probabilistic invariants

of pGCL[3] programs. This publicly available tool implements the technique advocated in [7], i.e., automatically computes the constraints under which a user-provided template is invariant, saving the user from tedious and error prone calculations. To the best of our knowledge, it is the first tool for synthesizing probabilistic invariants. Secondly, we show that the theory in [7] can be considerably simplified. In particular, we show that the usage of Motzkin's transposition theorem (a generalisation of Farkas' lemma) to turn an existentially quantified formula into a universally quantified one, is not needed. As a result, PRINSYS allows arbitrary formulas in templates and program guards. This allows for polynomial invariant templates and non-linear program expressions. So, an immediate consequence of this simplification is that the restriction to linear programs and linear invariants can be dropped. This is more of theoretical interest than of practical interest, as polynomial invariants—as for the traditional, non-probabilistic setting—are hard to synthesize in practice. Finally, we present some applications of the tool such as proving the equivalence of two programs computing a sample from $X-Y$ where X and Y are both geometrically distributed, and the generation of a fair coin from a biased one. We evaluate the experiments and give directions for future research.

Organization of the paper. Section 2 provides the preliminaries such as pGCL, probabilistic invariants, and expectations. Section 3 presents the steps of our approach and the simplification of [7]. Section 4 provides three examples to give insight about what PRINSYS can establish. Section 5 evaluates the tool and approach, whereas Sect. 6 concludes the paper and provides pointers to future work.

2 Background

When probabilistic programs are executed they determine a probability distribution over final values of program variables. For instance, on termination of

$$(x := 1 \ [0.75] \ x := 2);$$

the final value of x is 1 with probability $\frac{3}{4}$ or 2 with probability $1 - \frac{3}{4} = \frac{1}{4}$. An alternative way to characterise that probabilistic behaviour is to consider the expected values over random variables with respect to that distribution. For example, to determine the probability that x is set to 1, we can compute the expected value of the random variable "x is 1" which is $\frac{3}{4} \cdot 1 + \frac{1}{4} \cdot 0 = \frac{3}{4}$. Similarly, to determine the average value of x, we compute the expected value of the random variable "x" which is $\frac{3}{4} \cdot 1 + \frac{1}{4} \cdot 2 = \frac{5}{4}$. More generally, rather than a distribution-centred approach, we take an "expectation transformer" [11] approach. We annotate probabilistic programs with *expectations*.

[3] pGCL extends Dijkstra's guarded command language with a probabilistic choice operator.

Expectations. Expectations map program states to non-negative real values. They generalise Hoare's predicates for non-probabilistic programs towards real-valued functions. Intuitively, implication between predicates is generalised to pointwise inequality between expectations. For convenience we use square brackets to link Boolean truth values to numbers and by convention [true] = 1 and [false] = 0. In the example above, we call "x" the *post-expectation* and $\frac{5}{4}$ its *pre-expectation*. Thus the annotated program is $\langle \frac{5}{4} \rangle\ (x := 1\ [0.75]\ x := 2);\ \langle x \rangle$.

The formal mechanism for computing pre-expectations for a given program and post-expectation is the expectation transformer semantics [11]. Expectation transformers are the quantitative pendant to Dijkstra's predicate transformers. McIver and Morgan extend Dijkstra's concept and introduce a function wp(*prog*,*post*) which based on the program *prog* determines the *greatest pre-expectation* for any given post-expectation *post*. A summary of pGCL's expectation transformer semantics is given in Table 1 where f is a given post-expectation. From an operational perspective, pGCL programs can be viewed as (infinite state) MDPs with a reward structure induced by the given post-expectation f. Then the greatest pre-expectation can be computed as the expected cummulative reward on that model [4].

syntax *prog*	**semantics** wp(*prog*,f)
skip	f
abort	0
x := E	$f[x/E]$
P ; Q	$\mathrm{wp}(P, \mathrm{wp}(Q, f))$
if (G) { P } **else** { Q }	$[G] \cdot \mathrm{wp}(P, f) + [\neg G] \cdot \mathrm{wp}(Q, f)$
P [] Q	$\min\{\mathrm{wp}(P, f), \mathrm{wp}(Q, f)\}$
P [p] Q	$p \cdot \mathrm{wp}(P, f) + (1 - p) \cdot \mathrm{wp}(Q, f)$
while (G) { P }	$\mu X.([G] \cdot \mathrm{wp}(P, X) + [\neg G] \cdot f)$

Table 1. Syntax and expectation transformer semantics of pGCL

For loop-free programs, the pre-expectation is simply given by syntactic rules. However, loops pose a problem because their expectation over final values is given in terms of a least fixed point (over the domain of expectations with the ordering $\leq$, a pointwise ordering on expectations).

Invariants. Using special expectations which we call *invariants* we can avoid the calculation of a loop's fixed point. Assume we are given two expectations *pre* and *post* and we want to show that *pre* is a lower bound on the loop's actual pre-expectation, i.e.

$$pre \leq \mathrm{wp}(\mathbf{while}(G)\{body\}, post)\ .$$

Instead of computing the greatest pre-expectation $\mathrm{wp}(\mathbf{while}(G)\{\mathit{body}\}, \mathit{post})$ directly, it is more practical to divide this problem into simpler subtasks:

1. find an expectation $\mathcal{I}$ such that

$$\mathit{pre} \leq \mathcal{I} \quad \text{and} \quad \mathcal{I} \cdot [\neg G] \leq \mathit{post} \ ,$$

2. show $\mathcal{I}$ is *invariant*[4], that is $\mathcal{I} \cdot [G] \leq \mathrm{wlp}(\mathit{body}, \mathcal{I})$
3. show $\mathcal{I}$ is sound, that is $\mathcal{I} \leq \mathrm{wp}(\mathbf{while}(G)\{\mathit{body}\}, \mathcal{I} \cdot [\neg G])$

Points 2. and 3. may seem odd as they resemble the original problem of proving an inequality between an expectation and the greatest pre-expectation of a loop. However they are easier than the original problem, because in 2. the greatest pre-expectation can be explicitly computed because *body* is a loop-free program. In order to guarantee soundness (point 3.) the loop must terminate with probability one and the invariant $\mathcal{I}$ has to additionally meet one of the following sufficient conditions [11]:

- from every initial state of the loop only a finite state space is reachable
- or $\mathcal{I}$ is bounded above by some fixed constant
- or $\mathrm{wp}(\mathit{body}, \mathcal{I} \cdot [G])$ tends to zero as the number of iterations tends to infinity.

Remark 1. It is an open problem to give the *necessary and sufficient* conditions for soundness.

Put all together this proves the inequality above as

$$\mathit{pre} \leq \mathcal{I} \leq \mathrm{wp}(\mathbf{while}(G)\{\mathit{body}\}, \mathcal{I} \cdot [\neg G]) \leq^{5} \mathrm{wp}(\mathbf{while}(G)\{\mathit{body}\}, \mathit{post}) \ .$$

Example 1 (Application of invariants.). Consider the program *prog* in Lst. 2. On each iteration of the loop it sets x to -1 with probability 0.15, to 0 with probability 0.5 and to 1 with probability 0.35. We would like to prove that the probability to terminate in a state where $x = 1$ is 0.7 or equivalently

$$\mathrm{wp}(\mathit{prog}, [x = 1]) = 0.7 \ .$$

Instead of computing the least fixed point of the loop wrt. post-expectation $[x = 1]$, we can show that $\mathcal{I} = [x = 0] \cdot 0.7 + [x = 1]$ is invariant. If the loop terminates, we can establish:

$$\begin{aligned} [\neg G] \cdot \mathcal{I} &= [x \neq 0] \cdot [x = 0] \cdot 0.7 + [x = 1] \\ &= [x = 1] \ . \end{aligned}$$

[4] wlp is the "liberal" version of wp. Both expectation transformers coincide for almost surely terminating programs. Since in this paper we do not consider nested loops, i.e. *body* is loop-free (and hence surely terminates), we do not discuss the theoretical differences between wp and wlp here.

[5] wp is monotonic in its second argument [11].

Listing 2. A simple loop

```
x := 0;
while (x=0) {
      (x := 0;) [0.5] { (x := -1 [0.3] x := 1); }
}
```

At the beginning of the program the initialisation of x transforms the invariant to:

$$\begin{aligned} \mathrm{wp}(x := 0, \mathcal{I}) &= [0 = 0] \cdot 0.7 + [0 = 1] \\ &= 0.7 \ . \end{aligned}$$

In this way we obtain the annotation

$$\langle 0.7 \rangle \ x := 0; \ \langle \mathcal{I} \rangle \ \textbf{while}(x = 0)\{\ldots\} \ \langle [x \neq 0] \cdot [\mathcal{I}] = [x = 1] \rangle$$

as desired. It is sound because the program obviously terminates with probability one and $\mathcal{I}$ is bounded.

The crucial point in determining a pre-expectation of a program is to discover the necessary loop invariants for each loop. Checking soundness and carrying out subsequent calculations for the other program constructs turns out be easy in practice. In the following section we explain our approach to finding invariants step by step.

3 Our Approach

To explain the steps carried out by Prinsys we revisit the geometric distribution program from Lst. 1. In the next section, we will view it in a broader context.

Template. Consider the loop:

$$\textbf{while} \ (\mathrm{flip} \ = 0)\{ \ (\mathrm{flip} \ := 1 \ [\mathrm{p}] \ \mathrm{x} := \mathrm{x}{+}1); \ \}$$

and an expectation

$$\mathcal{T}_\alpha = [x \geq 0] \cdot x + [x \geq 0 \wedge \mathit{flip} = 0] \cdot \alpha$$

where α is an unknown (real) parameter. We call $\mathcal{T}_\alpha$ a *template*. Replacing α by a real value yields an *instance* of the template. Depending on this value, some instances may satisfy the invariance condition $\mathcal{T}_\alpha \cdot [G] \leq \mathrm{wlp}(\mathit{body}, \mathcal{T}_\alpha)$.

Goal. Prinsys gives a characterisation of all invariant instances of a given template. This characterisation is a formula which is true for all admissible values of the template parameters, α in our example. It is important to stress that this method is *complete* in the sense that for any given template the resulting formula captures *precisely* the invariant instances.

Workflow. Stage 1: After parsing the program text and template, PRINSYS traverses the generated control flow graph of the program and computes:

$$\begin{aligned}&\mathrm{wp}(\mathit{flip} := 1\ [\mathrm{p}]\ \mathrm{x} := \mathrm{x}{+}1,\ \mathcal{T}_\alpha)\\ &= [x \geq 0] \cdot px + (1-p) \cdot ([x+1 \geq 0] \cdot (x+1) + [x+1 \geq 0 \wedge \mathit{flip} = 0] \cdot \alpha)\quad .\end{aligned}$$

For details, cf. Table 1. After expanding this expression, the invariance condition amounts to:

$$\begin{aligned}\overbrace{[x \geq 0 \wedge \mathit{flip} = 0] \cdot (x+\alpha)}^{\mathcal{T}_\alpha \cdot [G]} \leq \quad & [x \geq 0] \cdot px \\ & + [x+1 \geq 0] \cdot ((1-p)x - p + 1) \\ & \underbrace{+[x+1 \geq 0 \wedge \mathit{flip} = 0] \cdot (1-p)\alpha}_{\mathrm{wlp}(\mathit{body}, \mathcal{T}_\alpha)}\quad .\end{aligned}$$

Our goal is to find all α such that the point-wise inequality is satisfied, i.e. it holds for every x and every *flip*. This can be done by pairwise comparison of the summands on the left-hand side and the right-hand side. But summands may overlap. This makes it necessary to rewrite the expectations in disjoint normal form (DNF).

Theorem 1 (Transformation to DNF [7]). *Given an expectation of the form*

$$f = [P_1] \cdot w_1 + \ldots + [P_n] \cdot w_n.$$

Then an equivalent expectation in DNF can be written as:

$$\sum_{I \in \mathcal{P}(\overline{n}) \setminus \emptyset} \left(\left[\bigwedge_{i \in I} P_i \wedge \neg \left(\bigwedge_{j \in \mathcal{P}(\overline{n}) \setminus I} P_j \right) \right] \cdot \left(\sum_{i \in I} w_i \right) \right)$$

where $\overline{n}$ is the index set $\{1, \ldots, n\}$ and $\mathcal{P}(\cdot)$ denotes the power set.

The left-hand side of the inequality for the example program above is already in DNF as there is only one summand. We apply the transformation to the right-hand side expression. The result is an expectation with 15 summands. For better readability we only show the summands that are not trivially zero:

$$\begin{aligned}&[x+1 \geq 0 \wedge x < 0 \wedge \mathit{flip} = 0)] \cdot ((1-p)x + (1-p)\alpha - p + 1) \\ &+[x \geq 0 \wedge \mathit{flip} = 0)] \cdot (x + (1-p)\alpha - p + 1) \\ &+[x+1 \geq 0 \wedge x < 0 \wedge \mathit{flip} \neq 0] \cdot ((1-p)x - p + 1) \\ &+[x \geq 0 \wedge \mathit{flip} \neq 0] \cdot (x - p + 1)\,.\end{aligned}$$

The following theorem provides a straightforward encoding of the inequality as a first-order formula.

Theorem 2. *Given two expectations over variables $x_1, \ldots, x_n$ in disjoint-normal form*

$$f = [P_1] \cdot u_1 + \ldots + [P_M] \cdot u_M \ , \qquad g = [Q_1] \cdot w_1 + \ldots + [Q_K] \cdot w_K \ .$$

The inequality $f \leq g$ holds if and only if

$$\begin{aligned} \forall x_1, \ldots, x_n \in \mathbb{R} : & \bigwedge_{m \in \overline{M}} \bigwedge_{k \in \overline{K}} (P_m \wedge Q_k \Rightarrow (u_m - w_k \leq 0)) \\ & \wedge \bigwedge_{m \in \overline{M}} \left(P_m \wedge \left(\bigwedge_{k \in \overline{K}} \neg Q_k \right) \Rightarrow u_m \leq 0 \right) \\ & \wedge \bigwedge_{k \in \overline{K}} \left(Q_k \wedge \left(\bigwedge_{m \in \overline{M}} \neg P_m \right) \Rightarrow 0 \leq w_k \right) \end{aligned}$$

holds, where $\overline{X}$ is the set of indices $\{1, 2, \ldots, X\}$.

The idea is that we consider individual summands on the left-hand and right-hand side of the inequality and compare their values. It may also be the case that for some evaluations, all predicates on the right-hand side are false and hence the expectation is zero (i.e., the zero function). Then it must be ensured that no summand is greater than zero on the left-hand side. Conversely, if none of the predicates on the left-hand side are satisfied, the summands on the right-hand side may be no less than zero.

Theorem 2 originally appears in [7] where the last case is omitted because expectations are assumed to be non-negative by definition. However it is crucial to encode such informal assumptions in the formula as the tools are not aware of such expectation properties and instead treat them as usual functions over real values. This issue remained undiscovered until its implementation in Prinsys caused incorrect results. The lesson learned is that bridging the gap between an idea and a working implementation requires more than "just" coding.

Continuing our example, the (simplified) first-order formula obtained is:

$$\begin{aligned} \forall x, \mathit{flip} : & (\alpha p + p - 1 \leq 0 \vee \mathit{flip} \neq 0 \vee x < 0) \\ & \wedge (\alpha p - \alpha + px + p - x - 1 \leq 0 \vee \mathit{flip} \neq 0 \vee x + 1 < 0 \vee x \geq 0) \\ & \wedge (\mathit{flip} = 0 \vee px + p - x - 1 \leq 0 \vee x + 1 < 0 \vee x \geq 0) \\ & \wedge (\mathit{flip} = 0 \vee p - x - 1 \leq 0 \vee x < 0) \, . \end{aligned}$$

The calculation of this formula by Prinsys concludes the first stage.

Stage 2: The formula is passed to Redlog which simplifies the formula by quantifier elimination. Sometimes the result returned by Redlog still contains redundant information and can be further reduced by its built-in simplifiers or by the Slfq tool. In the end the user is presented a formula that characterises all αs that make $\mathcal{T}_\alpha$ invariant:

$$\alpha p + p - 1 \geq 0 \ .$$

Listing 3. Annotated program from Lst. 1

$\langle \frac{1-p}{p} \rangle$
x := 0;
flip := 0;
$\langle [x \geq 0] \cdot x + [x \geq 0 \wedge \mathit{flip} = 0] \cdot \frac{1-p}{p} \rangle$
while (flip = 0) {
    (flip := 1 [p] x := x+1);
}
$\langle x \rangle$

We pick the greatest admissible α and obtain an invariant:

$$\mathcal{T}_{\frac{1-p}{p}} = [x \geq 0] \cdot x + [x \geq 0 \wedge \mathit{flip} = 0] \cdot \frac{1-p}{p} \ .$$

This can be used to prove that the program in Lst. 1 has an average outcome of $\frac{1-p}{p}$ which indeed is the mean of a geometric distribution with parameter p. The annotated program now looks as follows: The soundness of our invariant is given because there is always a non-zero probability to exit the loop, cf. definition of invariants above.

Figure 1 pictures the described workflow of PRINSYS.

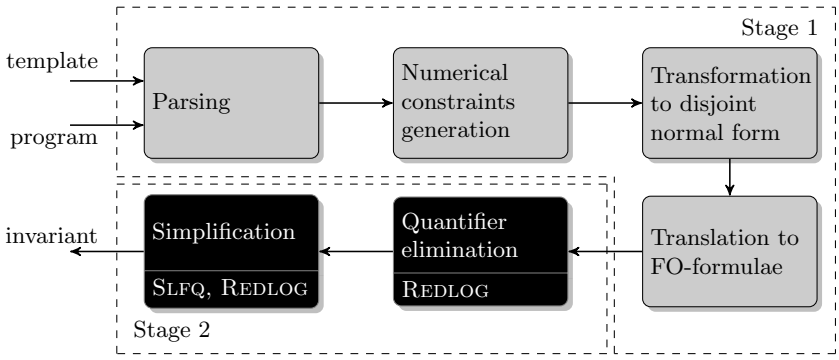


Fig. 1. Tool chain workflow

New insights. There are major differences with the approach sketched in [7]. In PRINSYS we skip the additional step of translating the universally quantified formula into an existential one using the Motzkin's transposition theorem. This step turns out to be not necessary. In fact it complicates matters as the existential formula will have more quantified variables which is bad for quantifier

Listing 4.

```
c := IC; // capital c (is set to some InitialCapital)
b := 1; // initially bet one unit
rounds := 0; //number of rounds played (survived)
while (b > 0){
        {// win with probability p
         c := c+b;
         b := 0;}
        [p]
        {// lose with probability 1-p
         c := c-b;
         b := 2*b;}
        rounds := rounds+1;
}
```

elimination. Furthermore, Motzkin's transposition theorem requires the universally quantified formula to be in a particular shape. Our implementation however does not have these restrictions and allows arbitrary predicates in the program's guards and in templates. Also the template and program do not have to be linear (theoretically at least) because Redlog and Slfq can work with polynomials. Moreover the invariant generation method remains complete in this case. This is because starting with the invariance condition all subsequent steps to obtain the simplified first-order formula are equivalence transformations.

This section has not only illustrated how the tool-chain works but also clearly shows the great amount of calculations that are done automatically for the user. Within seconds the user may try out different templates and play with the parameters until an invariant is found. The Prinsys tool saves the user a lot of tedious, error-prone work and pushes forward the automation of probabilistic program analysis.

4 Applications

This section presents three examples, for simplicity all based on our running example of the geometric distribution, that illustrate the possibilities of the Prinsys approach. Let us start with a relatively simple example.

Martingale betting strategy. Another variant of the geometric distribution appears in the following program, which models a gambler with infinite resources who is playing according to the martingale strategy. Note that this program has two unbounded variables. Using the same template as before, we discover that $\frac{1}{p}$ is the expected number of rounds played before the gambler stops. The expecta-

Listing 5.

```
x := 0;
flip := 0;
while (flip = 0) {
   (x := x+1 [p] flip := 1);
}
flip := 0;
while (flip = 0) {
   (x := x-1 [q] flip := 1);
}
```

Listing 6.

```
x := 0;
(flip := 0 [0.5] flip := 1);
if (flip = 0) {
   while (flip = 0) {
     (x := x+1 [p] flip := 1);
   }
} else {
   flip := 0;
   while (flip = 0) {
     x := x-1;
     (skip [q] flip := 1);
   }
}
```

tion differs from what we have computed for the program in Lst. 1 because here the counter is increased also on the last iteration before the loop terminates.

Geometric distribution. This example is taken from [8] where amongst others it has been shown that the two programs in Lst. 5 and Lst. 6 are equivalent for $p = \frac{1}{2}$ and $q = \frac{2}{3}$. The proof in [8] relies on language equivalence checking of probabilistic automata. Here, we show how the techniques supported by PRINSYS can be used to show that both programs are equivalent for any p and q satisfying $q = \frac{1}{2-p}$. Let us explain the example in more detail. The aim is to generate a sample x according to the distribution $X-Y$ where X is geometrically distributed with parameter $1-p$ and Y is geometrically distributed with $1-q$. Although it is not common to say that a distribution has a parameter $1-p$, it is natural in the context of these programs where x is manipulated with probability p and the loop is terminated with the remaining probability. The difference between the programs in Lst. 5 and Lst. 6 is that the first uses two loops in sequence whereas the latter needs only one out of two loops. Our goal is to determine when the two programs are equivalent, in the sense that they compute the same value for x on average.

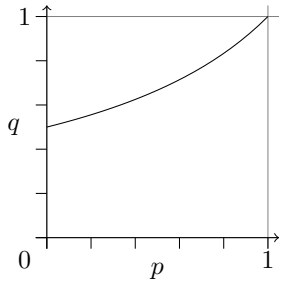


Fig. 2. Pairs (p, q) for which the programs in List. 5 and List. 6 produce the same x on average.

The PRINSYS tool generates invariants for single loops, so we consider each loop separately. Using the template $\mathcal{T}_\alpha = [x \geq 0] \cdot x + [x \geq 0 \wedge \mathit{flip} = 0] \cdot \alpha$ from our running example, PRINSYS yields the following invariants:

Listing 7. x is set to zero or one, each with probability 0.5

```
x := 0; // stores outcome of first biased coin flip
y := 0; // stores outcome of second biased coin flip

while (x-y = 0) {
    (x := 0 [p] x := 1);
    (y := 0 [p] y := 1);
}
```

- $\mathcal{I}_{11} = x + [\text{flip} = 0] \cdot \frac{p}{1-p}$,
- $\mathcal{I}_{12} = x + [\text{flip} = 0] \cdot \left(-\frac{q}{1-q}\right)$,
- $\mathcal{I}_{21} = \mathcal{I}_{11}$ and
- $\mathcal{I}_{22} = x + [\text{flip} = 0] \cdot \left(-\frac{1}{1-q}\right)$,

where $\mathcal{I}_{ij}$ is the invariant of the j-th loop in program i, $i, j \in \{1, 2\}$. With these invariants we can easily derive the expected value of x, which is $\frac{p}{1-p} - \frac{q}{1-q}$ and $\frac{p}{2(1-p)} - \frac{1}{2(1-q)}$ for the program in List. 5 and List. 6, respectively. The two programs thus are equivalent whenever these two expectations coincide; e. g. this is the case for $p = \frac{1}{2}$ and $q = \frac{2}{3}$ as discussed in [8]. Figure 2 visualises our result: for every point (p, q) on the graph the two programs are equivalent. This result cannot be obtained using the techniques in [8]; to the best of our knowledge there are no other automated techniques that can establish this.

Generating a fair coin from a biased coin. In [7], Hurd's algorithm to generate a sample according to a biased coin flip using only fair coin flips has been analysed. Using PRINSYS the calculations can be automated. This was elaborated in [3]. Here we consider an algorithm for the opposite problem. Using a coin with some arbitrary bias $0 < p < 1$, the algorithm in Lst. 7 generates a sample according to a fair coin flip. The loop terminates when the biased coin was flipped twice and showed different outcomes. Obviously the program terminates with probability one as on each iteration of the loop there is a constant positive chance to terminate. The value of x is taken as the outcome. The two possible outcomes are characterised by $x = 0 \wedge y = 1$ and $x = 1 \wedge y = 0$. We encode these two possibilities in the template:

$$[x = 0 \wedge y - 1 = 0] \cdot (\alpha) + [x - 1 = 0 \wedge y = 0] \cdot (\beta)$$

PRINSYS returns one constraint:

$$\alpha p^2 - \alpha p + \beta p^2 - \beta p \leq 0$$

As before we look for the maximum value, hence we consider equality with zero. The equation simplifies to $\alpha = -\beta$ because we know that $0 < p < 1$. Hence

$[x = 0 \land y - 1 = 0] - [x - 1 = 0 \land y = 0]$ is invariant[6] which, together with almost sure termination, gives us

$$\begin{aligned} &\mathrm{wp}(\mathit{prog}, [x = 0 \land y - 1 = 0] - [x - 1 = 0 \land y = 0]) \\ &= \mathrm{wp}(\mathit{prog}, [x = 0 \land y - 1 = 0]) - \mathrm{wp}(\mathit{prog}, [x - 1 = 0 \land y = 0]) \\ &= 0 \ . \end{aligned} \tag{1}$$

where *prog* is the entire program from Lst 7. The previous argument about almost sure termination and possible outcomes shows that

$$\begin{aligned} &\mathrm{wp}(\mathit{prog}, [x = 0 \land y - 1 = 0] + [x - 1 = 0 \land y = 0]) \\ &= \mathrm{wp}(\mathit{prog}, [x = 0 \land y - 1 = 0]) + \mathrm{wp}(\mathit{prog}, [x - 1 = 0 \land y = 0]) \\ &= 1 \ . \end{aligned} \tag{2}$$

The unique solution to (1) and (2) is

$$\begin{aligned} &\mathrm{wp}(\mathit{prog}, [x = 0 \land y - 1 = 0]) \\ &= \mathrm{wp}(\mathit{prog}, [x - 1 = 0 \land y = 0]) \\ &= 0.5 \ . \end{aligned}$$

This concludes the proof that x is distributed evenly for any p satisfying $0 < p < 1$.

5 Evaluation

We have seen three pGCL programs that were variants of the geometric distribution. Our approach allows us to exploit their common structure and enables us to calculate the expectation of these programs using the same template although they compute different (mean) values. Since our method does not rely on numerical calculation we are able to parameterise the programs and provide very general results. In particular we could decide when two programs have the same expectation depending on their parametric distributions. Another handy feature of reasoning with expectation-transformer wp is that we can exploit its properties as well. For example, the reasoning is modular with respect to sequential composition. That means we can compute the pre-expectation for individual loops and then add the results when we put the loops in sequence. The last example demonstrates yet another use of invariants. Instead of deriving a bound on the pre-expectation we have shown how an invariant may give constraints on the pre-expectation. Together with termination these constraints produced the sought pre-expectation. This exemplifies that invariants are not just a particular way to compute an expectation but rather they describe the behaviour of the program and can be used in different ways.

[6] We pick $\alpha = 1$ and $\beta = -1$ but in fact any non-zero pair of values $\alpha = -\beta$ would result in the same argument.

Together with the three (other) examples discussed in [4,7] we have a set of interesting programs which we can analyse with the help of Prinsys. Note, that our examples do not make use of the non-deterministic choice statement. This is because the algorithms we focused on do not need it, however Prinsys also supports non-deterministic pGCL programs. There is no commonly accepted benchmark suite that we can compare against as this area of research has not spawned many tools yet. We refrain from giving a table that shows for each program the state space size, the number of discovered invariants or running times. This is because the beauty of this approach is exactly that the number of states does not matter. In fact all programs that generate (a variant) of the geometric distribution have an infinite set of reachable states! The number of discovered invariants cannot be really be given as, first of all the result depends on the template provided and second we get a characterisation of *all* invariant instances of a template. Since we reason over the reals there are uncountably many.

The runtime of Prinsys depends on the size of the expressions that we have to handle. This means that if we have many choices in the loop (i.e. there are many paths in the control flow graph) this will blow up the size of $\mathrm{wp}(\mathit{body}, \mathcal{T})$. The same is true for templates that have many summands. Finally, the external tools used by Prinsys affect the overall running time. Their execution time cannot be predicted exactly but experience shows that the final simplification step takes considerably longer the more parameters we allow in the template. The overall runtime for the presented examples lies within a second on a laptop computer.

Since there is no software that could be easily adapted to support our methods, Prinsys was developed from scratch. It was recently redesigned to be more extensible and easier to maintain as we hope that future developments in the area of constraint-based methods will use our work as a basis. From the user's point of view, the usability was substantially increased with the introduction of a graphical user interface that allows an intuitive interaction.

Programs and templates considered in our examples are linear. This means all guards, assignments or terms are linear in the program variables. As pointed out earlier, our approach per se allows polynomial expressions as well. To see to what extent this applies in practice we have tried to generate polynomial invariants for variants of a bounded random walk, cf. Lst. 8. The goal is here to estimate the number of steps taken before x hits its lower bound zero or upper bound M where M is a fixed parameter. Surprisingly quantifier elimination works reasonably fast for formulas with polynomials but the returned quantifier-free formula is very big. The lack of powerful simplification methods makes it difficult to find a concise representation of the formula that describes all invariant instances of the template. Redlog's simplifier might increase the formula size or not terminate at all, whereas Slfq hits the memory bound quickly and crashes, even if the allocated memory is increased maximally.

Listing 8. Bounded random walk

```
counter := 0;
while (x > 0 and x-M < 0){
    (x := x+1 [p] x := x-1);
    counter := counter+1;
}
```

6 Conclusion

We have presented a new software tool called PRINSYS for probabilistic invariant generation. Its functionality was explained and its merits were assessed in the discussion. Also implementation details that deviate from the theoretic description of the method in [7] were pointed out. During our evaluation we have reached the next challenge, that is to extend invariant generation to polynomial templates. Related work, e.g. [12] suggests a workaround to find polynomial invariants for non-probabilistic programs. This comes at the price that they sacrifice completeness and limit the class of systems permitted. In the future we would like to work out a similar approximate invariant generation method for probabilistic systems and evaluate it within PRINSYS.

References

1. Barthe, G., Grégoire, B., Béguelin, S.Z.: Probabilistic relational Hoare logics for computer-aided security proofs. In: Int. Conf. on Mathematics of Program Construction (MPC). LNCS, vol. 7342, pp. 1–6 (2012)
2. Barthe, G., Köpf, B., Olmedo, F., Béguelin, S.Z.: Probabilistic relational reasoning for differential privacy. In: Symp. on Principles of Programming Languages (POPL). pp. 97–110. ACM (2012)
3. Gretz, F.: Invariant Generation for Linear Probabilistic Programs. Master's thesis, RWTH Aachen (2010), `http://www-i2.informatik.rwth-aachen.de/i2/gretz/`
4. Gretz, F., Katoen, J.P., McIver, A.: Operational versus Weakest Precondition Semantics for the Probabilistic Guarded Command Language. In: QEST. pp. 168–177 (2012)
5. Hahn, E.M., Hermanns, H., Wachter, B., Zhang, L.: Pass: Abstraction refinement for infinite probabilistic models. In: TACAS. pp. 353–357 (2010)
6. Hahn, E.M., Hermanns, H., Zhang, L.: Probabilistic Reachability for Parametric Markov Models. STTT 13(1), 3–19 (2011)
7. Katoen, J.P., McIver, A., Meinicke, L., Morgan, C.: Linear-Invariant Generation for Probabilistic Programs. In: SAS, pp. 390–406. LNCS (2011)
8. Kiefer, S., Murawski, A., Ouaknine, J., Wachter, B., Worrell, J.: On the Complexity of the Equivalence Problem for Probabilistic Automata. In: FoSSaCS. LNCS, vol. 7213, pp. 467–481 (2012)
9. Kiefer, S., Murawski, A.S., Ouaknine, J., Wachter, B., Worrell, J.: APEX: An Analyzer for Open Probabilistic Programs. In: CAV. LNCS, vol. 7358, pp. 693–698 (2012)

10. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of Probabilistic Real-time Systems. In: CAV. LNCS, vol. 6806, pp. 585–591 (2011)
11. McIver, A., Morgan, C.: Abstraction, Refinement And Proof For Probabilistic Systems (Monographs in Computer Science). SpringerVerlag (2004)
12. Sankaranarayanan, S., Sipma, H., Manna, Z.: Non-linear Loop Invariant Generation Using Gröbner Bases. In: POPL. pp. 318–329 (2004)
13. Ying, M.: Floyd-Hoare logic for quantum programs. ACM Trans. Program. Lang. Syst. 33(6), 19 (2011)