

# Semantics and Loop Invariant Synthesis for Probabilistic Programs

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der  
RWTH Aachen University zur Erlangung des akademischen Grades eines  
Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

**Dipl.-Inform. Friedrich Gretz**  
aus  
Sankt Petersburg, Russland

Berichter: Prof. Dr. Ir. Joost-Pieter Katoen  
Prof. Sriram Sankaranarayanan, PhD

Tag der mündlichen Prüfung: 25.09.2015

Diese Dissertation ist auf den Internetseiten der Universitätsbibliothek  
online verfügbar.

# 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 transformations 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.

# Zusammenfassung

In dieser Arbeit beschäftigen wir uns mit sequentiellen probabilistischen Programmen. Derartige Programme dienen als Modell für randomisierte Algorithmen in der Informatik. Sie erlauben beispielsweise die formale Analyse von Effektivität und Korrektheit von Algorithmen oder auch die Beurteilung von Sicherheitsaspekten in Protokollen.

Wir entwickeln eine operationelle Semantik für probabilistische Programme und zeigen, dass diese äquivalent zur Semantik nach McIver und Morgan ist, die auf Transformationen von Erwartungswerten basiert. Diese Äquivalenzbeziehung zweier Semantiken verschafft uns ein tiefer gehendes Verständnis für das Verhalten von probabilistischen Programmen. Weiterhin können nun Forschungsergebnisse, die auf Transitionssystemen wie zum Beispiel den Markow-Entscheidungsprozessen beruhen, in die Terminologie von deduktiven Verifikationsverfahren, beispielsweise Erwartungswerttransformationen, übersetzt werden – und umgekehrt.

In einem weiteren Schritt fügen wir ein Sprachkonzept für Beobachtungen hinzu. Beide Semantiken können entsprechend erweitert werden und erlauben es uns, bedingte Erwartungswerte zu ermitteln. Das heißt, wir berechnen beispielsweise den erwarteten Wert einer Programmvariable unter der Bedingung, dass alle Beobachtungen während des Programmablaufs eingehalten werden. Unser Hauptaugenmerk richtet sich hierbei auf die auftretenden Schwierigkeiten im Umgang mit nicht terminierenden, nicht deterministischen oder undurchführbaren Programmen. Der Beitrag dieser Arbeit ist es auch solchen Programmen eine wohl definierte Bedeutung zuzuordnen. Zusätzlich diskutieren wir Programmtransformationen, die es erlauben Beobachtungen aus den Programmen zu entfernen.

Im letzten Teil dieser Arbeit gehen wir zur Verifikation von probabilistischen Programmen über. Dabei interessieren wir uns für die Automatisierung von induktiven Überprüfungsverfahren. Wie üblich, stellen dabei Schleifen in Programmen das größte Hindernis dar, da sie die Berechnung von Fixpunkten oder das Auffinden von induktiven Schleifeninvarianten erfordern. Während die Generie-

Die Erzeugung von Schleifeninvarianten bereits für nicht probabilistische Programme eine zentrale Herausforderung darstellt, wird diese Aufgabe in unserer Situation noch schwieriger, da sie eine quantitative Beweisführung verlangt. Als Lösungsansatz konzentrieren wir uns auf ein Verfahren, das ausgehend von benutzerdefinierten Schablonen, quantitative Schleifeninvarianten generiert. Dieses Verfahren wird als Software-Tool PRINSYS implementiert und anhand von einigen Beispielen evaluiert.

# Аннотация

В представленной работе рассматриваются последовательные вероятностные программы. Такие программы служат в информатике моделью рандомизированных алгоритмов. Они позволяют формальный анализ эффективности и корректности алгоритмов или аспектов безопасности протоколов.

Мы разработали операционную семантику для вероятностных программ и показали, что она является эквивалентом семантики по McIver и Morgan, основанной на преобразователях ожиданий. Сочетание этих двух семантик даёт углублённое понимание поведения вероятностных программ. На основании этого сочетания возможны результаты исследований, которые базируются на таких системах, как процесс Маркова, перевести в терминологию дедуктивных методов верификации, как к примеру, преобразование математического ожидания – а также наоборот.

В следующем шаге добавляется в языке программирования понятие так называемых наблюдений. Это приводит к расширению обеих семантик и позволяет нам расчёт условных математических ожиданий. Например мы можем вычислить ожидаемую величину программной переменной, при условии, что все наблюдения удовлетворяются на протяжении выполнения программы. Наш главный вклад является исследованием программ не останавливающихся, не детерминистических или невыполнимых. В этой работе представленные семантики присваивают таким программам четко определенное значение. Кроме того рассматриваются трансформации программного текста позволяющие устранить наблюдение из данной программы.

Последняя часть работы обращает внимание на автоматизацию верификации вероятностных программ. Главным препятствием анализа программ, как обычно, являются циклы, так как они требуют вычисления неподвижных точек или нахождения индуктивных инвариантов. В то время, как поиск инвариантов является центральной проблемой для обычных, не вероятностных программ, эта задача ещё более усложняется тем, что вероят-

ностные программы требуют машинное мышление с количественными величинами. В частности мы рассматриваем метод для нахождения инвариантов исходя из шаблонов заданных пользователем. Этот метод реализован в предлагаемой программе PRINSYS и его практическое применение оценивается на основе нескольких примеров.

# 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” has also been submitted to Macquarie 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 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 on conditional expectations. In particular the result in Theorem 7 is due to Federico and the idea of Section 4.2.3 is due to Benjamin. 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 four year long journey.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
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
<b>2</b>	<b>Preliminaries</b>	<b>15</b>
2.1	Probability theory . . . . .	15
2.2	Markovian models . . . . .	16
2.3	The probabilistic Guarded Command Language . . . . .	22
2.4	Expectation transformers . . . . .	24
2.4.1	Distribution based: forward . . . . .	24
2.4.2	Expectation based: backward . . . . .	25
<b>I</b>	<b>Semantics</b>	<b>33</b>
<b>3</b>	<b>Linking operational and denotational semantics</b>	<b>35</b>
3.1	Operational semantics . . . . .	36
3.1.1	SOS rules . . . . .	37
3.2	Transfer theorem . . . . .	40
3.3	Extension to liberal wp semantics . . . . .	48
3.4	On non-implementable resolution of choices . . . . .	57
<b>4</b>	<b>Conditional probabilities and expectations</b>	<b>59</b>
4.1	Operational semantics for programs with conditioning . . . . .	59
4.2	Expectation transformer semantics for programs with conditioning . . . . .	63

4.2.1	Infeasible programs . . . . .	69
4.2.2	Alternative definition . . . . .	71
4.2.3	Expectation transformers and non-determinism . . . . .	71
4.3	Reasoning with conditioning . . . . .	72
4.3.1	Replacing observations by loops . . . . .	73
4.3.2	Replacing loops by observations . . . . .	77
4.3.3	Observation hoisting . . . . .	79
4.3.4	iid loops and hoisting: a case study . . . . .	87
4.3.5	Conditional expectations in loopy programs: the Crowds protocol . . . . .	91
<b>II</b>	<b>Verification</b>	<b>99</b>
<b>5</b>	<b>Automated analysis</b>	<b>101</b>
5.1	Proving properties of probabilistic programs . . . . .	101
5.1.1	Computing fixed points . . . . .	101
5.1.2	Invariants . . . . .	103
5.2	Feasible level of automation . . . . .	109
5.3	Prinsys . . . . .	110
5.3.1	Methodology . . . . .	111
5.3.2	Examples . . . . .	115
5.3.3	Problems . . . . .	121
<b>6</b>	<b>Conclusion and future work</b>	<b>125</b>

# Chapter 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 [69] 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 [17]. 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 on languages for 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<sup>1</sup> 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 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<sup>2</sup> 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 [50], Metropolis-Hastings [57] and other variants of Markov chain Monte Carlo sampling [39]. 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 [46] which allows the specification of finite state Markov chains or Markov decision processes<sup>3</sup>. 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 they have 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.

---

<sup>1</sup><http://projects.csail.mit.edu/church/wiki/Church>

<sup>2</sup><https://probmods.org/play-space.html>

<sup>3</sup>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 [25], and was introduced by McIver and Morgan [52]. 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 3 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 stop with probability  $p$  or to increase a variable  $x$  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 the negative binomial (Pascal) distribution. Finally, it is also known how to obtain a discrete uniform distribution using repeated fair coin flips [49]. This shows that in fact we have access to a variety of discrete distributions and

```

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

```

(a) Program text

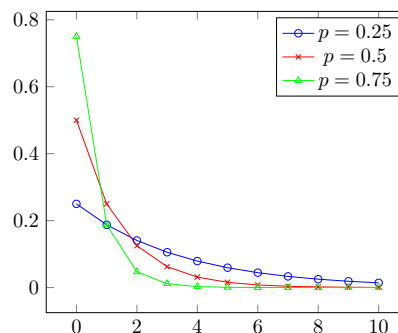
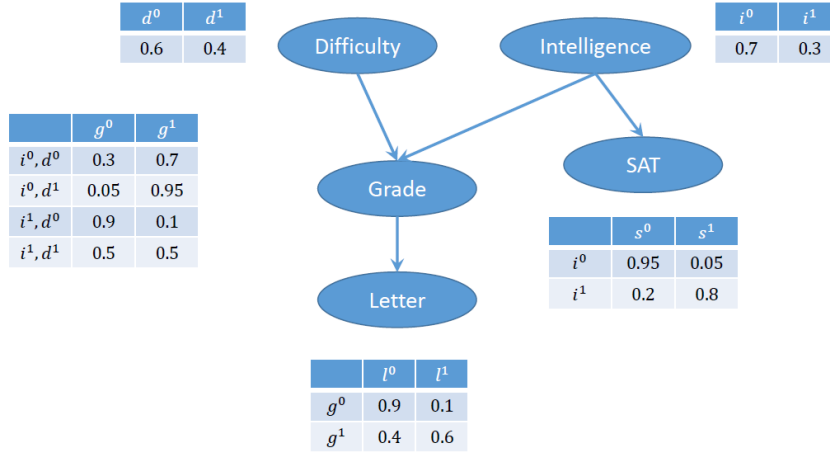
(b) Generated distributions over  $x$  for various values of  $p$ 

Figure 1.1: A probabilistic program implicitly models a distribution.

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.  $A \rightarrow B$  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 [28]. It models the likelihood that a student will receive a recommendation letter based on his performance. Bayesian inference [21] 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 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 4, the conditional probability of any of the variables being 1 can eas-



(a) Bayesian network [28] 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;          11  observe (g = 1);
2  d := 1 [0.4] d := 0;          12  if (i = 0)
3  if (i = 0 and d = 0)          13      s := 1 [0.05] s := 0;
4      g := 1 [0.7] g := 0;      14  else
5  else if (i = 0 and d = 1)     15      s := 1 [0.8] s := 0;
6      g := 1 [0.95] g := 0;     16  if (g = 0)
7  else if (i = 1 and d = 0)     17      l := 1 [0.1] l := 0;
8      g := 1 [0.1] g := 0;      18  else
9  else                            19      l := 1 [0.6] l := 0;
10     g := 1 [0.5] g := 0;

```

(b) Program adapted from [28] 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.

ily 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 4 we will see various other use cases. What the example in Figure 1.2 nicely shows – and what is also emphasised in [28] – 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 4.

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 [67]. The simplest form of a random walk is the unbounded symmetrical walk on a line. Its description in pGCL is shown in Figure 5.2. 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. [66, Ch. 2.4] and [26, 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 [16], which is a randomised 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

```
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

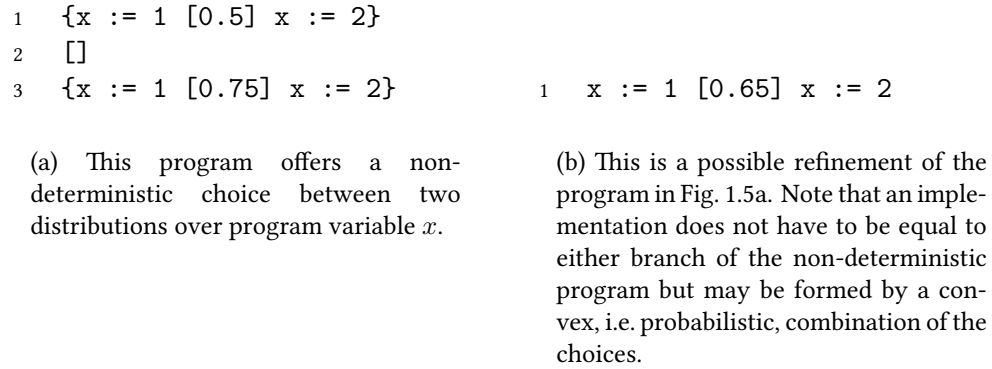


Figure 1.5: A non-deterministic probabilistic program and its refinement.

will occur in the network?” This example nicely shows how the probabilistic behaviour of an actual protocol can be modelled in pGCL. In Chapter 4 we will give 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. [52].

### 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  $\text{pGCL}$  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 [19]. 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 [62], security [6] or randomised algorithms design [55]. 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. [28]. 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  $\text{pGCL}$  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  $\text{pGCL}$  programs are not amenable to model checking. In Fig. 1.1 and Fig. 5.2 we have seen two examples of systems with a countably 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 [23, 44, 38], and progress has been made to tackle parametric probabilistic systems in model checking [42, 37]. As of today, runtime and the size and type 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 [25] gave the meaning of a GCL program  $P$  in terms of a function  $wp(P, \cdot)$  which maps a postcondi-

tion 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 [27] and Dijkstra [25] introduced predicate transformer semantics in the 1970s, it was not until nearly 20 years later that Likkien [48] 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 [52] 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 thoroughly 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 5 in Chapter 4. The transfer theorem is explained in Chapter 3. Our work appeared in a journal article [35] and in conference proceedings [33].

### 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. [18] and add the *observe* statement to the pGCL language. There, and in related work, e.g. [40, 57], 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. Finally two program transformations are presented that allow to remove observations from the programs. The details are outlined in Chapter 4, which is based on our conference paper [31] and technical report [32].

### 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 constitutes a theory that can be mechanised [41, 12, 19]. This allows to use theorem provers like HOL [29] or ISABELLE [56] to write down computer checkable proofs. How-

ever 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  $\text{pGCL}$  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 [52] have generalised the idea of invariants to probabilistic programs and established some proof rules for total correctness of probabilistic programs. Later, Katoen et al. [43] suggested that candidate expressions can be checked for invariance automatically. Subsequently we have revised and implemented their method for invariant generation. Chapter 5 evaluates the tool on several case studies. It is based on our work in [34].



## Chapter 2

# Preliminaries

In this chapter we give an overview of terminology and notations used throughout this thesis. The material covered here is taken from others' work and this chapter serves as a reference only. For a thorough treatment we give pointers to the relevant literature.

### 2.1 Probability theory

We follow the common notions in probability theory and assume a *sample space*  $\Omega$ . The elements of  $\Omega$  are called *samples* and subsets of  $\Omega$  are called *events*. It is required that the collection of events that we may choose from the sample space form a  $\sigma$ -algebra denoted  $\mathcal{A}$ . Further, we assume a function  $\Pr$  which assigns real values in the interval  $[0, 1]$  to events.

**Definition 1** (Probability measure). The mapping  $\Pr$  is called a *probability measure* if it adheres to the following three axioms, known as the Kolmogorov axioms:

1.  $\Pr(A) \geq 0$  for each event  $A \in \mathcal{A}$ .
2.  $\Pr(\Omega) = 1$ .
3.  $\Pr(\bigcup_{i=0}^{\infty} A_i) = \sum_{i=0}^{\infty} \Pr(A_i)$  for every collection of pairwise disjoint events  $A_1, A_2, \dots$

■

Together with the sample space and probability measure it forms the *probability space*  $(\Omega, \mathcal{A}, \Pr)$ . This is the mathematical representation of a random experiment. It is common to introduce an additional layer of abstraction using random variables. This is because the sample space  $\Omega$  can be any set of any objects  $\omega \in \Omega$

which are the outcomes of a random experiment. Random variables are used to map such outcomes to real numbers which is the domain suitable for calculus.

**Definition 2** (Random variable). Let  $X : \Omega \rightarrow \mathbb{R}$  be a mapping of samples to real values.  $X$  is called a *random variable*. ■

A *distribution* is the straightforward lifting of the probability measure of the probability space to a measure over the real numbers.

**Definition 3** (Distribution). A distribution  $\delta$  of a random variable  $X$  is a function  $\delta : \mathbb{R} \rightarrow [0, 1]$  such that

$$\Pr(A) = \int_{X(A)} \delta(x) dx .$$

**Remark 1** (Discrete distributions). Within the scope of this thesis we are concerned with distributions that underlie probabilistic programs. As we will see later, these programs generate a countable number of outcomes which means that  $\Omega$  is countable and the range of the corresponding random variable  $X(\Omega)$  is countable as well. Hence it is possible to define the probability  $\Pr(\omega)$  of individual outcomes and thus  $\delta(x) = \Pr(X^{-1}(x))$ . The above integral becomes a sum and the distribution  $\delta$  is called *discrete*. As a consequence we may safely use the same symbol  $\Pr$  for probability measures and distributions of random variables in this thesis. ■

The central notion of this thesis is that of an *expectation*.

**Definition 4** (Expectation). The expectation  $\mathbb{E}(X)$  of random variable  $X$  is defined as

$$\mathbb{E}(X) = \sum_{x \in \text{supp}(X)} x \cdot \Pr(X = x) ,$$

where the set  $\text{supp}(X)$  is called the *support* of  $X$  and contains all elements in the range of  $X$  that have positive probability. ■

## 2.2 Markovian models

We follow Baier and Katoen [3] in their presentation of Markov chains and decision processes.

**Definition 5** (Discrete-time Markov chain). Let

- $S$  be a countable set of states,
- $\mathbf{P} : S \times S \rightarrow [0, 1]$  be a transition probability matrix,

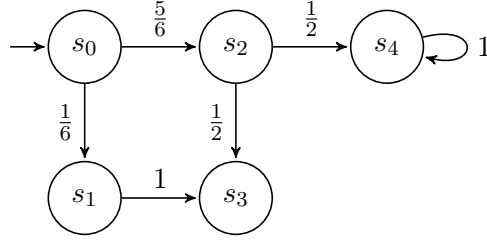


Figure 2.1: An example of a Markov chain.

- $s_0 \in S$  be an initial state,
- $AP$  be a set of atomic propositions and
- $L : S \rightarrow 2^{AP}$  be a labelling function.

The tuple  $\mathcal{M} = (S, \mathbf{P}, s_0, AP, L)$  is called a (discrete-time) Markov chain, MC for short. ■

An MC describes the behaviour of a probabilistic system. It can be thought of as a graph with nodes (states) and edges that have probabilities attached to them. For example, consider the MC depicted in Figure 2.1. The *execution* (or run) of this system starts in the initial state  $s_0$ . From the initial state a step is taken to  $s_1$  with probability  $\mathbf{P}(s_0, s_1) = 1/6$  or to  $s_2$  with probability  $\mathbf{P}(s_0, s_2) = 5/6$ . Let us assume that  $s_2$  is probabilistically chosen to be the successor. Then the MC is said to move to  $s_2$  and from there the run is continued in the same fashion. This process may terminate if a state  $s$  is reached that has no successors, e.g.  $\mathbf{P}(s_3, t) = 0$  for all  $t \in S$ . Every execution can be identified by the sequence of states that are visited.

**Definition 6** (Paths in an MC). A path  $\pi \in S^* \cup S^\omega$  is a *maximal* finite or infinite sequence of states such that for every state  $s$  and its successor  $s'$  on that path  $\mathbf{P}(s, s') > 0$  holds. For a state  $s$  the set  $Paths(s)$  contains all paths which start in  $s$ . The set of all paths in an MC  $\mathcal{M}$  is given by  $Paths(\mathcal{M}) = Paths(s_0)$ . Finally,  $Paths_{fin}(s)$  is the set of all *finite* paths that start in  $s$ . These paths need not be maximal and hence  $Paths_{fin}(s) \not\subseteq Paths(s)$ . ■

Based on paths we are able to define the probability space of a Markov chain. The idea is to use *cylinder sets*. For a given finite path  $\hat{\pi}$ , the cylinder set  $A_{\hat{\pi}}$  contains all maximal paths that share  $\hat{\pi}$  as their prefix:

$$A_{\hat{\pi}} = \{\pi \in Paths(\mathcal{M}) \mid \hat{\pi} \in \text{pref}(\pi)\} .$$

In particular this means that the probability of all these paths together is equal

to the probability of  $\hat{\pi}$ . The set of all cylinder sets forms the  $\sigma$ -algebra in the following definition.

**Definition 7** (Probability space of an MC). Let  $\mathcal{M}$  be an MC. The probability space  $(\Omega, \mathcal{A}, \Pr)$  of  $\mathcal{M}$  is given by

- $\Omega = \text{Paths}(\mathcal{M})$ ,
- $\mathcal{A} = \{A_{\hat{\pi}} \mid \hat{\pi} \in \text{Paths}_{\text{fin}}(\mathcal{M})\}$ , and
- $\Pr(A_{\hat{\pi}}) = \mathbf{P}(s_0, s_1) \cdot \mathbf{P}(s_1, s_2) \cdots \mathbf{P}(s_{n-1}, s_n)$  for cylinder  $A_{\hat{\pi}} \in \mathcal{A}$  where  $\hat{\pi} = s_0 s_1 \dots s_n$ .

■

We are interested in reachability events in MCs. 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 MC  $\mathcal{M}$  that hit some state  $s \in T$ . Formally  $\diamond T = \{\pi \in \text{Paths}(\mathcal{M}) \mid \exists i \geq 0. \pi[i] \in T\}$  where  $\pi[i]$  denotes the  $i$ -th state visited along  $\pi$ . We write  $\pi \models \diamond T$  whenever  $\pi$  belongs to  $\diamond T$ . It follows by standard arguments that  $\diamond T$  is a measurable event. For finite MCs efficient algorithms exist to calculate the probability of a reachability event, cf. [3]. Whenever we would like to denote the set of paths that reach  $T$  from some state  $s$  which is not necessarily the initial state of the MC, we write  $\text{Paths}(s, \diamond T)$ .

So far our system can model probabilistic *behaviour* but we would like to extend it such that it additionally models the *result of a computation* that is associated with this behaviour. For this purpose the states of a Markov chain may be annotated with values, which we call *rewards*.

**Definition 8** (Markov reward model). Let  $\mathcal{M}$  be an MC. The function  $r : S \rightarrow \mathbb{R}_{\geq 0}$  that assigns non-negative numbers to states is called a reward function and the tuple  $(\mathcal{M}, r)$  is called a Markov reward model (MRM). ■

In this model every run of the system accumulates a reward.

**Definition 9** (Cumulative reward). The *cumulative reward* of a finite path  $\hat{\pi} = s_0 s_1 \dots s_n$  in MRM  $(\mathcal{M}, r)$  is defined by

$$r(\hat{\pi}) = r(s_0) + r(s_1) + \dots + r(s_n) .$$

■

Note that in this definition a reward is earned upon *entering* a state. In other literature, e.g. [3] a reward is earned upon *leaving* a state. This difference is merely a technicality as the systems can be converted to suit either definition. For our purposes it is more intuitive to gain rewards upon entering a state as defined above.

**Definition 10** (Cumulative reachability reward). Let  $\pi = s_0 s_1 \dots$  be a maximal path in an MRM  $(\mathcal{M}, r)$  and  $T \subseteq S$  a set of target states. If  $\pi \models \diamond T$ , the cumulative reward along  $\pi$  before reaching  $T$  is defined by

$$r_T(\pi) = r(s_0) + \dots + r(s_k)$$

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  $\pi$  to reach  $T$  is the cumulative reward of the minimal prefix of  $\pi$  satisfying  $\diamond T$ . In case  $\pi$  never reaches a state in  $T$ , the cumulative reward is defined to be zero.

**Remark 2** (Reward for paths that fail to reach an objective). One may 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 will treat this as “undesired” behaviour that has reward zero. This agrees with the previous definition. ■

Finally, we are in the position to define the *expected* reward for reachability properties which measures the average outcome of a computation of a system. It is defined to be the expectation of the function  $r_T(\pi)$ .

**Definition 11** (Expected reward for reachability properties). Let  $(\mathcal{M}, r)$  be an MRM 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 *expected reward* until reaching  $T$  from  $s$ , denoted  $ExpRew^{(\mathcal{M}, r)}(s \models \diamond T)$ , is defined by:

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

■

The summation runs here over all possible cumulative reward values  $c \in \mathfrak{C}$ . On a countable state space there are only countably many different rewards that can be accumulated on the way from a state  $s$  to a target set  $T$ . The summation is therefore well defined. Again, efficient algorithms exist to determine the average outcome produced by an MRM and we refer to [3] for further details.

The models introduced so far are *fully probabilistic*. This means that even though a successor state is not chosen deterministically we at least know the distribution over successor states. However sometimes we do not even have this information. All we know is what are the possible successor states but we cannot say what is their individual probability to get selected. In that case the successor

is said to be selected *non-deterministically*. The following model generalises MCs by adding non-deterministic choices.

**Definition 12** (Markov decision process). Let

- $S$  be a countable set of states,
- $\rightarrow \subseteq S \times \text{Dist}(S)$  be a transition relation from a state to a finite set of distributions over states,
- $s_0 \in S$  be an initial state,
- $AP$  be a set of atomic propositions and
- $L : S \rightarrow 2^{AP}$  be a labelling function.

The tuple  $\mathcal{M} = (S, \rightarrow, s_0, AP, L)$  is called a Markov decision process, MDP for short. ■

We define  $\text{Dist}(s) = \{ \mu \mid s \rightarrow \mu \}$  to be the set of *enabled distributions* in state  $s$ . A *path*  $\pi$  of MDP  $\mathcal{M}$  is a maximal alternating sequence of states and distributions, written  $\pi = s_0 \xrightarrow{\mu_0} s_1 \xrightarrow{\mu_1} \dots$  such that  $\mu_i \in \text{Dist}(s_i)$  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  $\text{Dist}(s) = \emptyset$ . The set of all paths in  $\mathcal{M}$  is denoted  $\text{Paths}(\mathcal{M})$ . Reasoning about probabilities on sets of paths of an MDP relies on the resolution of non-determinism. This resolution is performed by an additional function, which is called a *scheduler*.

**Definition 13** (Scheduler). A *scheduler*<sup>1</sup> is a function  $\mathfrak{S} : S^+ \rightarrow \text{Dist}(S)$  that maps a sequence of states to an enabled distribution, i.e.  $\mathfrak{S}(s_0 s_1 \dots s_n) \in \text{Dist}(s_n)$ . ■

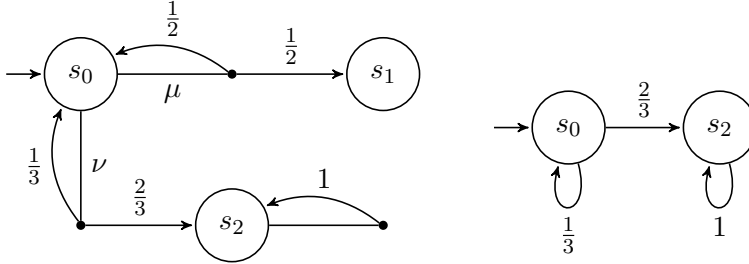
A scheduler keeps track of a sequence of states, which is referred to as the *history* of an execution, and at any decision point selects an enabled distribution depending on that history. Often schedulers are used that base their decision on the current state only.

**Definition 14** (Memoryless scheduler). A scheduler  $\mathfrak{S} : S \rightarrow \text{Dist}(S)$  that selects the next distribution based on the knowledge of the current state only is called a *memoryless scheduler*<sup>2</sup>. ■

We are now in the position to explain how MDPs are executed. For example, consider the MDP  $\mathcal{M}$  in Figure 2.2a. The execution starts in state  $s_0$ . This state offers a choice between two distributions, which we have labelled  $\mu$  and  $\nu$ . In order to resolve this choice, let us further assume we are given a scheduler  $\mathfrak{S}$  such

<sup>1</sup>Also known as policy, adversary, strategy or environment in other literature.

<sup>2</sup>We also refer to such schedulers as *positional*.



(a) An example of a Markov decision process  $\mathcal{M}$ . (b) Unrolling  $\mathcal{M}$  under the memoryless scheduler  $\mathfrak{S}$ , where  $\mathfrak{S}(S^*s_0) = \nu$ .

that  $\mathfrak{S}(s_0) = \nu$ . The system moves to either state  $s_2$  with probability  $\nu(s_2) = 2/3$  or remains in state  $s_0$  with probability  $\nu(s_0) = 1/3$ . The execution continues in the same fashion from that randomly chosen successor state. Note, that once a scheduler  $\mathfrak{S}$  is given, we can unroll the MDP  $\mathcal{M}$  to an MC  $\mathcal{M}^{\mathfrak{S}}$  where all non-deterministic choices have been resolved. For example, let  $\mathfrak{S}$  be a memoryless scheduler which always chooses  $\nu$  in  $s_0$ , then we can unroll  $\mathcal{M}$  to  $\mathcal{M}^{\mathfrak{S}}$  as shown in Figure 2.2b. Furthermore we use the superscript notation  $Paths^{\mathfrak{S}}(s, \diamond T)$  to denote the set of paths starting in  $s$  that eventually reach  $T$  under policy  $\mathfrak{S}$ .

**Remark 3** (On deterministic, probabilistic and non-deterministic behaviour). There is no consensus among different authors how the words *deterministic*, and *non-deterministic* are used when it comes to probabilistic systems. For example, Sampson et al. [63] call programs that exhibit random behaviour non-deterministic because the final *state* of the program is not determined at the beginning of execution. On the other hand, other authors such as McIver et al. [52] call the same programs deterministic because their *distribution* over the outcomes is indeed determined by the initial state.

In order to keep a clear distinction we call systems *deterministic* when their *outcome* is determined by the initial state, i.e. systems without probabilistic or non-deterministic behaviour. We call a system *fully probabilistic* when its *distribution over the outcomes* is determined by the initial state, i.e. there may be probabilistic choices but no non-deterministic choices. Markov chains are an example of such systems. Finally, we impose the least restriction on what we call *non-deterministic* systems. Thus they may have random as well as non-deterministic behaviour. MDPs are an example of such systems. ■

**Remark 4** (Countability of paths). In the context of this work we are only interested in finitely branching MDPs with bounded non-determinism. Therefore  $|Dist(s)| < \infty$  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 Lemma 2 later on. ■

Analogously to Definition 8 we can enrich MDPs with rewards.

**Definition 15** (Reward MDP). Let  $\mathcal{M}$  be an MDP. The function  $r : S \rightarrow \mathbb{R}_{\geq 0}$  that assigns non-negative numbers to states is called a reward function and the tuple  $(\mathcal{M}, r)$  is called a Reward Markov decision process (RMDP). ■

Again we are interested in the expected reward for a reachability property  $\diamond T$ . The definition of cumulative reachability reward for paths in MDPs is analogous to the one in MCs, cf. Definition 10, page 19.

**Definition 16** (Cumulative reachability reward). Let  $\pi = s_0 \xrightarrow{\mu_0} s_1 \xrightarrow{\mu_1} \dots$  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  $\pi$  before reaching  $T$  is defined by:  $r_T(\pi) = r(s_0) + \dots + r(s_k)$  where  $s_i \notin T$  for all  $i < k$  and  $s_k \in T$ . If  $\pi \not\models \diamond T$ , then  $r_T(\pi) = 0$ . ■

We can now define the *expected* reward for reachability.

**Definition 17** (Expected reward for reachability). Let  $(\mathcal{M}, r)$  be an RMDP with state space  $S$ ,  $T \subseteq S$  be a set of target states 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:

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

We omit the superscript  $(\mathcal{M}, r)$  when the underlying model is clear from the context. Further we will omit the subscript in  $r_T$ . ■

### 2.3 The probabilistic Guarded Command Language

Following the imperative programming paradigm, 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 is 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. Moreover we will see in the next chapter

that from any given initial state a pGCL program can reach only countably many states. Thus despite the uncountable range of program variables the reachable state space of a program is countable.

The language, as introduced in [52], has eight commands. We start with the primitive commands. There is the *no-operation* command

$$\textit{skip}$$

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

$$\textit{abort}$$

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:

$$x := E .$$

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:

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

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 *true* = 1 and *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 addition to Dijkstra's GCL [25]. 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:

$$\{P\} \square \{Q\} .$$

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 proper termination  $Q$  is executed. Again for the sake of readability some simplification of notation applies: we often 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 4, we will introduce one more command called *observe*, which however is not relevant for us at this stage.

## 2.4 Expectation transformers

### 2.4.1 Distribution based: forward

Instead of describing the meaning of a program operationally by giving all possible state transitions, we may 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 as done in [24]. In fact, since  $\text{pGCL}$  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 this way 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. This allows to *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. Analogously to expected rewards over RMDPs, we can choose the least of these expectations to make claims about a lower bound on the expected value of our random variable.

#### 2.4.2 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 program variable valuations to (non-negative) real values. We use  $wp(P, f)$  to denote the minimal expectation of random variable  $f$  with respect to  $\text{pGCL}$  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:

$$wp(\text{skip}, f) = f \text{ .}$$

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:

$$wp(\text{abort}, f) = 0 \ .$$

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

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

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:

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

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:

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

Thus  $wp(\{P\} \square \{Q\}, f)$  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:

$$wp(\text{while}(G) \{P\}, f) = \text{lfp}_x([G] \cdot wp(P, x) + [\neg G] \cdot f) . \quad (2.1)$$

Before we conclude the section with a final example, we need to explain the least fixed point semantics of the loop in (2.1) more carefully. To begin with, we need to explain the term “expectations” in the context of probabilistic programs.

In the terminology of McIver and Morgan [52], 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*.

**Definition 18** (Expectations (in probabilistic programming)). A function  $f : S \rightarrow [0, \infty]$  that maps variable valuations<sup>3</sup> to non-negative real values with an adjoined  $\infty$ -element is called an *expectation*. ■

In order to understand the definition in (2.1) above, we need to explain what is the structure in which the least fixed point is required. In the following we define the necessary notions.

**Definition 19** (Directed set). A non-empty set  $D$  with a preorder  $\lesssim$  is directed if for all  $x, y \in D$ , there exists  $z \in D$  such that  $x \lesssim z$  and  $y \lesssim z$ . ■

**Definition 20** (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  $C$ . ■

**Definition 21** (Scott-continuous function). Let  $C, C'$  be cpo's. A function  $F : C \rightarrow C'$  is Scott-continuous if

- for all directed subsets  $D \subseteq C$ , the image  $F(D)$  is directed and
- $F(\sup D) = \sup F(D)$

---

<sup>3</sup>Think of them as program states.

■

A pointwise ordering of expectations gives rise to a complete partial order.

**Proposition 1** (Cpo over expectations). Let  $\mathcal{E} = \{f : S \rightarrow [0, \infty]\}$  be the set of all expectations on  $S$ . Then  $(\mathcal{E}, \leq)$  is a cpo and the constant 0-function is its unique bottom element while the everywhere  $\infty$ -function is the unique top element. ■

Note that the expectation space  $(\mathcal{E}, \leq)$  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. The formalisation of the set of expectations as a cpo gives us a means to show the existence of the least fixed point in (2.1) using a well known result, often called “Kleene’s fixed point theorem”, which appears as Theorem 3 in [47]:

**Theorem 1** (Fixed point theorem). Every continuous function  $F$  over a cpo has a least fixed point which is  $\sup_{n \in \mathbb{N}} \{F^n(\perp)\}$ . ■

Note that “continuous” means Scott-continuous here. This theorem tells us that

$$\text{lfp}_x([G] \cdot wp(P, x) + [\neg G] \cdot f) = \sup_{n \in \mathbb{N}} \{(\lambda x. ([G] \cdot wp(P, x) + [\neg G] \cdot f))^n(0)\} .$$

In order to justify (2.1) using Theorem 1 it remains to show that the expectation transformer  $wp(P, \cdot)$  is indeed a Scott-continuous function. Consequently the behaviour of a loop is characterised by the supremum<sup>4</sup> over the behaviours of bounded loops.

**Lemma 1** (Continuity of expectation transformer  $wp$ ). For every pGCL program  $P$  the expectation transformer  $wp(P, \cdot)$  is a Scott-continuous function over  $(\mathcal{E}, \leq)$ . ■

In [52] the Lemma above appears as Lemma 5.6.6 where it is proven for programs with a finite state space. Subsequently, in Chapter 8 McIver and Morgan show how their results are be extended to infinite state spaces.

**Remark 5** (Expectation terminology). We have defined the term “expectation” twice! In Def. 4 this term denotes the (mathematical) expectation of a given random variable. However in Def. 18 above, an “expectation” is merely a non-negative function. It seems more adequate to call the latter simply a reward function as we did for RMDPs. The terminology was taken from [52]. ■

**Example 1** (Evaluating  $wp$  semantics). Our first example in Figure 1.1 was a program that generated samples according to a geometric distribution. While not

<sup>4</sup>Least upper bound (lub) and supremum (sup) mean the same.

```

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

```

Figure 2.3: Program  $P$  generates a random sample  $x$  according to the geometric distribution with parameter  $p$ .

very spectacular on its own, we will see variations of it reappearing in other programs such as the duelling cowboys, cf. Figure 3.2a, page 39 or the Crowds protocol, cf. Figure 4.7, page 92. We believe that the geometric distribution is at the heart of many other probabilistic programs and hence 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 2.3 once again. We are interested to find the mean value of the random variable  $x$ . According to the denotational semantics presented above (p. 25ff.), this quantity is given by

$$\begin{aligned}
& wp(P, x) \\
&= wp(x := 0, wp(flip := 0, wp(\text{while}(flip = 0)\{flip := 1[p]x := x + 1\}, x))) \tag{2.2}
\end{aligned}$$

$$\begin{aligned}
&= wp(x := 0, wp(flip := 0, \\
&\quad \text{lfp}_f \left( \underbrace{[flip = 0] \cdot wp(flip := 1[p]x := x + 1, f) + [flip \neq 0] \cdot x}_{\Phi(f)} \right)) \tag{2.3}
\end{aligned}$$

$$\begin{aligned}
&= wp(x := 0, wp(flip := 0, \\
&\quad \sup_k \left( \underbrace{[flip = 0] \cdot wp(flip := 1[p]x := x + 1, 0) + [flip \neq 0] \cdot x}_{\Phi(0)} \right)^k )) . \tag{2.4}
\end{aligned}$$

Equation (4.17) 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  $\Phi(f) = [flip = 0] \cdot wp(flip := 1[p]x := x + 1, f) + [flip \neq 0] \cdot x$  takes some expectation  $f$  and returns its pre-expectation with respect to one loop iteration. The solution to the fixed point equation in (4.18) is obtained using the Kleene fixed point theorem. It tells us that the least fixed point of  $\Phi$  can be

found by taking the supremum over  $k$  of  $\Phi^k$  which denotes the  $k$ -fold application of  $\Phi$ . This results in (4.19). For a detailed account of fixed point theorems, we refer to [47]. There, Theorem 3 applies in our setting where the cpo is the set of expectations with point-wise ordering  $(\mathcal{E}, \leq)$ . In order to find the supremum in (4.19) we consider the expression for several  $k$  and deduce a pattern:

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

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

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

$$\begin{aligned}\Phi^4(0) &= \Phi([\textit{flip} = 0] \cdot (px + (1 - p)p(x + 1)) + [\textit{flip} \neq 0] \cdot x) \\ &= \dots \\ &= [\textit{flip} = 0] \cdot (px + (1 - p)p(x + 1) + (1 - p)^2p(x + 2)) + [\textit{flip} \neq 0] \cdot x \\ &\vdots\end{aligned}$$

$$\Phi^{k+2}(0) = [\textit{flip} = 0] \cdot \sum_{i=0}^k (1 - p)^i \cdot p \cdot (x + i) + [\textit{flip} \neq 0] \cdot x$$

⋮

$$\begin{aligned}\sup_k \Phi^k(0) &= [\textit{flip} = 0] \cdot \sum_{i=0}^{\infty} (1 - p)^i \cdot p \cdot (x + i) + [\textit{flip} \neq 0] \cdot x \\ &= [\textit{flip} = 0] \cdot \left(\frac{1 - p}{p} + x\right) + [\textit{flip} \neq 0] \cdot x .\end{aligned}$$

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 (4.19). 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.

$$\begin{aligned}
& \Phi([flip = 0] \cdot (\frac{1-p}{p} + x) + [flip \neq 0] \cdot x) \\
&= [flip = 0] \cdot (px + (1-p)([flip = 0] \cdot (\frac{1-p}{p} + x + 1) + [flip \neq 0] \cdot (x + 1))) \\
&\quad + [flip \neq 0] \cdot x \\
&= [flip = 0] \cdot (px + (1-p)(\frac{1-p}{p} + x + 1)) + [flip \neq 0] \cdot x \\
&= [flip = 0] \cdot (\frac{1-p}{p} + x) + [flip \neq 0] \cdot x \quad . \tag{2.5}
\end{aligned}$$

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 [52, p. 182]. Using this result we can continue the calculation of the expected value of  $x$  in program 2.3 as follows:

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

This agrees with the expectation of the geometric distribution. ■



**Part I**

**Semantics**



## Chapter 3

# Linking operational and denotational semantics

In this chapter we study the semantics of  $\text{pGCL}$  programs. A program in this language is built from commands shown in the left column of Fig. 3.1. For every such command the right column shows a rule which defines how that command transforms a given expectation. Using these rules we can determine the pre-expectation for every given  $\text{pGCL}$  program and post-expectation. In this way the meaning of a program is defined by its expectation transformer. More details were given in Chapter 2.

Another way to explain the meaning of a program is to define its operational behaviour. An operational semantics assumes that a program moves in a stepwise fashion from one state to another. Given a program text and an initial state, the goal of the operational semantics is to explain how the program evolves from that state by executing the program text. For this we need to define rules dictating how from any state the next step is selected and what successor state is reached after taking this step. These rules induce some sort of transition system. Intuitively, for probabilistic programs written in  $\text{pGCL}$  such a transition system will be an instance of an MDP due to the probabilistic and non-deterministic choices present in the language.

However it is not obvious how the rules of Fig. 3.1 relate to some measure of an MDP. Stated differently, in this chapter we would like to understand what is the operational meaning of a  $\text{pGCL}$  program and how pre-expectations that we calculate using rules from Fig. 3.1 relate to the operational semantics.

As a first step we formally define how any program text in  $\text{pGCL}$  may be interpreted as an MDP.

syntax $prog$	semantics $wp(prog, f)$
$skip$	$f$
$abort$	$0$
$x := E$	$f[x/E]$
$if(G) \{P\} else \{Q\}$	$[G] \cdot wp(P, f) + [\neg G] \cdot wp(Q, f)$
$\{P\} [a] \{Q\}$	$a \cdot wp(P, f) + (1 - a) \cdot wp(Q, f)$
$\{P\} [] \{Q\}$	$\min\{wp(P, f), wp(Q, f)\}$
$P; Q$	$wp(P, wp(Q, f))$
$while(G) \{P\}$	$\text{lfp}_x([G] \cdot wp(P, x) + [\neg G] \cdot f)$

Figure 3.1: Syntax and expectation transformer semantics of pGCL. For details see Sections 2.3 and 2.4.

### 3.1 Operational semantics

We describe the translation of program text to (possibly infinite) Markov decision processes using *structured operational semantics* (SOS) rules [58]. Each rule consists of a premise and a conclusion written above and below a solid line. The premise may express some constraint about the current state that has to be fulfilled in order to apply the rule. Some rules do not need any constraints, i.e. the premise is simply *true*, in which case we leave them out. The conclusion describes the effect of a step. In the following we define a rule for every pGCL command. A state of a resulting MDP is a tuple  $\langle P, \eta \rangle$  that captures the remaining program text  $P$  that still needs to be executed and the current valuation of the program variables  $\eta$ .

**Remark 6** (Notation). Formally, a transition of an MDP moves from a state to a distribution over successor states non-deterministically. For the sake of readability we simplify our transitions whenever possible such that they move from a source to a target state directly in the following way: If only one distribution is enabled in state  $s$ , i.e.  $|Dist(s)| = 1$ , we leave out the name of the distribution. If 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 distribution and one successor with probability 1, we simply write  $s \rightarrow t$ . ■

We are now able to formally describe for any state that our program resides in how the next step is taken.

### 3.1.1 SOS rules

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

$$\langle \text{skip}, \eta \rangle \rightarrow \langle \epsilon, \eta \rangle .$$

The  $\epsilon$  denotes that there are no commands left to be executed. An *abort* state behaves like a trap to prevent the execution from reaching any proper terminal states

$$\langle \text{abort}, \eta \rangle \rightarrow \langle \text{abort}, \eta \rangle .$$

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

$$\langle x := E, \eta \rangle \rightarrow \langle \epsilon, \eta[x := \llbracket E \rrbracket_\eta] \rangle .$$

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 if-branch is taken

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

otherwise the else-branch is taken

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

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

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

and in the second

$$\frac{\eta \not\models G}{\langle \text{while}(G) \{P\}, \eta \rangle \rightarrow \langle \epsilon, \eta \rangle} .$$

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:

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

Analogously, for the non-deterministic choice command  $\{P\} \square \{Q\}$  there are two successors as well. This is encoded in the MDP as a non-deterministic choice between two distributions  $\mu$  and  $\nu$ . The distributions however deterministically pick a successor state, i.e.  $\mu(\langle P, \eta \rangle) = 1$  and  $\nu(\langle Q, \eta \rangle) = 1$ . With the notational conventions from Remark 6 this yields the rules:

$$\langle \{P\} \square \{Q\}, \eta \rangle \xrightarrow{\nu} \langle P, \eta \rangle \quad \langle \{P\} \square \{Q\}, \eta \rangle \xrightarrow{\mu} \langle Q, \eta \rangle .$$

In order to define sequential composition we assume that from a state  $\langle P, \eta \rangle$  a step to some distribution  $\mu$  can be taken. What we need to ensure is that from the 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

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

where  $\epsilon; Q = Q$ .

The rule above states that if in a state  $\langle P, \eta \rangle$  a distribution  $\mu$  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  $\langle \epsilon; Q, \eta'' \rangle$ , the MDP proceeds with the execution of  $Q$  starting with valuation  $\eta''$ .

Now we can formally define the operational semantics for a given pGCL program and an initial state.

**Definition 22** (MDP semantics). For a given pGCL program  $P$  and an initial valuation of program variables  $\eta_0$  the operational semantics is the MDP  $\mathcal{M} \llbracket P \rrbracket = (S, \rightarrow, s_0, AP, L)$  where

- $S$  is the set of pairs  $\langle Q, \eta \rangle$  with  $Q$  a pGCL-program or  $Q = \epsilon$ , and  $\eta$  is a variable valuation of the variables occurring in  $P$ ,
- $\rightarrow$  is the smallest relation that is induced by the inference rules in paragraph 3.1.1 above,

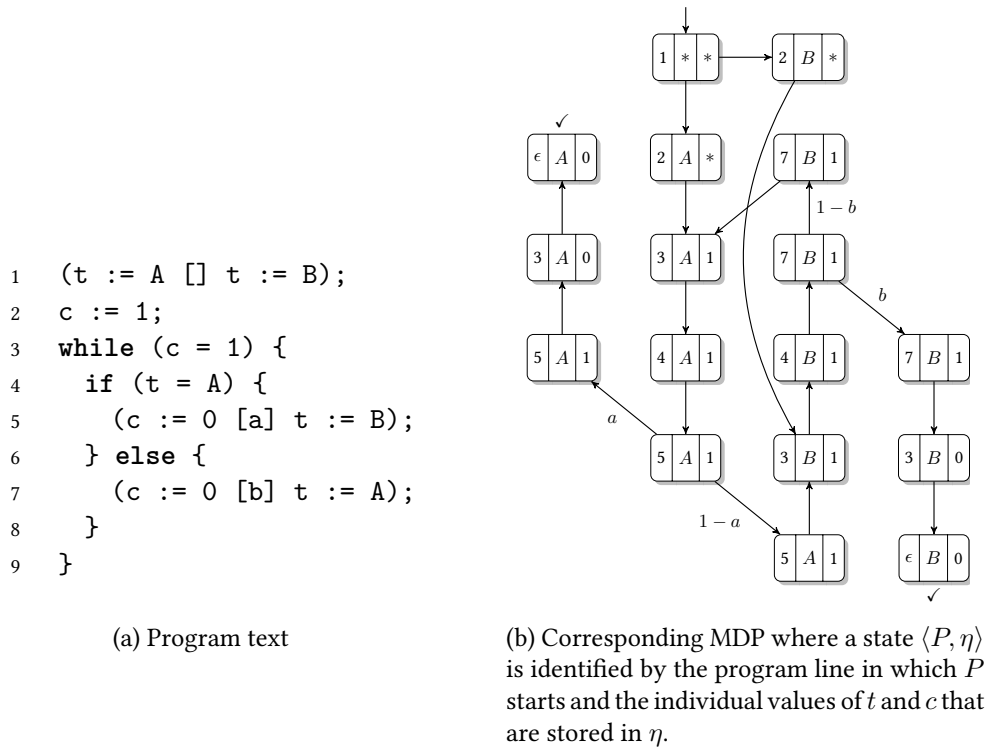


Figure 3.2: Duelling cowboys program

- $s_0$  is the state  $\langle P, \eta_0 \rangle$ ,
- $AP = \{\checkmark\}$ , and
- $L(\langle \epsilon, \eta \rangle) = \{\checkmark\}$ , for any  $\eta$  and  $L(s) = \emptyset$  otherwise.

■

**Example 2** (Operational semantics of programs). We illustrate the program semantics using a simple program which has a finite underlying state space. Figure 3.2a 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 3.2b shows the corresponding MDP semantics. In principle, there are four possible valuations of program variables that the initial state can assume. However since the first two lines of the program reset

all variables regardless of their initial valuation we make a simplification in our presentation and collapse all possible 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  $\langle \epsilon, A, 0 \rangle$  and  $\langle \epsilon, B, 0 \rangle$  depending on the resolution of the choice in the initial state.

We can now, for example, ask for the minimal 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  $\langle \epsilon, A, 0 \rangle$  where it is one. We can then compute the expected rewards for all resolutions of non-determinism. Assume we take the transition  $\langle 1, *, * \rangle \rightarrow \langle 2, A, * \rangle$  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

$$\sum_{i=0}^{\infty} ((1-a)(1-b))^i a = \frac{a}{a+b-ab} . \quad (3.1)$$

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

$$\sum_{i=0}^{\infty} ((1-a)(1-b))^i a(1-b) = \frac{a(1-b)}{a+b-ab} . \quad (3.2)$$

We see that (3.2) is less than (3.1) 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 3.2a is resolved, we can guarantee that cowboy A will win the duel with probability at least  $a(1-b)/a+b-ab$ . ■

## 3.2 Transfer theorem

As we have seen before the MDP semantics describes the possible executions of a program. However in order to reason about some quantity such as the probability of a specific outcome, 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. So the next step on our way to link operational and denotational semantics of pGCL is to explain how post-expectations relate to

reward functions. After that we will see what measure on the RMDP corresponds to the pre-expectation given by the  $w\mathcal{P}$  transformer.

Post-expectations can be interpreted as random variables: they assign real values to outcomes of the program. Thus they are evaluated over the final states of a program. In Example 2 we have used a reward function that assigned non-zero rewards only for particular final states. This reward function acted as a random variable as well, which indicated the state where cowboy  $A$  has won. This insight gives a straightforward embedding of a given post-expectation into an MDP as the following definition suggests.

**Definition 23** (RMDP of a  $\mathcal{P}$ GCL-program). Let  $P$  be a  $\mathcal{P}$ GCL-program and  $f$  a post-expectation for  $P$ . The reward-MDP associated with  $P$  and  $f$  is defined as  $\mathcal{R}_f\llbracket P \rrbracket = (\mathcal{M}\llbracket P \rrbracket, r)$  with  $\mathcal{M}\llbracket P \rrbracket$  the MDP of  $P$  as in Def. 22 and reward function  $r$  defined by  $r(s) = f(\eta)$  if  $L(s) = \{\checkmark\}$  and  $r(s) = 0$  otherwise. ■

In Def. 17 we defined minimal expected rewards for RMDPs is general. However  $\mathcal{R}_f\llbracket P \rrbracket$  has a specific reward structure: only terminal states may have a non-zero reward. We exploit this structure in the following lemma where it is explained how expected rewards can be computed in  $\mathcal{R}_f\llbracket P \rrbracket$ .

**Lemma 2** (Characterising expected rewards). For  $\mathcal{P}$ GCL program  $P$ , expectation  $f$  and a state  $s = \langle P, \eta \rangle$ , we have:

$$\text{ExpRew}^{\mathcal{R}_f\llbracket P \rrbracket}(s \models \diamond\checkmark) = \inf_{\mathfrak{S}} \sum_{\hat{\pi} \in \text{Paths}_{\min}^{\mathfrak{S}}(s, \diamond\checkmark)} \mathbf{P}^{\mathfrak{S}}(\hat{\pi}) \cdot r(\hat{\pi}) ,$$

where  $\text{Paths}_{\min}^{\mathfrak{S}}(s, \diamond\checkmark)$  is the set containing all (finite) paths of the form  $s_0 \dots s_k$  with  $s_0 = s$ ,  $L(s_k) = \{\checkmark\}$  and  $L(s_i) = \emptyset$  for all  $0 \leq i < k$  that adhere to the scheduler  $\mathfrak{S}$ . ■

*Proof.* The proof requires a property stated in Remark 4 namely that in an MDP there are only countably many finite paths that lead from one state to another. Consider the minimal expected reward as defined in Def. 17 where  $T$  is the set of  $\checkmark$ -states:

$$\inf_{\mathfrak{S}} \sum_{c \in \mathcal{C}} c \cdot \Pr^{\mathfrak{S}}\{ \pi \in \text{Paths}^{\mathfrak{S}}(s, \diamond\checkmark) \mid r(\pi) = c \} .$$

Given that  $\Pr^{\mathfrak{S}}(\pi \models \diamond\checkmark) = \mathbf{P}^{\mathfrak{S}}(\hat{\pi})$  where prefix  $\hat{\pi}$  of  $\pi$  is minimal and ends in a  $\checkmark$ -state, the above term equals:

$$\inf_{\mathfrak{S}} \sum_{c \in \mathcal{C}} c \cdot \mathbf{P}^{\mathfrak{S}}\{ \hat{\pi} \in \text{Paths}_{\min}^{\mathfrak{S}}(s, \diamond\checkmark) \mid r(\hat{\pi}) = c \} .$$

As in  $\mathcal{R}_f\llbracket P \rrbracket$  the number of finite path prefixes  $\hat{\pi}$  that reach some  $\checkmark$  state and accumulate a reward  $c$  is countable we can rewrite the sum into:

$$\inf_{\mathfrak{G}} \sum_{\hat{\pi} \in \text{Paths}_{\min}^{\mathfrak{G}}(s, \diamond\checkmark)} \mathbf{P}^{\mathfrak{G}}(\hat{\pi}) \cdot r(\hat{\pi}) .$$

□

Lemma 2 expresses the expected reward in terms of paths that reach a terminal state and their cumulative rewards. This provides a straightforward way to calculate expected rewards (for finite systems). In the next subsection, Lemma 2 will be helpful in the proofs of our main results.

Our terminology already indicated the similarity between the operational and denotational semantics and the following theorem makes this precise.

**Theorem 2** (Transfer theorem). For pGCL-program  $P$ , variable valuation  $\eta$ , and post-expectation  $f$ :

$$wp(P, f)(\eta) = \text{ExpRew}^{\mathcal{R}_f\llbracket P \rrbracket}(\langle P, \eta \rangle \models \diamond\checkmark) .$$

■

*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 Lemma 2.

Induction base:

- For  $P = \text{skip}$  we use the fact that *skip* does not change the post-expectation. We derive:

$$\begin{aligned} & \text{ExpRew}^{\mathcal{R}_f\llbracket \text{skip} \rrbracket}(\langle \text{skip}, \eta \rangle \models \diamond\checkmark) \\ &= \inf_{\mathfrak{G}} \sum_{\hat{\pi} \in \text{Paths}_{\min}^{\mathfrak{G}}(\langle \text{skip}, \eta \rangle, \diamond\checkmark)} \mathbf{P}^{\mathfrak{G}}(\hat{\pi}) \cdot r(\hat{\pi}) \\ &= \inf_{\mathfrak{G}} \mathbf{P}^{\mathfrak{G}}(\langle \text{skip}, \eta \rangle \langle \epsilon, \eta \rangle) \cdot f(\eta) \\ &= 1 \cdot f(\eta) \\ &= f(\eta) \\ &= wp(\text{skip}, f)(\eta). \end{aligned}$$

- For  $P = \text{abort}$  we use the fact that it fails to terminate and has a pre-expectation of zero. We derive:

$$\text{ExpRew}^{\mathcal{R}_f\llbracket \text{abort} \rrbracket}(\langle \text{abort}, \eta \rangle \models \diamond\checkmark)$$

$$\begin{aligned}
&= \inf_{\mathfrak{S}} \sum_{\hat{\pi} \in \text{Paths}_{\min}^{\mathfrak{S}}(\langle abort, \eta \rangle, \diamond \checkmark)} \mathbf{P}^{\mathfrak{S}}(\hat{\pi}) \cdot r(\hat{\pi}) \\
&= 0 \\
&= \text{wp}(abort, f)(\eta)
\end{aligned}$$

as there is no path starting from  $\langle abort, \eta \rangle$  that reaches a  $\checkmark$ -state.

- Let  $P$  be the assignment  $x := E$ . For this case we apply the substitution:

$$\begin{aligned}
&\text{ExpRew}^{\mathcal{R}_f} \llbracket x := E \rrbracket (\langle x := E, \eta \rangle \models \diamond \checkmark) \\
&= \inf_{\mathfrak{S}} \sum_{\hat{\pi} \in \text{Paths}_{\min}^{\mathfrak{S}}(\langle x := E, \eta \rangle, \diamond \checkmark)} \mathbf{P}^{\mathfrak{S}}(\hat{\pi}) \cdot r(\hat{\pi}) \\
&= \inf_{\mathfrak{S}} \mathbf{P}^{\mathfrak{S}}(\langle x := E, \eta \rangle \langle \epsilon, \eta[x/E] \rangle \cdot f(\eta[x/E])) \\
&= 1 \cdot f(\eta[x/E]) \\
&= f(\eta[x/E]) \\
&= f[x/E](\eta) \\
&= \text{wp}(x := E, f)(\eta).
\end{aligned}$$

Induction hypothesis: assume that for program  $P$  (and analogously for  $Q$ )

$$\text{wp}(P, f)(\eta) = \text{ExpRew}^{\mathcal{R}_f} \llbracket P \rrbracket (\langle P, \eta \rangle \models \diamond \checkmark) .$$

Induction step:

- Consider the probabilistic choice  $\{P\} [a] \{Q\}$  (this also covers conditional choice since it can be written as  $\{P\} [[G]] \{Q\}$ )<sup>1</sup>. The idea is to represent the expected reward as a weighted sum of the expected rewards computed from successor states. This corresponds to the weighted sum for the weakest pre-expectation:

$$\begin{aligned}
&\text{ExpRew}^{\mathcal{R}_f} \llbracket \{P\} [a] \{Q\} \rrbracket (\langle \{P\} [a] \{Q\}, \eta \rangle \models \diamond \checkmark) \\
&= \inf_{\mathfrak{S}} \sum_{\hat{\pi} \in \text{Paths}_{\min}^{\mathfrak{S}}(\langle \{P\} [a] \{Q\}, \eta \rangle, \diamond \checkmark)} \mathbf{P}^{\mathfrak{S}}(\hat{\pi}) \cdot r(\hat{\pi}) \\
&= \inf_{\mathfrak{S}} \sum_{\hat{\pi} \in \text{Paths}_{\min}^{\mathfrak{S}}(\langle P, \eta \rangle, \diamond \checkmark)} a \cdot \mathbf{P}^{\mathfrak{S}}(\hat{\pi}) \cdot r(\hat{\pi})
\end{aligned}$$

<sup>1</sup>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.

$$\begin{aligned}
& + \sum_{\hat{\pi} \in \text{Paths}_{\min}^{\mathfrak{S}}(\langle Q, \eta \rangle, \diamond\checkmark)} (1 - a) \cdot \mathbf{P}^{\mathfrak{S}}(\hat{\pi}) \cdot r(\hat{\pi}) \\
\stackrel{*}{=} & a \cdot \inf_{\mathfrak{S}_1} \sum_{\hat{\pi} \in \text{Paths}_{\min}^{\mathfrak{S}_1}(\langle P, \eta \rangle, \diamond\checkmark)} \mathbf{P}^{\mathfrak{S}}(\hat{\pi}) \cdot r(\hat{\pi}) \\
& + (1 - a) \cdot \inf_{\mathfrak{S}_2} \sum_{\hat{\pi} \in \text{Paths}_{\min}^{\mathfrak{S}_2}(\langle Q, \eta \rangle, \diamond\checkmark)} \mathbf{P}^{\mathfrak{S}}(\hat{\pi}) \cdot r(\hat{\pi}) \\
= & a \cdot \text{ExpRew}^{\mathcal{R}_f}[P](\langle P, \eta \rangle \models \diamond\checkmark) \\
& + (1 - a) \cdot \text{ExpRew}^{\mathcal{R}_f}[Q](\langle Q, \eta \rangle \models \diamond\checkmark) \\
\stackrel{I.H.}{=} & a \cdot \text{wp}(P, f)(\eta) + (1 - a) \cdot \text{wp}(Q, f)(\eta) \\
= & \text{wp}(\{P\} [a] \{Q\}, f)(\eta)
\end{aligned}$$

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  $\{P\} \square \{Q\}$  which is analogous to probabilistic choice, except that min replaces the weighted sum:

$$\begin{aligned}
& \text{ExpRew}^{\mathcal{R}_f}[\{P\} \square \{Q\}](\langle \{P\} \square \{Q\}, \eta \rangle \models \diamond\checkmark) \\
= & \inf_{\mathfrak{S}} \sum_{\hat{\pi} \in \text{Paths}_{\min}^{\mathfrak{S}}(\langle \{P\} \square \{Q\}, \eta \rangle, \diamond\checkmark)} \mathbf{P}^{\mathfrak{S}}(\hat{\pi}) \cdot r(\hat{\pi}) \\
= & \min \left\{ \inf_{\mathfrak{S}_1} \sum_{\hat{\pi} \in \text{Paths}_{\min}^{\mathfrak{S}_1}(\langle P, \eta \rangle, \diamond\checkmark)} \mathbf{P}^{\mathfrak{S}_1}(\hat{\pi}) \cdot r(\hat{\pi}), \right. \\
& \left. \inf_{\mathfrak{S}_2} \sum_{\hat{\pi} \in \text{Paths}_{\min}^{\mathfrak{S}_2}(\langle Q, \eta \rangle, \diamond\checkmark)} \mathbf{P}^{\mathfrak{S}_2}(\hat{\pi}) \cdot r(\hat{\pi}) \right\} \\
= & \min \{ \text{ExpRew}^{\mathcal{R}_f}[P](\langle P, \eta \rangle \models \diamond\checkmark), \\
& \text{ExpRew}^{\mathcal{R}_f}[Q](\langle Q, \eta \rangle \models \diamond\checkmark) \} \\
\stackrel{I.H.}{=} & \min \{ \text{wp}(P, f)(\eta), \text{wp}(Q, f)(\eta) \} \\
= & \text{wp}(\{P\} \square \{Q\}, f)(\eta)
\end{aligned}$$

- Consider the sequential composition  $P; Q$ . The idea is to 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:

$$\begin{aligned}
& \text{ExpRew}^{\mathcal{R}_f \llbracket P; Q \rrbracket}(\langle P; Q, \eta \rangle \models \diamond \checkmark) \\
&= \inf_{\mathfrak{S}} \sum_{\hat{\pi} \in \text{Paths}_{\min}^{\mathfrak{S}}(\langle P; Q, \eta \rangle, \diamond \checkmark)} \mathbf{P}^{\mathfrak{S}}(\hat{\pi}) \cdot r(\hat{\pi}) \\
&\stackrel{*}{=} \inf_{\mathfrak{S}} \sum_{\hat{\pi} \in \text{Paths}_{\min}^{\mathfrak{S}}(\langle P, \eta \rangle, \diamond \checkmark)} \mathbf{P}^{\mathfrak{S}}(\hat{\pi}) \cdot r^Q(\hat{\pi}) \\
&\quad \text{where } r^Q(\hat{\pi}) \text{ is the sum of rewards along } \hat{\pi} \text{ with} \\
&\quad r^Q(s) = \inf_{\mathfrak{S}'} \left( \sum_{\hat{\pi}' \in \text{Paths}_{\min}^{\mathfrak{S}'}(\langle Q, \eta' \rangle, \diamond \checkmark)} \mathbf{P}^{\mathfrak{S}'}(\hat{\pi}') \cdot r(\hat{\pi}') \right) \\
&\quad \text{if } s = \langle \epsilon, \eta' \rangle \text{ for some } \eta' \text{ and } r^Q(s) = 0 \text{ otherwise} \\
&= \text{ExpRew}^{\mathcal{R}_g \llbracket P \rrbracket}(\langle P, \eta \rangle \models \diamond \checkmark) \\
&\quad \text{where } g(\eta) = \text{ExpRew}^{\mathcal{R}_f \llbracket Q \rrbracket}(\langle Q, \eta \rangle \models \diamond \checkmark) \\
&\stackrel{I.H.}{=} \text{wp}(P, \text{wp}(Q, f))(\eta) \\
&= \text{wp}(P; Q, f)(\eta) \ .
\end{aligned}$$

In  $*$  we divide each path into the aforementioned pre- and suffixes. The minimal expected reward is calculated over the prefixes with respect to the reward function  $r^Q$ . We define  $r^Q$  to be the minimal expected reward over the suffixes with respect to the original reward function  $r$ . In this argument we use the positionality of schedulers, i.e. the choices made by  $\mathfrak{S}'$  on the suffix is independent of the choices made by  $\mathfrak{S}$  before.

- Consider the loop  $\text{while}(G) \{P\}$ . 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  $k \geq 0$  be

$$\text{while}^{k+1}(G) \{P\} = \text{if}(G) \{P; \text{while}^k(G) \{P\}\} \text{ else } \{ \text{skip} \}$$

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

$$\text{wp}(\text{while}^k(G) \{P\}, f)(\eta) = \text{ExpRew}^{\mathcal{R}_f \llbracket \text{while}^k(G) \{P\} \rrbracket}(\eta) \ . \quad (3.3)$$

The above equality ensures that the two sequences agree. To prove our claim we additionally need to show that

$$\sup_k wp(\text{while}^k(G) \{P\}, f)(\eta) = wp(\text{while}(G) \{P\}, f)(\eta) \quad (3.4)$$

$$\sup_k \text{ExpRew}^{\mathcal{R}_f} \llbracket \text{while}^k(G) \{P\} \rrbracket (\eta) = \text{ExpRew}^{\mathcal{R}_f} \llbracket \text{while}(G) \{P\} \rrbracket (\eta) \quad (3.5)$$

**Lemma 3.** Assume a pGCL program  $P$ , a boolean expression  $G$  and an expectation  $f$ . Let  $\Phi(x) = [G] \cdot wp(P, x) + [\neg G] \cdot f$  be an expectation transformer. Then for all  $n \in \mathbb{N}$

$$\Phi^n(0) = wp(\text{while}^n(G) \{P\}, f) .$$

■

*Proof.* The proof proceeds by induction on  $n$ .

Base case  $n = 0$ :

$$\Phi^0(0) = 0 = wp(\text{abort}, f) = wp(\text{while}^0(G) \{P\}, f) .$$

Induction hypothesis:

Assume that the claim holds for a fixed but arbitrary  $n$ .

Induction step  $n \mapsto n + 1$ :

$$\begin{aligned} & \Phi^{n+1}(0) \\ &= \Phi(\Phi^n(0)) \\ &= \Phi(wp(\text{while}^n(G) \{P\}, f)) \\ &= [G] \cdot wp(P, wp(\text{while}^n(G) \{P\}, f)) + [\neg G] \cdot f \\ &= [G] \cdot wp(P; \text{while}^n(G) \{P\}, f) + [\neg G] \cdot f \\ &= wp(\text{if}(G) \{P; \text{while}^n(G) \{P\}\} \text{ else } \{\text{skip}\}, f) \\ &= wp(\text{while}^{n+1}(G) \{P\}, f) . \end{aligned}$$

□

With the insight from Lemma 3, the equation (3.4) becomes straightforward:

$$\begin{aligned}
& \sup_k wp(\text{while}^k(P) \{G\}, f) \\
&= \sup_k \Phi^k(0) \\
&\stackrel{*}{=} \text{lfp}_x \underbrace{([G] \cdot wp(P, x) + [\neg G] \cdot f)}_{\Phi(x)} \\
&= wp(\text{while}(G) \{P\}, f)
\end{aligned}$$

In (\*) we apply the fixpoint theorem 3 from [47], see page 28.

The proof for equation (3.5) can be carried out in a similar way. For this we can characterise the minimal expected reward as a fixed point of a function. After showing that this function is Scott-continuous we can formulate a lemma analogous to Lemma 3 and subsequently prove the result. A more intuitive way to see that equation (3.5) holds is the following. Observe that  $\mathcal{R}_f \llbracket \text{while}^k(P) \{G\} \rrbracket$  is a subgraph of  $\mathcal{R}_f \llbracket \text{while}(P) \{G\} \rrbracket$ . Every path that terminates with a positive reward in the  $k$ -bounded RMDP is also present and has the same probability in the unbounded RMDP. The reverse is however not true, since there may be paths in the unbounded RMDP that correspond to more than  $k$  iterations of the loop and these would hit an *abort* state in the  $k$ -bounded RMDP. As we take the supremum over  $k$ , every terminating run of  $\mathcal{R}_f \llbracket \text{while}(P) \{G\} \rrbracket$  is found with the same probability in  $\sup_k \mathcal{R}_f \llbracket \text{while}^k(P) \{G\} \rrbracket$  and their expected rewards therefore coincide.

It remains to prove (3.3). This is done by induction on  $k$ .

Base case ( $k = 0$ ):

$$\begin{aligned}
& wp(\text{while}^0(G) \{P\}, f)(\eta) \\
&= wp(\text{abort}, f)(\eta) \\
&\stackrel{*}{=} \text{ExpRew}^{\mathcal{R}_f \llbracket \text{abort} \rrbracket}(\eta) \\
&= \text{ExpRew}^{\mathcal{R}_f \llbracket \text{while}^0(G) \{P\} \rrbracket}(\eta)
\end{aligned}$$

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

Induction hypothesis: equation (3.3) holds for some unspecified but fixed value of  $k$ .

Induction step:

$$\begin{aligned}
& wp(\text{while}^{k+1}(G) \{P\}, f)(\eta) \\
&= wp(\text{if}(G) \{P; \text{while}^k(G) \{P\}\} \text{else } \{\text{skip}\}, f)(\eta) \\
&= \left( [G] \cdot wp(P; \text{while}^k(G) \{P\}, f) + [\neg G] \cdot wp(\text{skip}, f) \right) (\eta) \\
&\stackrel{*}{=} \left( [G] \cdot \text{ExpRew}^{\mathcal{R}_f} \llbracket P; \text{while}^k(G) \{P\} \rrbracket + [\neg G] \cdot \text{ExpRew}^{\mathcal{R}_f} \llbracket \text{skip} \rrbracket \right) (\eta) \\
&= \text{ExpRew}^{\mathcal{R}_f} \llbracket \text{if}(G) \{P; \text{while}^k(G) \{P\}\} \text{else } \{\text{skip}\} \rrbracket (\eta) \\
&= \text{ExpRew}^{\mathcal{R}_f} \llbracket \text{while}^{k+1}(G) \{P\} \rrbracket (\eta)
\end{aligned}$$

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

With (3.3), (3.4) and (3.5) we finally have our claim:

$$\begin{aligned}
& wp(\text{while}(G) \{P\}, f)(\eta) \\
&= \sup_k wp(\text{while}^k(P) \{G\}, f) \\
&= \sup_k \text{ExpRew}^{\mathcal{R}_f} \llbracket \text{while}^k(G) \{P\} \rrbracket (\eta) \\
&= \text{ExpRew}^{\mathcal{R}_f} \llbracket \text{while}(G) \{P\} \rrbracket (\eta).
\end{aligned}$$

□

Theorem 2 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. Pre-expectations now have a precise meaning in terms of a measure on the RMDP that is underlying a given pGCL program. Furthermore this correspondence theorem allows to carry over results that are known in the probabilistic programming 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 2 that we can carry out the proof using either semantics and *transfer* it then onto the other. This motivates the theorem's name.

### 3.3 Extension to liberal wp semantics

So far we have studied reachability properties of probabilistic programs. Our measure was the expectation of the subdistribution over outcomes. We speak

of a subdistribution because not all executions terminate properly. Due to non-terminating runs some probability is “lost”. Consider the following program

$$P = \{x := 10\} [0.5] \{abort\} . \quad (3.6)$$

This program generates a subdistribution over the range of  $x$  where the outcome  $x = 10$  has probability  $wp(P, [x = 10]) = 0.5$ . The probability for all other values of  $x$  is 0 and half of the probability mass is “lost” due to the aborting run. However we might want to give a different value to non-terminating runs. Assume we want to know the probability of the safety property “the program  $P$  produces no other value than 10”. Program  $P$  satisfies this property with probability 1 since it either produces the value 10 or it does not terminate in which case we cannot say that any value is produced and the property is not violated as well. A similar concept is known from standard (i.e. non-probabilistic) program semantics. Dijkstra [25] introduced the *weakest liberal precondition (wlp)* which characterises the set of initial states of a program from which the program either establishes the post-condition *or does not terminate*. This notion is useful when we want to reason about partial correctness of the program. In that scenario we want to verify that the program *avoids* bad states or dually that it stays within a set of good states (possibly without ever terminating) which are characterised by a predicate usually called an *invariant*. In this section we consider the probabilistic extension of Dijkstra’s *wlp* which is the *minimal liberal pre-expectation* (abbreviated as *wlp* as well) due to McIver and Morgan [52]. Given a random variable  $f$ ,  $wlp(P, f)$  should be the expectation over all executions that terminate with some value of  $f$  *plus* the expectation over all non-terminating runs. But what should be the value that a non-terminating run contributes to that expectation? In particular,  $wlp(abort, f)$  should yield the maximal value for any  $f$ . One possibility would be to choose  $\infty$  as the top element but that would render all calculations useless. For the program  $P$  in (3.6), we would calculate

$$wlp(P, [x = 10]) = 0.5 \cdot [10 = 10] + 0.5 \cdot \infty = \infty$$

regardless of the chosen post-expectation. To avoid this problem we instead restrict the range of expectations that *wlp* transforms to the interval  $[0, 1]$ . Now we have a real-valued upper bound and may use *wlp* to characterise the probability that the program never produces any other outcome but 10 as

$$wlp(P, [x = 10]) = 0.5 \cdot [10 = 10] + 0.5 \cdot 1 = 1 .$$

Figure 3.3 shows the *wlp* semantics of pGCL. The only difference to the *wp* semantics studied before (cf. Fig. 3.1) are the pre-expectations for the *abort* statement and the *while* loop.

syntax <i>prog</i>	semantics $wlp(prog, f)$
<i>skip</i>	$f$
<i>abort</i>	1
$x := E$	$f[x/E]$
$if(G) \{P\} else \{Q\}$	$[G] \cdot wlp(P, f) + [\neg G] \cdot wlp(Q, f)$
$\{P\} [a] \{Q\}$	$a \cdot wlp(P, f) + (1 - a) \cdot wlp(Q, f)$
$\{P\} [] \{Q\}$	$\min\{wlp(P, f), wlp(Q, f)\}$
$P; Q$	$wlp(P, wlp(Q, f))$
$while(G) \{P\}$	$\text{gfp}_x([G] \cdot wlp(P, x) + [\neg G] \cdot f)$

Figure 3.3: Liberal expectation transformer semantics of pGCL.

**Proposition 2** (Cpo over expectations for  $wlp$  semantics). Let  $\mathcal{E}_{\leq 1} = \{f : S \rightarrow [0, 1]\}$  be the set of 1-bounded expectations. These form a cpo  $(\mathcal{E}_{\leq 1}, \geq)$  with the constant 1-expectation as its bottom and the constant 0-expectation as its top element. For any program  $P$ , the minimal liberal pre-expectation

$$wlp(P, \cdot) : \mathcal{E}_{\leq 1} \rightarrow \mathcal{E}_{\leq 1}$$

acts on the set of 1-bounded expectations. ■

In Chapter 4 we will use  $wlp$  to quantify the probability of avoiding undesired states. Later, in Chapter 5 the transformer  $wlp$  will serve as the basis for the definition of a probabilistic loop invariant. In what follows we define a measure  $LExpRew$  on RMDPs along the lines of Definition 17, introduce a lemma that allows to characterise this measure analogously to Lemma 2 and finally show the correspondence between  $LExpRew$  and  $wlp$  as has been previously done for the non-liberal transformer  $wp$  in Theorem 2.

In analogy to Definition 17 we define a *liberal* expected reward which takes into account non-terminating runs of a program.

**Definition 24** (Liberal 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 *liberal* expected reward until reaching some state in

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

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

■

The above definition sums over possible reward values. We can rewrite the sum to run over finite paths as suggested in the following lemma.

**Lemma 4** (Characterising liberal expected rewards). For pGCL program  $P$  and variable valuation  $\eta$ , we have:

$$\begin{aligned} & LExpRew^{\mathcal{R}_f \llbracket P \rrbracket}(\langle P, \eta \rangle \models \diamond \checkmark) \\ &= \inf_{\mathfrak{S}} \left\{ \Pr^{\mathfrak{S}}(\langle P, \eta \rangle \not\models \diamond \checkmark) + \sum_{\hat{\pi} \in Paths_{\min}^{\mathfrak{S}}(s, \diamond \checkmark)} \mathbf{P}^{\mathfrak{S}}(\hat{\pi}) \cdot r_{P \checkmark}(\hat{\pi}) \right\} . \end{aligned}$$

■

*Proof.* Follows immediately from Def. 17 and Lemma 2. □

Finally, we are in the position to state the main result of this section.

**Theorem 3** (Transfer theorem for liberal semantics). For pGCL-program  $P$ , variable valuation  $\eta$ , and post-expectation  $f$ :

$$wlp(P, f)(\eta) = LExpRew^{\mathcal{R}_f \llbracket P \rrbracket}(\langle P, \eta \rangle \models \diamond \checkmark) .$$

■

*Proof.* By structural induction over the pGCL program  $P$  (analogously to the proof of Theorem 2). Similarly we apply Lemma 4 here. To avoid repetition we skip the base cases which are rather simple.

Induction hypothesis: assume that for program  $P$  (and analogously for  $Q$ )

$$wlp(P, f)(\eta) = LExpRew^{\mathcal{R}_f \llbracket P \rrbracket}(\langle P, \eta \rangle \models \diamond \checkmark) .$$

Induction step:

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

$$LExpRew^{\mathcal{R}_f \llbracket \{P\} [a] \{Q\} \rrbracket}(\langle \{P\} [a] \{Q\}, \eta \rangle \models \diamond \checkmark)$$

$$\begin{aligned}
&= \inf_{\mathfrak{G}} \left( \Pr^{\mathfrak{G}}(\langle \{P\} [a] \{Q\}, \eta \rangle \not\models \diamond\checkmark) \right. \\
&\quad \left. + \sum_{\hat{\pi} \in \text{Paths}_{\min}^{\mathfrak{G}}(\langle \{P\} [a] \{Q\}, \eta \rangle, \diamond\checkmark)} \mathbf{P}^{\mathfrak{G}}(\hat{\pi}) \cdot r(\hat{\pi}) \right) \\
&= \inf_{\mathfrak{G}} \left( a \cdot \Pr^{\mathfrak{G}}(\langle P, \eta \rangle \not\models \diamond\checkmark) \right. \\
&\quad + \sum_{\hat{\pi} \in \text{Paths}_{\min}^{\mathfrak{G}}(\langle P, \eta \rangle, \diamond\checkmark)} a \cdot \mathbf{P}^{\mathfrak{G}}(\hat{\pi}) \cdot r(\hat{\pi}) \\
&\quad + (1 - a) \cdot \Pr^{\mathfrak{G}}(\langle Q, \eta \rangle \not\models \diamond\checkmark) \\
&\quad \left. + \sum_{\hat{\pi} \in \text{Paths}_{\min}^{\mathfrak{G}}(\langle Q, \eta \rangle, \diamond\checkmark)} (1 - a) \cdot \mathbf{P}^{\mathfrak{G}}(\hat{\pi}) \cdot r(\hat{\pi}) \right) \\
&= a \cdot \inf_{\mathfrak{G}_1} \left( \Pr^{\mathfrak{G}_1}(\langle P, \eta \rangle \not\models \diamond\checkmark) \right. \\
&\quad \left. + \sum_{\hat{\pi} \in \text{Paths}_{\min}^{\mathfrak{G}_1}(\langle P, \eta \rangle, \diamond\checkmark)} \mathbf{P}^{\mathfrak{G}_1}(\hat{\pi}) \cdot r(\hat{\pi}) \right) \\
&\quad + (1 - a) \cdot \inf_{\mathfrak{G}_2} \left( \Pr^{\mathfrak{G}_2}(\langle Q, \eta \rangle \not\models \diamond\checkmark) \right. \\
&\quad \left. + \sum_{\hat{\pi} \in \text{Paths}_{\min}^{\mathfrak{G}_2}(\langle Q, \eta \rangle, \diamond\checkmark)} \mathbf{P}^{\mathfrak{G}_2}(\hat{\pi}) \cdot r(\hat{\pi}) \right) \\
&= a \cdot \text{LExpRew}^{\mathcal{R}_f \llbracket P \rrbracket}(\langle P, \eta \rangle \models \diamond\checkmark) \\
&\quad + (1 - a) \cdot \text{LExpRew}^{\mathcal{R}_f \llbracket Q \rrbracket}(\langle Q, \eta \rangle \models \diamond\checkmark) \\
&\stackrel{I.H.}{=} a \cdot \text{wlp}(P, f)(\eta) + (1 - a) \cdot \text{wlp}(Q, f)(\eta) \\
&= \text{wlp}(\{P\} [a] \{Q\}, f)(\eta) .
\end{aligned}$$

- Consider the non-deterministic choice  $\{P\} \square \{Q\}$ :

$$\text{LExpRew}^{\mathcal{R}_f \llbracket P \square Q \rrbracket}(\langle P \square Q, \eta \rangle \models \diamond\checkmark)$$

$$\begin{aligned}
&= \inf_{\mathfrak{G}} \left( \Pr^{\mathfrak{G}}(\langle \{P\} \parallel \{Q\}, \eta \rangle \not\models \diamond\checkmark) \right. \\
&\quad \left. + \sum_{\hat{\pi} \in \text{Paths}_{\min}^{\mathfrak{G}}(\langle \{P\} \parallel \{Q\}, \eta \rangle, \diamond\checkmark)} \mathbf{P}^{\mathfrak{G}}(\hat{\pi}) \cdot r(\hat{\pi}) \right) \\
&= \min \left\{ \inf_{\mathfrak{G}_1} \left( \Pr^{\mathfrak{G}_1}(\langle P, \eta \rangle \not\models \diamond\checkmark) + \sum_{\hat{\pi} \in \text{Paths}_{\min}^{\mathfrak{G}_1}(\langle P, \eta \rangle, \diamond\checkmark)} \mathbf{P}^{\mathfrak{G}_1}(\hat{\pi}) \cdot r(\hat{\pi}) \right), \right. \\
&\quad \left. \inf_{\mathfrak{G}_2} \left( \Pr^{\mathfrak{G}_2}(\langle Q, \eta \rangle \not\models \diamond\checkmark) + \sum_{\hat{\pi} \in \text{Paths}_{\min}^{\mathfrak{G}_2}(\langle Q, \eta \rangle, \diamond\checkmark)} \mathbf{P}^{\mathfrak{G}_2}(\hat{\pi}) \cdot r(\hat{\pi}) \right) \right\} \\
&= \min\{LExpRew^{\mathcal{R}_f[P]}(\langle P, \eta \rangle \models \diamond\checkmark), LExpRew^{\mathcal{R}_f[Q]}(\langle Q, \eta \rangle \models \diamond\checkmark)\} \\
&\stackrel{I.H.}{=} \min\{wlp(P, f)(\eta), wlp(Q, f)(\eta)\} \\
&= wlp(\{P\} \parallel \{Q\}, f)(\eta) .
\end{aligned}$$

- Consider the sequential composition  $P; Q$ :

$$\begin{aligned}
&LExpRew^{\mathcal{R}_f[P;Q]}(\langle P; Q, \eta \rangle \models \diamond\checkmark) \\
&= \inf_{\mathfrak{G}} \left( \Pr^{\mathfrak{G}}\{\langle P; Q, \eta \rangle \not\models \diamond\checkmark\} \right. \\
&\quad \left. + \sum_{\hat{\pi} \in \text{Paths}_{\min}^{\mathfrak{G}}(\langle P; Q, \eta \rangle, \diamond\checkmark)} \mathbf{P}^{\mathfrak{G}}(\hat{\pi}) \cdot r(\hat{\pi}) \right) \\
&\stackrel{*}{=} \inf_{\mathfrak{G}} \left( \Pr^{\mathfrak{G}}\{\langle P, \eta \rangle \not\models \diamond\checkmark\} \right. \\
&\quad \left. + \sum_{\hat{\pi} \in \text{Paths}_{\min}^{\mathfrak{G}}(\langle P, \eta \rangle, \diamond\checkmark)} \mathbf{P}^{\mathfrak{G}}(\hat{\pi}) \cdot r^Q(\hat{\pi}) \right)
\end{aligned}$$

where  $r^Q(\hat{\pi})$  is the sum of rewards along  $\hat{\pi}$  with

$$r^Q(s) = \inf_{\mathfrak{G}'} \left( \Pr^{\mathfrak{G}'}\{\langle Q, \eta' \rangle \not\models \diamond\checkmark\} \right)$$

$$\begin{aligned}
& + \left( \sum_{\hat{\pi}' \in \text{Paths}_{\min}^{\mathcal{G}'}(\langle Q, \eta' \rangle, \diamond\checkmark)} \mathbf{P}^{\mathcal{G}'}(\hat{\pi}') \cdot r(\hat{\pi}') \right) \\
& \text{if } s = \langle \epsilon, \eta' \rangle \text{ for some valuation } \eta' \text{ and } r^Q(s) = 0 \text{ otherwise.} \\
& = \text{LExpRew}^{\mathcal{R}_g \llbracket P \rrbracket}(\langle P, \eta \rangle \models \diamond\checkmark) \\
& \quad \text{where } g(\eta) = \text{LExpRew}^{\mathcal{R}_f \llbracket Q \rrbracket}(\langle Q, \eta \rangle \models \diamond\checkmark) \\
& \stackrel{I.H.}{=} \text{wlp}(P; \text{wlp}(Q, f))(\eta) \\
& = \text{wlp}(P; Q, f)(\eta) .
\end{aligned}$$

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 a terminal state of  $P$  and paths that do reach the end of  $P$  but diverge before reaching a terminal state of  $Q$ . The probability of the former is captured by  $\Pr^{\mathcal{G}}\{\langle P, \eta \rangle \not\models \diamond\checkmark\}$  and the probability of the latter is the product of the probability of the prefix and the suffix whose probability is captured by  $r^Q$ .

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

$$\text{wlp}(\text{while}^k(G) \{P\}, f)(\eta) = \text{LExpRew}^{\mathcal{R}_f \llbracket \text{while}^k(G) \{P\} \rrbracket}(\eta) . \quad (3.7)$$

Equation (3.7) shows that the  $k$ -bounded loop semantics agree. As before this can be done by induction on  $k$ . Additionally we need to show that

$$\sup_k \text{wlp}(\text{while}^k(G) \{P\}, f)(\eta) = \text{wlp}(\text{while}(G) \{P\}, f)(\eta) \quad (3.8)$$

$$\sup_k \text{LExpRew}^{\mathcal{R}_f \llbracket \text{while}^k(G) \{P\} \rrbracket}(\eta) = \text{LExpRew}^{\mathcal{R}_f \llbracket \text{while}(G) \{P\} \rrbracket}(\eta) . \quad (3.9)$$

This can be done analogously to the proof of equations (3.4) and (3.5) in the proof of Theorem 2. In particular, for (3.8) with  $\Phi(x) = [G] \cdot \text{wlp}(P, x) + [\neg G] \cdot f$  it needs to be shown that  $\Phi^n(0) = \text{wlp}(\text{while}^n(G) \{P\}, f)$ . This is done by induction on  $n$ . For (3.9), again an argument over the RMDPs can be given. Since we have spelled out the details before we do not repeat all the same steps here again.

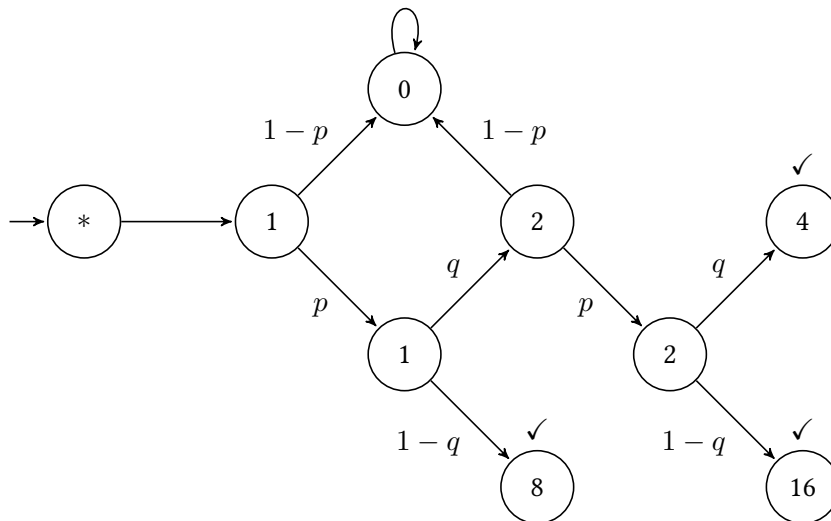
□

```

1  i := 1;
2  while (i < 4) {
3    (skip [p] i := 0);
4    (i := 2*i [q] i := 8*i);
5  }

```

(a) This program probabilistically multiplies  $i$  and terminates with 4, 8, 16 or does not terminate at all.



(b) Corresponding RMDP where each state contains the value of  $i$ . For the sake of readability we skip intermediate states which would be generated by our SOS rules but do not show any update of the variable valuation.

Figure 3.4: A program and its transition system.

**Example 3 (Semantics).** We review all our semantics on a simple program shown in Figure 3.4a. In the following we refer to this program as *prog*. It starts with  $i=1$  and on every loop iteration it either decides probabilistically to not terminate by setting  $i = 0$  or it multiplies  $i$  probabilistically by either 2 or 8. Once  $i$  has been multiplied by 8, *prog* stops. Now let us assume we need to find out the probability that *prog* terminates in a state where  $i$  is at least 16. In the operational semantics we set the reward  $f$  of the single final state that meets this condition to 1 and 0 everywhere else. The sought probability is given by

$$\text{ExpRew}^{\mathcal{R}_f[\![prog]\!] }(\langle prog, \langle * \rangle \rangle \models \diamond \checkmark) = p^2q(1 - q) .$$

Equivalently, we may define a post-expectation  $f = [i \geq 16]$  and determine  $wp(prog, f)$ . For this a fixed point of the loop needs to be found and we set up a fixed point iteration as we did in Example 1, page 28 before. Here, let *body* denote the loop's body in lines 3 – 4. Then

$$\Phi(X) = [i < 4] \cdot wp(\text{body}, X) + [i \geq 4] \cdot [i \geq 16] .$$

After four iterations we find the least fixed point as

$$\Phi^3(0) = [1 \leq i < 2] \cdot p^2q(1 - q) + [2 \leq i < 4] \cdot p(1 - q) + [i \geq 16] = \Phi^4(0)$$

where we use the fact that  $i$  is an integer. Plugging in the initialisation of  $i$  in the first line we obtain

$$wp(prog, f) = p^2q(1 - q) .$$

Now, if we evaluate the liberal semantics, we find the probability to either terminate with  $i$  at least 16 or not terminate at all. This is given by

$$\text{LEXPrew}^{\mathcal{R}_f[\![prog]\!] }(\langle prog, \langle * \rangle \rangle \models \diamond \checkmark) = p^2q(1 - q) + 1 - p + pq(1 - p) .$$

Equivalently we obtain the same result using the *wlp* semantics. As above we set up a fixed point iteration with

$$\Phi(X) = [i < 4] \cdot wlp(\text{body}, X) + [i \geq 4] \cdot [i \geq 16] .$$

But since we are now looking for the greatest fixed point we start with  $\Phi(1)$ . Again, using the fact that  $i$  is a positive integer, after four iterations we find the fixed point

$$\begin{aligned} & [1 \leq i < 2] \cdot p^2q(1 - q) + [2 \leq i < 4] \cdot p(1 - q) + [i < 4] \cdot (1 - p) + [i \geq 16] \\ & + [i < 2] \cdot pq(1 - p) . \end{aligned}$$

With the initialisation of  $i = 1$  we obtain

$$wlp(prog, f) = p^2q(1 - q) + 1 - p + pq(1 - p)$$

as expected. ■

**Remark 7** (On expectations being bound). Our definitions above assume that expectations that are transformed by  $wlp$  must range over the  $[0, 1]$  interval only. In the original work [52] the authors allow a more flexible definition. They assume that an expectation  $f$  is bounded within the context of a given program. This means that there exists some real number  $\alpha$  such that for every final state  $\eta$  of program  $P$ ,  $f(\eta) \leq \alpha$ . Then—for this particular program  $P$  and expectation  $f$ —they can define  $wlp(\text{abort}, g) = \alpha$  for any  $g$ . Our Theorem 3 still holds but the non-termination probability has to be scaled by that  $\alpha$ . This framework allows to reason about more general expectations but has some problems:

1. Different post-expectation require different bounds  $\alpha$  and thus induce different semantics. For example, reconsider program  $P$  from (3.6), page 49. In order to calculate  $wlp(P, x) = 10$ , we need  $\alpha = 10$ . But for  $wlp(P, x^2)$  we need  $\alpha$  to be at least 100 which changes the semantics of  $\text{abort}$ .
2. Unless  $\alpha$  is chosen tightly it is not clear how to interpret the result of  $wlp$ . For example take  $\alpha = 100$ , then  $wlp(P, x) = 0.5 \cdot 10 + 0.5 \cdot 100 = 55$ .
3. Also different programs may require different bounds. Thus the inequality  $wlp(P_1, f) \leq wlp(P_2, f)$  may be interpreted as refinement of  $P_1$  by  $P_2$  only if we know that both  $wlp$ 's were calculated with the same  $\alpha$ .

The above issues motivate our restriction to indicator random variables (naturally bounded by 1) when reasoning with  $wlp$ . ■

### 3.4 On non-implementable resolution of choices

Here we elaborate why the “infimum” in the definition of the operational equivalent of  $wlp$

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

is crucial. We demonstrate this by means of an example, which was inspired by [9]. Let  $P$  be the program in Figure 3.5, page 58. Obviously we have  $wlp(P, [x = 1]) = 0$  because the least favourable choice is to terminate the first loop after the first iteration and thus never increase  $i$ . Consequently the second loop is not executed and the program terminates with  $x = i = 0$ . However  $wlp(P, [x = 0])$  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.

```

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

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, 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 definition of  $LExpRew$  uses an infimum. 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 scheduler that does achieve the minimal expected reward.

## Chapter 4

# 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.

### 4.1 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

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

```

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

```

Figure 4.1: The “fishbowl” program

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

$$\begin{aligned}
 \mathbb{E}(X|\mathbb{1}_B = 1) &= \sum_{x \in \text{range}(X)} x \cdot \mathbf{P}(X = x|\mathbb{1}_B = 1) \\
 &= \frac{\sum_{x \in \text{range}(X)} x \cdot \mathbf{P}(X = x, \mathbb{1}_B = 1)}{\mathbf{P}(\mathbb{1}_B = 1)}. \tag{4.1}
 \end{aligned}$$

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 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 [66, 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 4.1. We are looking for the probability that the fish  $f_1$ , initially contained in the fishbowl, was a piranha. The *observation* that the fish removed has been a piranha is built into the program directly using the *observe* statement. In order to understand this program’s behaviour consider Figure 4.2. 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<sup>1</sup>. It proceeds to set  $f_1$

<sup>1</sup>As we did before in Figure 3.2b, page 39, we collapse all possible initial states where the variables have some particular values into one where we do not care about the variable valuation be-

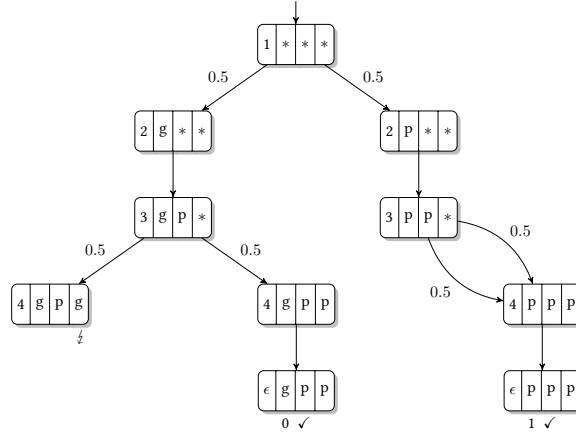


Figure 4.2: Operational semantics of the fishbowl program

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.  $f_1 = \text{piranha}$ . This gives rise to a random variable  $\mathbb{1}_A$  which we evaluate on the terminal states for the RMDP and hence we assign a reward of 1 to the state  $\langle \epsilon, p, p, p \rangle$  and 0 everywhere else. Let event  $B$  be that the observation is met, i.e.  $sample = \text{piranha}$  holds. In order to characterise this event in the RMDP, we label  $\langle 4, g, p, g \rangle$  where  $sample \neq \text{piranha}$  with  $\frac{1}{2}$ . Consequently,  $B$  contains only those runs of the RMDP that avoid reaching a  $\frac{1}{2}$ -state. Then the answer to the puzzle is given by

$$\mathbb{E}(\mathbb{1}_A | \mathbb{1}_B = 1) = \frac{1 \cdot 0.5 + 0 \cdot 0.25}{0.5 + 0.25} = \frac{1/2}{3/4} = \frac{2}{3}.$$

In order to reason formally about conditional rewards of pGCL programs with observations the operational semantics needs to be defined. However this requires only a little adaptation of Definition 22 on page 38. The SOS rules defining  $\rightarrow$  are simply extended by one rule, namely

$$\frac{\eta \models G}{\langle \text{observe}(G), \eta \rangle \rightarrow \langle \epsilon, \eta \rangle}.$$

Thus if the current state satisfies the observation, the execution continues. Otherwise the execution is blocked and cannot reach a final,  $\checkmark$ -labelled state any more. Additionally, we wish to differ between successful termination, non-termination

---

cause the program will overwrite these values anyway.

and non-termination due to a violation of an observation. Therefore the labelling function of  $\mathcal{R}_f\llbracket P \rrbracket$  is extended to be

$$L(\langle s, \eta \rangle) = \begin{cases} \checkmark & \text{if } s = \epsilon \\ \not\checkmark & \text{if } s = \text{observe}(G); P \text{ where } \eta \not\models G \\ & \text{and } P \text{ is a (possibly empty) pGCL program} \\ \emptyset & \text{otherwise .} \end{cases}$$

This leads to the following definition<sup>2</sup> of conditional minimal expected rewards in RMDPs for pGCL programs with observations.

**Definition 25** (Conditional expected reward). For a pGCL program  $P$  with observations and a post-expectation  $f$  let  $\mathcal{R}_f\llbracket P \rrbracket$  be the corresponding RMDP with state space  $S$ , a set of final states labelled with  $\checkmark$  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  $\checkmark$ -states in  $\mathcal{R}_f\llbracket P \rrbracket$ . The *conditional minimal expected reward* until reaching  $\checkmark$  from  $s$  avoiding  $\not\checkmark$ -states, denoted  $CExpRew^{\mathcal{R}_f\llbracket P \rrbracket}(s \models \diamond\checkmark \mid \neg\diamond\not\checkmark)$ , is defined by:

$$\inf_{\mathfrak{C}} \frac{\sum_{c \in \mathfrak{C}} c \cdot \Pr^{\mathfrak{C}}\{\pi \in Paths^{\mathfrak{C}}(s, \diamond\checkmark) \mid r_{\checkmark}(\pi) = c\}}{1 - \Pr^{\mathfrak{C}}\{\pi \in Paths^{\mathfrak{C}}(s, \diamond\not\checkmark)\}} .$$

■

By construction of  $\mathcal{R}_f\llbracket P \rrbracket$ , a path which reached a final  $\checkmark$ -state cannot have visited a  $\not\checkmark$ -state. Therefore, for the above definition it suffices to specify (in the numerator) that a  $\checkmark$ -state is reached without mentioning the implicit constraint that on this path no  $\not\checkmark$ -states were encountered. Previously, it was emphasised that we condition on *avoiding*  $\not\checkmark$ -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. To further elaborate on this point, let us consider the following example:

$$\begin{aligned} & \{ \text{abort} \} [0.5] \{ \{ x := 0 \} [0.5] \{ x := 1 \} \}; \\ & \text{observe}(x = 1) . \end{aligned}$$

With probability 0.5 the program diverges and does not reach the observation—and hence does not violate it. Accordingly, Definition 25 determines the conditional expectation of  $x$  to be

$$\frac{1 \cdot 0.25 + 0 \cdot 0.25}{1 - 0.25} = \frac{1}{3} .$$

<sup>2</sup>We are of course not the first to consider conditioning on Markov models, cf. [2, 4], for example.

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 introduces inconsistencies. In the given example the expectation of  $x$  conditioned on *termination and passing all observations* 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

$$\begin{aligned} &\{abort\} [0.5] \{\{x := 0\} [0.5] \{x := 1\}\}; \\ &observe(true) \end{aligned}$$

would produce different expectations for  $x$ . The first program does not have any observation and thus the average value of  $x$  would be determined to be 0.25. Contrary to this the vacuously true observation in the second program would implicitly condition the expectation of  $x$  on the termination of the program and hence rescale the result to  $0.25/0.5 = 0.5$ . In *our* semantics (Definition 25), both programs are equivalent and produce the same expected value. Section 4.2.2 revisits 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.

## 4.2 Expectation transformer semantics for programs with conditioning

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

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

where  $G$  is a boolean predicate. Additionally recall the notion of the greatest *liberal* pre-expectation  $wlp$  from Section 3.3, page 48. 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. For more details, see Figure 3.3, page 50. The restriction to 1-bounded expectations comes naturally in the context of conditional expectations because we will use  $wlp$  to measure the *probability* to pass all encountered observations. In Chapter 5 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

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

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

**Definition 26** (Conditional pre-expectation). For fully probabilistic<sup>3</sup> pGCL programs  $P$  with *observe* statements we define the *conditional pre-expectation*  $\underline{cwp}$  as

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

■

The definition of  $\underline{cwp}$  is deliberately restricted to programs without non-deterministic choices. This is because the non-determinism could be resolved differently when evaluating the numerator and the denominator. In Section 4.2.3 we return to this problem and show why we cannot give a better definition for  $\underline{cwp}$ .

**Example 4** (Evaluation of  $\underline{cwp}$ ). Let us revisit the fishbowl example and let  $P$  be the program from Figure 4.1, page 60. We write  $P_{i-j}$  to denote the subprogram in lines  $i$  to  $j$ . Again, the goal is to determine the probability that the fish originally contained in the fishbowl was a piranha, given that the sampled fish is a piranha. This quantity is given by

$$\begin{aligned} & \underline{cwp}(P, [f_1 = \text{piranha}]) \\ &= \frac{wp(P, [f_1 = \text{piranha}])}{wlp(P, 1)} \\ &= \frac{wp(P_{1-3}, wp(\text{observe}(\text{sample} = \text{piranha}), [f_1 = \text{piranha}]))}{wlp(P_{1-3}, wlp(\text{observe}(\text{sample} = \text{piranha}), 1))} \\ &= \frac{wp(P_{1-3}, [\text{sample} = \text{piranha} \wedge f_1 = \text{piranha}])}{wlp(P_{1-3}, [\text{sample} = \text{piranha}])} \\ &= \dots \\ &= \frac{1/2}{3/4} = \frac{2}{3} . \end{aligned}$$

<sup>3</sup>Programs without non-deterministic choice.

We see how the conditional pre-expectation amounts to determining the expectation of  $[f_1 = \textit{piranha}]$  and the indicator random variable corresponding to the observation  $\textit{sample} = \textit{piranha}$  and dividing this by the probability that  $P$  does not terminate or establishes  $\textit{sample} = \textit{piranha}$ . This reminds us of the definition of conditional expectations in (4.1), page 60. 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 anywhere in the program text. The steps that were left out above are purely syntactical and follow the rules given in Figure 3.1, page 36 and Figure 3.3, page 50 respectively. ■

Our new transformer is a natural extension of  $\textit{wp}$  because it behaves identically on observation-free (fully probabilistic) programs:

**Proposition 3** ( $\textit{cwp}$  generalises  $\textit{wp}$ ). For program  $P$  that does not contain any observations, it holds

$$\textit{cwp}(P, f) = \textit{wp}(P, f) .$$

■

*Proof.* With Theorem 3 and Lemma 4 on page 51 we have that for an observation-free program  $P$  and any variable valuation  $\eta$ :

$$\begin{aligned} \textit{wlp}(P, 1)(\eta) &= \textit{LEXPrew}^{\mathcal{R}_1}[\![P]\!](\langle P, \eta \rangle \models \diamond\checkmark) \\ &= \inf_{\mathfrak{S}} \left\{ \textit{Pr}^{\mathfrak{S}}(\langle P, \eta \rangle \not\models \diamond\checkmark) + \textit{Pr}^{\mathfrak{S}}(\langle P, \eta \rangle \models \diamond\checkmark) \right\} \\ &= 1 . \end{aligned}$$

Thus

$$\textit{cwp}(P, f) = \frac{\textit{wp}(P, f)}{\textit{wlp}(P, 1)} = \frac{\textit{wp}(P, f)}{1} = \textit{wp}(P, f) .$$

□

Proposition 3 does not make any restrictions to probabilistic programs even though we previously defined  $\textit{cwp}$  only for such programs. The reason is that in the special case of observation-free programs  $\textit{wlp}(P, 1)$  is 1 regardless of the chosen scheduler as shown above. Therefore  $\textit{cwp}$  is well-defined even for non-deterministic programs in this special case.

A remark about notation: our new transformer is denoted  $\textit{cwp}$  where the “c” stands for *conditional* and we keep the “wp” because it essentially behaves like  $\textit{wp}$ . 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. The underscore in

$cwp$  reminds us that there are two quantities we have to determine first and then divide them to get the desired result.

In the following we set out to establish a transfer theorem for conditional pGCL programs. Before doing so, the properties of  $wp$  and  $wlp$  need to be re-examined for the extended language. First note that our pre-expectation definition of  $observe(G)$  is merely syntactic sugar with respect to the  $wp$  transformer while it is a genuinely new language construct with respect to the  $wlp$  transformer. For  $wp$  it holds

$$\begin{aligned} wp(observe(G), f) &= [G] \cdot f + [\neg G] \cdot 0 \\ &= wp(if(\neg G) \{abort\} else \{skip\}, f) \end{aligned}$$

for all expectations  $f$ . However 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 in pGCL. We state this fact without proof and refer to a result on a dual programming concept which is usually referred to as *magic* [53]. Nonetheless we have the following result:

**Proposition 4** (Transfer theorem for pGCL with *observe*). Our results

- i) Theorem 2 and
- ii) Theorem 3

remain valid for pGCL with *observe* statements. ■

*Proof.* i) This is straightforward because we have seen above that

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

Thus the  $wp$  semantics of *observe* allows to syntactically rewrite *observe* using pGCL statements for which Theorem 2 already holds.

- ii) We can extend the induction proof of Theorem 3 with an extra base case:

Let  $P$  be the observation statement  $observe(G)$ . We derive:

$$LExpRew^{\mathcal{R}_f \llbracket observe(G) \rrbracket} (\langle observe(G), \eta \rangle \models \diamond \checkmark)$$

$$\begin{aligned}
&= \inf_{\mathfrak{S}} \left( \Pr^{\mathfrak{S}}(\langle \text{observe}(G), \eta \rangle \not\models \diamond\checkmark) \right. \\
&\quad \left. + \sum_{\hat{\pi} \in \text{Paths}_{\min}^{\mathfrak{S}}(\langle \text{observe}(G), \eta \rangle, \diamond\checkmark)} \mathbf{P}^{\mathfrak{S}}(\hat{\pi}) \cdot r(\hat{\pi}) \right) \\
&= \begin{cases} \inf_{\mathfrak{S}} \mathbf{P}^{\mathfrak{S}}(\langle \text{observe}(G), \eta \rangle \langle \epsilon, \eta \rangle) \cdot r(\eta) & \text{if } \eta \models G, \text{ and} \\ \inf_{\mathfrak{S}} \mathbf{P}^{\mathfrak{S}}(\langle \text{observe}(G), \eta \rangle \langle \epsilon, \eta \rangle) \cdot 0 & \text{else} \end{cases} \\
&= [\eta \models G] \cdot f(\eta) \\
&= ([G] \cdot f)(\eta) \\
&= \text{wlp}(\text{observe}(G), f)(\eta).
\end{aligned}$$

□

The next proposition explains our choice of  $\text{wlp}(P, 1)$  in the denominator of the  $\text{cwp}$  definition above.

**Proposition 5** (The probability of all admissible runs). Given a pGCL program with observations,  $\text{wlp}(P, 1)$  gives the probability of all paths that either terminate in a  $\checkmark$ -state or that do not terminate. Equivalently, this is the probability of all paths that avoid the violation of any observation in the program. ■

*Proof.*

$$\begin{aligned}
&\text{wlp}(P, 1)(\eta) \\
&\stackrel{\text{Prop. 4ii}}{=} \text{LEXPrew}^{\mathcal{R}_1[P]}(\langle P, \eta \rangle \models \diamond\checkmark) \\
&= \inf_{\mathfrak{S}} \left( \Pr^{\mathfrak{S}}(\langle P, \eta \rangle \not\models \diamond\checkmark) \quad + \quad \sum_{\hat{\pi} \in \text{Paths}_{\min}^{\mathfrak{S}}(\langle P, \eta \rangle, \diamond\checkmark)} \mathbf{P}^{\mathfrak{S}}(\hat{\pi}) \cdot 1 \right) \\
&= \inf_{\mathfrak{S}} \left( 1 - \Pr^{\mathfrak{S}}\{ \pi \in \text{Paths}^{\mathfrak{S}}(s, \diamond\checkmark) \} \right) .
\end{aligned}$$

To show the validity of the last equation, fix any scheduler  $\mathfrak{S}$ . By construction of  $\mathcal{R}_1^{\mathfrak{S}}[P]$ , any path that visits a  $\checkmark$ -state cannot diverge nor visit an  $\checkmark$ -state before or after the  $\checkmark$ -state. Conversely, any path that terminates in a  $\checkmark$ -state or is infinite cannot visit a  $\checkmark$ -state. Thus we can partition the set of paths in  $\mathcal{R}_1^{\mathfrak{S}}[P]$  into two subsets: one with all paths that reach a  $\checkmark$ -state and one with all paths that do not. This partitioning is used in the last step of the proof. □

Finally we can extend our transfer theorem to conditional expectations over (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 26 on page 64 this leads to the following result.

**Theorem 4** (Transfer theorem for conditional programs). For any fully probabilistic pGCL-program  $P$ , possibly with *observe* statements, variable valuation  $\eta$ , and post-expectation  $f$ :

$$\underline{cwp}(P, f)(\eta) = CExpRew^{\mathcal{R}_f[P]}(\langle P, \eta \rangle \models \diamond\checkmark \mid \neg\diamond\checkmark) .$$

*Proof.*

$$\begin{aligned} & CExpRew^{\mathcal{R}_f[P]}(\langle P, \eta \rangle \models \diamond\checkmark \mid \neg\diamond\checkmark) \\ = & \inf_{\mathfrak{S}} \frac{\sum_{c \in \mathfrak{C}} c \cdot \Pr^{\mathfrak{S}}\{\pi \in Paths^{\mathfrak{S}}(s, \diamond\checkmark) \mid r_{\checkmark}(\pi) = c\}}{1 - \Pr^{\mathfrak{S}}\{\pi \in Paths^{\mathfrak{S}}(s, \diamond\checkmark)\}} \\ \stackrel{i)}{=} & \frac{\sum_{c \in \mathfrak{C}} c \cdot \Pr\{\pi \in Paths(s, \diamond\checkmark) \mid r_{\checkmark}(\pi) = c\}}{1 - \Pr\{\pi \in Paths(s, \diamond\checkmark)\}} \\ \stackrel{ii)}{=} & \frac{ExpRew^{\mathcal{R}_f[P]}(\langle P, \eta \rangle \models \diamond\checkmark)}{1 - \Pr\{\pi \in Paths(s, \diamond\checkmark)\}} \\ \stackrel{iii)}{=} & \frac{wp(P, f)}{wlp(P, 1)} \\ = & \underline{cwp}(P, f) . \end{aligned}$$

Where the equations hold with the following arguments:

- i) As  $P$  is fully probabilistic no scheduler is needed and the numerator and denominator are thus decoupled.
- ii) Apply Definition 17, page 22 in the numerator.
- iii) Apply Proposition 4, page 66 in the numerator and Proposition 5, page 67 in the denominator.

□

### 4.2.1 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  $x$ .

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

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 the 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, thanks to Theorem 4, we may evaluate the expectation transformer  $\underline{cwp}$  and see that

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

This example<sup>4</sup> shows that the conditional pre-expectation of probabilistic choice depends on its context and is not obtained as the weighted average between conditional pre-expectations of subprograms (as was the case with  $wp$ ).

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

<sup>4</sup>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  $x$ .

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  $Q$ .

$$\begin{array}{ll}
 P : & x := 1; \\
 & \text{while}(x = 1) \{ \\
 & \quad x := 1 \\
 & \} \\
 Q : & x := 1; \\
 & \text{while}(x = 1) \{ \\
 & \quad \{x := 1\} [0.5] \{x := 2\}; \\
 & \quad \text{observe}(x = 1); \\
 & \}
 \end{array}$$

Clearly the non-probabilistic program  $P$  does not terminate and thus its weakest liberal pre-expectation  $wlp(P, 1)$  is 1 and its conditional pre-expectation  $\underline{cwp}(P, f)$  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 applying the  $wlp$  semantics, which involves finding the fixed point of the loop.

$$\begin{aligned}
 & wlp(Q, 1) \\
 = & wlp(x := 1, wlp(\text{while}(x = 1) \{ \{x := 1\} [0.5] \{x := 2\}; \\
 & \quad \text{observe}(x = 1) \}, 1)) \\
 = & wlp(x := 1, \underset{z}{\text{gfp}}([x = 1] \cdot wlp(\{x := 1\} [0.5] \{x := 2\}; \\
 & \quad \text{observe}(x = 1), z) + [x \neq 1] \cdot x)) \\
 = & wlp(x := 1, \underset{n}{\text{inf}}([x = 1] \cdot wlp(\{x := 1\} [0.5] \{x := 2\}; \\
 & \quad \text{observe}(x = 1), 1) + [x \neq 1] \cdot x)^n) \\
 = & wlp(x := 1, \underset{n}{\text{inf}}([x = 1] \cdot wlp(\{x := 1\} [0.5] \{x := 2\}; \text{observe}(x = 1), \\
 & \quad [x = 1] \cdot 0.5 + [x \neq 1] \cdot x) + [x \neq 1] \cdot x)^{n-1}) \\
 = & wlp(x := 1, \underset{n}{\text{inf}}([x = 1] \cdot wlp(\{x := 1\} [0.5] \{x := 2\}; \text{observe}(x = 1), \\
 & \quad [x = 1] \cdot 0.5^2 + [x \neq 1] \cdot x) + [x \neq 1] \cdot x)^{n-2}) \\
 & \vdots \\
 = & wlp(x := 1, [x \neq 1] \cdot x)
 \end{aligned}$$

= 0

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.

#### 4.2.2 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

$$\frac{wp(P, f)}{wp(P, 1)}. \quad (4.2)$$

In fact this is the definition taken in the work of Nori et al. [40] and Claret et al. [18]. But as already discussed in Section 4.1, page 62ff., 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 this definition does not generalise  $wp$  for observation-free programs as in Proposition 3, page 65.

#### 4.2.3 Expectation transformers and non-determinism

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 determine 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 4.3. 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  $CExpRew^{\mathcal{R}_\mathfrak{S}_\mu}(\diamond T \mid \neg \diamond \frac{1}{2}) = 1.5$  and  $CExpRew^{\mathcal{R}_\mathfrak{S}_\nu}(\diamond T \mid \neg \diamond \frac{1}{2}) = 1.4$ . Hence  $CExpRew^{\mathcal{R}}(\diamond T \mid \neg \diamond \frac{1}{2}) = 1.4$  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  $CExpRew^{\mathcal{R}'_\mathfrak{S}_\mu}(\diamond T \mid \neg \diamond \frac{1}{2}) = 2$  and  $CExpRew^{\mathcal{R}'_\mathfrak{S}_\nu}(\diamond T \mid \neg \diamond \frac{1}{2}) = 2.2$ . So

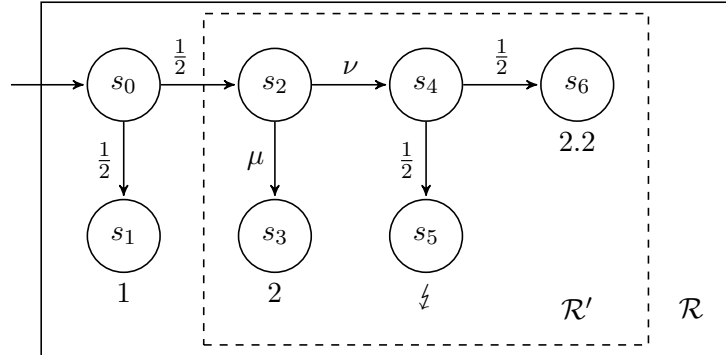


Figure 4.3: Schedulers are not compositional: the choice which minimises the conditional expected reward in  $\mathcal{R}'$  is not minimal inside the larger context of  $\mathcal{R}$ .

in  $\mathcal{R}'$  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

$$\mathcal{T}(\{P\} \parallel \{Q\}, X) = \min_{\preceq} \{\mathcal{T}(P, X), \mathcal{T}(Q, X)\}$$

must fail for any representation of conditional expectations  $X$  and any order  $\preceq$  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.

### 4.3 Reasoning with conditioning

Note that most of 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 apply to programs with an arbitrary number of observations written at arbitrary positions within the program text.

### 4.3.1 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. Others [18, 40] have pointed out that *observe* statements can be replaced by a non-terminating loops. This is due to their choice to define conditional semantics that normalise with respect to the terminating behaviour of programs as discussed in Sections 4.1, page 62 and 4.2.2, page 71. In view of that definition the following equivalence between program constructs proves the claim:

$$\begin{aligned} wp(\text{observe}(\neg G), f) &= wp(\text{if}(G) \{ \text{abort} \} \text{else} \{ \text{skip} \}, f) \\ &= wp(\text{while}(\neg G) \{ \text{skip} \}, f) . \end{aligned}$$

The *conditional* expectations can be then computed on this transformed observation-free program.

In contrast, our semantics does not require that a run terminates but only that it does not violate any observation. In the following it is shown that also in our semantics the observations can be removed but the program transformation turns out to be slightly 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  $P'$  according to the following steps:

1. Initialise *rerun* to *false*.
2. Apply the following rewriting rules:

$$\begin{aligned} \text{observe}(G) &\rightarrow \text{if}(\neg G) \{ \text{rerun} := \text{true} \} \text{else} \{ \text{skip} \} \\ x := E &\rightarrow \text{if}(\neg \text{rerun}) \{ x := E \} \text{else} \{ \text{skip} \} \\ \text{abort} &\rightarrow \text{if}(\neg \text{rerun}) \{ \text{abort} \} \text{else} \{ \text{skip} \} \\ \text{while}(G) \{ \dots \} &\rightarrow \text{while}(G \wedge \neg \text{rerun}) \{ \dots \} . \end{aligned} \tag{4.3}$$

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

$$\begin{aligned} & \{x := 0\} [0.5] \{x := 1\} \\ & \text{observe}(x \neq 0); \\ & y := y/x; \end{aligned}$$

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

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

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:

$$\begin{aligned} & s_1, \dots, s_n := x_1, \dots, x_n; \\ & \text{rerun} := \text{true}; \\ & \text{while}(\text{rerun}) \{ \\ & \quad x_1, \dots, x_n := s_1, \dots, s_n; \\ & \quad P'; \\ & \} \end{aligned} \tag{4.4}$$

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

**Theorem 5.** Let programs  $P$  and  $P''$  be as above. Then

$$CExpRew^{\mathcal{R}_f \llbracket P \rrbracket}(\langle P, \eta \rangle \models \diamond\checkmark \mid \neg\diamond\checkmark) = ExpRew^{\mathcal{R}_f \llbracket P'' \rrbracket}(\langle P'', \eta \rangle \models \diamond\checkmark) .$$

Additionally, for fully probabilistic programs it holds that

$$\underline{cwp}(P, f) = \underline{wp}(P'', f) .$$

*Proof.* We begin with the more general claim over the operational semantics. Let  $\eta$  be some initial state of  $P$ .

$$CExpRew^{\mathcal{R}_f \llbracket P \rrbracket}(\langle P, \eta \rangle \models \diamond\checkmark \mid \neg\diamond\checkmark) \quad \text{in } P \quad (4.5)$$

$$= CExpRew^{\mathcal{R}_f \llbracket P' \rrbracket}(\langle P, \eta \rangle \models \diamond\checkmark \mid \neg\diamond rerun) \quad \text{in } P' \quad (4.6)$$

$$= \inf_{\mathfrak{S}} \frac{\sum_{c \in \mathcal{C}} c \cdot \Pr^{\mathfrak{S}} \{ \pi \in Paths^{\mathfrak{S}}(s, \diamond\checkmark) \mid r_{\checkmark}(\pi) = c \}}{1 - \Pr^{\mathfrak{S}} \{ \pi \in Paths^{\mathfrak{S}}(s, \diamond rerun) \}} \quad (4.7)$$

$$= \inf_{\mathfrak{S}} \frac{\sum_{\hat{\pi} \in Paths_{\min}^{\mathfrak{S}}(s, \diamond\checkmark)} \mathbf{P}^{\mathfrak{S}}(\hat{\pi}) \cdot r(\hat{\pi})}{1 - \Pr^{\mathfrak{S}} \{ \pi \in Paths^{\mathfrak{S}}(s, \diamond rerun) \}} \quad (4.8)$$

$$= \inf_{\mathfrak{S}} \sum_{i=0}^{\infty} (\Pr^{\mathfrak{S}} \{ \pi \in Paths^{\mathfrak{S}}(s, \diamond rerun) \})^i \cdot \sum_{\hat{\pi} \in Paths_{\min}^{\mathfrak{S}}(s, \diamond\checkmark)} \mathbf{P}^{\mathfrak{S}}(\hat{\pi}) \cdot r(\hat{\pi}) \quad (4.9)$$

$$= \inf_{\mathfrak{S}} \sum_{\hat{\pi} \in Paths_{\min}^{\mathfrak{S}}(s, \diamond\checkmark)} \sum_{i=0}^{\infty} \left( (\Pr^{\mathfrak{S}} \{ \pi \in Paths^{\mathfrak{S}}(s, \diamond rerun) \})^i \cdot \mathbf{P}^{\mathfrak{S}}(\hat{\pi}) \cdot r(\hat{\pi}) \right) \quad (4.10)$$

$$= \inf_{\mathfrak{S}} \sum_{\hat{\pi} \in Paths_{\min}^{\mathfrak{S}}(s, \diamond\checkmark)} \mathbf{P}^{\mathfrak{S}}(\hat{\pi}) \cdot r(\hat{\pi}) \quad \text{in } P'' \quad (4.11)$$

$$= ExpRew^{\mathcal{R}_f \llbracket P'' \rrbracket}(\langle P'', \eta \rangle \models \diamond\checkmark) . \quad (4.12)$$

The equality (4.6) holds because, by construction, the probability to violate an observation in  $P$  agrees with the probability to reach a state in  $P'$  where *rerun* is *true*. In order to obtain equation (4.9) we use the fact that for a fixed real value  $r$  and probability  $a$  it holds

$$\frac{r}{1-a} = \sum_{i=0}^{\infty} a^i r .$$

Rewriting (4.9) into (4.10) precisely captures the expected cumulative reward of all terminating paths in  $P''$  which is the expression in the following line.

The claim for  $\underline{cwp}$  and  $\underline{wp}$  for fully probabilistic programs directly follows from Theorem 4.  $\square$

We illustrate the two main steps of our transformation using our fishbowl example.

**Example 5** (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 in (4.3):

```

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

```

Final result  $P''$  using transformations in (4.4):

```

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] {
11        if (!rerun) {f1 := piranha;}}
12    {if (!rerun) {f2 := piranha;}}
13    {if (!rerun) {sample := f1;}} [0.5] {
14        if (!rerun) {sample := f2;}}
15    if (sample != piranha) {rerun := true;}

```

14 } ■

### 4.3.2 Replacing loops by observations

Theorem 5 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 5. 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 5 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 5 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 5. We see that the variables  $s_1, \dots, s_3$  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 which ensures that the loop is executed at least once. 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 27** (iid loop). A loop  $\text{while}(G) \{P\}$  is called *iid*, if  $\text{wp}(P, f) = \text{wp}(P^k, f)$  for all expectations  $f$  and  $k \in \mathbb{N} \setminus \{0\}$ , where

$$P^k = \underbrace{P; P; \dots P}_k$$

is the  $k$ -fold repetition of  $P$ . ■

The aim of Definition 27 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  $\omega$ , such that  $\mathbb{E}(\mathbb{1}_\omega)$  differs between the two distributions.

**Example 6** (iid loops). Consider the programs  $P$  and  $Q$ :

$$\begin{array}{ll}
 P : & b := 0 \\
 & \text{while}(c) \{ \\
 & \quad b := 1 - b \\
 & \quad \{c := 0\} [1/2] \{c := 1\} \\
 & \} \\
 \\
 Q : & \text{while}(c) \{ \\
 & \quad b := 1 \\
 & \quad \{c := 0\} [1/2] \{c := 1\} \\
 & \quad \text{if}(c) \{ \\
 & \quad \quad b := 0 \\
 & \quad \quad \{c := 0\} [1/2] \{c := 1\} \\
 & \quad \} \\
 & \}
 \end{array}$$

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  $\text{body}$  denote the loop body in  $P$ . We have

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

Thus running the loop once produces an expectation which is different from the expectation produced after two runs and that by Definition 27 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.:

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

This example also shows 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 rewritten as an iid loop. ■

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

**Theorem 6** (Transformation of iid loops). Let  $\text{loop} = \text{while}(G) \{P\}$  be an iid loop and let  $Q = \text{if}(G) \{P; \text{observe}(\neg G)\} \text{else} \{\text{skip}\}$ . Then for any expectation  $f$

$$wp(\text{loop}, f) = \underline{cwp}(Q, f) .$$

*Proof.* Apply Theorem 5 to program  $Q$ . Let the resulting program be  $\text{loop}'$ . Since  $\text{loop}$  is iid, from Definition 27 we have that  $wp(P, f) = wp(P^k, f)$  for all  $f$  and  $k$  and therefore  $\text{loop}'$  is iid, too. Thus the same simplification steps as in Example 5 apply: there is only one *observe* statement at the end of  $\text{loop}'$  and furthermore there is no data flow between iterations of  $\text{loop}'$ . 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  $\text{loop}$ . □

### 4.3.3 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. 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 4.4 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.

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

Figure 4.4: Program transformation for hoisting *observe* statements. In the transformation of the while-loop the function  $\pi_2$  is the projection to the second component of  $\mathcal{T}$ .

With the transformation rules from Figure 4.4 we establish the following result.

**Theorem 7** (Correctness of hoisting). Let  $P$  be a fully probabilistic program and admit at least one feasible run for every initial state and  $\mathcal{T}(P, 1) = (\hat{P}, \hat{h})$ . Then for any expectation  $f$ ,

$$\underline{cwp}(P, f) = wp(\hat{P}, f) .$$

*Proof.* We have that  $\underline{cwp}(P, f) = \frac{wp(P, f)}{wlp(P, 1)}$ . Thus the stronger two equations below are proven and Theorem 7 follows with  $h = 1$ .

$$\hat{h} \cdot wp(\hat{P}, f) = wp(P, h \cdot f) \tag{4.13}$$

$$\hat{h} = wlp(P, h) \tag{4.14}$$

where  $\mathcal{T}(P, h) = (\hat{P}, \hat{h})$ . The proof proceeds by induction on the structure of  $P$ .  
Induction base:

- For  $P = \text{skip}$  we have  $\mathcal{T}(\text{skip}, h) = (\text{skip}, h)$  and the statement follows immediately since

$$h \cdot \text{wp}(\text{skip}, f) = h \cdot f = \text{wp}(\text{skip}, h \cdot f)$$

and

$$h = \text{wlp}(\text{skip}, h) .$$

- For  $P = \text{abort}$  we have  $\mathcal{T}(\text{abort}, h) = (\text{abort}, 1)$  and the statement follows immediately since

$$h \cdot \text{wp}(\text{abort}, f) = 0 = \text{wp}(\text{abort}, h \cdot f)$$

and

$$1 = \text{wlp}(\text{abort}, h) .$$

- For  $P = x := E$  we have  $\mathcal{T}(x := E, h) = (x := E, h[x/E])$  and the statement follows immediately since

$$\begin{aligned} h[x/E] \cdot \text{wp}(x := E, f) &= h[x/E] \cdot f[x/E] \\ &= (h \cdot f)[x/E] = \text{wp}(x := E, h \cdot f) \end{aligned}$$

and

$$h[x/E] = \text{wlp}(x := E, h) .$$

- For  $P = \text{observe}(G)$  we have  $\mathcal{T}(\text{observe}(G), h) = (\text{skip}, [G] \cdot h)$  and the statement follows immediately since

$$[G] \cdot h \cdot \text{wp}(\text{skip}, f) = [G] \cdot h \cdot f = \text{wp}(\text{observe}(G), h \cdot f)$$

and

$$[G] \cdot h = \text{wlp}(\text{observe}(G), h) .$$

Induction hypotheses: assume that for program  $P$  (and analogously for  $Q$ )

$$\text{IH}_1: \hat{h} \cdot \text{wp}(\hat{P}, f) = \text{wp}(P, h \cdot f)$$

$$\text{IH}_2: \hat{h} = \text{wlp}(P, h)$$

where  $\mathcal{T}(P, h) = (\hat{P}, \hat{h})$ .

Induction step:

- Consider the sequential composition  $P; Q$ . Let  $(\hat{Q}, \hat{h}_Q) = \mathcal{T}(Q, h)$  and  $(\hat{P}, \hat{h}_P) = \mathcal{T}(P, \hat{h}_Q)$  and hence  $\mathcal{T}(P; Q, h) = (\hat{P}; \hat{Q}, \hat{h}_P)$ . Now

$$\begin{aligned} & \hat{h}_P \cdot \text{wp}(\hat{P}; \hat{Q}, f) \\ &= \hat{h}_P \cdot \text{wp}(\hat{P}, \text{wp}(\hat{Q}, f)) \\ &= \text{wp}(P, \hat{h}_Q \cdot \text{wp}(\hat{Q}, f)) && \text{(IH}_1 \text{ on } P) \\ &= \text{wp}(P, \text{wp}(Q, h \cdot f)) && \text{(IH}_1 \text{ on } Q) \\ &= \text{wp}(P; Q, h \cdot f) \end{aligned}$$

and

$$\begin{aligned} \hat{h}_P &= \text{wlp}(P, \hat{h}_Q) && \text{(IH}_2 \text{ on } P) \\ &= \text{wlp}(P, \text{wlp}(Q, h)) && \text{(IH}_2 \text{ on } Q) \\ &= \text{wlp}(P; Q, h) . \end{aligned}$$

Consider the probabilistic choice  $\{P\} [a] \{Q\}$ , which as usual covers if-then-else as a special case. Let  $(\hat{P}, \hat{h}_P) = \mathcal{T}(P, h)$  and  $(\hat{Q}, \hat{h}_Q) = \mathcal{T}(Q, h)$ . We obtain

$$\mathcal{T}(\{P\} [a] \{Q\}, h) = (\{\hat{P}\} [a \cdot \hat{h}_P / \hat{h}] \{\hat{Q}\}, \hat{h})$$

with  $\hat{h} = a \cdot \hat{h}_P + (1 - a) \cdot \hat{h}_Q$ .

To prove the first claim

$$\hat{h} \cdot \text{wp}(\{\hat{P}\} [a \cdot \hat{h}_P / \hat{h}] \{\hat{Q}\}, f) = \text{wp}(\{P\} [a] \{Q\}, h \cdot f)$$

of the lemma we need to make a case distinction between those states that are mapped by  $\hat{h}$  to a positive number and those that are mapped to 0. In

the first case, i.e. if  $\hat{h}(\eta) > 0$ , we reason as follows:

$$\begin{aligned}
& \hat{h}(\eta) \cdot \mathbf{wp}(\{\hat{P}\} [a \cdot \hat{h}_P / \hat{h}] \{\hat{Q}\}, f)(\eta) \\
&= \hat{h}(\eta) \cdot \left( \frac{a \cdot \hat{h}_P}{\hat{h}}(\eta) \cdot \mathbf{wp}(\hat{P}, f)(\eta) \right. \\
&\quad \left. + \frac{(1-a) \cdot \hat{h}_Q}{\hat{h}}(\eta) \cdot \mathbf{wp}(\hat{Q}, f)(\eta) \right) \\
&= a(\eta) \cdot \hat{h}_P(\eta) \cdot \mathbf{wp}(\hat{P}, f)(\eta) \\
&\quad + (1-a)(\eta) \cdot \hat{h}_Q(\eta) \cdot \mathbf{wp}(\hat{Q}, f)(\eta) \\
&= a(\eta) \cdot \mathbf{wp}(P, h \cdot f)(\eta) \\
&\quad + (1-a)(\eta) \cdot \mathbf{wp}(Q, h \cdot f)(\eta) \quad (\text{IH}_1) \text{ and } (\text{IH}_2) \\
&= \mathbf{wp}(\{P\} [a] \{Q\}, h \cdot f)(\eta)
\end{aligned}$$

while in the second case, i.e. if  $\hat{h}(\eta) = 0$ , the claim holds because we will have  $\mathbf{wp}(\{P\} [a] \{Q\}, h \cdot f)(\eta) = 0$ . To see this note that if  $\hat{h}(\eta) = 0$  then either

- $a(\eta) = 0 \wedge \hat{h}_Q(\eta) = 0$ , or
- $a(\eta) = 1 \wedge \hat{h}_P(\eta) = 0$ , or
- $\hat{h}_P(\eta) = 0 \wedge \hat{h}_Q(\eta) = 0$

holds. Now assume we are in the first case (an analogous argument works for the other cases); using the  $\text{IH}_1$  over  $Q$  we obtain

$$\begin{aligned}
\mathbf{wp}(\{P\} [0] \{Q\}, h \cdot f)(\eta) &= \mathbf{wp}(Q, h \cdot f)(\eta) \\
&= \hat{h}_Q(\eta) \cdot \mathbf{wp}(Q, f)(\eta) = 0 .
\end{aligned}$$

The proof of the second claim of the lemma is straightforward:

$$\begin{aligned}
& a \cdot \hat{h}_P + (1-a) \cdot \hat{h}_Q \\
&= a \cdot \mathbf{wlp}(P, h) + (1-a) \cdot \mathbf{wlp}(Q, h) \quad (\text{IH}_2) \\
&= \mathbf{wlp}(\{P\} [a] \{Q\}, h) .
\end{aligned}$$

- Consider the loop  $\mathbf{while}(G) \{P\}$ . Let  $\hat{h} = \text{gfp } F$  where  $F(X) = [G] \cdot \mathcal{T}_P(X) + [\neg G] \cdot h$  and  $\mathcal{T}_P(\cdot)$  is a short-hand for  $\pi_2 \circ \mathcal{T}(P, \cdot)$ . Now if we let  $(\hat{P}, \theta) = \mathcal{T}(P, \hat{h})$  by definition of  $\mathcal{T}$  we obtain

$$\mathcal{T}(\mathbf{while}(G) \{P\}, h) = (\mathbf{while}(G) \{\hat{P}\}, \hat{h}).$$

The first claim of the lemma says that

$$\hat{h} \cdot \text{wp}(\text{while}(G) \{\hat{P}\}, f) = \text{wp}(\text{while}(G) \{P\}, h \cdot f) .$$

Now if we let  $H(X) = [G] \cdot \text{wp}(\hat{P}, X) + [\neg G] \cdot f$  and  $I(X) = [G] \cdot \text{wp}(P, X) + [\neg G] \cdot h \cdot f$ , the claim can be rewritten as  $\hat{h} \cdot \text{lfp } H = \text{lfp } I$  and a straightforward argument using the Kleene fixed point theorem (and the continuity of  $\text{wp}$ ) shows that it is entailed by the formula  $\forall n : \hat{h} \cdot H^n(0) = I^n(0)$ . We prove the formula by induction on  $n$ . The base case  $n = 0$  is trivial. For the induction step we reason as follows:

$$\begin{aligned} & \hat{h} \cdot H^{n+1}(0) \\ &= F(\hat{h}) \cdot H^{n+1}(0) && \text{(def. } \hat{h}) \\ &= ([G] \cdot \mathcal{T}_P(\hat{h}) + [\neg G] \cdot h) \cdot H^{n+1}(0) && \text{(def. } F) \\ &= ([G] \cdot \mathcal{T}_P(\hat{h}) + [\neg G] \cdot h) \\ &\quad \cdot ([G] \cdot \text{wp}(\hat{P}, H^n(0)) + [\neg G] \cdot f) && \text{(def. } H) \\ &= [G] \cdot \mathcal{T}_P(\hat{h}) \cdot \text{wp}(\hat{P}, H^n(0)) \\ &\quad + [\neg G] \cdot h \cdot f \\ &= [G] \cdot \theta \cdot \text{wp}(\hat{P}, H^n(0)) + [\neg G] \cdot h \cdot f && \text{(def. } \theta) \\ &= [G] \cdot \text{wp}(P, \hat{h} \cdot H^n(0)) + [\neg G] \cdot h \cdot f && \text{(IH}_1 \text{ on } P) \\ &= I(\hat{h} \cdot H^n(0)) && \text{(def. } I) \\ &= I^{n+1}(0) \end{aligned}$$

We now turn to proving the second claim

$$\hat{h} = \text{wlp}(\text{while}(G) \{P\}, h)$$

of the lemma. By letting  $J(X) = [G] \cdot \text{wlp}(P, X) + [\neg G] \cdot h$ , the claim reduces to  $\text{gfp } F = \text{gfp } J$ , which we prove showing that  $\hat{h} = \text{gfp } F$  is a fixed point of  $J$  and  $\text{gfp } J$  is a fixed point of  $F$ . (These assertions basically

imply that  $\text{gfp } F \geq \text{gfp } J$  and  $\text{gfp } J \geq \text{gfp } F$ , respectively.)

$$\begin{aligned}
J(\hat{h}) &= [G] \cdot \text{wlp}(P, \hat{h}) + [\neg G] \cdot h && \text{(def. } J) \\
&= [G] \cdot \theta + [\neg G] \cdot h && \text{(IH}_2 \text{ on } P) \\
&= [G] \cdot \mathcal{T}_P(\hat{h}) + [\neg G] \cdot h && \text{(def. } \theta) \\
&= F(\hat{h}) && \text{(def. } F) \\
&= \hat{h} && \text{(def. } \hat{h})
\end{aligned}$$

$$\begin{aligned}
F(\text{gfp } J) &= [G] \cdot \mathcal{T}_P(\text{gfp } J) + [\neg G] \cdot h && \text{(def. } F) \\
&= [G] \cdot \text{wlp}(P, \text{gfp } J) + [\neg G] \cdot h && \text{(IH}_2 \text{ on } P) \\
&= J(\text{gfp } J) && \text{(def. } J) \\
&= \text{gfp } J && \text{(def. } \text{gfp } J)
\end{aligned}$$

□

Note that we apply the transformation rules from Figure 4.4 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. For readability in all hoisting examples the transformation steps are given as program text. Instead of writing tuples such as  $(P; Q', f')$  as the result of a transformation step  $\mathcal{T}(P; Q, f)$ , we literally *hoist* the observation through the program text and write  $Q; f'; P'$ .

**Example 7** (Hoisting *observe*).

Original program

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

First step of the hoisting transformation

$$\begin{aligned}
&\{f_1 := \text{goldfish}\} [0.5] \{f_1 := \text{piranha}\}; \\
&f_2 := \text{piranha}; \\
&\text{observe}([f_1 = \text{piranha}] \cdot 0.5 + [f_2 = \text{piranha}] \cdot 0.5);
\end{aligned}$$

$$\{sample := f_1\} \left[ \frac{[f_1 = piranha] \cdot 0.5}{[f_1 = piranha] \cdot 0.5 + [f_2 = piranha] \cdot 0.5} \right] \{sample := f_2\}$$

Second step of the hoisting transformation

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

Last step of the hoisting transformation

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

■

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 as 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 by Nori et al. [57], but there instead of coin flips they introduce probability by sampling from distributions. Therefore their transformation rule is weaker as 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.

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 5) 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 4.4 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  $\omega$ -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.

#### 4.3.4 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 8** (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 [16] 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 4.5. 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

```

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 4.5: A pGCL program modelling the probabilistic behaviour of the Zeroconf protocol. We use  $\text{for}(1..N) \{ \dots \}$  to abbreviate  $i := 0; \text{while}(i < N) \{ \dots; i := i + 1 \}$ .

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<sub>1</sub>.

```

1  configured := false;
2  while (!configured) {
3      (collision := true [q] collision := false);
4      configured := true;
5      {
6          skip;
7      }
8      [p^N]
9      {
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 6 may be applied to replace the loop by an *observe* statement. Thus we obtain a program ZEROCONF<sub>2</sub>.

```

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 4.4 yielding the final program

ZEROCONF<sub>3</sub>.

```

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 + [collision=
   false](1-p^N))]
9  {
10     if (collision) {
11         configured := false;
12     }
13 }
```

By Theorem 6, we have

$$wp(\text{ZEROCONF}_1, [\text{collision} = \text{true}]) = \underline{cwp}(\text{ZEROCONF}_2, [\text{collision} = \text{true}])$$

and by Theorem 7,

$$\underline{cwp}(\text{ZEROCONF}_2, [\text{collision} = \text{true}]) = wp(\text{ZEROCONF}_3, [\text{collision} = \text{true}]) .$$

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 4.6 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 4.6. 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  $1.53 \cdot 10^{-6}$ .

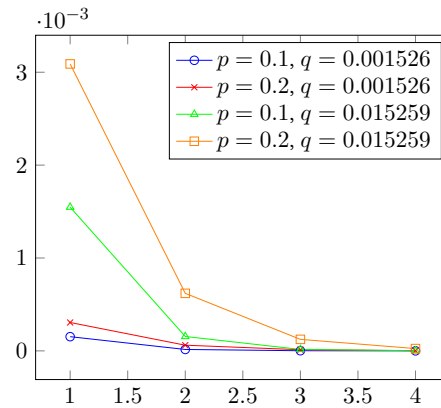


Figure 4.6: Collision probabilities as functions of the number of probes sent.

### 4.3.5 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 9** (Crowds). To demonstrate the applicability of the *cwp*-semantics to a practical example, we consider the Crowds protocol [60]. 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  $\text{pGCL}$ -program CROWDS in Figure 4.7 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*

```
1 // Let  $c$  be the fraction of corrupted nodes on the
2 // network and let  $p$  be the forward probability
3 intercepted := false;
4 delivered := false;
5 // Initiate communication path to server by sending
6 // the message 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 4.7: PGCL program of the Crowds protocol where the length of the communication path through the network is bounded by some fixed number  $k$ .

*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 *cwp* directly. The probability that a message is not intercepted given that it was rerouted no more than  $k$  times is given by

$$\underline{cwp}(\text{CROWDS}, [\neg \text{intercepted}]) = \frac{\text{wp}(\text{CROWDS}, [\neg \text{intercepted}])}{\text{wlp}(\text{CROWDS}, 1)} \quad (4.15)$$

The computation of this quantity requires to find fixed points.

**Detailed calculations** Let *init* be a short-hand notation for the program text in lines 3–8 and *loop* denote lines 9–18 in the program CROWDS in Figure 4.7. Further let *body* denote the program in the loop’s body in lines 10–17. For readability we abbreviate the variable names *delivered* as *del*, *counter* as *cnt* and *intercepted* as *int*. In the following we consider *del* and *int* as boolean variables assuming *true* = 1 and *false* = 0. In order to determine (4.15) we first start with the numerator. This quantity is given by

$$\text{wp}(\text{init}; \text{loop}; \text{observe}(cnt \leq k), [\neg \text{int}]) \quad (4.16)$$

$$= \text{wp}(\text{init}, \text{wp}(\text{loop}, [cnt \leq k \wedge \neg \text{int}])) \quad (4.17)$$

$$= \text{wp}(\text{init}, \text{lfp}_F([\neg \text{del}] \cdot \text{wp}(\text{body}, F) + [\text{del} \wedge cnt \leq k \wedge \neg \text{int}])) \quad (4.18)$$

$$= \text{wp}(\text{init}, \sup_n([\neg \text{del}] \cdot \text{wp}(\text{body}, 0) + [\text{del} \wedge cnt \leq k \wedge \neg \text{int}])^n) \quad (4.19)$$

where the supremum is taken over the range of the function

$$\Phi(f) = [\neg \text{del}] \cdot \text{wp}(\text{body}, f) + [\text{del} \wedge cnt \leq k \wedge \neg \text{int}]$$

and  $\Phi^n$  denotes the  $n$ -fold application of  $\Phi$ . Equation (4.17) is given directly by the semantics of sequential composition of pGCL commands. In the next line we apply the definition of loop semantics in terms of the least fixed point. Finally, (4.19) is given by the Kleene fixed point theorem as a solution to the fixed point equation in (4.18). We can explicitly find the supremum by considering the expression for several  $n$  and deducing a pattern.

$$\begin{aligned} \Phi(0) &= [\neg \text{del}] \cdot \text{wp}(\text{body}, 0) + [\text{del} \wedge cnt \leq k \wedge \neg \text{int}] \\ &= [\text{del} \wedge cnt \leq k \wedge \neg \text{int}] \end{aligned}$$

$$\begin{aligned}
\Phi^2(0) &= \Phi([del \wedge cnt \leq k \wedge \neg int]) \\
&= [\neg del] \cdot wp(body, [del \wedge cnt \leq k \wedge \neg int]) \\
&\quad + [del \wedge cnt \leq k \wedge \neg int] \\
&= [\neg del] \cdot (p(1-c) \cdot [del \wedge cnt + 1 \leq k \wedge \neg int] \\
&\quad + (1-p) \cdot [cnt \leq k \wedge \neg int]) \\
&\quad + [del \wedge cnt \leq k \wedge \neg int] \\
&= [\neg del \wedge cnt \leq k \wedge \neg int] \cdot (1-p) \\
&\quad + [del \wedge cnt \leq k \wedge \neg int]
\end{aligned}$$

$$\begin{aligned}
\Phi^3(0) &= \Phi([\neg del \wedge cnt \leq k \wedge \neg int] \cdot (1-p) \\
&\quad + [del \wedge cnt \leq k \wedge \neg int]) \\
&= \dots \\
&= [\neg del \wedge cnt \leq k \wedge \neg int] \cdot (1-p) \\
&\quad + [\neg del \wedge cnt + 1 \leq k \wedge \neg int] \cdot (1-p)p(1-c) \\
&\quad + [del \wedge cnt \leq k \wedge \neg int]
\end{aligned}$$

As we continue to compute  $\Phi^n(0)$  in the  $i$ -th step we add a summand of the form

$$[\neg del \wedge cnt + i \leq k \wedge \neg int] \cdot (1-p)(p(1-c))^i$$

However we see that the predicate evaluates to *false* for all  $i > k - cnt$ . Hence the non-zero part of the fixed point is given by

$$\begin{aligned}
&[del \wedge cnt \leq k \wedge \neg int] \\
&+ \sum_{i=0}^{k-cnt} [\neg del \wedge cnt + i \leq k \wedge \neg int] \cdot (1-p)(p(1-c))^i \\
&= [del \wedge cnt \leq k \wedge \neg int] \\
&\quad + [\neg del \wedge cnt \leq k \wedge \neg int] \cdot \sum_{i=0}^{k-cnt} (1-p)(p(1-c))^i \\
&= [del \wedge cnt \leq k \wedge \neg int] \\
&\quad + [\neg del \wedge cnt \leq k \wedge \neg int] \cdot (1-p) \frac{1 - (p(1-c))^{k-cnt+1}}{1 - p(1-c)} .
\end{aligned}$$

where for the last equation we use a property of the finite geometric series, namely that for  $r \neq 1$

$$\sum_{k=0}^{n-1} ar^k = a \frac{1 - r^n}{1 - r} .$$

The result coincides with the intuition that in a state where  $del = false$ , the probability to fail to reach the goal  $\neg int \wedge cnt \leq k$  is distributed geometrically with probability  $p(1 - c)$ . It is easy to verify that our educated guess is correct by checking that we indeed found a fixed point of  $\Phi$ :

$$\begin{aligned}
& \Phi([\mathit{del} \wedge \mathit{cnt} \leq k \wedge \neg \mathit{int}]) \\
& + [\neg \mathit{del} \wedge \mathit{cnt} \leq k \wedge \neg \mathit{int}] \cdot (1 - p) \frac{1 - (p(1 - c))^{k - \mathit{cnt} + 1}}{1 - p(1 - c)} \\
= & [\mathit{del} \wedge \mathit{cnt} \leq k \wedge \neg \mathit{int}] \\
& + [\neg \mathit{del}] \cdot \left( (1 - p) \cdot [\mathit{cnt} \leq k \wedge \neg \mathit{int}] \right. \\
& \quad \left. + p(1 - c) \left( [\mathit{del} \wedge \mathit{cnt} + 1 \leq k \wedge \neg \mathit{int}] \right. \right. \\
& \quad \quad \left. \left. + [\neg \mathit{del} \wedge \mathit{cnt} + 1 \leq k \wedge \neg \mathit{int}] \cdot (1 - p) \frac{1 - p(1 - c)^{k - \mathit{cnt}}}{1 - p(1 - c)} \right) \right) \\
= & [\mathit{del} \wedge \mathit{cnt} \leq k \wedge \neg \mathit{int}] \\
& + [\neg \mathit{del} \wedge \mathit{cnt} \leq k \wedge \neg \mathit{int}] \cdot (1 - p) \\
& + [\neg \mathit{del} \wedge \mathit{cnt} + 1 \leq k \wedge \neg \mathit{int}] \cdot (1 - p)(p(1 - c)) \frac{1 - p(1 - c)^{k - \mathit{cnt}}}{1 - p(1 - c)} \\
= & [\mathit{del} \wedge \mathit{cnt} \leq k \wedge \neg \mathit{int}] \\
& + [\neg \mathit{del} \wedge \mathit{cnt} = k \wedge \neg \mathit{int}] \cdot (1 - p) \\
& + [\neg \mathit{del} \wedge \mathit{cnt} + 1 \leq k \wedge \neg \mathit{int}] \cdot (1 - p) \\
& + [\neg \mathit{del} \wedge \mathit{cnt} + 1 \leq k \wedge \neg \mathit{int}] \cdot (1 - p)(p(1 - c)) \frac{1 - p(1 - c)^{k - \mathit{cnt}}}{1 - p(1 - c)} \\
= & [\mathit{del} \wedge \mathit{cnt} \leq k \wedge \neg \mathit{int}] \\
& + [\neg \mathit{del} \wedge \mathit{cnt} = k \wedge \neg \mathit{int}] \cdot (1 - p) \\
& + [\neg \mathit{del} \wedge \mathit{cnt} + 1 \leq k \wedge \neg \mathit{int}] \\
& \cdot (1 - p) \frac{1 - p(1 - c) + (p(1 - c))(1 - p(1 - c))^{k - \mathit{cnt}}}{1 - p(1 - c)} \\
= & [\mathit{del} \wedge \mathit{cnt} \leq k \wedge \neg \mathit{int}] \\
& + [\neg \mathit{del} \wedge \mathit{cnt} \leq k \wedge \neg \mathit{int}] \cdot (1 - p) \frac{1 - p(1 - c)^{k - \mathit{cnt} + 1}}{1 - p(1 - c)}
\end{aligned}$$

Since the loop terminates almost surely it has only one fixed point [52] which is trivially also the least. We can now continue our calculation from (4.19).

$$\begin{aligned}
& wp(\text{init}, \sup_n ([\neg del] \cdot wp(\text{body}, 0) + [del \wedge cnt \leq k \wedge \neg int]^n)) \\
&= wp(\text{init}, [del \wedge cnt \leq k \wedge \neg int] \\
&\quad + [\neg del \wedge cnt \leq k \wedge \neg int] \\
&\quad \cdot (1-p) \frac{1 - (p(1-c))^{k-cnt+1}}{1 - p(1-c)})
\end{aligned} \tag{4.20}$$

$$= (1-c)(1-p) \frac{1 - (p(1-c))^k}{1 - p(1-c)} . \tag{4.21}$$

This concludes the calculation of the numerator of (4.15). Analogously we find the denominator

$$\begin{aligned}
& wlp(\text{init}, \text{loop}; \text{observe}(cnt \leq k), 1) \\
&= wlp(\text{init}, wlp[\text{loop}]([cnt \leq k])) \\
&= wlp(\text{init}, \text{gfp}_F ([\neg del] \cdot wlp(\text{body}, F) \\
&\quad + [del \wedge cnt \leq k])) \\
&= wlp(\text{init}, \sup_n ([\neg del] \cdot wlp(\text{body}, 1) \\
&\quad + [del \wedge cnt \leq k]^n)) \\
&= wlp(\text{init}, [del \wedge cnt \leq k] \\
&\quad + [\neg del \wedge cnt \leq k] \cdot (1 - p^{k-cnt+1})) \\
&= 1 - p^k .
\end{aligned} \tag{4.22}$$

The only difference is that here the supremum is taken with respect to the reversed order  $\geq$  in which 1 is the bottom and 0 is the top element. However as mentioned earlier *loop* terminates with probability one and the notions of *wp* and *wlp* coincide. We divide (4.21) by (4.22) to finally arrive at a closed form solution parameterised in  $p$ ,  $c$ , and  $k$ :

$$\begin{aligned}
& \underline{cwp}(\text{CROWDS}, [\neg \text{intercepted}]) \\
&= (1-c)(1-p) \frac{1 - (p(1-c))^k}{1 - p(1-c)} \cdot \frac{1}{1 - p^k} .
\end{aligned}$$

One can visualise it as a function in  $k$  by fixing the parameters  $c$  and  $p$ . For example, Figure 4.8 shows the conditional probability plotted for various parameter settings. The automation of this analysis requires to find the fixed points in (4.15) automatically. This issue is addressed in the next chapter. ■

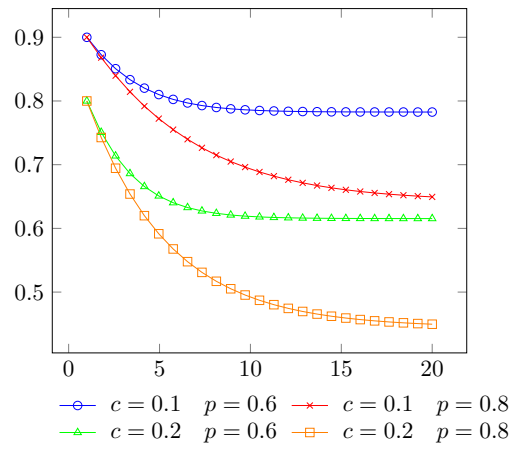


Figure 4.8: The conditional probability that a message is intercepted as a function of  $k$  for fixed  $c$  and  $p$ .



**Part II**

**Verification**



## Chapter 5

# 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  $\text{pGCL}$  programs as defined in Chapter 3 *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 a technique for fixed point calculation is discussed. Then we summarise how invariants are used as an alternative means to reason about pre-expectations of loops. Subsequently, we show one approach for the analysis of probabilistic programs using invariants and discuss to what extent our software tool PRINSYS helps the user in this process.

### 5.1 Proving properties of probabilistic programs

#### 5.1.1 Computing fixed points

In Chapter 2, page 28 we gave a detailed example how a fixed point can be found by hand. Now we are interested in automated techniques which may be implemented on a computer. 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 prob-

abilistic 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

$$[\textit{flip} = 0] \cdot (a_2 \cdot \textit{flip} + a_1 \cdot x + a_0) + [\textit{flip} \neq 0] \cdot (b_2 \cdot \textit{flip} + b_1 \cdot x + b_0) \quad (5.1)$$

and after some iterations converge to

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

The template (5.1) is instantiated with the values above and yields the fixed point of the loop. 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. [18]. 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 2.4.1) 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 [18], 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 (2.5), page 31 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 mathematical formulas.

### 5.1.2 Invariants

Before we elaborate more on invariants for probabilistic programs, let us consider invariants for standard, non-probabilistic programs and how they are used.

**Definition 28** (Standard loop invariant). A predicate  $\mathcal{I}$  is called a *loop invariant* for a loop  $\text{while}(G) \{P\}$  if

$$\mathcal{I} \wedge G \Rightarrow \text{wlp}(P, \mathcal{I}) . \quad (5.2)$$

■

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 (5.2) 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). Often this is emphasised by saying that  $\mathcal{I}$  is an *inductive* invariant. Colloquially, we may say that “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:

$$\mathcal{I} \Rightarrow \text{wlp}(\text{while}(G) \{P\}, \mathcal{I} \wedge \neg G) .$$

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*:

$$\mathcal{I} \Rightarrow \text{wp}(\text{while}(G) \{P\}, \mathcal{I} \wedge \neg G) .$$

In practice one usually wants to prove that given some precondition  $\psi$  at the beginning of the loop, the postcondition  $\varphi$  will hold after the loop's execution.

The straightforward way is to show this by directly applying  $wp$  semantics, i.e. proving

$$\psi \Rightarrow wp(\text{while}(G) \{P\}, \varphi) .$$

But it turns out to be hard because this requires to find the least fixed point of the loop with respect to  $\varphi$ . Instead it may be easier to

1. find a predicate  $\mathcal{I}$  such that

$$\psi \Rightarrow \mathcal{I} \quad \text{and} \quad \mathcal{I} \wedge \neg G \Rightarrow \varphi ,$$

2. show  $\mathcal{I}$  is invariant for the loop  $\text{while}(G) \{P\}$ , cf. Definition 28 and
3. prove that the loop terminates from any state in  $\mathcal{I} \wedge G$ .

Via this detour the same relation between  $\psi$  and  $\varphi$  is established as

$$\psi \Rightarrow \mathcal{I} \Rightarrow wp(\text{while}(G) \{P\}, \mathcal{I} \wedge \neg G) \quad \text{and} \quad \mathcal{I} \wedge \neg G \Rightarrow \varphi .$$

Let us return to probabilistic programs. McIver and Morgan [52, Ch. 2] have generalised the terms of partial and total correctness to probabilistic programs. If, for a probabilistic program  $P$  and expectations  $g$  and  $f$ , we can establish

$$g = wp(\text{while}(G) \{P\}, f) , \tag{5.3}$$

then we have shown the total correctness of  $\text{while}(G) \{P\}$  with respect to post-expectation  $f$ . For an initial valuation  $\eta$  of program variables,  $g(\eta)$  is the expectation of  $f$  computed over all terminating runs of  $\text{while}(G) \{P\}$ . In this sense,  $wp$  takes care of both, probabilistic termination and expectation calculation. As was the case for standard programs, we would like to separate concerns and prove partial correctness by means of a probabilistic invariant independently of proving (almost sure) termination. Therefore loop invariants are introduced for probabilistic programs.

**Definition 29** (Probabilistic loop invariant). An expectation  $\mathcal{I}$  is called a *probabilistic loop invariant* for  $\text{while}(G) \{P\}$  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 (cf. Definition 28) except that our invariant is an expectation and not a predicate and thus implication between predicates is lifted to an inequality between expectations. From now on we refer to probabilistic

loop invariants simply as invariants for readability. Previously, we have used the probabilistic version of the  $wlp$  transformer to measure the probability to avoid a bad event. Hence it was naturally bound to be in the range  $[0, 1]$ . As discussed in Remark 7, page 57, in principle the range can be extended to  $[0, \alpha]$  for some  $\alpha \in \mathbb{R}_{\geq 0}$  provided that  $\alpha$  is an upper bound on the pre- and post-expectations. Fortunately, in practice the programs which we study allow to circumvent this issue. All programs examined in the rest of the chapter will terminate almost surely. As have been pointed out before, it is known that for almost surely terminating programs  $wp$  and  $wlp$  are identical. So for the rest of this chapter we may write  $wlp$  to follow the notation of the above definition or to stress certain statements but in fact when it comes to the practical analysis of programs,  $wp$  and  $wlp$  become interchangeable. In this way we save ourselves from discussing the meaning and range of  $wlp$  for each and every program.

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 instances of martingales [68]. 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 29 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].

In summary, given a probabilistic loop  $while(G) \{P\}$ , a pre-expectation  $g$  and a post-expectation  $f$ , the goal is to establish

$$g \leq wp(while(G) \{P\}, f) .$$

Instead of computing the fixed-point of the loop we divide the problem into sub-tasks:

1. find an expectation  $\mathcal{I}$  such that

$$g \leq \mathcal{I} \quad \text{and} \quad \mathcal{I} \wedge \neg G \leq f ,$$

2. show  $\mathcal{I}$  is invariant for the loop  $while(G) \{P\}$ , cf. Definition 29,
3. show  $\mathcal{I}$  is sound, that is  $\mathcal{I} \leq wp(while(G) \{P\}, [\neg G] \cdot \mathcal{I})$  .

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  $P$  is a loop-free program (as assumed above). In order to guarantee soundness (point 3.) the loop must terminate with probability 1 and the invariant  $\mathcal{I}$  has to additionally meet one of the following sufficient conditions [52, pp. 71–72]:

- show that from every initial state of the loop only a finite state space is reachable
- or show 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.

It is an open problem to give the *necessary and sufficient* conditions for soundness. Following the steps above we have established the desired relationship between  $g$  and  $f$  because

$$g \leq \mathcal{I} \leq wp(\text{while}(G) \{P\}, [-G] \cdot \mathcal{I}) \leq^1 wp(\text{while}(G) \{P\}, f) . \quad (5.4)$$

From the line above we see another crucial difference between standard and probabilistic invariants, namely 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. In practice we often want to find the pre-expectation of a loop precisely, not just some under-approximation. For this we need to establish

$$g = \mathcal{I} \quad \text{and} \quad [-G] \cdot \mathcal{I} = f$$

and we require an *exact* invariant [52, pp. 67–68] which satisfies

$$\mathcal{I} = wp(P, \mathcal{I}) . \quad (5.5)$$

Indeed in all our examples throughout this chapter the invariants are chosen such that they establish the pre-expectation exactly.

**Example 10** (Application of invariants). Consider the program *prog* in Figure 5.1. 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

$$wp(\text{prog}, [x = 1]) = 0.7 .$$

---

<sup>1</sup> $wp$  is monotonic in its second argument [52].

```

1  x := 0;
2  while (x = 0) {
3    {x := 0} [0.5] {{x := -1} [0.3] {x := 1}}
4  }

```

Figure 5.1: A simple loop

Instead of computing the least fixed point of the loop with respect to post-expectation  $[x = 1]$ , we can show that  $\mathcal{I} = [x = 0] \cdot 0.7 + [x = 1]$  is an exact invariant. If the loop terminates, we can establish:

$$[\neg G] \cdot \mathcal{I} = [x \neq 0] \cdot [x = 0] \cdot 0.7 + [x = 1] = [x = 1] .$$

At the beginning of the program the initialisation of  $x$  transforms the invariant to:

$$wp(x := 0, \mathcal{I}) = [0 = 0] \cdot 0.7 + [0 = 1] = 0.7 .$$

In this way we have shown the claim

$$wp(\text{prog}, [x = 1]) = 0.7$$

as desired. It is sound because the program obviously terminates with probability 1 and  $\mathcal{I}$  is bounded. ■

There is an intricate difference between non-probabilistic and probabilistic programs that may lead to unsound reasoning with invariants. For standard programs it is sufficient to show the termination of the loop in order to apply invariants. Probabilistic programs additionally require checking the extra soundness criteria mentioned above and here we would like to elaborate on the reason by means of an example. 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 different. Consider the example in Figure 5.2. This loop will terminate with probability one 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 Definition 29. Hence,

$$n \leq wlp(\text{while}(n \neq 0) \{ \{n := n - 1\} [0.5] \{n := n + 1\} \}, n \cdot [n = 0]) .$$

```

1  n := 1;
2  while (n != 0) {
3    (n := n - 1 [0.5] n := n + 1);
4  }

```

Figure 5.2: Symmetric random walk over  $\mathbb{N}$  with absorbing barrier at zero [52].

And we have already shown that the loop terminates almost surely. We could therefore falsely conclude that

$$n \leq wp(\text{while}(n \neq 0) \{\{n := n - 1\} [0.5] \{n := n + 1\}\}, n \cdot [n = 0]) .$$

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 would be 1. 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 with respect to  $n$ . The result is 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.

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

$$wp((\text{while}(G) \{P\})^n, e) \geq 0 \quad \forall n \in \mathbb{N} \quad (5.6)$$

where  $(\text{while}(G) \{P\})^n$  is the  $n$ -fold unrolling of the loop and they assume  $(\text{while}(G) \{P\})^0 = \text{skip}$ . 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

$$wp(\text{while}(G) \{P\}, 2x - 1) \geq 0$$

and since  $wp$  is linear, we can equivalently write

$$\begin{aligned} &\Leftrightarrow 2 \cdot wp(\text{while}(G) \{P\}, x) - 1 \geq 0 \\ &\Leftrightarrow wp(\text{while}(G) \{P\}, x) \geq 0.5 . \end{aligned}$$

In this way a lower bound on the expectation of  $x$  can be derived from  $e$ . 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 29 relate to IEI  $e$  in (5.6)? 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 (5.6) 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 no less than 0). So in [14] they generate expressions  $e$  that in our framework satisfy

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

where  $\eta_0$  is the initial state from which the loop starts its execution<sup>2</sup>. 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.

## 5.2 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 on techniques and their implementations we refer to published results on standard [61] and probabilistic [14] programs.

---

<sup>2</sup>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  $\eta_0$ .

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

$$[G] \cdot \mathcal{I} \leq \text{wlp}(P, \mathcal{I}) \quad (5.7)$$

$$[\neg G] \cdot \mathcal{I} \leq f \quad (5.8)$$

where  $\mathcal{I}$  is the invariant to be determined. In principle,  $\mathcal{I}$  may be an arbitrary 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 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 in which factors or additive constants may be parameterised. We have seen an example of a template earlier, cf. (5.1), page 102. 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 an invariant that satisfies (5.8). 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 the user has to provide a lot of information before the computer takes over. For standard programs, implementations exist, e.g. [36, 64, 20].

In the following section we discuss our implementation of a constraint based method due to [43] and evaluate its applicability on some examples. Its key benefit is that it checks (5.7) automatically and thereby saves the user the hardest calculations that are needed to establish pre- and post-expectations of probabilistic loops.

### 5.3 Prinsys

PRINSYS implements an invariant generation method suggested by Katoen et al. [43]. It is available for download on

<http://moves.rwth-aachen.de/research/tools/prinsys/> .

In the following section a detailed explanation of PRINSYS’s methodology and implementation is given. Thereafter we proceed to discuss other programs which were studied in the scope of this work.

### 5.3.1 Methodology

Here we reconsider our example program from Figure 2.3, page 29, which generates a geometrically distributed variable, and work out all steps which the tool performs to find the desired invariant. Essentially the program consists of the loop

$$\text{while } (flip = 0) \{ \{ flip := 1 \} [p] \{ x := x + 1 \} \} .$$

**Template.** To begin with, we need a template. This is a possibly parameterised expectation that defines the “shape” of our invariant. For this example we choose

$$T_\alpha = [x \geq 0] \cdot x + [x \geq 0 \wedge flip = 0] \cdot \alpha$$

where  $\alpha$  is an unknown (real) parameter. The intuition behind this template is that the average value of  $x$  is represented by the unknown  $\alpha$ . Before the execution of the loop,  $x$  is zero and  $flip$  is zero and at this point  $T_\alpha$  evaluates to  $\alpha$ . Then with every further iteration of the loop the probability that  $flip$  remains zero decreases while at the same time the value of  $x$  increases. Thus the average value of  $T_\alpha$  remains constant. Eventually the loop terminates and  $T_\alpha$  equals the post-expectation  $x$ .

**Goal.** Replacing  $\alpha$  by a real value yields an *instance* of the template. Depending on this value, some instances may or may not satisfy the invariance condition

$$[G] \cdot T_\alpha \leq \text{wlp}(P, T_\alpha) . \quad (5.9)$$

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,  $\alpha$  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:

$$\begin{aligned} & \text{wlp}(\{ flip := 1 \} [p] \{ x := x + 1 \}, T_\alpha) \\ = & [x \geq 0] \cdot px + (1 - p) \cdot ([x + 1 \geq 0] \cdot (x + 1) + [x + 1 \geq 0 \wedge flip = 0] \cdot \alpha) . \end{aligned}$$

After expanding this expression, the invariance condition amounts to:

$$\overbrace{[x \geq 0 \wedge \mathit{flip} = 0] \cdot (x + \alpha)}^{[G] \cdot T_\alpha} \leq \underbrace{[x \geq 0] \cdot px + [x + 1 \geq 0] \cdot ((1 - p)x - p + 1) + [x + 1 \geq 0 \wedge \mathit{flip} = 0] \cdot (1 - p)\alpha}_{wlp(P, T_\alpha)} .$$

Our goal is to find all  $\alpha$  such that the point-wise inequality is satisfied, i.e., it holds for every value of  $x$  and  $\mathit{flip}$ . 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 8** (Transformation to DNF [43]). Given an expectation of the form

$$f = [P_1] \cdot w_1 + \dots + [P_n] \cdot w_n .$$

Then an equivalent expectation in DNF can be written as:

$$\sum_{I \in \mathcal{P}(\bar{n}) \setminus \emptyset} \left( \left[ \bigwedge_{i \in I} P_i \wedge \neg \left( \bigwedge_{j \in \bar{n} \setminus I} P_j \right) \right] \cdot \left( \sum_{i \in I} w_i \right) \right)$$

where  $\bar{n}$  is the index set  $\{1, \dots, n\}$  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 8 summands. For better readability we only show the summands that are not trivially zero:

$$\begin{aligned} & [x + 1 \geq 0 \wedge x < 0 \wedge \mathit{flip} = 0] \cdot ((1 - p)x + (1 - p)\alpha - p + 1) \\ & + [x \geq 0 \wedge \mathit{flip} = 0] \cdot (x + (1 - p)\alpha - p + 1) \\ & + [x + 1 \geq 0 \wedge x < 0 \wedge \mathit{flip} \neq 0] \cdot ((1 - p)x - p + 1) \\ & + [x \geq 0 \wedge \mathit{flip} \neq 0] \cdot (x - p + 1) . \end{aligned}$$

At this point the invariance condition is still given as an inequality between real-valued functions. In a further step we encode this as a decision problem by means of the following theorem.

**Theorem 9.** Given two expectations over variables  $x_1, \dots, x_n$  in disjoint-normal form

$$f = [P_1] \cdot u_1 + \dots + [P_M] \cdot u_M, \quad g = [Q_1] \cdot w_1 + \dots + [Q_K] \cdot w_K.$$

The inequality  $f \leq g$  holds if and only if

$$\begin{aligned} \forall x_1, \dots, x_n \in \mathbb{R} : & \bigwedge_{m \in \overline{M}} \bigwedge_{k \in \overline{K}} (P_m \wedge Q_k \Rightarrow (u_m - w_k \leq 0)) \\ & \wedge \bigwedge_{m \in \overline{M}} \left( P_m \wedge \left( \bigwedge_{k \in \overline{K}} \neg Q_k \right) \Rightarrow u_m \leq 0 \right) \\ & \wedge \bigwedge_{k \in \overline{K}} \left( Q_k \wedge \left( \bigwedge_{m \in \overline{M}} \neg P_m \right) \Rightarrow 0 \leq w_k \right) \end{aligned}$$

holds, where  $\overline{X}$  is the set of indices  $\{1, 2, \dots, X\}$ . ■

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 9 originally appears in [43] 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:

$$\begin{aligned} \forall x, flip : & (\alpha p + p - 1 \leq 0 \vee flip \neq 0 \vee x < 0) \\ & \wedge (\alpha p - \alpha + px + p - x - 1 \leq 0 \vee flip \neq 0 \vee x + 1 < 0 \vee x \geq 0) \\ & \wedge (flip = 0 \vee px + p - x - 1 \leq 0 \vee x + 1 < 0 \vee x \geq 0) \\ & \wedge (flip = 0 \vee p - x - 1 \leq 0 \vee x < 0). \end{aligned}$$

The calculation of this formula by PRINSYS concludes the first stage.

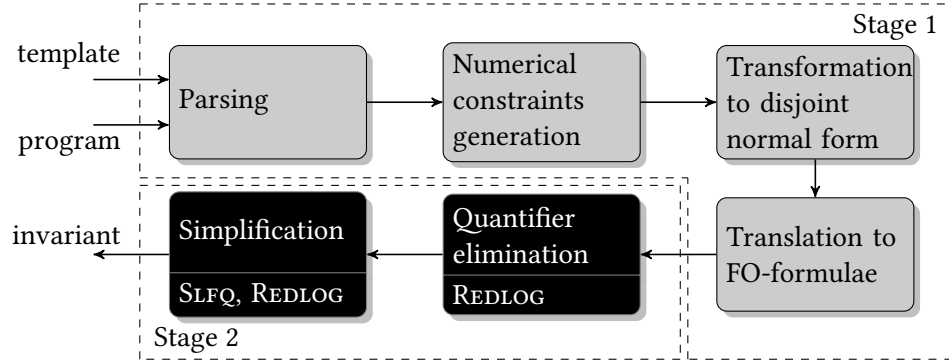


Figure 5.3: Tool chain workflow

Stage 2: The formula is passed to REDLOG which simplifies the formula using 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  $\alpha$ s that make  $T_\alpha$  invariant:

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

This simplifies to

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

We pick the greatest admissible  $\alpha$  and obtain an invariant:

$$T_{\frac{1-p}{p}} = [x \geq 0] \cdot x + [x \geq 0 \wedge flip = 0] \cdot \frac{1-p}{p} .$$

This is a stronger expression than the fixed point in (2.5), page 31 but it suffices to show that the geometric distribution program has an average outcome of  $\frac{1-p}{p}$  which indeed is the mean of a geometric distribution with parameter  $p$ . The soundness of our invariant is ensured because the loop terminates almost surely and in every iteration there is a non-zero probability to exit the loop.

Figure 5.3 summarises the discussed workflow of PRINSYS.

**New insights.** There are major differences with the approach sketched in [43]. 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 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.

If a parameter-free template  $\mathcal{T}$  is used, then PRINSYS will simply report whether the inequality (5.9) is satisfied or not. For such invariance checks, we recently have added an option to use Z3 [22] 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 (5.9). 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 (5.9) does not hold. This may serve as valuable information to the user to refine his guess of what  $\mathcal{T}$  should be.

### 5.3.2 Examples

In the following we discuss several examples in which we were able to successfully identify an invariant with the help of PRINSYS.

**Martingale betting strategy.** Another variant of the geometric distribution appears in the program in Figure 5.4, 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 expectation differs from what we have computed for the geometric distribution program in the previous example because here the counter is increased also on the last iteration before the loop terminates.

```

1  c := IC; // capital c (is set to some InitialCapital)
2  b := 1; // initially bet one unit
3  rounds := 0; // number of rounds played (survived)
4  while (b > 0) {
5      // win with probability p
6      c := c+b;
7      b := 0;}
8      [p]
9      // lose with probability 1-p
10     c := c-b;
11     b := 2*b;}
12     rounds := rounds+1;
13 }

```

Figure 5.4: The martingale betting strategy doubles the stake until the gambler wins once.

**Program equivalence.** This example is taken from [45] where amongst others it has been shown that the two programs in Figure 5.5a and Figure 5.5b are equivalent for  $p = \frac{1}{2}$  and  $q = \frac{2}{3}$ . The proof in [45] 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 Figure 5.5a and Figure 5.5b 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.

The PRINSYS tool generates invariants for single loops, so we consider each loop separately. Using the template  $T_\alpha = [x \geq 0] \cdot x + [x \geq 0 \wedge \text{flip} = 0] \cdot \alpha$  from our running example, PRINSYS yields the following invariants:

- $\mathcal{I}_{11} = x + [\text{flip} = 0] \cdot \frac{p}{1-p},$
- $\mathcal{I}_{12} = x + [\text{flip} = 0] \cdot \left(-\frac{q}{1-q}\right),$

```

1  x := 0;
2  flip := 0;
3  while (flip = 0) {
4    (x := x+1 [p] flip := 1);
5  }
6  flip := 0;
7  while (flip = 0) {
8    (x := x-1 [q] flip := 1);
9  }

1 x := 0;
2 (flip := 0 [0.5] flip := 1);
3 if (flip = 0) {
4   while (flip = 0) {
5     (x := x+1 [p] flip := 1);
6   }
7 } else {
8   flip := 0;
9   while (flip = 0) {
10    x := x-1;
11    (skip [q] flip := 1);
12  }
13 }

```

(a) Two loops in sequence.

(b) Choice between two loops.

Figure 5.5: Different implementations that may produce the same average outcome.

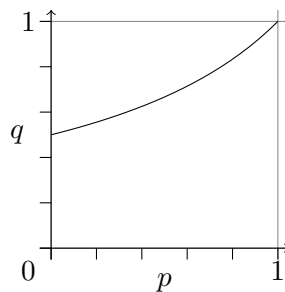


Figure 5.6: Pairs  $(p, q)$  for which the programs in Figure 5.5a and Figure 5.5b produce the same  $x$  on average.

```

1  x := 0; // stores outcome of first biased coin flip
2  y := 0; // stores outcome of second biased coin flip

4  while (x-y = 0) {
5      (x := 0 [p] x := 1);
6      (y := 0 [p] y := 1);
7  }
```

Figure 5.7: Algorithm which generates  $x = 0$  and  $x = 1$  with equal probability by repeatedly flipping a coin with an arbitrary bias  $p$ .

- $\mathcal{I}_{21} = \mathcal{I}_{11}$  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 Figure 5.5a and Figure 5.5b, 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 [45]. Figure 5.6 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 [45]; to the best of our knowledge there are no other automated techniques that can establish this.

**Generating a fair coin from a biased coin.** Using a coin with some arbitrary bias  $0 < p < 1$ , the algorithm in Figure 5.7 generates a sample according to a fair coin flip. The loop terminates when the biased coin is flipped twice and shows 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:

$$[x = 0 \wedge y = 1 = 0] \cdot (\alpha) + [x = 1 \wedge y = 0] \cdot (\beta)$$

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

$[x = 0 \wedge y - 1 = 0] - [x - 1 = 0 \wedge y = 0]$  is invariant<sup>3</sup> which, together with almost sure termination, gives us

$$\begin{aligned} & wp(\text{prog}, [x = 0 \wedge y - 1 = 0] - [x - 1 = 0 \wedge y = 0]) \\ &= wp(\text{prog}, [x = 0 \wedge y - 1 = 0]) - wp(\text{prog}, [x - 1 = 0 \wedge y = 0]) \\ &= 0 . \end{aligned} \tag{5.10}$$

where *prog* is the entire program from Figure 5.7. The previous argument about almost sure termination and possible outcomes shows that

$$\begin{aligned} & wp(\text{prog}, [x = 0 \wedge y - 1 = 0] + [x - 1 = 0 \wedge y = 0]) \\ &= wp(\text{prog}, [x = 0 \wedge y - 1 = 0]) + wp(\text{prog}, [x - 1 = 0 \wedge y = 0]) \\ &= 1 . \end{aligned} \tag{5.11}$$

The unique solution to (5.10) and (5.11) is

$$\begin{aligned} & wp(\text{prog}, [x = 0 \wedge y - 1 = 0]) \\ &= wp(\text{prog}, [x - 1 = 0 \wedge y = 0]) \\ &= 0.5 . \end{aligned}$$

This concludes the proof that  $x$  is distributed evenly for any  $p$  satisfying  $0 < p < 1$ .

**Generating a biased coin from a fair one.** In [43], 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 5.8. With PRINSYS we have successfully verified that

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

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.

**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 5.9

---

<sup>3</sup>We pick  $\alpha = 1$  and  $\beta = -1$  but in fact any non-zero pair of values  $\alpha = -\beta$  would result in the same argument.

```

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

```

Figure 5.9: Algorithm which generates a sample  $x$  distributed binomially with parameters  $p$  and  $M$ .

gives a  $\text{rGCL}$  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

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

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

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

This invariant allows to show that the expectation of  $x/M$  is  $p$ . Since  $w\mathcal{P}$  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 [30, p. 45ff.].

### 5.3.3 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 [59], this transformation seems impractical since parts of the program's structure are 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.

```

while (G) {
  P;
  while (H) {
    R
  }
  Q;
}

```

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 (5.7) and (5.8) generalise to

$$[G] \cdot \mathcal{I} \leq wlp(P, \mathcal{J}) \quad (5.12)$$

$$[H] \cdot \mathcal{J} \leq wlp(R, \mathcal{J}) \quad (5.13)$$

$$[\neg H] \cdot \mathcal{J} \leq wlp(Q, \mathcal{I}) \quad (5.14)$$

$$[\neg G] \cdot \mathcal{I} \leq f \quad (5.15)$$

Condition (5.13) ensures that  $\mathcal{J}$  is an invariant of the inner loop. Together with conditions (5.12) and (5.14) this further ensures that  $\mathcal{I}$  is an invariant of the outer loop. Finally (5.15) establishes a lower bound on the given post-expectation  $f$  upon termination of the loops—in the same way as (5.8) did before. These constraints can be encoded in a 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 (5.9) 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. Therefore strategies to simplify the output need to be devised.

**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 [51].

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 (cf. Definition 29). 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.



## Chapter 6

# Conclusion and future work

In this thesis we have studied the semantics and analysis techniques for probabilistic programs. We have given  $\text{pGCL}$  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 has proven itself to be fundamental for the understanding of the behaviour of probabilistic programs. It furthermore opened up a second option to reason about properties of  $\text{pGCL}$  programs. By means of an example we have shown that the liberal expected reward may not be achieved by a particular minimising scheduler. However for the expected reward, which corresponds to  $wp$ , it seems likely that the infimum can be substituted by a minimum and that in fact a minimising scheduler may be implemented. However to the best of our knowledge this result has only been proven for finite state RMDPs [54, 3]. The proof for infinite state RMDPs remains an intriguing objective for further research. Looking further at the relationship of operational and denotation semantics opens up more research problems. Currently we have not considered *recursion* in  $\text{pGCL}$ . Moreover, McIver and Morgan [52] have studied an abstraction and refinement notion for  $\text{pGCL}$  programs on the level of  $wp$  semantics. It would be interesting to recover these notions within our operational semantics—mostly likely as a variant of a (bi-)simulation relation on MDPs [65].

Our transfer theorem also guided our way when we enriched  $\text{pGCL}$  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 feature. This gave rise to the notion of conditional minimal expected rewards over RMDPs and the conditional 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 *cwp* style semantics have been left for future work. Our treatment of semantics for conditional probabilistic programs was concluded with a discussion of interesting program transformations that allow to remove conditioning. In that context we defined so called “iid loops” and it would be interesting to automate their detection and subsequently their analysis. On top of that, the observation hoisting could be implemented inside our tool PRINSYS as this transformation relies on probabilistic invariants.

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. Here we see a lot of potential for further research. Technically, PRINSYS needs to be tuned towards better performance. As we have mentioned before, the expressions’ size explodes particularly due to the DNF transformation. A clever way needs to be found that allows to encode the invariance condition as a first-order formula without first blowing it up and then subsequently removing all unsatisfiable subexpressions. Additionally we have seen that the constraints which are returned by REDLOG contain redundant subformulas. These may render the output unreadable for the user and hence simplification strategies are needed that would allow to rewrite the returned constraint to an equivalent one of considerably smaller size. Very recently progress on the constraint-based invariant generation for probabilistic programs has been made [15]. PRINSYS provides a good framework to evaluate the new technique on various examples and hence its implementation inside PRINSYS seems desirable. Finally, we have discussed the possibility of developing patterns for invariants. There the lack of a larger set of case studies has been pointed out as the main obstacle. In fact, a diversified set of examples would improve our overall tool development. The relatively new research area of probabilistic program verification currently does not have a comprehensive benchmark set and thus more example programs need to be found. 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  $\text{rGCL}$  programs.



# Bibliography

- [1] Monty hall problem. [http://en.wikipedia.org/wiki/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] Tomáš 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] Yu-Fang Chen, Chih-Duo Hong, Bow-Yaw Wang, and Lijun Zhang. Counterexample-guided polynomial loop invariant generation by Lagrange

- interpolation. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 658–674. Springer, 2015.
- [16] S. Cheshire, B. Aboba, and E. Guttman. Dynamic configuration of ipv4 link-local addresses, 2005.
- [17] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137 – 167, 1959.
- [18] 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, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 92–102. ACM, 2013.
- [19] 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.
- [20] 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.
- [21] Adnan Darwiche. Bayesian networks. *Commun. ACM*, 53(12):80–90, 2010.
- [22] 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.
- [23] 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.

- [24] JI den Hartog. *Probabilistic Extensions of Semantical Models*. PhD thesis, Vrije Universiteit Amsterdam, 2002.
- [25] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
- [26] William Feller. *An Introduction to Probability Theory and its Applications*, volume 1 of *Wiley mathematical statistics series*. Wiley, 1966.
- [27] Robert W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.
- [28] 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.
- [29] 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.
- [30] Friedrich Gretz. Invariant Generation for Linear Probabilistic Programs. Master’s thesis, RWTH Aachen, 2010. <http://moves.rwth-aachen.de/people/fgretz/>.
- [31] Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Annabelle McIver, and Federico Olmedo. Conditioning in probabilistic programming. *To appear in Mathematical Foundations of Programming Semantics 2015*.
- [32] Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Annabelle McIver, and Federico Olmedo. Conditioning in probabilistic programming. *CoRR*, abs/1504.00198, 2015.
- [33] 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.

- [34] 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.
- [35] 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.
- [36] 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.
- [37] 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.
- [38] 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.
- [39] 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.
- [40] 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.
- [41] Joe Hurd, Annabelle McIver, and Carroll Morgan. Probabilistic guarded commands mechanized in HOL. *Theor. Comput. Sci.*, 346(1):96–112, 2005.

- [42] 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.
- [43] 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.
- [44] 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.
- [45] Stefan Kiefer, Andrzej Murawski, Joel Ouaknine, Bjoern Wachter, and James Worrell. On the complexity of the equivalence problem for probabilistic automata. In *FoSSaCS*, volume 7213 of *LNCS*, pages 467–481, 2012.
- [46] 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.
- [47] 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.
- [48] Johan J. Lukkien. Operational semantics and generalized weakest preconditions. *Sci. Comput. Program.*, 22(1-2):137–155, 1994.
- [49] Jérémie Lumbroso. Optimal discrete uniform generation from coin flips, and applications. *CoRR*, abs/1304.1916, 2013.
- [50] 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.
- [51] 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.

- [52] Annabelle McIver and Carroll Morgan. *Abstraction, Refinement And Proof For Probabilistic Systems (Monographs in Computer Science)*. Springer Verlag, 2004.
- [53] 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.
- [54] Carroll Morgan and Annabelle McIver. Memoryless strategies for stochastic games via domain theory. *Electr. Notes Theor. Comput. Sci.*, 130:23–37, 2005.
- [55] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
- [56] 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.
- [57] 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.
- [58] Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.
- [59] T. M. Rabejaha 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.
- [60] Michael K. Reiter and Aviel D. Rubin. Crowds: Anonymity for web transactions. *ACM Trans. Inf. Syst. Secur.*, 1(1):66–92, 1998.
- [61] 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.
- [62] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

- [63] Adrian Sampson, Pavel Panekha, Todd Mytkowicz, Kathryn S. McKinley, Dan Grossman, and Luis Ceze. Expressing and verifying probabilistic assertions. 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 14. ACM, 2014.
- [64] 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.
- [65] Roberto Segala and Nancy A. Lynch. Probabilistic simulations for probabilistic processes. *Nord. J. Comput.*, 2(2):250–273, 1995.
- [66] Henk C. Tijms. *Understanding Probability: Chance Rules in Everyday Life*. Cambridge University Press, 2004.
- [67] George H. Weiss. Random walks and their applications. *American Scientist*, 71(1):pp. 65–71, 1983.
- [68] David Williams. *Probability With Martingales*. Cambridge University Press, 1991.
- [69] Mingsheng Ying, Nengkun Yu, Yuan Feng, and Runyao Duan. Verification of quantum programs. *Sci. Comput. Program.*, 78(9):1679–1700, 2013.