ADVANCED WEAKEST PRECONDITION CALCULI 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

BENJAMIN LUCIEN KAMINSKI, M.SC.

aus

BONN

Berichter:	Prof. Dr. Ir. Dr. h. c. Joost-Pieter Katoen
	Prof. Annabelle McIver
Tag der mündlichen Prüfung:	8. Februar 2019

Benjamin Lucien Kaminski. *Advanced Weakest Precondition Calculi for Probabilistic Programs.* February 15, 2019

ABSTRACT

We study quantitative reasoning about probabilistic programs. In doing so, we investigate two main aspects: The reasoning techniques themselves and the computational hardness of that reasoning.

As for the former aspect, we first give a comprehensive introduction to *weakest preexpectation reasoning* á la McIver & Morgan — a reasoning technique for the verification of probabilistic programs that builds on Dijkstra's *weakest precondition calculus* for programs with nondeterminism and Kozen's *probabilistic propositional dynamic logic* for probabilistic programs. We then develop advanced weakest–preexpectation–style calculi for probabilistic programs that enable reasoning about

1. expected runtimes,
2. conditional expected values and conditional probabilities, and
3. expected values of mixed–sign random variables.

As with Dijkstra's calculus, our calculi are defined inductively on the program structure and thus allow for compositional reasoning on source code level. We put a special emphasis on proof rules for reasoning about loops.

The second aspect we study is the inherent computational hardness of reasoning about probabilistic programs, which is independent from the employed analysis technique. In particular, we study the hardness of approximating expected values and (co)variances. We show that lower bounds on expected values are not computable but computably enumerable, whereas upper bounds are not computably enumerable. For covariances, we show that neither lower nor upper bounds are computably enumerable.

Furthermore, we study the hardness of deciding termination of probabilistic programs. While we study different notions of probabilistic termination, for instance almost–sure termination or termination within finite expected time (also known as positive almost–sure termination), we show that deciding termination of probabilistic programs is generally strictly harder than deciding termination of nonprobabilistic programs.

KURZFASSUNG

Wir studieren die quantitative Analyse probabilistischer Programme. Dabei untersuchen wir vornehmlich zwei Aspekte: Die Analysetechniken selbst, sowie die komplexitäts- bzw. berechenbarkeitstheoretische Schwere der entsprechenden Analyseprobleme.

In Bezug auf die Analysetechniken geben wir zunächst eine umfassende Einführung in den *Kalkül der Schwächsten Vorerwartungen* á la McIver & Morgan — ein Kalkül für die Verifikation probabilistischer Programme, der auf Dijkstras *Kalkül der Schwächsten Vorbedingungen* für Programme mit Nichtdeterminismus und Kozens *Probabilistischer Dynamischer Aussagenlogik* für probabilistische Programme aufbaut. Anschließend entwickeln wir weitergehende Kalküle für probabilistische Programme im Stile McIver & Morgans, welche dazu geeignet sind, Analysen über

1. erwartete Laufzeiten,
2. bedingte Erwartungswerte und bedingte Wahrscheinlichkeiten, und
3. Erwartungswerte von Zufallsvariablen mit gemischtem Vorzeichen

zu fahren. Wie auch Dijkstras Kalkül sind unsere Kalküle induktiv über die Programmstruktur definiert und erlauben somit eine modulare Analyse auf Quelltextebene. Ein besonderes Augenmerk legen wir auf Regeln, welche die Analyse von Schleifen ermöglichen.

Der zweite Aspekt, den wir untersuchen, ist die inhärente berechenbarkeitstheoretische Schwere der Analyse probabilistischer Programme, welche unabhängig von der verwendeten Analysetechnik selbst ist. Im Speziellen untersuchen wir dazu die Schwere der Approximation von Erwartungswerten und Kovarianzen. Wir zeigen, dass untere Schranken für Erwartungswerte nicht berechenbar, aber rekursiv aufzählbar sind, obere Schranken jedoch nicht rekursiv aufzählbar sind. Für Kovarianzen zeigen wir, dass weder obere noch untere Schranken rekursiv aufzählbar sind.

Desweiteren untersuchen wir die Schwere der Entscheidbarkeit der Terminierung probabilistischer Programme. Während wir dazu zwar unterschiedliche Auffassungen eines probabilistischen Terminierungsbegriffs untersuchen, beispielsweise fast–sichere Terminierung oder Terminierung innerhalb endlicher erwarteter Zeit (auch positive fast–sichere Terminierung genannt), zeigen wir, dass die Terminierung probabilistischer Programme im Allgemeinen echt schwerer zu entscheiden ist als die Terminierung nicht-probabilistischer Programme.

ABSTRAKT

Zkoumáme kvantitativní analýzu pravděpodobnostních programů. Přitom se zabýváme především dvěma aspekty: Samotnými technikami analýzy jako takovými, jakož i teoretickou složitostí rozhodnutelnosti daných analytických problémů.

Ohledně analytických technik uvádíme nejprve komplexní úvod do takzvaného *Kalkulu nejslabších předočekávání* á la Mclverová & Morgan — kalkul pro verifikaci pravděpodobnostních programů, který se opírá o Dijkstrův *Kalkul nejslabších vstupních podmínek* pro programy s nedeterminismem a Kozenovu *Pravděpodobnostní dynamickou výrokovou logiku* pro pravděpodobnostní programy. Následně vyvíjíme komplexnější verze tohoto kalkulu pro pravděpodobnostní programy ve stylu Mclverové & Morgana, které jsou vhodné pro analýzu

- ✧ očekávaných časů,
- ✧ podmíněných očekávaných hodnot a podmíněných pravděpodobností a
- ✧ očekávaných hodnot náhodných proměnných se smíšeným znaménkem

Stejně jako Dijkstrův kalkul jsou naše kalkuly vzhledem ke struktuře programu induktivně definované a umožňují tak kompozicionální analýzu na úrovni zdrojového kódu. Zvláštní pozornost věnujeme pravidlům, které umožňují analýzu cyklu.

Druhým námi zkoumaným aspektem je inherentní teoretický stupeň nerozhodnutelnosti analýzy pravděpodobnostních programů, nezávisle na aplikované technice analýzy jako takové. K tomu zkoumáme zejména složitost aproximace očekávaných hodnot a kovariancí. Dokazujeme, že dolní meze pro očekávané hodnoty nejsou rekurzivní, ale rekurzivně spočetné, kdežto horní meze nejsou rekurzivně spočetné. U kovariancí dokazujeme, že jak horní, tak i dolní meze nejsou rekurzivně spočetné.

Dále zkoumáme složitost rozhodnutelnosti terminace pravděpodobnostních programů. Při zkoumání různých pojetí pravděpodobnostního pojmu terminace, například skoro jisté terminace nebo terminace v rámci konečného očekávaného času (také nazývané pozitivní skoro jistá terminace), dokazujeme, že rozhodnutí o terminaci pravděpodobnostních programů je obecně ostře těžší než terminace nepravděpodobnostních programů.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank Joost-Pieter Katoen for his truly great supervision. He introduced me to the interesting and vibrant field of probabilistic program research and I am thankful for his guidance in authoring the publications that lead to this thesis. Out of his many good qualities as a supervisor, I would like to particularly highlight that Joost-Pieter tirelessly promotes his advisee's research among experts all across the world, which I have always found extremely encouraging and which, I believe, should not be taken for granted! Moreover, Joost-Pieter knows how to create a truly great work and research environment!

Speaking of that great environment: I had the opportunity to work with many wonderful people in Aachen. I am grateful that right from the beginning I shared an office with two postdoctoral researchers, Nils Jansen and Federico Olmedo. Doing research with them was (and still is) great! Apart from research itself, I am thankful that Nils and Joost-Pieter taught me about „non–scientific" aspects of research very early on (and they continue to do so). It was also my friend Nils who introduced me to the art of working–at–a–café (a very effective technique for finishing one's thesis) as well as to a delicious regional product called Monschauer Els.

I would like to thank my second examiner Annabelle McIver. It was great meeting her at various places all over the world and having really inspiring discussions. I am particularly thankful that Carroll Morgan and her offered the opportunity to collaborate with them. I am also really grateful that Annabelle came all the way from Sidney to attend my defense in Aachen. Again something I do not take for granted. I would also like to take this opportunity and thank the remaining two members of my doctoral examination committee, Wil van der Aalst and Erich Grädel, for creating a very friendly and cordial atmosphere during my defense.

So far my most fruitful collaboration was with Christoph Matheja. Ever since Federico and I initially got him interested in doing research with us, Christoph and I have successfully continued to work together and I sincerely hope this will continue in the future. Another collaboration I enjoy very much is with Kevin Batz, whom I wish all the best for his own PhD studies.

I am grateful to Alexandra Silva for giving me the opportunity to visit her and her group in London. I have had a truly amazing time and met so many great people! Thank you Alexandra, Paul Brunet, Fredrik Dahlqvist, Carsten Fuhs, Gerco van Heerdt, Tobias Kappé, Joshua Moerman, Louis Parlant, Matteo Sammartino, Fabio Zanasi for all the nice discussions and for making my stay in London such a great experience.

I am grateful to Gilles Barthe for constantly feeding us with interesting challenges and for the nice collaboration we have had. I am also grateful to my coauthors Alejandro Aguirre, Jürgen Giesl, Marcel Hark, Christian Hensel, Justin Hsu, Maurice van Keulen, Thomas Noll, Raimondas Sasnauskas, and Carsten Weise for many fun and fruitful discussions.

Coming back to the great work environment, I would like to thank Erika Ábrahám, Philipp Berger, Helen Bolke-Hermanns, Harold Bruintjes, Florian Corzilius, Luis María Ferrer Fioriti, Predrag Filipovikj, Florian Frohn, Martin Grohe, Rebecca Haehn, Arnd Hartmanns, Jonathan Heinen, Jera Hensel, Christina Jansen, Sebastian Junges, Mojgan Kamali, Shahid Khan, Gereon Kremer, Tim Lange, Francesco Leofante, Anna Lukina, Elke Ohlenforst, Shashank Pathak, Tim Quatmann, Pascal Richter, Stefan Schupp, Jip Spel, Marcin Szymczak, Wolfgang Thomas, Matthias Volk, Birgit Willms, Gerhard Woeginger, and the many people I surely forgot to mention here. Each and everyone of them has always been very approachable, helpful, and open for discussions. I thank them not only for their scientific and non–scientific input but especially for making my time at i2 (and the computer science department at RWTH Aachen in general) just great!

I would like to thank my parents Monika and Gustav Kaminski for all their love and their unconditional support throughout my entire life. My mum and dad raised me bilingual and my mum introduced me very early on to higher mathematics. Both helped me a lot for forming an ability of abstraction and without their influence, I would have probably not done a PhD. I would also like to thank Stefanie Riske for all her love and support, especially during my PhD years.

A big „Thank you!“ goes out to my super talented cousin Benoît Texier, who designed the amazing cover art for this thesis. The drawing is just awesome and has far exceeded all my expectations!

I would like to thank my dad for translating the abstract of my thesis into Czech and Petr Novotný for proof reading it with regard to mathematical and technical terminology.

I would like to especially thank my fellow students, friends, and Gruppe 0 members Johannes van der Giet, Russ Jukić, and Philipp Kaiser for the years we have spent studying computer science in Aachen. Without you guys, I cannot imagine to have pulled this off!

I would like to thank my friends Agnes Nießen, Alexander Hofmann, Anne Hofmanns, Benedikt Flerus, David Renger, Fabian Klaes, Florian Kratz, Henrik Schwaeppe, Janine Lückgen, Judith Moos, Julian Kemp, Kevin Xiang, Lena Sellmeier, Luis Böttcher, Maira Kryschewski, Martin Moos, Max Rauch, Nicole Kramorz, Pascal Marquardt, Paul Walterscheid, Roland Fischer, Sebastian Palm, Sebastian Taron, Stephanie Ulmen, Tobias Haas, Ulla Krüger, and Valentin Ziemons for attending and celebrating my defense. You made it an amazing day! Once again something I do not take for granted!

CONTENTS

1 OVERVIEW

Reasoning about programs is an indispensable but very difficult task in software engineering. Today, in times of ever–increasing complexity and omnipresence of software systems, the ability to reason about programs is becoming more and more important, but already the pioneers of computing had this problem on their minds some 80 years ago.

In 1941, the civil engineer Konrad Zuse completed building what is today considered the world's first programmable computer, the Z3. He deliberately designed it *not* to be a universal computer [Roj97]: First off, the Z3 featured *no loops* (which could be surmounted by literally *glueing together* the two ends of the punched tape containing the program). Secondly, the Z3 featured *no conditional branching*, effectively rendering the control flow graph of a Z3 program a straight line. While today we find it hard to imagine any programming language being able to get by without conditionals, Zuse himself even considered conditional branching *harmful*, as he feared that it would render programs too difficult to comprehend and understand [Zus90].

Twenty–seven years later, Edsger Dijkstra considered the `goto` statement harmful because it is „too much of an invitation to make a mess of one's program" [Dij68]. Instead, he advocated the use of guarded repetition constructs like while loops, which would give programs more structure and thereby open up an angle of attack for reasoning about them.

Both Zuse and Dijkstra explicitly expressed their concern about being able to comprehend programs and reason about what they compute. Even though the programming constructs which they discouraged were fully deterministic, they deemed them too complicated to reason about and it was explicitly for that reason that they recommended against their use. Given their concern about comprehensibility of programs, we can only imagine the discomfort they might have experienced when faced with the task of reasoning about probabilistic programs which *behave randomly* in the first place.[1] Randomization, however, has always played an important role in computing, even since its early days. In 1962, Tony Hoare proposed a randomized variant of his Quicksort algorithm [Hoa62]. A year later, Michael Rabin presented the notion of *probabilistic automata*, a randomized variant of finite automata [Rab63]. Today, probabilistic automata (and their numerous variations) continue to be subject of intensive research [Kat16].

1 Indeed, Dijkstra even provided written testimony of his discomfort, though in the context of a different form of uncertain program behavior, namely *nondeterminism*. He wrote in [Dij75]: „I myself had to overcome a considerable mental resistance before I found myself willing to consider non–deterministic programs seriously."

In this thesis, we study reasoning about the more general notion of *probabilistic programs*, which are computational procedures that may conditionally branch, depending on the outcome of a random experiment. The initial spark for formal reasoning about probabilistic programs was given by Dexter Kozen's seminal work on *semantics* of probabilistic programs in the late 1970s and early 1980s [Koz79; Koz81]. Subsequently, seminal work on *verification* of probabilistic programs was presented by Sergiu Hart, Micha Sharir, and Amir Pnueli [HSP82; HSP83] and, independently, by Kozen [Koz83; Koz85]. While many advances have been made since then, the repertoire of techniques for reasoning about probabilistic programs is arguably much less developed than for deterministic ones. It is likely that part of the reason for this discrepancy is that reasoning about probabilistic programs is more difficult — a fact which we will make mathematically precise in Part III of this thesis.

In Parts I and II of this thesis, we endeavor to extend the repertoire of reasoning techniques for probabilistic programs. To that end, we present advanced calculi for reasoning about structured probabilistic programs in a compositional manner. Our calculi are suitable for *quantitative reasoning*, which is relevant not only for probabilistic programs but, indeed, also for deterministic ones, as Thomas Henzinger points out [Hen13]:

> *The Boolean partition of software into correct and incorrect programs falls short of the practical need to assess the behavior of software in a more nuanced fashion against multiple criteria.*

Most of the calculi we present in this thesis are applicable to deterministic programs as well[2] and hence we see our work in the broader context of *quantitative verification* rather than just *probabilistic program verification*.

In the remainder of this section, we give a very brief overview of use cases and analysis problems for probabilistic programs, formal verification of programs in general, and probabilistic program verification in particular. Furthermore, we provide a short overview of the original contributions covered in this thesis, as well as an extensive synopsis of the parts and chapters of this thesis for the reader's convenience.

1.1 PROBABILISTIC PROGRAMS

Today, probabilistic programs serve (at least) two purposes: describing *randomized algorithms* and encoding *complex probability distributions*. In the following, we will survey these two use cases very briefly and point out why quantitative reasoning is necessary in these cases.

2 With the exception of the calculus for reasoning about *conditional expected values* presented in Chapter 8. While in principle applicable to deterministic programs, the calculus is simply not very meaningful in that context.

1.1.1 *Describing Randomized Algorithms*

Randomized algorithms use access to some source of randomness to provide a more efficient means of solving problems that are computationally difficult otherwise. Some problems are even impossible to solve without randomization, e.g. certain consensus problems [Ben83; FLP83]. Another use case where deterministic algorithms fail is symmetry breaking. As an example, the IEEE 802.3 Ethernet standard uses a so–called *Exponential Backoff Algorithm* for avoiding collisions between two coequal parties communicating on a single Ethernet line. For achieving collision avoidance, each party runs a randomized algorithm that lowers the probability of both parties attempting to occupy the Ethernet line at the same time [Iee].

As an example of speedup through randomization, Hoare's randomized Quicksort algorithm selects a pivot element uniformly at random which reduces the expected worst–case complexity of Quicksort to $\mathcal{O}\big(n\log(n)\big)$. In contrast, Quicksort with deterministic pivot selection has a worst–case complexity of $\mathcal{O}\big(n^2\big)$. Randomized Quicksort is an example of a *Las Vegas algorithm* which is *certainly correct* while only *probably fast*.

Another type of randomized algorithms are *Monte Carlo algorithms*, which are *certainly fast* while only *probably correct*. A prime example is the randomized matrix multiplication verification algorithm discovered by Rūsiņš Freivalds, which runs certainly in quadratic time [Fre79], but with probability $1/2$ fails to refute an incorrect matrix multiplication. The fastest known deterministic method reduces to performing an actual matrix multiplication, the fastest known algorithm for which was found by Virginia Williams and runs in $\mathcal{O}(n^{2.373})$ time [Wil14], i.e. *certainly slower* than Freivalds' algorithm.

Lastly, there is also a mixed type of randomized algorithms known as *Atlantic City algorithms*, which are both *probably fast and probably correct*. The nature of the properties that make up the distinction between Las Vegas, Monte Carlo, and Atlantic City algorithms demonstrates that we have an intrinsic need for *quantitative reasoning* when analyzing randomized algorithms: An Atlantic City algorithm, for instance, is not just either *correct* or *incorrect*. Instead, we need to *measure the probability* of the algorithm yielding correct results in order to asses the correctness of the algorithm.

Furthermore, an Atlantic City algorithm is not just always efficient or not. It does not suffice to look at a worst–case outcome of random events and reason about the longest possible runtime in that worst case. Instead, we need to consider *all* possible outcomes that may emerge from executing the randomized algorithm and then *average* their runtime in order to obtain an *expected runtime*. We thus see that not only functional correctness plays a role for the analysis of randomized algorithms. *Nonfunctional* and, in particular, *quantitative requirements*, such as expected runtimes, are of paramount importance as well. In this thesis, we present a calculus specifically tailored to the task of reasoning about expected runtimes.

1.1.2 *Describing Complex Probability Distributions*

Besides describing randomized algorithms, probabilistic programs are used in machine learning, artificial intelligence, or cognitive sciences, to describe complex probability distributions. This discipline is commonly known as *probabilistic programming* [Pro]. Various languages for probabilistic programming have been proposed, for instance Church (functional) [Goo+08], Figaro (object oriented) [Pfe09], ProbLog (logic) [RKT07], R2 (imperative) [Nor+14], Stan (functional) [Car+17], and Tabular (Excel spreadsheets) [Bor+16]. A key desideratum in probabilistic programming is to be able to describe complex *conditional* probability distributions, while at the same time achieving good accessibility of the description.

Before the advent of probabilistic programming, distributions were often encoded using *probabilistic graphical models*. However, the most promising models that have emerged from the machine learning and artificial intelligence communities outstrip the expressive power of such graphical models. As a workaround, models are encoded using a mixture of graphical and textual representation [Pro]. Probabilistic programming languages, on the other hand, provide a unified way to encode distributions in an accessible, yet mathematically rigorous way. Moreover, probabilistic programs are accessible to a working programmer who might not be knowledgeable in data science or probability theory [Gor+14].

As for their expressivity, universal probabilistic programming languages can encode any (discrete) probability distribution for which probabilities of events are semi–computable [Ica17]. In addition to their expressivity, probabilistic programs have the advantage of encoding distributions in a *structured* manner. They are thus highly amenable to *formal reasoning*.

A key analysis problem for probabilistic graphical models as well as for probabilistic programming is *inference* [Gor+14]: Given an event E (i.e. some set of outcomes of the probabilistic computation) and possibly some observed evidence O (again some set of outcomes of the probabilistic computation), what is the probability that event E will occur, given that O occurs? The inference problem in probabilistic programming is closely related to determining the correctness probability of a randomized algorithm. After all, yielding a correct result is also just some particular event.

In this thesis, we present a calculus for inference on general probabilistic programs with conditioning. Devising inference mechanisms on general probabilistic programs bears one big advantage: it disconnects the inference task from the model. Even *sampling* algorithms for *approximate inference* often have to be heavily hand–tuned and have to make use of substantial domain knowledge about the model at hand, in order to make them efficient [KF09, Chapter 12.3]. A more desirable approach would be to eradicate the need for that domain knowledge altogether and come up with more general inference methods [LBW17].

1.2 FORMAL VERIFICATION

Apart from Zuse and Dijkstra, another pioneer of computing, who saw a need for the ability to reason about programs very early on, was Alan Turing. In 1949, he presented his paper „Checking a Large Routine“ at the inaugural conference of the *Electronic Delay Storage Automatic Calculator* at the University of Cambridge Mathematical Laboratory [Tur49; MJ84]. Turing began his paper by asking:

> *How can one check a routine in the sense of making sure that it is right?*

What Turing back then considered a „large routine“ consists of about *12 lines of code*.[3] As of 2009, the code base of an average modern high–end car was estimated to consist of about *100 million lines of code* [Cha09]. Even the size of the safety–critical parts of that code base alone will exceed what Turing considered a „large routine“ by multiple orders of magnitude.

With the advent of *autonomous* car driving during the last decade or two, the need for guarantees on software correctness has increased significantly, as the importance of software for such autonomous systems can hardly be overrated. Elon Musk, founder of the electric car manufacturer Tesla, even considers Tesla to be a *software company*. In a 2015 interview [Hir15], he said:

> *We really designed the* [car] *to be a very sophisticated computer on wheels. Tesla is a software company as much as it is a hardware company.*

Between 2016 and 2018 alone, at least four people were killed in car accidents while autonomous driving systems were engaged. Other software failures with disastrous consequences include the failed maiden flight of the European Space Agency's Ariane 5 carrier rocket (material damage: USD 370 million) [Wika] or overdoses applied by Atomic Energy of Canada Limited's Therac–25 radiotherapy unit [Wikl] (at least three fatalities caused directly by severe radiation overdose, *several people seriously injured*). More recently, in 2018, the security vulnerabilities *Meltdown* [Lip+18] and *Spectre* [Koc+18] were made public. They affect almost *any* modern computer system and the full extent of their impact can still not be estimated.

So how can we avoid such software failures? One way is *testing*. In a nutshell, testing amounts to heuristically selecting a large number of critical inputs and checking whether the program being tested complies with its specification on all selected inputs. In general, however, there are *infinitely many* or at least *too many* inputs. Finding an error thus amounts to finding a needle in an infinitely large haystack. Trying to *prove the absence of an*

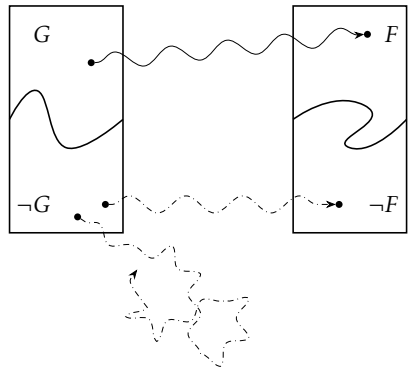
3 Turing provides his program in the form of a flow chart. A one–to–one translation to Python, for instance, can be done using 12 lines of code.

error amounts to asserting that there is no needle at all in the infinitely large haystack. As Dijkstra famously put it in his Turing Award Lecture [Dij72],

> *Testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.*

So, how can we show the absence of bugs? In his 1949 paper, Turing already outlined a method for systematically and formally verifying software. This method he outlines anticipated more matured and workable calculi for program verification developed later by Bob Floyd [Flo67a], Tony Hoare [Hoa69], and Edsger Dijkstra [Dij75]. In this thesis, we build directly on the work of the latter, namely on Dijkstra's *weakest precondition calculus*.

In a nutshell, the weakest precondition calculus works as follows: Say we need to verify the program C. Furthermore, we are given a *specification* in the form of a postcondition F on final states. Our goal is to verify that C terminates only in states satisfying the postcondition F. Dijkstra's calculus now allows us to obtain a *weakest precondition* G on initial states (i.e. the largest set of initial states), so that executing C on any state satisfying precondition G guarantees that C terminates in a final state satisfying postcondition F. Expressed graphically, we have the following picture:



Starting from an initial state satisfying the weakest precondition G, the computation of C will terminate in a final state satisfying the specification F. Starting from an initial state not satisfying G, there are two possibilities: *Either* the computation of C will terminate in a final state *not* satisfying F, *or* the computation of C will *not terminate* at all. In any case, for all initial states not satisfying the weakest precondition, the program C does not satisfy the specification. On the other hand, if the weakest precondition happens to describe the set of *all* states, then C satisfies its specification from *any* initial state.

A principal advantage of the weakest precondition calculus is that it can be defined *inductively* on the structure of the programming language itself. It can thus be applied directly *on source code level*, which means that it allows for directly verifying the software that is actually going to be deployed.

There is no need to first translate the source code to some abstract model on which the verification can then be performed. Such translations would naturally bear the potential for abstraction or translation errors.

Another advantage of the calculus' inductive nature is that it allows for *compositional reasoning*, which constitutes a classical divide–and–conquer principle. Compositionality allows for proving parts of a program correct and then inserting those correct parts into a context of a larger program.

1.3 VERIFICATION OF PROBABILISTIC PROGRAMS

> *One grain of wheat does not constitute a pile, nor do two grains, nor three and so on. On the other hand, everyone will agree that a hundred million grains of wheat do form a pile. What then is the threshold number? Can we say that 325,647 grains of wheat do not form a pile, but that 325,648 grains do? If it is impossible to fix a threshold number, it will also be impossible to know what is meant by a pile of wheat.*
>
> — Émile Borel

Reasoning about probabilistic programs is naturally quantitative. For example, we mentioned earlier that efficiency through randomization often takes its toll in the form of a small error probability. So when is a randomized algorithm correct? We could of course fix thresholds and say, for instance, that a probabilistic algorithm is sufficiently correct if it yields an incorrect result with probability at most $1/3$. Correctness of a randomized algorithm would then become a Boolean property: *Either* the error probability is at most $1/3$, *or* the probability is strictly higher.

Let us compare an exponential–time algorithm with error probability $1/3$ to a linear–time algorithm with error probability $5/12$. The linear–time algorithm is *incorrect* according to the fixed threshold since $5/12 > 1/3$. However, running the linear–time algorithm thrice and taking a majority vote over the results, reduces the error probability to approximately $1/4$ while retaining a linear runtime. Thus, the linear–time algorithm is arguably preferable.

The little gedankenexperiment above shows that fixing thresholds a priori is not very helpful, and hence we cannot just state that a randomized algorithm is either correct or incorrect. Instead, we need formal guarantees on *how correct* the algorithm is, i.e. we must *measure the probability* of the algorithm yielding correct results. The same goes for its runtime. We cannot just state that a randomized algorithm's runtime is either fast or slow. Instead, we need a guarantee that it is fast *on average*, i.e. we must *measure the expected runtime* of the algorithm. Simply put, verification of randomized algorithms comes down to providing *guarantees on quantities*.

So how do we verify probabilistic programs? One way is to execute the program a large number of times and check whether or not it satisfies its specification when averaging over the so–obtained sample set. This approach for probabilistic programs corresponds in some sense to *testing* for deterministic programs and can only ever yield *statistical confidence*, but no hard guarantees.

Towards providing formal guarantees on probabilistic programs, Dexter Kozen developed the *probabilistic propositional dynamic logic* (PPDL) [Koz83; Koz85] — a *modal logic* for verification of probabilistic programs, in which the modalities are annotated with probabilistic programs and the ground terms are measurable functions. Whereas Dijkstra's weakest precondition calculus is a *predicate transformer* transforming postconditions to preconditions, Kozen generalized to a *measurable function transformer*. This yields the following table that Peter Selinger calls „Kozen's Rosetta Stone"[4] [Sel17]:

deterministic	**probabilistic**
state transformer	distribution transformer
postcondition F	measurable function f
weakest precondition of F	expected value of f

A deterministic program is executed on an initial state and (if it terminates) yields a unique final state. A probabilistic program, on the other hand, yields in general a *probability distribution* over final states. If we want to verify a probabilistic program, we choose as postcondition a measurable function f, for instance the indicator function $[F]$ of a predicate F. If we can obtain guarantees on the expected value of $[F]$ after execution of the program, we have in effect obtained a formal guarantee on the *probability* that the program terminates in a state satisfying F. Kozen's PPDL, however, is more expressive and allows for answering questions about more general quantities, such as the expected value of program variable x, a program's termination probability, etc.

For PPDL, Kozen did not consider nondeterminism as Dijkstra did, but instead *replaced* it by probabilistic choice. The work of Kozen was later furthered by Annabelle McIver and Carroll Morgan [MMS96; MM05] mainly by (A) (re)incorporating nondeterminism into the probabilistic programming language and (B) intensively studying how Hoare's invariant–based reasoning carries over to probabilistic programs.

Referring to Dijkstra's weakest precondition calculus, McIver & Morgan called their calculus the *weakest preexpectation calculus*. In this thesis, we build heavily on their calculus, present it in their style, and — constituting the main contribution of this thesis — present similar but more advanced and novel calculi for reasoning about quantitative properties of probabilistic programs, such as expected runtimes, conditional expected values, etc.

4 The missing „third language" for making up a proper Rosetta Stone [Wikj] is chosen by Selinger to be the language of *quantum programming*.

1.4 CONTRIBUTIONS AND SYNOPSIS OF THIS THESIS

In this thesis, we attempt to further the line of work that has been developed over the past 70 years or so by Turing, Floyd, Hoare, Dijkstra, Kozen, McIver, and Morgan (amongst others). We advance the analysis of structured probabilistic programs in the style of Dijkstra's predicate transformer approach. A special focus is placed on reasoning about loops using invariant–based techniques. In short, the main contributions covered in this thesis are:

A. We give a more or less comprehensive and novel *introduction to weakest preexpectation reasoning* à la McIver & Morgan.

B. We survey *rules for reasoning about weakest preexpectations* and present *a new rule for proving termination* of probabilistic loops.

C. We present a novel calculus for compositional reasoning about *expected runtimes* of programs. In contrast to a naïve approach of annotating a program with a runtime–counter variable, our approach is sound, even if the program does not terminate with probability 1.

D. We present a novel calculus for weakest–preexpectation–style *inference for probabilistic programs with conditioning*. In contrast to existing approaches, we explicitly do *not* assume a priori that programs terminate with probability 1.

E. We present a novel calculus that extends McIver & Morgan's weakest preexpectation calculus for reasoning about expected values of non–negative random variables to the case for *mixed–sign random variables*.

F. We investigate the *computational hardness* of approximating expected values and covariances and of deciding probabilistic termination.

In the following, we give a more detailed synopsis of this thesis. Furthermore, we list for each chapter the publications that emerged from developing the contributions of the respective chapter. A graph roughly depicting the dependencies of the individual chapters on each other is provided for the reader's convenience in Figure 1.1.

1.4.1 *Part I: Classical Weakest Preexpectation Reasoning*

The first part of this thesis is intended to be an introduction to *weakest preexpectation reasoning* for probabilistic programs, which we gradually develop from the early ideas of Floyd, Hoare, and Dijkstra. It is the base–layer technique which all the advanced calculi we present later build on. We also present a variety of techniques for dealing with loops — arguably one of the most difficult tasks in program verification.

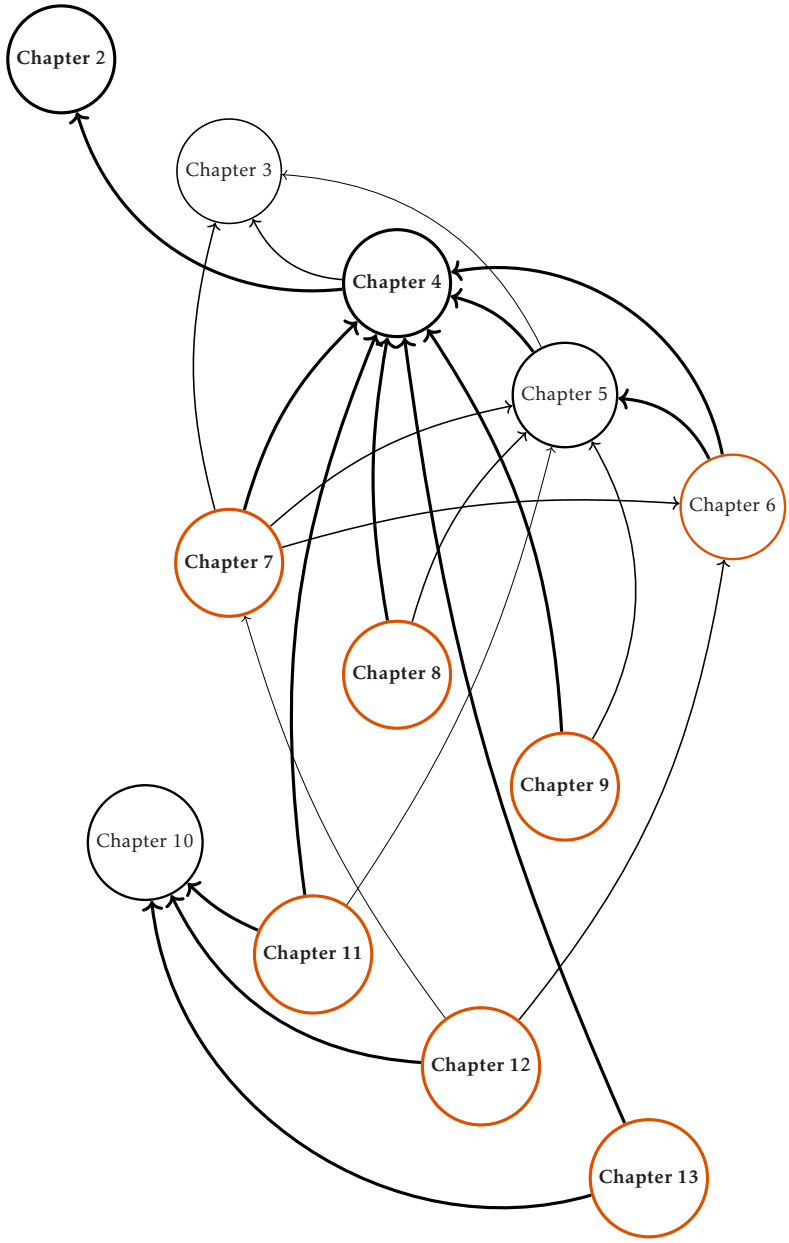


Figure 1.1: A rough dependency graph for the chapters of this thesis. A thicker arrow symbolizes a stronger dependency. Orange chapters contain original contributions of the author covered in this thesis.

Chapter 2: Weakest Precondition Reasoning. We present Dijkstra's *Guarded Command Language* (GCL) and techniques for reasoning about both *qualitative* and *quantitative properties* of deterministic GCL programs. As for qualitative reasoning, we explain the notion of *Hoare triples* and develop from those Dijkstra's *weakest precondition calculus*. We also discuss reasoning about nondeterministic programs as well as partial correctness. Finally, we show how the weakest precondition calculus can be generalized from reasoning about the satisfaction of Boolean–valued predicates to *anticipating values* of more general real–valued functions.

Chapter 3: Probabilistic Computations. We present McIver & Morgan's *probabilistic Guarded Command Language* (pGCL) — a language that features both *probabilistic* and *nondeterministic uncertainty*. We give *small–step operational semantics* to pGCL programs in the form of *probabilistic computation trees* as well as in the form of *Markov decision processes*. We also provide a semantics in the form of *probability distributions over final states*.

Chapter 4: Weakest Preexpectation Reasoning. We give an introduction to quantitative reasoning about pGCL programs by means of the *weakest preexpectation calculus* à la McIver & Morgan. This calculus can be used to reason about probabilities of events as well as more general expected values of real–valued functions. All the advanced calculi we present later in this thesis build on the weakest preexpectation calculus.

We also present a *liberal* calculus for *partial correctness*. We establish connections between the liberal and the nonliberal calculi, and to the operational semantics from Chapter 3. We also discuss properties of the calculi, such as *continuity*, *monotonicity*, or *linearity*.

Chapter 5: Proof Rules for Loops. We present techniques for reasoning about weakest preexpectations of loops. For that, we recall Hoare's method of using *invariants* and show how these can be lifted to our more general quantitative setting. Using invariants, we show how one can obtain *upper and lower bounds* on preexpectations and how to refine such bounds. We also discuss how *reasoning about lower bounds is conceptually harder* than reasoning about upper bounds. For loops of a certain form (*independent identically distributed loops*), we show how *exact preexpectations* can be determined. Section 5.2.8 is based on:

[Bat+18b] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. „How long, O Bayesian network, will I sample thee? A program analysis perspective on expected sampling times." In: *Proc. of the European Symposium on Programming Languages and Systems (ESOP)*. vol. 10801. Lecture Notes in Computer Science. Springer, 2018, pp. 186–213

Chapter 6: Probabilistic Termination. We study rules for proving two different forms of *probabilistic termination*: *almost–sure termination* and *positive almost–sure termination*. We then survey dedicated rules for proving probabilistic termination such as e.g. Chakarov & Sankaranarayanan's *supermartingale ranking functions*, Fioriti & Hermanns' *ranking supermartingales* or McIver & Morgan's *zero–one law*. We also present *a new rule for proving almost–sure termination*. Lastly, we present case studies on applying this new rule. Sections 6.2.3 and 6.2.4 are based on:

[McI+18] Annabelle McIver, Carroll Morgan, Benjamin Lucien Kaminski, and Joost-Pieter Katoen. „A New Proof Rule for Almost–sure Termination." In: *Proc. of the Symposium on Principles of Programming Languages (POPL)* 2.POPL (2018), 33:1–33:28

1.4.2 *Part II: Advanced Weakest Preexpectation Reasoning*

In the second part of this thesis, we present three *advanced weakest precondition calculi* for reasoning about probabilistic programs, each tailored to a different task. All those calculi build upon the weakest preexpectation calculus presented in Chapter 4.

Chapter 7: Expected Runtimes. We present the *expected runtime calculus* made for — as the name suggests — reasoning about expected runtimes of probabilistic programs. We show how an obvious approach annotating the program with runtime counters and reasoning about their expected value is unsound, in general, and thus justify the need for a dedicated calculus. Our ert calculus is *sound* while still being appealingly simple. We discuss *basic properties* of the calculus. Finally, we present *proof rules based* on our notion of invariants for reasoning about loops and show their effectiveness by reasoning about the expected runtime of the well–known *coupon collector's problem*. Chapter 7 is based in part on:

[Kam+18] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. „Weakest Precondition Reasoning for Expected Runtimes of Randomized Algorithms." In: *Journal of the ACM* 65.5 (2018), 30:1–30:68

[Kam+16] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. „Weakest Precondition Reasoning for Expected Run–Times of Probabilistic Programs." In: *Proc. of the European Symposium on Programming Languages and Systems (ESOP)*. vol. 9632. Lecture Notes in Computer Science. Springer, 2016, pp. 364–389

Chapter 8: Conditioning. We endow pGCL with a *conditioning* feature to obtain the *probabilistic Guarded Command Language with conditioning* (cpGCL). We present the *conditional weakest preexpectation calculus* (cwp) for reasoning about conditional expected values yielded by cpGCL programs. We also discuss partial correctness, as well as two other alternatives for defining cwp calculi and discuss how these can be interpreted and how our calculus is more natural. Furthermore, we discuss *nondeterminism* and *basic properties* of the cwp calculus. Finally, we present invariant–based *proof rules* for reasoning about loops. Chapter 8 is based in part on:

[Olm+18] Federico Olmedo, Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Annabelle McIver. „Conditioning in Probabilistic Programming." In: *Trans. on Programming Languages and Systems* 40.1 (2018), 4:1–4:50

[Gre+16] Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Federico Olmedo. „On the Semantic Intricacies of Conditioning." In: *Extended Abstracts of the 1st Workshop on Probabilistic Programming Semantics (PPS)* (2016)

[Kat+15] Joost-Pieter Katoen, Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, and Federico Olmedo. „Understanding Probabilistic Programs." In: *Correct System Design — Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday*. Vol. 9360. Lecture Notes in Computer Science. Springer, 2015, pp. 15–32

[Jan+15a] Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Federico Olmedo, Friedrich Gretz, and Annabelle McIver. „Conditioning in Probabilistic Programming." In: *Proc. of the Conference on Mathematical Foundations of Programming Semantics (MFPS)* 319 (2015), pp. 199–216

Chapter 9: Mixed–sign Expectations. The classical weakest preexpectation calculus can be used to reason about expected values of *non–negative* real–valued functions. We extend this calculus to allow for *mixed–sign* real–valued functions as well. We carefully discuss problems that arise in reasoning about mixed–sign functions and show how these can be mitigated. We discuss *basic properties* of the calculus and *proof rules* for reasoning about loops. Chapter 9 is based on:

[KK17b] Benjamin Lucien Kaminski and Joost-Pieter Katoen. „A Weakest Pre–expectation Semantics for Mixed–sign Expectations." In: *Proc. of the Annual Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, 2017, pp. 1–12

[KK17a] Benjamin Lucien Kaminski and Joost-Pieter Katoen. „A Weakest Pre–Expectation Semantics for Mixed–Sign Expectations."

In: *Extended Abstracts of the 2nd Workshop on Probabilistic Programming Semantics (PPS)*. 2017

1.4.3 *Part III: Computational Hardness*

Whereas the subject of the second part of this thesis was techniques for reasoning about probabilistic programs, the last part of this thesis is concerned with the *computational hardness* of that reasoning.

Chapter 10: The Arithmetical Hierarchy. We present the notion of the *arithmetical hierarchy*, originally due to Stephen Kleene and independently to Andrzej Mostowski. This framework allows us to place decision problems that are undecidable in the first place in a hierarchy, thereby stating in a mathematically precise sense „how undecidable" a decision problem is. We will place all analysis problems we consider for probabilistic programs in the arithmetical hierarchy.

Chapter 11: Approximating Preexpectations. We study the *hardness of computing weakest preexpectations*, i.e. expected values of random variables with respect to distributions yielded by executing a probabilistic program. For that, we study the hardness of approximating *lower bounds*, approximating *upper bounds*, and deciding whether some value equals the *exact* preexpectation. Chapter 11 is based in part on:

[KKM18] Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. „On the Hardness of Analyzing Probabilistic Programs." In: *Acta Informatica* (2018)

[KK15b] Benjamin Lucien Kaminski and Joost-Pieter Katoen. „On the Hardness of Almost–Sure Termination." In: *Proc. of the International Symposium on Mathematical Foundations of Computer Science (MFCS)*. vol. 9234. Lecture Notes in Computer Science. Springer, 2015, pp. 307–318

[KK15a] Benjamin Lucien Kaminski and Joost-Pieter Katoen. „Analyzing Expected Outcomes and (Positive) Almost–sure Termination of Probabilistic Programs is Hard." In: *Proc. of the Young Researchers' Conference „Frontiers of Formal Methods" (FFM)*. vol. 9234. Aachener Informatik Berichte. 2015, pp. 179–184

Chapter 12: Deciding Probabilistic Termination. We study the *hardness of deciding probabilistic termination*. More specifically, we study the hardness of deciding *almost–sure termination* and *positive almost–sure termination* on a specified input, as well as their *universal* variants. Based on our findings,

we engage in a discussion about the *proper notion of probabilistic termination*. Chapter 12 is based in part on:

[KKM18] Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. „On the Hardness of Analyzing Probabilistic Programs." In: *Acta Informatica* (2018)

[KK15b] Benjamin Lucien Kaminski and Joost-Pieter Katoen. „On the Hardness of Almost–Sure Termination." In: *Proc. of the International Symposium on Mathematical Foundations of Computer Science (MFCS)*. vol. 9234. Lecture Notes in Computer Science. Springer, 2015, pp. 307–318

[KK15a] Benjamin Lucien Kaminski and Joost-Pieter Katoen. „Analyzing Expected Outcomes and (Positive) Almost–sure Termination of Probabilistic Programs is Hard." In: *Proc. of the Young Researchers' Conference „Frontiers of Formal Methods" (FFM)*. vol. 9234. Aachener Informatik Berichte. 2015, pp. 179–184

Chapter 13: Approximating Covariances. We study the *hardness of computing covariances and variances* of random variables with respect to distributions yielded by executing a probabilistic program. As we did for preexpectations, we study the hardness of approximating *lower bounds*, approximating *upper bounds*, and deciding whether some value equals the *exact* covariance or variance. Chapter 13 is based in part on:

[KKM18] Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. „On the Hardness of Analyzing Probabilistic Programs." In: *Acta Informatica* (2018)

[KKM16] Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. „Inferring Covariances for Probabilistic Programs." In: *Proc. of the International Conference on Quantitative Evaluation of Systems (QEST)*. vol. 9826. Lecture Notes in Computer Science. Springer, 2016, pp. 191–206

Chapter 14: Conclusion and Future Work. We conclude and summarize discussions of potential directions for future work which would otherwise remain scattered in the various chapters of this thesis.

1.4.4 *Part IV: Appendices*

We provide mathematical preliminaries and calculations that have been omitted in the main text. More specifically, we provide in Appendix A basic preliminaries on domain theory, or rather on the fixed point theory of monotonic self–maps on complete lattices. In Appendix B, we provide basic preliminar-

ies on Markov chains and Markov decision processes. Finally, we provide in Appendix C calculations that are omitted in the main text.

1.5 A NOTE ON CONTRIBUTIONS OF THE AUTHOR

In this section, I give a complete list of peer–reviewed publications I coauthored that emerged from the research done in the course of writing this thesis. Additionally, I also provide a list of additional peer–reviewed publications I coauthored, but that are not covered in this thesis. Under current doctoral regulations of the RWTH Aachen University Faculty of Mathematics, Computer Science and Natural Sciences, I am required to discuss in detail my own contributions to the publications covered in this thesis. This discussion is found in Appendix D.

1.5.1 *List of Publications Covered in this Thesis*

[Kam+18] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. „Weakest Precondition Reasoning for Expected Runtimes of Randomized Algorithms." In: *Journal of the ACM* 65.5 (2018), 30:1–30:68

[Bat+18a] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. „A Program Analysis Perspective on Expected Sampling Times." In: *Extended Abstracts of the International Conference on Probabilistic Programming (PROBPROG)*. 2018

[KKM18] Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. „On the Hardness of Analyzing Probabilistic Programs." In: *Acta Informatica* (2018)

[Bat+18b] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. „How long, O Bayesian network, will I sample thee? A program analysis perspective on expected sampling times." In: *Proc. of the European Symposium on Programming Languages and Systems (ESOP)*. vol. 10801. Lecture Notes in Computer Science. Springer, 2018, pp. 186–213

[McI+18] Annabelle McIver, Carroll Morgan, Benjamin Lucien Kaminski, and Joost-Pieter Katoen. „A New Proof Rule for Almost–sure Termination." In: *Proc. of the Symposium on Principles of Programming Languages (POPL)* 2.POPL (2018), 33:1–33:28

[Olm+18] Federico Olmedo, Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Annabelle McIver. „Conditioning in Probabilistic Programming." In: *Trans. on Programming Languages and Systems* 40.1 (2018), 4:1–4:50

[KK17b] Benjamin Lucien Kaminski and Joost-Pieter Katoen. „A Weakest Pre–expectation Semantics for Mixed–sign Expectations." In: *Proc. of the Annual Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, 2017, pp. 1–12

[KK17a] Benjamin Lucien Kaminski and Joost-Pieter Katoen. „A Weakest Pre–Expectation Semantics for Mixed–Sign Expectations." In: *Extended Abstracts of the 2nd Workshop on Probabilistic Programming Semantics (PPS)*. 2017

[KKM16] Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. „Inferring Covariances for Probabilistic Programs." In: *Proc. of the International Conference on Quantitative Evaluation of Systems (QEST)*. vol. 9826. Lecture Notes in Computer Science. Springer, 2016, pp. 191–206

[Kam+16] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. „Weakest Precondition Reasoning for Expected Run–Times of Probabilistic Programs." In: *Proc. of the European Symposium on Programming Languages and Systems (ESOP)*. vol. 9632. Lecture Notes in Computer Science. Springer, 2016, pp. 364–389

[Gre+16] Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Federico Olmedo. „On the Semantic Intricacies of Conditioning." In: *Extended Abstracts of the 1st Workshop on Probabilistic Programming Semantics (PPS)* (2016)

[Kat+15] Joost-Pieter Katoen, Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, and Federico Olmedo. „Understanding Probabilistic Programs." In: *Correct System Design — Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday*. Vol. 9360. Lecture Notes in Computer Science. Springer, 2015, pp. 15–32

[KK15b] Benjamin Lucien Kaminski and Joost-Pieter Katoen. „On the Hardness of Almost–Sure Termination." In: *Proc. of the International Symposium on Mathematical Foundations of Computer Science (MFCS)*. vol. 9234. Lecture Notes in Computer Science. Springer, 2015, pp. 307–318

[Jan+15a] Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Federico Olmedo, Friedrich Gretz, and Annabelle McIver. „Conditioning in Probabilistic Programming." In: *Proc. of the Conference on Mathematical Foundations of Programming Semantics (MFPS)* 319 (2015), pp. 199–216

[KK15a] Benjamin Lucien Kaminski and Joost-Pieter Katoen. „Analyzing Expected Outcomes and (Positive) Almost–sure Termination of Probabilistic Programs is Hard." In: *Proc. of the Young*

Researchers' Conference „Frontiers of Formal Methods" (FFM). vol. 9234. Aachener Informatik Berichte. 2015, pp. 179–184

1.5.2 *List of Additional Publications Not Covered in this Thesis*

Below I list other peer–reviewed publications that I coauthored during the time I was writing this thesis or during my undergraduate studies, but whose contributions are not covered in this thesis:

[Bat+19] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. „Quantitative Separation Logic — A Logic for Reasoning about Probabilistic Programs." In: *Proc. of the Symposium on Principles of Programming Languages (POPL)*. [to appear]. ACM, 2019

[Keu+18] Maurice van Keulen, Benjamin Lucien Kaminski, Christoph Matheja, and Joost-Pieter Katoen. „Rule–based Conditioning of Probabilistic Data Integration." In: *Proc. of the International Conference on Scalable Uncertainty Management (SUM)*. Lecture Notes in Artificial Intelligence. Springer, 2018

[Jan+16] Nils Jansen, Christian Dehnert, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Lukas Westhofen. „Bounded Model Checking for Probabilistic Programs." In: *Proc. of the International Symposium on Automated Technology for Verification and Analysis (ATVA)*. vol. 9938. Lecture Notes in Computer Science. 2016, pp. 68–85

[Olm+16] Federico Olmedo, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. „Reasoning about Recursive Probabilistic Programs." In: *Proc. of the Annual Symposium on Logic in Computer Science (LICS)*. ACM, 2016, pp. 672–681

[Jan+15b] Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. „Probabilistic Programs — A Natural Model for Approximate Computations." In: *Extended Abstracts of the Workshop on Approximate Computing (AC 15)*. 2015

[Sas+11] Raimondas Sasnauskas, Oscar Soria Dustmann, Benjamin Lucien Kaminski, Klaus Wehrle, Carsten Weise, and Stefan Kowalewski. „Scalable Symbolic Execution of Distributed Systems." In: *Proc. of the International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society, 2011, pp. 333–342

Although not covered in this thesis, two of the above papers are closely related to this thesis. In [Olm+16], we have developed a weakest preexpec-

tation calculus as well as an expected runtime calculus for *recursive* probabilistic programs. We also discussed proof rules and demonstrated their effectiveness in a case study, namely a randomized binary search.

In [Bat+19], we have developed a *quantitative separation logic* for *local reasoning* about probabilistic programs that may allocate *dynamic memory*. We extended classical separation logic [IO01; Rey02] in two regards: First off all, we lifted the two separating connectives $\star$ and $\mathrel{-\!\!*}$ from connecting two predicates to connecting two *quantities*. Properties of the classical separating connectives, such as the adjointness of $\star$ and $\mathrel{-\!\!*}$, carry over to the quantitative connectives. Secondly, we developed a weakest preexpectation calculus for reasoning about probabilistic programs with dynamic memory using the quantitative versions of $\star$ and $\mathrel{-\!\!*}$.

Part I

CLASSICAL WEAKEST PREEXPECTATION REASONING

The first part of this thesis is intended as a *more or less comprehensive guide to weakest preexpectation reasoning* for probabilistic programs in the style of McIver & Morgan. While most of the material presented here are not my original ideas, I believe that the didactical approach that I take here is new and insightful (at least it was for me). I first recap Dijkstra's classical weakest *precondition* calculus for nonprobabilistic programs before I *gradually* (with a stopover at *quantitative* reasoning for nonprobabilistic programs) develop its extension to the weakest *preexpectation* calculus for probabilistic programs. Thereafter, I present an extensive *collection of proof rules* for reasoning about loops. In particular, I present reasoning techniques based on *quantitative invariants*, which will be a reoccurring motif in Part II of this thesis.

WEAKEST PRECONDITION REASONING 2

Weakest precondition reasoning [Dij75; Dij76] is a technique developed by Edsger Wybe Dijkstra for formal reasoning about the correctness of nonprobabilistic programs. In this chapter, we will introduce this technique and show how it can be generalized to allow for quantitative reasoning. Later, in Chapter 4, we will see how those quantitative techniques can be extended to reasoning about probabilistic programs.

This chapter is organized as follows: We first present the programming language GCL. This language (as well as probabilistic and recursive extensions of it) will be used throughout this entire thesis. We then recap Dijkstra's original weakest precondition calculus which allows for reasoning at the level of predicates. Finally, we show how to extend this style of reasoning beyond predicates to quantities, e.g. values of program variables.

2.1 THE GUARDED COMMAND LANGUAGE (GCL)

Dijkstra's Guarded Command Language (GCL) [Dij76] is a very simple, yet Turing–complete [Wikm], imperative model programming language that still provides enough structure to produce readable programs. In that sense the GCL formalism lies between elementary models of computation like e.g. Turing–machines or the Lambda calculus (very simple formalism, less readable programs) and higher programming languages like e.g. C++ or Java (more complex formalism, much more readable programs).

We use GCL to describe ordinary, i.e. nonprobabilistic, programs. There is thus no access to any source of randomness within GCL. On the other hand, we will later see that GCL does include features for modeling uncertainty, namely *nondeterministic choices*. Later in Chapter 3, we will see how this language can be extended further to express probabilistic computations as well. For now, though, let us consider only ordinary, deterministic programs:

Definition 2.1 (The Guarded Command Language [Dij76]):

a. *Let* Vars *be a countable set of program variables and let* Vals *be a countable set of values. If not explicitly stated otherwise, we let* $\mathsf{Vals} = \mathbb{Q}$*, where* $\mathbb{Q}$ *is the set of rational numbers. For later use, let* $\aleph : \mathbb{N} \to \mathsf{Vals}$*. be a* bijective *canonical enumeration of* Vals*.*

b. *The set of program states is given by*

$$\Sigma \;=\; \{\sigma \mid \sigma : \mathsf{Vars} \to \mathsf{Vals}\}\ .$$

c. *The set of programs in guarded command language, denoted GCL, is given by the grammar*

$$\begin{aligned} C \;\longrightarrow\; & \texttt{skip} && \text{(effectless program)}\\ \mid\; & \texttt{diverge} && \text{(freeze)}\\ \mid\; & x := E && \text{(assignment)}\\ \mid\; & C \,;\, C && \text{(sequential composition)}\\ \mid\; & \texttt{if}\,(\varphi)\,\{C\}\,\texttt{else}\,\{C\} && \text{(conditional choice)}\\ \mid\; & \texttt{while}(\varphi)\{C\}\,, && \text{(while loop)} \end{aligned}$$

where $x \in \mathsf{Vars}$ is a program variable, E is an arithmetic expression over program variables, and φ is a boolean expression over program variables guarding a choice or a loop.

d. *A program containing no* `diverge` *or while loops is loop–free.*

e. *Given a program state σ, we denote by $\sigma(E)$ the evaluation of expression E in σ, i.e. the value obtained by evaluating E after replacing any occurrence of any program variable x in E by the value $\sigma(x)$. Analogously, we denote by $\sigma(\varphi)$ the evaluation of φ in σ to either true or false. Furthermore, for a value $v \in \mathsf{Vals}$ we write $\sigma\,[x \mapsto v]$ to indicate that in σ we set x to v, i.e.*[1]

$$\sigma\,[x \mapsto v] \;=\; \lambda y.\, \begin{cases} v, & \text{if } y = x \\ \sigma(y), & \text{if } y \neq x\,. \end{cases}$$

f. *We use the Iverson bracket notation [Wikg] to associate with each guard its according indicator function. The Iverson bracket $[\varphi]$ of guard φ is thus defined as the function*

$$[\varphi]\colon \quad \Sigma \to \{0,1\}, \quad [\varphi](\sigma) \;=\; \begin{cases} 1, & \text{if } \sigma(\varphi) = \text{true} \\ 0, & \text{if } \sigma(\varphi) = \text{false}. \end{cases}$$

Let us examine the computational effects of all GCL constructs. We start with the atomic programs: The effectless program `skip` does nothing, meaning that it terminates immediately in an unaltered program state. Starting in an initial state σ, the program `skip` will terminate in the same final state σ.

The freezing program `diverge` is a program that immediately enters an endless busy loop and therefore it does not terminate (diverges) regardless of the initial state it is started in. It can be thought of as a shorthand notation for the endless loop `while(true){skip}` (for the effects of loops, see below).

1 We use λ–expressions to construct functions: $\lambda\xi.\,\epsilon$ stands for the function that, when applied to an argument α, evaluates to ϵ in which every occurrence of ξ is replaced by α.

The assignment $x := E$ is the (only) statement that directly alters the program state. It evaluates E in the current program state and sets program variable x to the thusly obtained value. When executed on an initial state σ, the assignment $x := E$ thus terminates in final state $\sigma[x \mapsto \sigma(E)]$.

We proceed with the composed statements: The sequentially composed program $C_1 \fatsemi C_2$ has exactly the effect one would expect: First C_1 is executed and after its termination C_2 is executed. So if C_1 transforms state σ into σ' and C_2 transforms σ' into σ'', then $C_1 \fatsemi C_2$ transforms σ into σ''. Notice that if C_1 diverges on σ (e.g. if $C_1 = \texttt{diverge}$), then so does $C_1 \fatsemi C_2$.

The construct $\texttt{if}\,(\varphi)\,\{C_1\}\,\texttt{else}\,\{C_2\}$ is a conditional choice. Guard φ is a boolean expression over program variables, thus e.g. of the form $x > 0$ or $(x + y \leq z + 17) \wedge (z \neq 0)$. If φ evaluates to true in the current program state, then C_1 is executed and if φ evaluates to false, then C_2 is executed.

EXAMPLE 2.2 (Deterministic GCL Programs):
Consider the following loop–free GCL program:

$$
\begin{aligned}
C_{2.2} \rhd \quad & \texttt{if}\,(y > 0)\,\{x := 5\}\,\texttt{else}\,\{x := 2\} \fatsemi \\
& y := x - 3 \fatsemi \\
& \texttt{skip}
\end{aligned}
$$

First, this program checks whether y is strictly larger than 0. If this is the case, it sets x to 5. Otherwise (i.e. if $y \leq 0$) it sets x to 2. After that, the program sets y to the value of x decreased by 3. Finally, the program does an effectless operation by performing a `skip` statement.

The last construct $\texttt{while}(\varphi)\{C\}$ is a guarded while loop, that is executed as follows: If in the current program state σ the guard φ evaluates to false, then the whole loop immediately terminates without any effect. If on the other hand φ evaluates to true, then the loop body C is executed. After C has terminated in some state σ' (if C in fact terminates), the whole loop construct is invoked all over again but now starting from initial state σ': If φ evaluates to false in state σ', the loop terminates, otherwise C is executed to obtain a next state σ'', and so on. In principle the loop $\texttt{while}(\varphi)\{C\}$ is thus equivalent to the *infinitely long* (thus not well–formed) program

$$\texttt{if}\,(\varphi)\,\{C \fatsemi \texttt{if}\,(\varphi)\,\{C \fatsemi \cdots\}\,\texttt{else}\,\{\texttt{skip}\}\}\,\texttt{else}\,\{\texttt{skip}\}$$

which is an infinite nesting of simple conditional choices.

Notice that loops need not terminate on all initial states: Consider

$$\texttt{while}(x \neq 0)\{x := x - 1\},$$

which is a program that terminates only from those initial states σ in which $\sigma(x)$ is positive and moreover an integer.

EXAMPLE 2.3 (Deterministic GCL Programs):
Consider the following GCL program:

$$
\begin{array}{ll}
C_{2.3} \rhd & z := y \fatsemi \\
& \texttt{while}(x > 0)\{ \\
& \quad z := z + 1 \fatsemi \\
& \quad x := x - 1 \\
& \}
\end{array}
$$

$C_{2.3}$ first sets z to y and then, as long as x is strictly larger than 0, it repeats the following two steps: It adds 1 to z and subtracts 1 from x. These two steps (i.e. the effect of the loop body) will be repeated in total $\lceil x_0 \rceil$ times, where x_0 is the initial value of x.

Effectively this program therefore adds to y the rounded up value that x initially had and stores that result in variable z. As a (possibly unwanted) side–effect, the program also „forgets" about the value of x by effectively setting it to a value between 0 and -1.

2.2 REASONING ABOUT PREDICATES

We now develop formal reasoning about correctness of GCL programs at the level of predicates. For us, a predicate represents simply an arbitrary subset of program states, i.e. we can think of a predicate F as

$$F \in \mathcal{P}(\Sigma) ,$$

where for a set S we denote by $\mathcal{P}(S)$ its powerset. The predicate false stands for the empty set $\emptyset$ and dually true stands for the entire set Σ. Furthermore, $\neg F$ stands for the set $\Sigma \setminus F$, and $F \wedge G$ and $F \vee G$ stand respectively for the intersection and the union of the sets represented by F and G. We write

$$\sigma \models F$$

and say „σ satisfies F" to indicate that state σ is in the set represented by predicate F and $\sigma \not\models F$ to indicate that σ is not in that set. We write

$$F_1 \implies F_2$$

and say „F_1 implies F_2" to indicate that the set represented by predicate F_1 is a subset of the set represented by predicate F_2.

Our goal will be to associate to each program C and each predicate F (interpreted as a set of *final* states) a predicate G (interpreted as a set of *initial* states) such that if and only if the program C is started in any state $\sigma \models G$, then C terminates in a state $\tau \models F$. In the following we will gradually develop a calculus for achieving this.

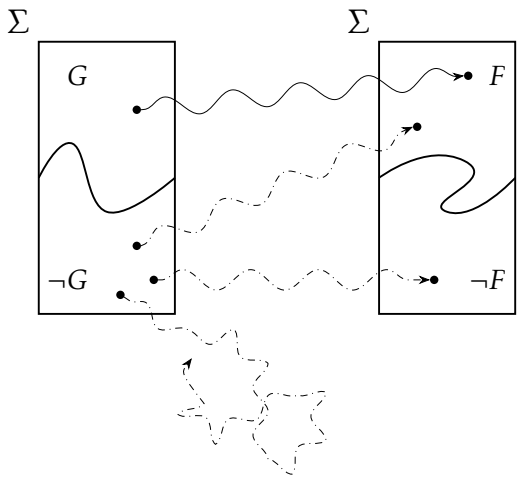


Figure 2.1: Hoare triple $\langle G \rangle\ C\ \langle F \rangle$ is valid: Starting in G, C will terminate in F. Starting in $\neg G$, we do not know whether C diverges or terminates in F or $\neg F$.

2.2.1 *Hoare Triples*

Hoare logic is a formal verification method seeded by the works of Robert (Bob) W Floyd[2] [Flo67a] and later developed further by Charles Antony (Tony) Richard Hoare [Hoa69]. It is therefore also called Floyd–Hoare logic.

The crucial concept of this technique are *Hoare triples*: Given two predicates F and G and a program C, a Hoare triple $\langle G \rangle\ C\ \langle F \rangle$ is said to be *valid* iff the following holds:

> If program C is started in some initial state $\sigma \models G$,
> then C *terminates* in a final state $\tau \models F$.

Notice our notion of validity means „valid for total correctness“[3] as presented by Manna & Pnueli [MP74]. In a more diagrammatic style, the situation is depicted in Figure 2.1. We call F a *postcondition* since we interpret it as a predicate over final states, i.e. F shall hold after (post) the execution of C. Dually, we call G a *precondition* since we interpret it as a predicate over initial states, i.e. G shall hold before (pre) the execution of C.

There are two things we would like to emphasize here: First, the validity of $\langle G \rangle\ C\ \langle F \rangle$ still does not tell us *anything* about what happens when C is

2 Floyd's middle name is in fact just W: „[Floyd] was indeed born with another middle name, but he had it legally changed to ‚W'—just as President Truman's middle name was simply ‚S'. Bob liked to point out that ‚W.' is a valid abbreviation for ‚W'.“ [Hai04]

3 As opposed to „valid for partial correctness“.

executed on some initial state $\sigma \not\models G$. In particular, it might be the case that C will terminate in a final state $\tau \models F$ nonetheless. This will be different for the notion of weakest preconditions.

Secondly, if $\langle G \rangle\ C\ \langle F \rangle$ is valid for some F, then it is *guaranteed that C terminates* from any state $\sigma \models G$. This means in particular that the validity of $\langle G \rangle\ C\ \langle \mathsf{true} \rangle$ simply states that C terminates from every state $\sigma \models G$, but it does not tell us anything about the final state τ, since $\forall \tau\colon\ \tau \models \mathsf{true}$.

2.2.2 *Weakest Preconditions*

To circumvent the dissatisfactory situation that validity of $\langle G \rangle\ C\ \langle F \rangle$ gives no information about the states satisfying $\neg G$ we now introduce the notion of weakest preconditions [Dij75; Dij76]. Imagine for that a Hoare triple

$$\langle \ldots\ldots \rangle\ C\ \langle F \rangle\ ,$$

where the precondition is left open just like in a cloze — so to speak: a Hoare triple with a *blank*. We would like to fill this blank in a very general way, namely with the *weakest possible predicate* G in the following sense: Any predicate G' for which $\langle G' \rangle\ C\ \langle F \rangle$ is valid should imply the more general (weaker) predicate G. Put more formally:

$$\forall G'\colon \qquad G' \implies G \quad \text{iff} \quad \langle G' \rangle\, C\, \langle F \rangle \text{ is valid} \tag{2.1}$$

Given program C and postcondition F, we call the (unique) predicate G that satisfies Condition (2.1) the

weakest precondition of C with respect to postcondition F,

denoted as $\mathsf{wp}\,[\![C]\!]\,(F)$. As a diagram, the situation is depicted in Figure 2.2: From any state $\sigma \models G$ the program C terminates in some state $\tau \models F$. Moreover, if C is started in a state $\sigma \not\models G$, then

- either C terminates in a state $\tau \not\models F$,
- or C does not terminate at all.

It is easy to see that the Hoare triple $\langle \mathsf{wp}\,[\![C]\!]\,(F) \rangle\ C\ \langle F \rangle$ is always valid and moreover that the following holds:

$$\langle G' \rangle\, C\, \langle F \rangle \text{ is valid} \qquad \text{iff} \qquad G' \implies \mathsf{wp}\,[\![C]\!]\,(F)$$

We can also see that the situation has now changed in comparison to Hoare triples in the sense that executing C on initial state $\sigma \not\models \mathsf{wp}\,[\![C]\!]\,(F)$ will *definitely not* terminate in a state $\tau \models F$.

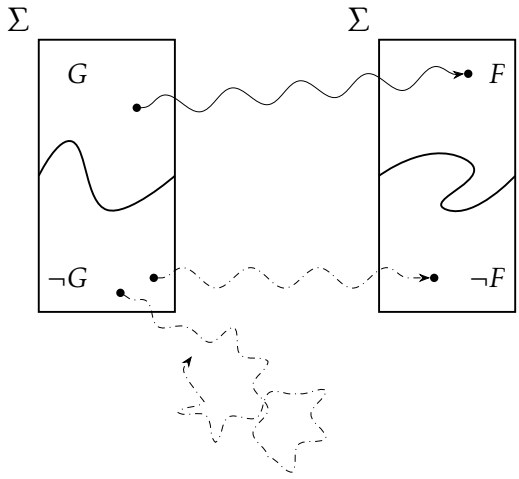


Figure 2.2: G is the weakest precondition of C with respect to postcondition F: Starting in G, C terminates in F. Starting in $\neg G$, C either diverges or terminates in $\neg F$.

2.2.3 *The Weakest Precondition Calculus*

Though we have defined what weakest preconditions are, given a program and a postcondition we yet have no *method* of finding out what the respective weakest precondition is. In the following we will therefore show how to obtain weakest preconditions in a systematic way, namely with the aid of a *backward moving continuation–passing style weakest precondition transformer*.

2.2.3.1 *Continuation–passing*

The principle of a continuation–passing style transformer is depicted in Figure 2.3: Assume we want to know the weakest precondition of the composed program $C_1 \fatsemi C_2$ with respect to postcondition F. Then we start from the end of $C_1 \fatsemi C_2$ with continuation F and move *backward* to the position between C_1 and C_2. While moving that position, we also transition from F to the weakest precondition of C_2 with respect to F, i.e. to $\mathrm{wp}\,[\![C_2]\!]\,(F)$.

Let us denote by G the intermediate predicate $\mathrm{wp}\,[\![C_2]\!]\,(F)$. Then G represents by definition exactly those states from which execution of C_2 will terminate in F. Therefore, we want precisely G to be *the* postcondition that the execution of C_1 should terminate in, so that the execution of the entire program $C_1 \fatsemi C_2$ terminates in F.

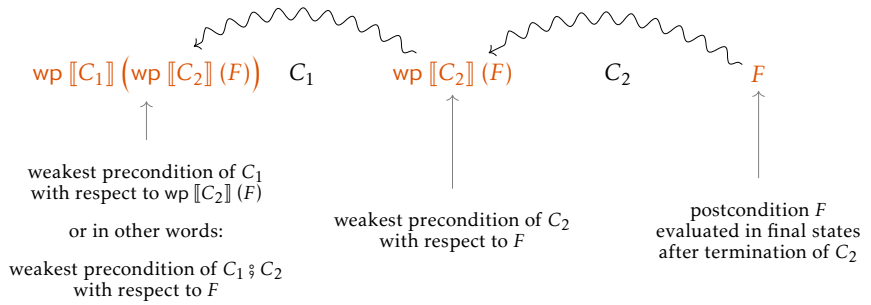


Figure 2.3: Continuation–passing style weakest precondition transformer.

To the program C_1 the predicate G is of course just an arbitrary continuation as any other. We can therefore determine the weakest precondition of C_1 with respect to postcondition G by moving to the beginning of program C_1 while transitioning from G to $\mathsf{wp}\,[\![C_1]\!]\,(G)$. But since $G = \mathsf{wp}\,[\![C_2]\!]\,(F)$, we have $\mathsf{wp}\,[\![C_1]\!]\,(G) = \mathsf{wp}\,[\![C_1]\!]\,(\mathsf{wp}\,[\![C_2]\!]\,(F))$, and therefore we have effectively moved from the end of program $C_1 \fatsemi C_2$ to its beginning while transitioning from F to the weakest precondition of $C_1 \fatsemi C_2$ with respect to F.

Now that we have some familiarity with the basic concept of continuation–passing, we will use it to give precise rules on how to obtain weakest preconditions for any program with respect to any postcondition. It turns out that this can be done in a very structured way, namely by induction on the structure of the programs: For every program C, we will show how to construct a continuation–passing style transformer $\mathsf{wp}\,[\![C]\!]$ of type

$$\mathsf{wp}\,[\![C]\!]\colon \quad \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)\ ,$$

that takes as input a postcondition F and returns the weakest precondition of C with respect to postcondition F. We first show how to construct these transformers for loop–free programs and then proceed with loops.

2.2.3.2 *Weakest Preconditions of Loop–free Programs*

The rules for constructing the wp transformer are given in Table 2.1. Let us ignore the definition for while loops for the time being and let us go over the other definitions one by one: Since the program `skip` has no effect, the postcondition F has to be transformed to the very same precondition F, i.e.

$$\mathsf{wp}\,[\![\texttt{skip}]\!]\,(F) \;=\; F\ .$$

C	$\mathsf{wp}\,[\![C]\!]\,(F)$
`skip`	F
`diverge`	false
$x := E$	$F\,[x/E]$
$C_1 \,;\, C_2$	$\mathsf{wp}\,[\![C_1]\!]\,\big(\mathsf{wp}\,[\![C_2]\!]\,(F)\big)$
`if` $(\varphi)\,\{C_1\}$ `else` $\{C_2\}$	$\big(\varphi \wedge \mathsf{wp}\,[\![C_1]\!]\,(F)\big) \vee \big(\neg\varphi \wedge \mathsf{wp}\,[\![C_2]\!]\,(F)\big)$
`while`$(\varphi)\{C'\}$	$\mathsf{lfp}\,X.\,\big(\neg\varphi \wedge F\big) \vee \big(\varphi \wedge \mathsf{wp}\,[\![C']\!]\,(X)\big)$

Table 2.1: The weakest precondition transformer acting on predicates.

For the assignment $x := E$, we essentially *replace every occurrence of x in F by E*. However, since E does not actually *occur* in F, we define

$$\mathsf{wp}\,[\![x := E]\!]\,(F) \;=\; F\,[x/E]\,,$$

where $F\,[x/E]$ is a predicate whose indicator function is given by

$$\lambda\sigma.\;[F](\sigma\,[x \mapsto \sigma(E)])\,.$$

Notice that we have again used the Iverson bracket notation $[F]$ above.

Example 2.4 (Weakest Preconditions of Assignments):

A. $\mathsf{wp}\,[\![x := 5]\!]\,(x \leq 0) \;=\; (5 \leq 0) \;=\; \mathsf{false}$

B. $\mathsf{wp}\,[\![z := 18]\!]\,(x = 0) \;=\; (x = 0)$

C. $$\begin{aligned}\mathsf{wp}\,[\![x := 5]\!]\,((x > 2) \to (y = 7)) \;&=\; (5 > 2) \to (y = 7)\\ &=\; \mathsf{true} \to (y = 7)\\ &=\; (y = 7)\end{aligned}$$

In the predicates above, the symbol $\to$ (logical implication) is the usual abbreviation for $\neg A \vee B$. Notice that $\to$ is syntactic construct while $\Longrightarrow$ is a semantic one: $A \to B$ is *one predicate* while $A \implies B$ is a *statement concerning the two predicates A and B*.

Next, we turn to sequential composition: We have already seen the principle of continuation–passing and this very principle is implemented in the def-

inition of the transformer for sequential composition: Given the two transformers $\mathsf{wp}\,[\![C_1]\!]$ and $\mathsf{wp}\,[\![C_2]\!]$, we define

$$\mathsf{wp}\,[\![C_1 \,;\, C_2]\!]\,(F) \;=\; \mathsf{wp}\,[\![C_1]\!]\,(\mathsf{wp}\,[\![C_2]\!]\,(F))\;.$$

The intuition for the conditional choice $\texttt{if}\,(\varphi)\,\{C_1\}\,\texttt{else}\,\{C_2\}$ is the following: If in the initial state σ, the guard φ is satisfied, then C_1 will be executed. We thus need to associate with that case the weakest precondition of C_1 with respect to F. As a predicate, this reads as[4]

$$\varphi \to \mathsf{wp}\,[\![C_1]\!]\,(F)\;.$$

Dually, if we have $\sigma \not\models \varphi$, then C_2 is executed and we thus need to associate with that case the weakest precondition of C_2. As a predicate, this reads as

$$\neg\varphi \to \mathsf{wp}\,[\![C_2]\!]\,(F)\;.$$

We can now express that both cases (the case for $\sigma \models G$ and the case for $\sigma \not\models G$) must hold true in one predicate, namely by

$$(\varphi \to \mathsf{wp}\,[\![C_1]\!]\,(F)) \;\wedge\; (\neg\varphi \to \mathsf{wp}\,[\![C_2]\!]\,(F))\;,$$

which is logically equivalent to

$$(\varphi \wedge \mathsf{wp}\,[\![C_1]\!]\,(F)) \;\vee\; (\neg\varphi \wedge \mathsf{wp}\,[\![C_2]\!]\,(F))\;.$$

We choose the latter over the former representation in Table 2.1 because we will later associate $\wedge$ with $\cdot$ and $\vee$ with $+$, whereas an arithmetic representation of $\to$ is more cluttered and inconvenient.[5]

Before we turn our attention to the definition of wp for while loops, let us take a look at how we can formally reason about the program from Example 2.2 on page 25 by using the wp transformer:

EXAMPLE 2.5 (Weakest Preconditions of Loop–Free Programs):
We will reconsider the program $C_{2.2}$ from Example 2.2 and reason about the set of initial states from which the execution of $C_{2.2}$ terminates in a state satisfying postcondition $y^2 > 2$.

Throughout this thesis, we will use the notation

$$\begin{array}{l} /\!/\!/\ \boldsymbol{G'} \\ /\!/\!/\ \boldsymbol{G} \\ C \\ /\!/\!/\ \boldsymbol{F} \end{array}$$

4 As usually, $\neg$ binds stronger than $\to$.
5 Namely $\alpha \to \beta$ would need to be associated with $(1-\alpha)+\beta$.

to express the fact that $G = \mathsf{wp}\,[\![C]\!]\,(F)$ and moreover that G' is logically equivalent to G. It is thus more intuitive to read annotated programs from bottom to top, just like the wp transformer moves from the back to the front. Using this notation, we can annotate the program $C_{2.2}$ simply by applying the wp rules from Table 2.1 as shown in Figure 2.4.

By these annotations, we have established $\mathsf{wp}\,[\![C_{2.2}]\!]\,\big(y^2 > 0\big) = (y > 0)$. This tells us that from any initial state in which y is larger than 0 the execution of $C_{2.2}$ terminates in some final state τ in which y^2 is larger than 2.

Notice that $y > 0$ and $y^2 > 2$ are evaluated in different states, namely in initial and final states, respectively.

2.2.3.3 *Weakest Preconditions of Loops*

We now study weakest preconditions of loops. For the freezing program `diverge`, notice that for any postcondition F there is no initial state σ from which `diverge` terminates in some final state $\tau \models F$ (because `diverge` does not terminate at all). So whatever precondition we assign to `diverge` with respect to F, it may not be satisfiable by any σ. Therefore, the weakest precondition of `diverge` with respect to any postcondition F must be defined as

$$\mathsf{wp}\,[\![\texttt{diverge}]\!]\,(F) \;=\; \mathsf{false}\,,$$

since $\forall \sigma\colon \sigma \not\models \mathsf{false}$.

If we take a look at the definition of wp for while loops in Table 2.1, we see that it is defined using a *least fixed point operator* (lfp), namely as

$$\mathsf{lfp}\; X.\; \underbrace{(\neg\varphi \wedge F) \vee (\varphi \wedge \mathsf{wp}\,[\![C']\!]\,(X))}_{=:\,\Phi(X)}\,,$$

by which we mean the least fixed point of the characteristic function $\Phi(X)$. This function is of type $\mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$, thus mapping predicates to predicates. A fixed point of Φ is a predicate G such that $\Phi(G) = G$. But in what sense can the fixed point be the *least* one? For that, we need to introduce some notion of order on the set of predicates, i.e. on $\mathcal{P}(\Sigma)$. More concretely, $\Longrightarrow$ induces a complete lattice (see Definition A.1) on $\mathcal{P}(\Sigma)$, i.e.

$$F_1 \text{ „is smaller than or equal to“ } F_2 \qquad \text{iff} \qquad F_1 \Longrightarrow F_2\,.$$

The least element of the *complete lattice* $(\mathcal{P}(\Sigma), \Longrightarrow)$ is false. The supremum of a chain $S \subseteq \mathcal{P}(\Sigma)$ is given by

$$\sup S \;=\; \bigvee_{F \in S} F\,,$$

```
/// y > 0
/// (y > 0 ∧ true) ∨ (y ≤ 0 ∧ false)
if (y > 0) {
    /// true
    /// (5 − 3)^2 > 2
    x := 5
    /// (x − 3)^2 > 2
}else{
    /// false
    /// (2 − 3)^2 > 2
    x := 2
    /// (x − 3)^2 > 2
};
/// (x − 3)^2 > 2
y := x − 3;
/// y^2 > 2
skip
/// y^2 > 2
```

Figure 2.4: Weakest precondition annotations for Example 2.5.

which is the predicate that corresponds to the union of all sets corresponding to the predicates in S.

One can now show that Φ is a continuous function (Definition A.2) and we thus know by the Kleene fixed point theorem (Theorem A.5) that Φ has a least fixed point, given by

$$\mathsf{lfp}\,\Phi \;=\; \sup_{n\in\mathbb{N}} \Phi^n(\mathsf{false})\,.$$

By the existence of the least fixed point, we have ensured that wp for while loops is well–defined.

The evaluation of $\Phi^0(\mathsf{false})$, $\Phi^1(\mathsf{false})$, $\Phi^2(\mathsf{false})$, ... is called a *fixed point iteration* and we call $\Phi^i(\mathsf{false})$ the i-th iteration or i-th step of that fixed point iteration. A very important fact about the fixed point iteration of continuous functions is that, if started from the least element of the underlying complete lattice, it converges *monotonically* to the least fixed point, meaning that in our particular case we have an ascending chain

$$\Phi^0(\mathsf{false}) \implies \Phi^1(\mathsf{false}) \implies \Phi^2(\mathsf{false}) \implies \Phi^3(\mathsf{false}) \implies \ldots\,.$$

This follows by induction from continuity of Φ which implies monotonicity of Φ (see Definition A.3 and A.4). For the base case, we have

$$\Phi^0(\mathsf{false}) \;=\; \mathsf{false} \implies \Phi(\mathsf{false})$$

trivially, since false implies anything. Then, by monotonicity, we can perform the induction step. Assuming $\Phi^n(\mathsf{false}) \implies \Phi^{n+1}(0)$, we get

$$\Phi^{n+1}(\mathsf{false}) \implies \Phi^{n+2}(0)$$

by monotonicity of Φ. Let us revisit Example 2.3, and reason about a possibly unwanted side–effect of that program (setting x to 0) using the wp calculus.

Example 2.6 (Weakest Preconditions of Loops):
Reconsider the program $C_{2.3}$ from Example 2.3:

$C_{2.3} \rhd$
```
z := y;
while(x > 0){
    z := z + 1;
    x := x − 1
}
```

We would like to reason about whether the program sets x exactly to 0, i.e. about postcondition $x = 0$. The characteristic function of the while loop with respect to postcondition $x = 0$ is given by

$$\begin{aligned}\Phi(X) &= (x \leq 0 \land x = 0) \lor (x > 0 \land \mathsf{wp}\,[\![z := z+1 \fatsemi x := x-1]\!]\,(X)) \\ &= (x = 0) \lor (x > 0 \land \mathsf{wp}\,[\![z := z+1 \fatsemi x := x-1]\!]\,(X))\,.\end{aligned}$$

Let us do the first three iterations of the fixed point iteration for Φ:

$$\begin{aligned}\Phi^0(\mathsf{false}) &= \mathsf{false} \\ \Phi^1(\mathsf{false}) &= (x = 0) \\ \Phi^2(\mathsf{false}) &= (x = 0) \lor (x = 1) \\ \Phi^3(\mathsf{false}) &= (x = 0) \lor (x = 1) \lor (x = 2)\end{aligned}$$

Detailed calculations can be found in Appendix C.1 on page 323.

After three iterations, we can already start seeing a pattern for $n > 1$:

$$\Phi^n(\mathsf{false}) = (x = 0) \lor (x = 1) \lor \ldots \lor (x = n-1) = \bigvee_{i=0}^{n-1} (x = i)$$

We could prove this pattern correct by induction on n, which we however omit here. The above fixed point iteration will converge to the precondition

$$\sup_{n \in \mathbb{N}} \Phi^n(\mathsf{false}) = \bigvee_{i=0}^{\omega} (x = i) = (x \in \mathbb{N})\,,$$

and thus

$$\mathsf{wp}\,[\![\mathtt{while}\,(x > 0)\,\{z := z+1 \fatsemi x := x-1\}]\!]\,(x = 0) = (x \in \mathbb{N})\,. \tag{2.2}$$

For the whole program, we can finally make these annotations:

$$\begin{aligned}&/\!/\!/\ x \in \mathbb{N} \\ &z := y \fatsemi \\ &/\!/\!/\ x \in \mathbb{N} \qquad \text{(by Equation 2.2)} \\ &\mathtt{while}\,(x > 0)\,\{ \\ &\quad z := z+1 \fatsemi \\ &\quad x := x-1\ \} \\ &/\!/\!/\ x = 0\end{aligned}$$

We have thus proven that from all initial states where x is a natural number, the program sets x to 0.

While the above reasoning about the while loop was more or less ad–hoc, formal reasoning about such fixed points in a systematic way is one of the most difficult tasks in program verification. In general, this is not automatable, as

this would contradict Rice's Theorem [Ric53] and therefore ultimately contradict the undecidability of the Halting Problem [Chu36; Tur37]. We show how to reason about loops in a possibly more automatable way in Chapter 5.

2.2.4 *Reasoning about Nondeterminism*

So far, all GCL constructs were of deterministic nature: Given an initial state, the behavior of the program was completely determined. We will now introduce some notion of uncertainty into our GCL programming language: the *nondeterministic choice* construct

$$\{C_1\} \square \{C_2\}\,.$$

We call programs that contain such nondeterministic choices *nondeterministic programs*. Analogously, we call programs that do not contain any nondeterministic choice constructs *deterministic programs*. The concept of programs containing nondeterministic choices was already present in Dijkstra's original weakest precondition calculus [Dij75], although the idea of „nondeterministic algorithms" is due to Floyd and dates back further [Flo67b]. Even earlier, as a precursor to nondeterministic programs, Rabin & Scott introduced nondeterministic finite automata [RS59].

As for the semantics of nondeterministic choice, the program $\{C_1\} \square \{C_2\}$ executes *either* C_1 *or* C_2. Both scenarios are possible and we simply have no information on which branch is going to be executed. In particular, we would like to stress that it is *not meaningful to associate a probability* to either executing C_1 or C_2. Especially assigning the probability of $1/2$ to either possibility is only seemingly self–evident, but not meaningful. Nondeterministic choice is thus a *possibilistic*, not a probabilistic construct. For semantics of nondeterministic and possibilistic programs based on *possibility theory* instead of probability theory, see [CW08; WC12; WC11].

Example 2.7 (A Nondeterministic Loop–Free GCL Program):
Consider the following program that extends Example 2.2 on page 25:

$$\begin{aligned} C_{2.7} \rhd \quad & \{y := 1\} \square \{y := y - 1\}\fatsemi \\ & \texttt{if}\,(y > 0)\,\{x := 5\}\,\texttt{else}\,\{x := 2\} \\ & y := x - 3\fatsemi \\ & \texttt{skip} \end{aligned}$$

This program first nondeterministically either sets y to 1 or decreases y by 1. Then it performs the same steps as the program from Example 2.2, starting with the check for $y > 0$. Notice that this check can either evaluate to true, i.e. in case that the left branch of the nondeterministic choice was executed

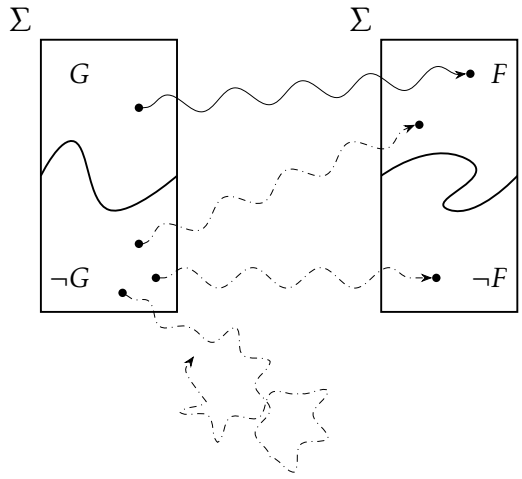


Figure 2.5: G is the weakest precondition of nondeterministic program C with respect to postcondition F: Starting in G, C will terminate in F. Starting in $\neg G$, we cannot guarantee anything about the computation of C.

and y was set to 1, or it can depend on the initial value of y, i.e. in case that the right branch was executed.

Let us now look at weakest preconditions of nondeterministic choices: Recall that we are interested in a precondition that *guarantees* both termination and establishment of the postcondition F. In order to guarantee this regardless whether C_1 or C_2 is executed, the weakest precondition of $\{C_1\} \square \{C_2\}$ with respect to F must be a weakest precondition of *both* C_1 *and* C_2 with respect to F. The weakest precondition transformer for $\{C_1\} \square \{C_2\}$ is thus given by

$$\mathsf{wp}\,[\![\{C_1\} \square \{C_2\}]\!]\,(F) \;=\; \mathsf{wp}\,[\![C_1]\!]\,(F) \;\wedge\; \mathsf{wp}\,[\![C_2]\!]\,(F)\;.$$

We might recall at this point that for a deterministic program C we could make the following statement:

> If $\sigma \not\models \mathsf{wp}\,[\![C]\!]\,(F)$ then we know that executing C on state σ will *definitely not* terminate in a state $\tau \models F$.

For a nondeterministic C, however, the statement must be:

> If $\sigma \not\models \mathsf{wp}\,[\![C]\!]\,(F)$ then it is *not guaranteed* that executing C on state σ will terminate in a state $\tau \models F$.

The situation is depicted in Figure 2.5. Notice that this diagram is *the same* as the one in Figure 2.1. There is a hidden difference in the possible compu-

tation starting in $\neg G$ and terminating in F, though: For Hoare triples with deterministic program C, this possible path stems from the fact that validity of the Hoare triple $\langle G \rangle\ C\ \langle F \rangle$ is too weak a statement to exclude this path. For weakest preconditions of nondeterministic programs on the other hand, the path from $\neg G$ to F is instead due to the nondeterminism of C: The path from $\neg G$ to F might actually be a possible computation of C. However: computations starting from $\neg G$ are *not guaranteed* to terminate in F.

Example 2.8 (Weakest Preconditions and Nondeterminism):
We will reconsider the program $C_{2.7}$ from Example 2.7 and again, as in Example 2.2, reason about postcondition $y^2 > 2$. Using the annotation style from earlier, we annotate $C_{2.7}$ as shown in Figure 2.6. By these annotations, we establish $\text{wp}\,[\![C_{2.7}]\!]\left(y^2 > 2\right) = (y > 1)$. This means that from any initial state in which y is larger than 1, it is guaranteed that execution of $C_{2.7}$ will terminate in a state in which y^2 is larger than 2.

Notice that even from a state in which $y \leq 1$ it is still possible that the program terminates in a state satisfying $y^2 > 2$, namely if in the nondeterministic choice the left branch $y := 1$ is executed. However, this is *not guaranteed*. This situation is reflected exactly by the path from $\neg G$ to F in Figure 2.5, when instantiating G with $y > 1$ and F with $y^2 > 2$.

2.2.5 *Weakest Liberal Preconditions*

We have already encountered the phenomenon that certain programs do not terminate from certain initial states. For instance, the program

$$\texttt{while}(x \neq 0)\{x := x - 1\}$$

terminates only on initial states where $x \in \mathbb{N}$. Our notion of weakest preconditions, however, captures only the fact that a program terminates in a state satisfying a given postcondition. Sometimes (e.g. later in this thesis), it is necessary, though, to reason about partial correctness, namely that a postcondition has to be satisfied only in case that the program terminates (but termination itself is not guaranteed).

2.2.5.1 *The Notion of Weakest Liberal Preconditions*

The type of reasoning we require here can be carried out using the notion of *weakest liberal preconditions*: Given a program C and a postcondition F, we call the (unique) predicate G the

weakest liberal precondition of C with respect to F,

```
/// y > 1
/// true ∧ y > 1
{
    /// true
    /// 1 > 0
    y := 1
    /// y > 0
} □ {
    /// y > 1
    /// y − 1 > 0
    y := y − 1
    /// y > 0
};
/// y > 0                                          (see Example 2.5)
if (y > 0) {x := 5} else {x := 2};
y := x − 3;
skip
/// y² > 2
```

Figure 2.6: Weakest precondition annotations for Example 2.8.

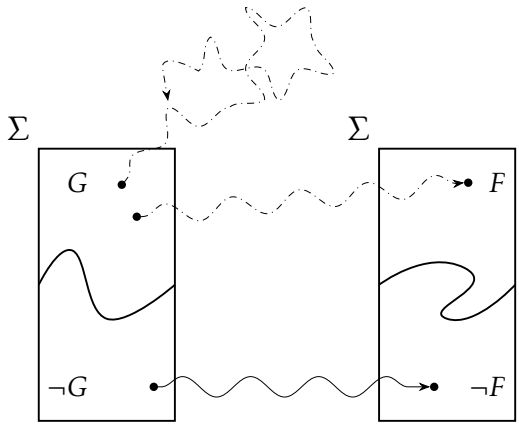


Figure 2.7: G is the weakest liberal precondition of deterministic program C with respect to postcondition F: Starting in G, C cannot terminate in $\neg F$, i.e. it will either diverge or terminate in F. Starting in $\neg G$, C will terminate in $\neg F$.

denoted $\mathsf{wlp}\,[\![C]\!]\,(F)$, if it satisfies the following: From any state $\sigma \models G$

- either C terminates in a state $\tau \models F$,
- or C does not terminate at all.

Moreover, if C is started in a state $\sigma \not\models G$, then C terminates in a state $\tau \not\models F$. As a diagram, the situation is depicted Figure 2.7 for deterministic programs and in Figure 2.8 for nondeterminstic programs. The difference is only in those computations starting from $\neg G$: For the possible computation path from $\neg G$ to F, recall the explanations on page 38. The possible diverging path emanating from $\neg G$ is also caused by the nondeterminism of the program: It is possible that C diverges from $\neg G$ but it is not guaranteed.

If F is some correctness property and we can prove the above, then we say that C is *partially correct*, whereas if we additionally require termination (as it was the case with weakest preconditions) we say that C is *totally correct*. In that terminology, weakest preconditions are suited for reasoning about total correctness whereas weakest liberal preconditions are suited for reasoning about partial correctness.

2.2.5.2 *The Weakest Liberal Precondition Calculus*

We now show how to obtain weakest liberal preconditions in a way similar to the weakest precondition transformer, namely by a backward moving

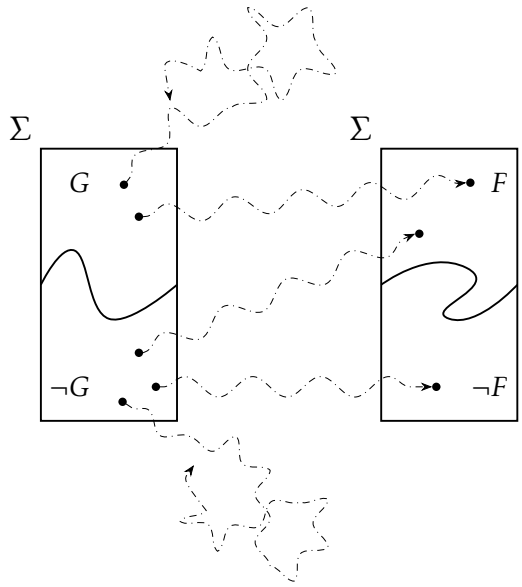


Figure 2.8: G is the weakest liberal precondition of nondeterministic program C with respect to postcondition F: Starting in G, C will not terminate in $\neg F$.

continuation–passing style weakest liberal precondition transformer.

Weakest Liberal Preconditions of Loop–Free Programs. The rules for constructing the wlp transformer are given in Table 2.2. Let us again ignore the definition for while loops for the time being and inspect the remaining rules: For the atomic programs `skip` and $x := E$ the rules are exactly the same.

For the remaining loop–free programs, the definitions differ only in the fact that the right hand sides use wlp instead of wp on subprograms. From that observation, we can easily conclude that wlp and wp coincide for any loop–free program, i.e.

$$\forall \text{ loop–free } C \ \forall F\colon \quad \mathsf{wlp}\,[\![C]\!]\,(F) \;=\; \mathsf{wp}\,[\![C]\!]\,(F)\ .$$

This does not only make sense when looking at the formal definitions, but it also makes sense intuitively: Differences between wp and wlp occur only for nontermination, but this cannot occur in loop–free programs.[6]

Weakest Liberal Preconditions of Loops. We now turn towards weakest liberal preconditions of loops. As for `diverge`, consider the following: Ac-

6 Recall that programs containing `diverge` are *not* loop–free.

C	$\mathsf{wlp}\,[\![C]\!]\,(F)$
`skip`	F
`diverge`	true
$x := E$	$F[x/E]$
$C_1 \fatsemi C_2$	$\mathsf{wlp}\,[\![C_1]\!]\,\big(\mathsf{wlp}\,[\![C_2]\!]\,(F)\big)$
`if` $(\varphi)\,\{C_1\}$ `else` $\{C_2\}$	$\big(\varphi \wedge \mathsf{wlp}\,[\![C_1]\!]\,(F)\big) \vee \big(\neg\varphi \wedge \mathsf{wlp}\,[\![C_2]\!]\,(F)\big)$
$\{C_1\} \,\square\, \{C_2\}$	$\mathsf{wlp}\,[\![C_1]\!]\,(F) \;\wedge\; \mathsf{wlp}\,[\![C_2]\!]\,(F)$
`while`$(\varphi)\{C'\}$	$\mathsf{gfp}\; X.\, \big(\neg\varphi \wedge F\big) \vee \big(\varphi \wedge \mathsf{wlp}\,[\![C']\!]\,(X)\big)$

Table 2.2: The weakest liberal precondition transformer.

cording to the definition of weakest liberal preconditions, the weakest liberal precondition of `diverge` with respect to postcondition F must be a predicate such that either `diverge` terminates in a state $\tau \models F$ (however, `diverge` never terminates, so this is never the case), or `diverge` does not terminate (this is always the case). Therefore, the weakest liberal precondition of `diverge` with respect to *any* postcondition can only be true.

Dually to weakest preconditions, we see in Table 2.2 that the weakest liberal precondition of a while loop is defined using a *greatest fixed point operator* (gfp) instead of a least one as

$$\mathsf{gfp}\; X.\, \underbrace{(\neg\varphi \wedge F) \vee (\varphi \wedge \mathsf{wlp}\,[\![C']\!]\,(X))}_{=:\,\Phi(X)},$$

i.e. the greatest fixed point of the characteristic function $\Phi(X)$.

Dually to least elements and suprema, a complete lattice also always has a greatest element and every subset also has an infimum. The greatest element of the complete lattice $(\mathcal{P}(\Sigma),\ \Longrightarrow)$ is true. The infimum of a subset $S \subseteq \mathcal{P}(\Sigma)$ is given by

$$\inf S \;=\; \bigwedge_{F \in S} F\,,$$

which is the predicate that corresponds to the intersection of all sets corresponding to the predicates in S.

One can now show that Φ is continuous and we thus know by the Kleene fixed point theorem (A.5) that Φ has a greatest fixed point, given by

$$\mathsf{gfp}\,\Phi \;=\; \inf_{n \in \mathbb{N}} \Phi^n(\mathsf{true})\,,$$

and therefore wlp for while loops is well–defined.

Dually to the situation with least fixed points, a very important fact about the fixed point iteration of continuous functions is that, if started from the *greatest* element of the underlying complete lattice, it converges *monotonically* to the *greatest fixed point*. This means in our particular case that we have a *descending* chain

$$\Phi^0(\mathsf{true}) \Longleftarrow \Phi^1(\mathsf{true}) \Longleftarrow \Phi^2(\mathsf{true}) \Longleftarrow \dots .$$

This fact also follows from the monotonicity of Φ which is implied by its continuity (see A.4).

We can now reconsider `diverge` from a gfp point of view. Recall that `diverge` is a shorthand for `while(true){skip}`. Then the characteristic functional with respect to any postcondition F is given by

$$\Phi(X) \;=\; (\mathsf{false} \wedge F) \;\vee\; (\mathsf{true} \wedge \mathsf{wp}\,[\![\mathtt{skip}]\!]\,(X)) \;=\; X\,,$$

i.e. the identity function on $\mathcal{P}(\Sigma)$. Its largest fixed point is obviously the largest element of $\mathcal{P}(\Sigma)$, namely true. Therefore,

$$\mathsf{wp}\,[\![\mathtt{diverge}]\!]\,(F) \;=\; \mathsf{wp}\,[\![\mathtt{while(true)\{skip\}}]\!]\,(F) \;=\; \mathsf{true}$$

Let us again revisit an example and prove the partial correctness of a program using the wlp calculus:

Example 2.9 (Weakest Liberal Preconditions):
Reconsider the following program from earlier in this section:

$$C_{2.9} \rhd \qquad \mathtt{while}(x \neq 0)\{x := x - 1\}$$

We would like to reason about the fact that if $C_{2.9}$ terminates, it sets x to 0. We can do so by reasoning about the weakest liberal precondition of $C_{2.9}$ with respect to postcondition $x = 0$. The characteristic function of the loop with respect to postcondition $x = 0$ is given by

$$\begin{aligned}\Phi(X) \;&=\; (x = 0 \wedge x = 0) \;\vee\; (x \neq 0 \wedge \mathsf{wp}\,[\![x := x - 1]\!]\,(X)) \\ &=\; (x = 0) \;\vee\; (x \neq 0 \wedge \mathsf{wp}\,[\![x := x - 1]\!]\,(X))\,.\end{aligned}$$

Let us perform the fixed point iteration for Φ (for wlp the fixed point iteration for the greatest fixed point goes true, $\Phi(\mathsf{true})$, $\Phi^2(\mathsf{true})$, $\Phi^3(\mathsf{true})$, … instead of false, $\Phi(\mathsf{false})$, $\Phi^2(\mathsf{false})$, $\Phi^3(\mathsf{false})$, … as for wp):

$$\begin{aligned}\Phi(\mathsf{true}) \;&=\; (x = 0) \;\vee\; (x \neq 0 \wedge \mathsf{wp}\,[\![x := x - 1]\!]\,(\mathsf{true})) \\ &=\; (x = 0) \;\vee\; (x \neq 0 \wedge \mathsf{true}) \\ &=\; (x = 0) \;\vee\; (x \neq 0)\end{aligned}$$

$= \mathsf{true}$

We see that after only one iteration we have reached a fixed point. By monotonicity of Φ, this is the greatest fixed point and we hence have

$$\mathsf{wlp}\,[\![\mathtt{while}\,(x \neq 0)\,\{x := x-1\}]\!]\,(x=0) \;=\; \mathsf{true}\,.$$

We have thus formally proven the partial correctness property that from *all* initial states the program $C_{2.9}$ sets x to 0 if it terminates.

2.3 REASONING ABOUT VALUES

Up until now, we have recapped Dijkstra's original weakest precondition calculus which enables reasoning at the level of predicates over program states. We will now see how to take this method of reasoning beyond the level of predicates to more general functions.

Recall for this purpose our notion of predicates: We have identified a predicate F with a subset of program states, i.e. $F \in \mathcal{P}(\Sigma)$. We also introduced the Iverson bracket $[F]$ which is the indicator function of F and is of type $\Sigma \to \{0,1\}$. It is obvious that, *in principle*, predicates $F \in \mathcal{P}(\Sigma)$ and their indicator functions $[F] : \Sigma \to \{0,1\}$ are the same.

As a first step to go beyond predicates, we reformulate Dijkstra's weakest precondition calculus in terms of indicator functions, i.e. functions f of type $\Sigma \to \{0,1\}$. The resulting definitions are given in Table 2.3. Let us go exemplarily over the rules for assignment, conditional, and nondeterministic choice: For the assignment, we have

$$\mathsf{wp}\,[\![x := E]\!]\,(f) \;=\; f\,[x/E]\,,$$

where $f\,[x/E]$ is defined analogously to the case for predicates as

$$f\,[x/E] \;=\; \lambda\sigma.\, f(x[x \mapsto \sigma(E)])\,.$$

This mimics the definition of wp of assignments for predicates.

For `diverge`, we have

$$\mathsf{wp}\,[\![\mathtt{diverge}]\!]\,(f) \;=\; 0\,,$$

This mimics the definition of wp of divergence for predicates since $0 = [\mathsf{false}]$.

For the conditional choice we have

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

where $\cdot$ and $+$ are to be understood pointwise, i.e.

$$f_1 \cdot f_2 \;=\; \lambda\sigma.\, f_1(\sigma)\cdot f_2(\sigma) \quad\text{and}\quad f_1 + f_2 \;=\; \lambda\sigma.\, f_1(\sigma) + f_2(\sigma)\,.$$

C	$\mathsf{wp}\,[\![C]\!]\,(f)$
`skip`	f
`diverge`	0
$x := E$	$f\,[x/E]$
$C_1 \mathbin{;} C_2$	$\mathsf{wp}\,[\![C_1]\!]\,\big(\mathsf{wp}\,[\![C_2]\!]\,(f)\big)$
$\texttt{if}\,(\varphi)\,\{C_1\}\,\texttt{else}\,\{C_2\}$	$[\varphi]\cdot \mathsf{wp}\,[\![C_1]\!]\,(f) \;+\; [\neg\varphi]\cdot \mathsf{wp}\,[\![C_2]\!]\,(f)$
$\{C_1\}\,\square\,\{C_2\}$	$\min\big\{\mathsf{wp}\,[\![C_1]\!]\,(f),\, \mathsf{wp}\,[\![C_2]\!]\,(f)\big\}$
$\texttt{while}(\varphi)\{C'\}$	$\mathsf{lfp}\; X.\; [\neg\varphi]\cdot f \;+\; [\varphi]\cdot \mathsf{wp}\,[\![C']\!]\,(X)$

Table 2.3: The weakest precondition transformer acting on indicator functions. This transformer serves also as an anticipation transformer acting on more general functions of type $f\colon \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$.

The definition of the conditional choice using $\cdot$ and $+$ instead of $\wedge$ and $\vee$ is meaningful since for predicates $F, G \in \mathcal{P}(\Sigma)$ we have that

$$[F \wedge G] \;=\; [F]\cdot[G] \quad \text{and} \quad [F \vee G] \;=\; [F]+[G]\,,$$

and thus $\wedge$ corresponds to $\cdot$ and $\vee$ corresponds to $+$.

For the nondeterministic choice construct we have

$$\{C_1\}\,\square\,\{C_2\} \;=\; \min\{\mathsf{wp}\,[\![C_1]\!]\,(f),\, \mathsf{wp}\,[\![C_2]\!]\,(f)\}\,,$$

where min is also to be understood pointwise, i.e.

$$\min\{f_1, f_2\} \;=\; \lambda\sigma.\; \min\{f_1(\sigma), f_2(\sigma)\}\,.$$

This is meaningful since for predicates $F, G \in \mathcal{P}(\Sigma)$ we have that

$$[F \wedge G] \;=\; \min\{[F], [G]\}\,,$$

and therefore $\wedge$ not only corresponds to $\cdot$ but also to min. The choice of $\cdot$ for the conditional choice and min for the nondeterministic is somewhat arbitrary at this point but we will say more about the role of min shortly. A very high–level intuition at this point is that we want to express by $a \cdot b$ the *logical connective* „both a and b must be true“, whereas by $\min\{a, b\}$ we want to *select* the „least true option from a and b“.

2.3.1 *Anticipated Values*

We saw how to reformulate the weakest precondition calculus to act on functions of type $f\colon \Sigma \to \{0, 1\}$. In terms of the reformulated calculus, we can

reason about whether program C will terminate in a state satisfying a predicate F by calculating

$$\mathsf{wp}\,[\![C]\!]\,([F])\ .$$

So from any state σ with $\mathsf{wp}\,[\![C]\!]\,([F])(\sigma) = 1$ the program C will terminate in a state $\tau \models F$, and from any state σ with $\mathsf{wp}\,[\![C]\!]\,([F])(\sigma) = 0$ the program C will either terminate in a state $\tau \not\models F$ or not terminate at all. This means that $\mathsf{wp}\,[\![C]\!]\,([F])$ is a function that *anticipates the truth of* F after termination of C, or in other words:

$\mathsf{wp}\,[\![C]\!]\,([F])$ is the *anticipated value* of $[F]$.

Now that we know that we can use wp to anticipate values of functions of type $\Sigma \to \{0, 1\}$, a natural question arises:

Can we anticipate values of more general functions?

For example: can we anticipate the value of program variable x after termination of C; or as another example: can we anticipate the value of $y^2 + |\sin z|$? It turns out that the answer to that question is *yes*.

2.3.2 *An Anticipated Value Calculus for Deterministic Programs*

Consider for the moment only *deterministic* programs. We would now like to reason about anticipated values of a more general class of functions, namely functions from the set of *anticipations*:

Definition 2.10 (Anticipations):

A. *The set of anticipations is defined as*

$$\mathbb{A} \;=\; \left\{ f \;\middle|\; f \colon \Sigma \to \mathbb{R}^{\infty}_{\geq 0} \right\},$$

where $\mathbb{R}^{\infty}_{\geq 0}$ is the set of non–negative real numbers with an adjoined ∞ element which is larger than every real number.

B. *A complete lattice on $\mathbb{A}$ is induced by the partial order*

$$f_1 \;\preceq\; f_2 \qquad \text{iff} \qquad \forall \sigma \in \Sigma\colon \quad f_1(\sigma) \;\leq\; f_2(\sigma)\,.$$

The least element of the complete lattice $(\mathbb{A}, \preceq)$ is the function that maps every program state to 0, i.e. the function

$$\lambda\sigma\mathbf{.}\ 0\,,$$

which we (overloadingly) also denote by 0. The supremum of a subset $S \subseteq \mathbb{A}$ is constructed pointwise as

$$\sup\, S \;=\; \lambda\sigma\mathbf{.}\ \sup_{f \in S} f(\sigma)\,.$$

It turns out that for anticipating values of functions, we can just reuse the calculus from Table 2.3 but have the f's be taken from $\mathbb{A}$. In that sense, the transformer from Table 2.3 also serves as an *anticipated value transformer*. So if we want to know the value that an $f \in \mathbb{A}$ has after executing C, we just use f as the *postanticipation* and determine the *preanticipation* $\mathsf{wp}\,[\![C]\!]\,(f)$ according to Table 2.3. In that way we obtain the sought–after anticipated value of f. The completeness of the lattice $(\mathbb{A}, \preceq)$ ensures existence of least fixed points and thereby well–definedness of wp for loops. Notice that functions as

$$x \;=\; \lambda\sigma.\,\sigma(x) \quad \text{and} \quad y^2 + |\sin z| \;=\; \lambda\sigma.\,\sigma(y)^2 + |\sin \sigma(z)|$$

are both members of $\mathbb{A}$.[7] Notice furthermore that the wp calculus acting on $\mathbb{A}$ subsumes Dijkstra's original calculus since for every predicate F we have $[F] \in \mathbb{A}$ and *to all intents and purposes*, $\mathsf{wp}\,[\![C]\!]\,(F) = \mathsf{wp}\,[\![C]\!]\,([F])$. Even the order $\preceq$ on anticipations subsumes the order $\Longrightarrow$ on predicates since for predicates F_1 and F_2 we have

$$F_1 \Longrightarrow F_2 \quad \text{iff} \quad [F_1] \preceq [F_2]\,.$$

Example 2.11 (Anticipated Values of Deterministic Programs):

A. We reconsider the program $C_{2.2}$ from Example 2.2 on page 25 and instead of reasoning whether $y^2 > 2$, we will now directly anticipate the value of y^2 after execution of $C_{2.2}$. We will reuse our annotation style from earlier, i.e.

$$\begin{array}{l} /\!/\!/\ g' \\ /\!/\!/\ g \\ C \\ /\!/\!/\ f \end{array}$$

expresses the fact that $g = \mathsf{wp}\,[\![C]\!]\,(f)$ and moreover that $g' = g$. Since we want to anticipate the value of y^2, we will use the function y^2 as postanticipation and annotate $C_{2.2}$ using the rules from Table 2.3 as shown in Figure 2.9 (again: read from bottom to top).

In words, $\mathsf{wp}\,[\![C]\!]\,\left(y^2\right) = [y > 0] \cdot 4 \;+\; [y \leq 0]$ tells us that from any initial state σ with $y > 0$ we will end up in some final state with $y^2 = 4$, whereas if initially $y \leq 0$ we will end up in some final state with $y^2 = 1$.

B. Reconsider the program $C_{2.3}$ from Example 2.3:

7 We tacitly assume that x takes only positive values. Otherwise $\lambda\sigma.\,\sigma(x)$ would technically not be a member of $\mathbb{A}$. A more appropriate choice would be the function $[x > 0]\cdot x = \lambda\sigma.\,[x > 0](\sigma)\cdot\sigma(x)$ which *is* a member of $\mathbb{A}$, but we did not want to clutter the presentation above.

$C_{2.3}$ ▷

```
z := y;
while(x > 0){
    z := z + 1;
    x := x − 1
}
```

We want to reason about the value that program variable z has after the execution of $C_{2.3}$, i.e. about postanticipation z. The characteristic function of the loop with respect to postanticipation z is given by

$$\Phi(X) \;=\; [x \leq 0] \cdot z \;+\; [0 < x] \cdot \mathsf{wp}\,[\![z := z+1\,;\, x := x-1]\!]\,(X)\;.$$

The first three iterations of the fixed point iteration for Φ are:

$$\begin{aligned}
\Phi(0) &= [x \leq 0] \cdot z \;+\; [0 < x \leq 0] \cdot \lceil x \rceil \\
\Phi^2(0) &= [x \leq 1] \cdot z \;+\; [0 < x \leq 1] \cdot \lceil x \rceil \\
\Phi^3(0) &= [x \leq 2] \cdot z \;+\; [0 < x \leq 2] \cdot \lceil x \rceil
\end{aligned}$$

Detailed calculations can be found in Appendix C.2. After three iterations, we can already start seeing a pattern for $n > 1$:

$$\Phi^n(0) \;=\; [x \leq n-1] \cdot z \;+\; [0 < x \leq n-1] \cdot \lceil x \rceil$$

Again we omit proving the above pattern correct. By inspection of this pattern, we see that the preanticipation of the loop converges to

$$\begin{aligned}
&\mathsf{wp}\,[\![\texttt{while}(x > 0)\{\ldots\}]\!]\,(z) \\
&= \sup_{n \in \mathbb{N}} [x \leq n-1] \cdot z \;+\; [0 < x \leq n-1] \cdot \lceil x \rceil \\
&= z \;+\; [0 < x] \cdot \lceil x \rceil
\end{aligned}$$

For the whole program, we can finally make these annotations:

```
/// y + [0 < x] · ⌈x⌉
z := y;
/// z + [0 < x] · ⌈x⌉                    (see above)
while(x > 0){
    z := z + 1;
    x := x − 1
}
/// z
```

We have thus proven $\mathsf{wp}\,[\![C_{2.3}]\!]\,(z) = y + [0 < x] \cdot \lceil x \rceil$. This means that from all initial states $C_{2.3}$ terminates and the value of z after termination is the initial value of y plus — in case that x was initially positive — the initial value of $\lceil x \rceil$.

One issue we have not investigated so far is the anticipated value of a nonterminating program execution. Since it is not immediately clear, what the anticipated value should be, a remark on that matter is in order:

Remark 2.12 (Anticipated Values and Nontermination). Evaluation of the weakest preexpectation $\mathsf{wp}\,[\![C]\!]\,(f)$ at σ is (and indeed has to be) 0 if C does not terminate on σ. Thus, when observing e.g. $\mathsf{wp}\,[\![C]\!]\,(x)(\sigma) = 0$ alone, we cannot know offhand whether C terminates on σ in a state with $x = 0$ or whether C does not terminate on σ.

While it might seem somewhat arbitrary at first glance, we can get an intuition for that 0 by looking at the anticipated value of C with respect to $1 = [\mathsf{true}]$. Recall that $\mathsf{wp}\,[\![C]\!]\,(1)(\sigma)$ evaluates to 0 exactly if C does not terminate on σ and for reasons of continuity $\mathsf{wp}\,[\![C]\!]\,(f)(\sigma)$ has to evaluate to 0 for *any* f in case C does not terminate on σ. △

2.3.3 *Anticipated Value Calculi for Nondeterministic Programs*

We now turn towards anticipated values of nondeterministic programs. As we know, the nondeterministic choice $\{C_1\} \,\square\, \{C_2\}$ executes either C_1 or C_2 and we have no information on what is going to happen. We can therefore not speak of *the* anticipated value of a function f since there might be multiple values that a program can yield. For instance, the program

$$
\begin{array}{l}
x := 0 \,;\\
\{c := 0\} \,\square\, \{c := 1\} \,;\\
\mathtt{while}(c = 1)\{\\
\quad x := x + 1 \,;\\
\quad \{c := 0\} \,\square\, \{c := 1\}\\
\}
\end{array}
$$

may even yield any natural number for x or not terminate at all. The range of anticipated values of x is therefore infinite here.

For weakest preconditions of nondeterministic programs it made sense to choose the least true value $\min\{\mathsf{wp}\,[\![C_1]\!]\,(F), \mathsf{wp}\,[\![C_2]\!]\,(F)\}$ as the weakest precondition of $\{C_1\} \,\square\, \{C_2\}$ with respect to postcondition F. For anticipated values, this is a meaningful possible choice as well, i.e. we can define

$$\mathsf{wp}\,[\![\{C_1\} \,\square\, \{C_2\}]\!]\,(f) \;=\; \min\{\mathsf{wp}\,[\![C_1]\!]\,(f), \mathsf{wp}\,[\![C_2]\!]\,(f)\}\,,$$

```
/// [y > 0] · 4 + [y ≤ 0]
/// [y > 0] · 4 + [y ≤ 0] · 1
if (y > 0){
    /// 4
    /// (5 − 3)²
    x := 5
    /// (x − 3)²
}else{
    /// 1
    /// (2 − 3)²
    x := 2
    /// (x − 3)²
};
/// (x − 3)²
y := x − 3;
/// y²
skip
/// y²
```

Figure 2.9: Anticipated value annotations for Example 2.11 A.

where min is again meant pointwise, i.e.

$$\min\{f_1, f_2\} \;=\; \lambda\sigma.\, \min\{f_1(\sigma), f_2(\sigma)\}\ .$$

$\mathsf{wp}\,[\![\{C_1\} \,\square\, \{C_2\}]\!]\,(f)$ thus assigns to each initial state the *least* anticipated value of f. We call this the *demonic* model of nondeterminism. This model enjoys the nice property that it subsumes the weakest *precondition* calculus for nondeterministic programs and hence we will continue to use the symbol wp in the context of demonic nondeterminism.

While we just saw that min is a quite natural choice, there are use cases where choosing max instead of min is more meaningful, e.g. when reasoning about expected runtimes (see Chapter 7). In this case, we employ a so–called *angelic* model of nondeterminism, which gives us a different transformer awp for *greatest* anticipated values. This transformer is defined analogously to wp except on nondeterministic choice constructs, on which it is given as

$$\mathsf{awp}\,[\![\{C_1\} \,\square\, \{C_2\}]\!]\,(f) \;=\; \max\{\mathsf{awp}\,[\![C_1]\!]\,(f), \mathsf{awp}\,[\![C_2]\!]\,(f)\}\ .$$

Note that wp and awp obviously coincide on deterministic programs.

Both angelic and demonic nondeterminism are in some sense extremal and one could certainly think of other models. The advantage of these two models, however, is that they yield relatively easy definitions and the resulting calculi enjoy several nice properties.

Example 2.13 (Anticipated Values of Nondeterministic Programs): We will reconsider the program $C_{2.7}$ from Example 2.7 and reason about the *least* anticipated value of y^2 as shown in Figure 2.10. By these annotation, we have established $\mathsf{wp}\,[\![C]\!]\left(y^2\right) = 1 + [y > 1] \cdot 3$. This tells us that from any initial state in which y is larger than 1, C will terminate in a state where y is at least 4, and if initially $y \leq 1$ then in a state τ where y is at least 1.

Notice that even from a state in which $y \leq 1$ it is still possible that the program terminates in a state where y^2 is at least 4, namely if in the nondeterministic choice the left branch $y := 1$ is executed.

```
/// 1 + [y > 1] · 3
/// min{4, [y > 1] · 4 + [y ≤ 1]}
{
    /// 4
    /// [1 > 0] · 4 + [1 ≤ 0]
    y := 1
    /// [y > 0] · 4 + [y ≤ 0]
} □ {
    /// [y > 1] · 4 + [y ≤ 1]
    /// [y − 1 > 0] · 4 + [y − 1 ≤ 0]
    y := y − 1
    /// [y > 0] · 4 + [y ≤ 0]
};
/// [y > 0] · 4 + [y ≤ 0]                    (see Example 2.11)
if (y > 0) {x := 5} else {x := 2};
y := x − 3;
skip
/// y²
```

Figure 2.10: Anticipated value annotations for Example 2.13.

PROBABILISTIC COMPUTATIONS 3

In this chapter, we introduce syntax and semantics of the *probabilistic Guarded Command Language* (pGCL). This model programming language is a syntactic superset of Dijkstra's GCL and enriches it by probabilistic constructs for modeling probabilistic computations.

Semantics of structured probabilistic programs have been first studied by Kozen in the late 70's and early 80's [Koz79; Koz81; Koz83; Koz85]. He did not consider nondeterministic choice as Dijkstra did in his GCL but instead *replaced* it with probabilistic choice. Later, McIver & Morgan (re)introduced nondeterministic choices [MMS96; MM05]. We follow their approach here and present a variant of their pGCL, i.e. a programming language that features both kinds of uncertainties: randomness and nondeterminism.[1] We begin with a note on the difference between these two sources of uncertainty.

3.1 RANDOMNESS VERSUS NONDETERMINISM

Consider a fair (random) coin. If we flip the coin, we do not know upfront what the outcome will be: heads or tails. Since the coin is fair, there is not even a bias towards one of the two outcomes. We could thus think that all we know is that the outcome will be either heads or tails. However, that is actually not quite *all* we know. We do have an additional grain of information in knowing that both outcomes are equally likely to occur, each with probability $1/2$.

Suppose a gambler approaches us and proposes the following bet:

> „I bet you a troy ounce of gold that if you flip this fair coin 10 times in a row, it will land on heads each of those times."

Should we bet on it? By knowing that the coin is fair, we actually have a tremendous amount of information about this bet, namely that the chance of winning is

$$1 - \frac{1}{2^{10}} \;=\; 99.90234375\%\,,$$

and we would thus be well–advised to take the chances, unless we are somewhat overly risk–averse.

We now consider a *nondeterministic coin* that is modeled by the following mechanism: A fair (random) coin is flipped by us and then a blindfolded

1 We will use the terms *random* and *probabilistic* synonymously.

oracle is asked to announce the outcome of the coin flip. The outcome of this nondeterministic coin flip is then either *correct* (in case the oracle's announcement is correct) or *incorrect* (in case the announcement is incorrect) and this is in fact really *all we know* about the outcome of the entire experiment. Does the oracle just randomly (i.e. probabilistically) guess an answer? If yes, is the oracles guess biased in any way? Would that even matter? Or does the oracle even have the superhuman ability to correctly announce the outcome of the coin toss every single time? (Or with high probability?) Is it even possible that we are being tricked in some way and the oracle is somehow reliably informed about the outcome of the coin toss? To make a long story short: There is a myriad of possibilities for the oracle to come to its conclusion and we have no clue about the underlying mechanism.

Suppose that the gambler approaches us and proposes the following bet:

> „I bet you a troy ounce of gold that the oracle will make a correct announcement 10 times in a row."

Should we bet on it? By knowing basically nothing about the outcomes of the nondeterministic coin, we can only *hope* that the oracle fails to make 10 correct announcements in a row, *but we cannot reasonably associate a probability or even any quantity to this outcome*. We can only state that this unfavorable outcome is *possible*. If we want to play it reasonably safe, we should perhaps not take those chances.

To summarize: Randomness is a kind of uncertainty where we can meaningfully assign a quantity — namely a *probability* — to each of the possible outcomes, whereas nondeterminism is a kind of uncertainty where we can only say for each outcome whether there is a *possibility* for it to occur or not. Random behavior thus allows for quantitative reasoning, whereas nondeterministic behavior only allows for qualitative reasoning. Mixing both behaviors in a single computational process is widely acknowledged to be problematic and is an active area of research (see e.g. [VW06; Var03; TKP09; Mis00; KP17]). We will later present the calculus of McIver & Morgan that attempts to do quantitative reasoning about programs in a language which features both probabilistic and nondeterministic uncertainty by *resolving* nondeterminism in the least favorable way.

3.2 pGCL — A PROBABILISTIC GCL

Access to some source of randomness is a key ingredient for a programming language that models probabilistic computations. There are different concepts for introducing randomness into the computation, e.g. random inputs, internal coin flips, sampling values from predefined probability distributions, etc. We (and many others) choose for our development two sources of randomness:

- ✧ coin flips, biased according to some rational probability, and
- ✧ sampling of values from *discrete* probability distributions.

For incorporating the aforementioned two sources of randomness, we endow Dijkstra's GCL with a *probabilistic choice* and a *random assignment* construct, thereby obtaining a *probabilistic* GCL. Formally, this probabilistic programming language is defined as follows:

Definition 3.1 (A Probabilistic Guarded Command Language):

A. *A function* $\pi : \mathsf{Vals} \to [0,1]$ *is called a probability distribution over values if* $\sum_{v \in \mathsf{Vals}} \pi(v) = 1$. *We denote the set of all probability distributions over values by* $\mathcal{D}(\mathsf{Vals})$. *A distribution expression is a function*

$$\mu : \Sigma \to \mathcal{D}(\mathsf{Vals})$$

that maps every program state to a probability distribution over values. Recall from Definition 2.1 *that* Vals *is always required to be countable and therefore every probability distribution over values is a* discrete *probability distribution.*

B. *A probability expression is a function*

$$p : \Sigma \to [0,1] \cap \mathbb{Q}$$

that maps every program state to a rational *probability.*

C. *The set of programs in probabilistic guarded command language, denoted* pGCL*, is given by the grammar*

$$\begin{aligned}
C \;\longrightarrow\; & \texttt{skip} && \text{(effectless program)}\\
\mid\; & \texttt{diverge} && \text{(freeze)}\\
\mid\; & x := E && \text{(assignment)}\\
\mid\; & x :\approx \mu && \text{(random assignment)}\\
\mid\; & C \,;\, C && \text{(sequential composition)}\\
\mid\; & \texttt{if}\,(\varphi)\,\{C\}\,\texttt{else}\,\{C\} && \text{(conditional choice)}\\
\mid\; & \{C\} \,\square\, \{C\} && \text{(nondeterministic choice)}\\
\mid\; & \{C\}\,[p]\,\{C\} && \text{(probabilistic choice)}\\
\mid\; & \texttt{while}(\varphi)\{C\}\,, && \text{(while loop)}
\end{aligned}$$

where $x \in \mathsf{Vars}$ *is a program variable,* E *is an arithmetic expression over program variables,* μ *is a distribution expression,* φ *is a boolean expression over program variables guarding a choice or a loop, and* p *is a probability expression.*

D. *A* pGCL *program containing no* `diverge` *statements and no while loops is called loop–free.*

E. *A* pGCL *program that contains neither random assignments nor probabilistic choices is called nonprobabilistic. A* pGCL *program that contains no nondeterministic choices is called tame. A* pGCL *program that contains neither constructs of randomness nor constructs of nondeterminism is called deterministic.*

Every language construct from GCL is also contained in pGCL. Their computational effects are exactly the same in pGCL as they are in GCL (see Section 2.1). We thus only go over the probabilistic constructs introduced in pGCL here, starting with the conceptually simpler one.

Probabilistic choices. The probabilistic choice construct

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

behaves as follows: It evaluates probability expression p in the current program state σ to obtain a probability $p(\sigma)$. Then, it executes C_1 with probability $p(\sigma)$ and C_2 with probability $1 - p(\sigma)$.

EXAMPLE 3.2 (Probabilistic Choices):

A. The program

$$\{x := x + 1\}\,[2/3]\,\{z := 17\}$$

increments variable x by 1 with probability $2/3$ and it assigns the constant 17 to variable z with probability $1 - 2/3 = 1/3$.

B. If the current program state is σ, then the program

$$\{x := x + 1\}\,[1/|x|+1]\,\{x := x - 1\}$$

increments variable x by 1 with probability $1/|\sigma(x)|+1$ and decrements x by 1 with probability $1 - 1/|\sigma(x)|+1 = |\sigma(x)|/|\sigma(x)|+1$. Therefore, the further away from 0 the value of x is, the less likely it is that x is incremented.

Random Assignments. The random assignment construct

$$x :\approx \mu$$

behaves as follows: It evaluates distribution expression μ in tn the current program state σ to obtain a discrete probability distribution $\pi = \mu(\sigma)$. Then it samples a value from π, thus obtaining a sample value $v \in \mathsf{Vals}$ with probability $\pi(v)$. This value v is then assigned to variable x.

For denoting distribution expressions, we use bra–ket notation [Wikc]. For example, distribution expression

$$\tfrac{1}{2} \cdot |a\rangle + \tfrac{1}{3} \cdot |b\rangle + \tfrac{1}{6} \cdot |c\rangle$$

denotes a distribution where value a is sampled with probability $1/2$, b with probability $1/3$, and c with probability $1/6$.

Example 3.3 (Random Assignments):

A. The program

$$x :\approx \tfrac{1}{2} \cdot |x+1\rangle + \tfrac{1}{2} \cdot |x-1\rangle$$

increments or decrements variable x by 1, each with probability $1/2$. It does so by first evaluating the distribution expression $1/2 \cdot |x+1\rangle + 1/2 \cdot |x-1\rangle$ in the current program state σ. This gives the distribution

$$\pi(v) \;=\; \begin{cases} \frac{1}{2}, & \text{if } v = \sigma(x)+1 \text{ or } v = \sigma(x)-1 \\ 0, & \text{otherwise.} \end{cases}$$

Then the program samples from π. By that, the values $\sigma(x)+1$ and $\sigma(x)-1$ are each sampled with probability $1/2$. The sampled value is then assigned to variable x.

B. Like the program from Example 3.2 B., the program

$$x :\approx \tfrac{1}{|x|+1} \cdot |x+1\rangle + \tfrac{|x|}{|x|+1} \cdot |x-1\rangle$$

too increments variable x by 1 with probability $1/|\sigma(x)|+1$ and decrements x by 1 with probability $|\sigma(x)|/|\sigma(x)|+1$. It does so by first evaluating the distribution expression $1/|x|+1 \cdot |x+1\rangle + |x|/|x|+1 \cdot |x-1\rangle$ in the current program state σ. This gives the probability distribution

$$\pi(v) \;=\; \begin{cases} \frac{1}{|\sigma(x)|+1}, & \text{if } v = \sigma(x)+1 \\ \frac{|\sigma(x)|}{|\sigma(x)|+1}, & \text{if } v = \sigma(x)-1 \\ 0, & \text{otherwise.} \end{cases}$$

Then the program samples from π. By that, the value $\sigma(x)+1$ is sampled with probability $1/|\sigma(x)|+1$ and the value $\sigma(x)-1$ is sampled with probability $|\sigma(x)|/|\sigma(x)|+1$. The sampled value is then assigned to variable x.

c. The program

$$k :\approx \mathsf{Unif}[1 \ldots 10]$$

assigns to program variable k one of the integers between 1 and 10, each with probability $1/10$.

d. If the current program state is σ, the program

$$k :\approx \mathsf{Unif}[1 \ldots n]$$

assigns to variable k an integer value between 1 and $\sigma(n)$, where n is a program variable,[2] each with probability $1/\sigma(n)$.

We have just provided a more or less formal intuition on what the two probabilistic choice constructs do computationally. In the following sections, we will provide several precise semantics to probabilistic programs.

3.3 SEMANTICS OF pGCL

Semantics of programming languages are precise mathematical descriptions of a program's computational effects. For deterministic (nondeterministic) programs, it often provides a mapping from initial states to (sets of) final states. It thus tells us what the (possible) outcome(s) of excuting a program on a given initial state is (are). Such mappings are qualitative in that a given initial state is either mapped to a given final state or not. In contrast to that, probabilistic programs clearly require *quantitative* information in order to make for a meaningful semantics.

In this section, we describe two different operational semantics of pGCL. They are operational in the sense that they describe a step–by–step, i.e. instruction–by–instruction, execution of a program and the according evolution of program states over the course of the execution. Such semantics are called *small–step semantics* or *structural operational semantics* [Plo04]. We will also show how the two operational semantics are related. Moreover, we will learn how we can think of *the outcome* of a probabilistic computation as a probability distribution over final states. This can be thought of as a *big–step semantics*, that maps input states directly to an outcome.

3.3.1 *Computation Tree Semantics*

Computation trees naturally occur in studies of nondeterministic Turing machines (e.g. [PZ83]) and alternating Turing machines [CKS81], but also in studies of deterministic computations like recursive functions [Gri99]. In

2 We tacitly assume that $\sigma(n)$ is a natural number.

the latter, computation trees model multiple subcomputations that may have to be evaluated. For alternating Turing machines, we can think of computations as games played on trees, where branching represents alternatives for the players to choose from. For nondeterministic Turing machines, branching of their computation tree also represents different alternatives in which a computation may proceed. Computation trees constitute one of the most basic and robust representations of computation and we will thus view them as *the* base layer small–step semantics of probabilistic computations against which all our soundness results will be proved.

For pGCL we will present a computation tree semantics in the vein of nondeterministic Turing machines, where branching represents alternatives in which the computation may proceed either due to randomness or nondeterminism. The nodes of the tree represent current configurations of the computation and the edges computation steps, i.e. progress in computation. Formally, this computation tree semantics is defined as follows:

DEFINITION 3.4 (Computation Tree Semantics of pGCL):

A. *A configuration* $\kappa = \langle C, \sigma, n, \theta, \eta, q \rangle$ *comprises of*

- *either a program* $C \in \text{pGCL}$ *that is left to be executed or a symbol* $C = \downarrow$ *indicating successful termination,*
- *a program state, i.e. a variable valuation,* $\sigma \in \Sigma$,
- *the number* $n \in \mathbb{N}$ *of computation steps that have been executed in the past over the course of the computation,*
- *the history* $\theta \in \mathbb{N}^*$ *of all probabilistic choices that were made,*
- *the history* $\eta \in \{L, R\}^*$ *of all nondeterministic choices that were made, and*
- *the probability* $q \in [0, 1] \cap \mathbb{Q}$ *of reaching the configuration if nondeterministic choices are resolved according to* η.

We denote the set of all configurations by $\mathbb{K}$. *Notice that* $\mathbb{K}$ *is* countable, *since* pGCL, Σ, $\mathbb{N}^*$, $\{L, R\}^*$, *and* $[0, 1] \cap \mathbb{Q}$ *are countable.*

B. *A transition relation* $\vdash \subseteq \mathbb{K} \times \mathbb{K}$ *between configurations is defined as the smallest relation satisfying the rules given in* Figure 3.1. *As usual, we denote by* $\vdash^k$ *reachability within* k *applications of* $\vdash$ *and by* $\vdash^*$ *the reflexive–transitive closure of* $\vdash$.

C. *The computation tree of executing program* $C \in \text{pGCL}$ *on input* $\sigma \in \Sigma$, *denoted* $\mathcal{T}^{C,\sigma}$, *is a tree* (κ_0, K, E) *where*

- $\kappa_0 = \langle C, \sigma, 0, \varepsilon, \varepsilon, 1 \rangle$ *is the root of the tree,*
- $K = \{ \kappa \mid \kappa_0 \vdash^* \kappa \}$ *is the set of nodes in the tree, and*
- $E = (K \times K) \cap \vdash$ *is the set of edges of the tree.*

$$\frac{}{\langle \texttt{skip}, \sigma, n, \theta, \eta, q\rangle \vdash \langle \downarrow, \sigma, n+1, \theta, \eta, q\rangle} \text{(skip)}$$

$$\frac{}{\langle \texttt{diverge}, \sigma, n, \theta, \eta, q\rangle \vdash \langle \texttt{diverge}, \sigma, n+1, \theta, \eta, q\rangle} \text{(diverge)}$$

$$\frac{v = \sigma(E)}{\langle x := E, \sigma, n, \theta, \eta, q\rangle \vdash \langle \downarrow, \sigma[x \mapsto v], n+1, \theta, \eta, q\rangle} \text{(assign)}$$

$$\frac{\mu(\sigma)(v) = a > 0 \qquad \aleph^{-1}(v) = i}{\langle x :\approx \mu, \sigma, n, \theta, \eta, q\rangle \vdash \langle \downarrow, \sigma[x \mapsto v], n+1, \theta i, \eta, q \cdot a\rangle} \text{(rnd–assign)}$$

$$\frac{\langle C_1, \sigma, n, \theta, \eta, q\rangle \vdash \left\langle C_1', \sigma', n+1, \theta', \eta', q'\right\rangle \qquad C_1' \neq \downarrow}{\langle C_1 \fatsemi C_2, \sigma, n, \theta, \eta, q\rangle \vdash \left\langle C_1' \fatsemi C_2, \sigma', n+1, \theta', \eta', q'\right\rangle} \text{(seq1)}$$

$$\frac{\langle C_1, \sigma, n, \theta, \eta, q\rangle \vdash \langle \downarrow, \sigma', n+1, \theta', \eta', q'\rangle}{\langle C_1 \fatsemi C_2, \sigma, n, \theta, \eta, q\rangle \vdash \langle C_2, \sigma', n+1, \theta', \eta', q'\rangle} \text{(seq2)}$$

$$\frac{\varphi(\sigma) = \text{true}}{\langle \texttt{if}\,(\varphi)\,\{C_1\}\,\texttt{else}\,\{C_2\}, \sigma, n, \theta, \eta, q\rangle \vdash \langle C_1, \sigma, n+1, \theta, \eta, q\rangle} \text{(if1)}$$

$$\frac{\varphi(\sigma) = \text{false}}{\langle \texttt{if}\,(\varphi)\,\{C_1\}\,\texttt{else}\,\{C_2\}, \sigma, n, \theta, \eta, q\rangle \vdash \langle C_2, \sigma, n+1, \theta, \eta, q\rangle} \text{(if2)}$$

$$\frac{}{\langle \{C_1\} \Box \{C_2\}, \sigma, n, \theta, \eta, q\rangle \vdash \langle C_1, \sigma, n+1, \theta, \eta L, q\rangle} \text{(nondet1)}$$

$$\frac{}{\langle \{C_1\} \Box \{C_2\}, \sigma, n, \theta, \eta, q\rangle \vdash \langle C_2, \sigma, n+1, \theta, \eta R, q\rangle} \text{(nondet2)}$$

$$\frac{p(\sigma) = a}{\langle \{C_1\}\,[p]\,\{C_2\}, \sigma, n, \theta, \eta, q\rangle \vdash \langle C_1, \sigma, n+1, \theta 0, \eta, q \cdot a\rangle} \text{(prob1)}$$

$$\frac{p(\sigma) = a}{\langle \{C_1\}\,[p]\,\{C_2\}, \sigma, n, \theta, \eta, q\rangle \vdash \langle C_2, \sigma, n+1, \theta 1, \eta, q \cdot (1-a)\rangle} \text{(prob2)}$$

$$\frac{\varphi(\sigma) = \text{true}}{\langle \texttt{while}(\varphi)\{C\}, \sigma, n, \theta, \eta, q\rangle \vdash \langle C \fatsemi \texttt{while}(\varphi)\{C\}, \sigma, n+1, \theta, \eta, q\rangle} \text{(while1)}$$

$$\frac{\varphi(\sigma) = \text{false}}{\langle \texttt{while}(\varphi)\{C\}, \sigma, n, \theta, \eta, q\rangle \vdash \langle \downarrow, \sigma, n+1, \theta, \eta, q\rangle} \text{(while2)}$$

Figure 3.1: Inference rules for the transition relation $\vdash$ between configurations. Recall from Definition 2.1 A. that $\aleph$ is a bijective enumeration of Vals.

A configuration represents the current state of a program execution together with a history of probabilistic and nondeterministic choices that have been made in past computation steps and together with the resulting probability with which this configuration is reached. Given a concrete configuration

$$\kappa \;=\; \langle C, \sigma, n, \theta, \eta, q \rangle \;,$$

component C indicates the rest of the program that is left to be executed. C thus plays the role of a *program counter*, except that it does not contain a line number but instead it contains the entire program that is left to be executed. Notice that if C is a while loop $\texttt{while}(\varphi)\{body\}$ that is about to perform one more iteration of its loop body (i.e. $\varphi(\sigma) = \mathsf{true}$), then the (while1)–rule *prepends* $\texttt{while}(\varphi)\{body\}$ with a copy of $body$ (thus obtaining $body \mathbin{;} \texttt{while}(\varphi)\{body\}$) in order to account for the iteration of the loop body. As a consequence, the „program component" of the configurations along consecutive $\vdash$–transitions does not necessarily get shorter and shorter just as the line numbers of a program counter would not grow strictly larger and larger when performing several iterations through a loop.

Component σ is the current program state that contains the variable valuations. We can thus think of σ as the *memory* which the program instructions can read from and write to.

The component n of configuration κ is the number of computation steps that have been executed in the past. We can think of n as a *runtime tracker*.

Components θ and η are the *histories of choices* that have been made in the past. Whenever a probabilistic or nondeterministic choice is made, this choice is recorded by appending it in either to θ or η, respectively. The history of nondeterministic choices η is a sequence of letters L and R for *L*eft and *R*ight. For the history of probabilistic choices θ, we need infinitely many symbols because for the random assignment it is conceivable that we can choose a value v from arbitrarily (or even countably infinitely) many values.[3] We thus make use of the canonical enumeration $\aleph$ of Vals introduced in Definition 2.1 A. and record $\aleph^{-1}(v)$ in the history of probabilistic choices whenever we sample value v at a random assignment.

Lastly, component q is the probability with which configuration κ was reached, i.e. the multiplied probabilities with which all probabilistic choices on the path from the root of a computation tree to κ were made.

The transition relation $\vdash$ represents single computation steps. Thus, if $\kappa \vdash \kappa'$, then executing a single atomic instruction, checking a single guard (of a conditional choice or a while loop), or flipping a single coin (probabilistic or nondeterministic) takes the computation from configuration κ to configuration κ'.

The computation tree $\mathcal{T}^{C,\sigma}$ as a whole is a tree–representation of all computations that can emanate from executing program C on input σ. The root

3 This is for instance the case for the random assignment $x :=\mathsf{Unif}[0 \ldots n]$.

$\kappa_0 = \langle C, \sigma, 0, \varepsilon, \varepsilon, 1\rangle$ of the tree is the initial configuration. It is trivially reached with probability 1 and no (i.e. 0) execution steps have been executed in the past. Also, neither probabilistic nor nondeterministic choices have been made so far. Both histories are thus the empty word ε. The set of nodes $K = \{\kappa \mid \kappa_0 \vdash^* \kappa\}$ is the set of all configurations reachable with non–zero probability under some strategy for resolving nondeterministic choices. The set of edges $E = (K \times K) \cap \vdash$ connects the reachable configurations according to a small–step execution semantics.

Notice that in case of a deterministic program C, the computation „tree" degenerates to a sequential list because branching only occurs when probabilistic or nondeterministic coins are flipped. In particular notice that conditional choices and while loops do *not* cause branching in the computation tree, since the current program state either satisfies the guard or not and the subsequent configuration is therefore completely determined by the current configuration (in particular the program state). Let us now look at an example of a computation tree:

Example 3.5 (Computation Trees of Probabilistic Programs):
Consider the program C given by

```
while(c = 1){
    {c := 0} [1/2] {skip}
}
```

and some initial state σ with $\sigma(c) = 1$. Then the computation tree $\mathcal{T}^{C,\sigma}$ of executing C on σ is shown in Figure 3.2.

We can observe that the entire computation tree $\mathcal{T}^{C,\sigma}$ from Example 3.5 is isomorphic to its own subtree emanating from $\langle \texttt{while}(\cdots\}, \sigma, 3, 1, \varepsilon, 1/2\rangle$ and also to the subtree emanating from $\langle \texttt{while}(\cdots\}, \sigma, 6, 1, \varepsilon, 1/4\rangle$. This is due to the fact that future progress in computation in general does not depend on the (histories of) probabilistic and nondeterministic choices that were made in the past but solely on the current program C that is left to be executed and the current program state σ. This independence from the histories can be captured formally by an *equivalence of* $\vdash$*–transitions* as follows:

Proposition 3.6 (Equivalence of $\vdash$–Transitions):
The following equivalence holds:

$$\langle C, \sigma, n, \theta, \eta, q\rangle \vdash \langle C', \sigma', n+1, \theta w, \eta u, q'\rangle$$
$$\text{iff} \quad \langle C, \sigma, 0, \varepsilon, \varepsilon, 1\rangle \vdash \langle C', \sigma', 1, w, u, q'/q\rangle$$

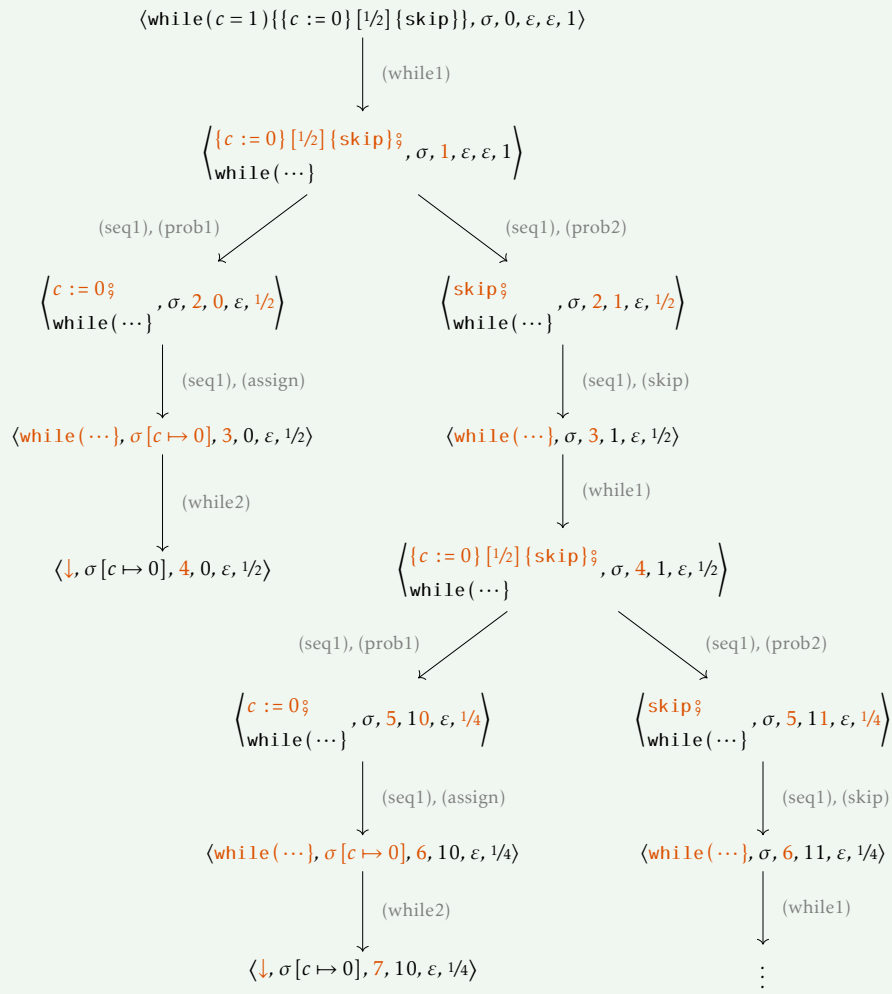


Figure 3.2: Computation tree $\mathcal{T}^{C,\sigma}$ of executing the probabilistic program $C = \texttt{while}(c = 1)\{\{c := 0\}\,[1/2]\,\{\texttt{skip}\}\}$ on a state σ with $\sigma(c) = 1$. Edge labels indicate which $\vdash$–rules are used to derive this transition. Notice that $\mathcal{T}^{C,\sigma}$ is *infinite* since all configurations of the form $\langle \texttt{while}(\cdots), \sigma, 3n, 1^n, \varepsilon, 1/2^n\rangle$ for $n \in \mathbb{N}$ are reachable along a rightmost path.

Proposition 3.6 can be interpreted as a form of reconditioning or rescaling: If we arrive at some configuration $\langle C, \sigma, n, \theta, \eta, q\rangle$ with probability q and transition from there to a configuration $\langle C', \sigma', n+1, \theta w, \eta u, q'\rangle$ then this happens with probability q'/q. We can then renormalize q to 1, reset the runtime tracker, and delete all histories: *Given* that we have somehow reached a configuration with program C and state σ we can restart from there and transit with probability q'/q to configuration $\langle C', \sigma', 1, w, u, q'/q\rangle$. This rescaling can be interpreted as a *Markov property* showing that the probability of transiting to a next configuration depends solely on the current state of the execution (comprising of program and program state) and not on any historic events. In Section 3.3.3, we thus „quotient out" the runtime tracker, the histories, and the probability q and by that obtain a Markov decision process semantics for pGCL.

3.3.2 *Distributions over Final States*

In the previous section we saw that execution of a probabilistic program can result in multiple (even infinitely many) possible computation paths since — in contrast to deterministic programs — the behavior of the program depends not only on the initial state but also on nondeterminism and randomness. This goes so far that a program may terminate only with a certain probability strictly between 0 and 1.

So how can we now conceive of *the outcome* of a probabilistic computation? Let us leave nondeterminism out of the picture for the moment, i.e. let us consider only tame programs. Despite the fact that executing such programs does not necessarily yield a unique final state, tame probabilistic programs do yield a unique *probability distribution* over final states. To be more precise, we are dealing with *subdistributions*, i.e. distributions with a total mass that may be less than 1. The „mass defect" of the resulting subdistribution then represents the probability of nontermination. To see that this is not just a convenient way for modeling, consider the program

$$\{\texttt{diverge}\}\,[1/3]\,\{x := 5\}\,.$$

Starting this program in some initial state σ will give a proper subdistribution over final states μ given by

$$\mu(\tau) \;=\; \begin{cases} \frac{2}{3}, & \text{if } \tau = \sigma\,[x \mapsto 5] \\ 0, & \text{otherwise.} \end{cases}$$

For each potential final state τ, this subdistribution μ gives us by $\mu(\tau)$ the precise, *unnormalized* probability that the execution of the above program will terminate in final state τ. The total mass of μ is $2/3$ and the missing

$$\frac{\langle C, \sigma, n, \theta, \eta, q\rangle \vdash \langle C', \sigma, n+1, \theta, \eta L, q\rangle \quad \langle C, \sigma, n, \theta, \eta, q\rangle \vdash \langle C'', \sigma, n+1, \theta, \eta R, q\rangle \quad \mathfrak{s}(C,\sigma) = L}{\langle C, \sigma, n, \theta, \eta, q\rangle \vdash_{\mathfrak{s}} \langle C', \sigma, n+1, \theta, \eta L, q\rangle} \text{(}\mathfrak{s}\text{–sched1)}$$

$$\frac{\langle C, \sigma, n, \theta, \eta, q\rangle \vdash \langle C', \sigma, n+1, \theta, \eta L, q\rangle \quad \langle C, \sigma, n, \theta, \eta, q\rangle \vdash \langle C'', \sigma, n+1, \theta, \eta R, q\rangle \quad \mathfrak{s}(C,\sigma) = R}{\langle C, \sigma, n, \theta, \eta, q\rangle \vdash_{\mathfrak{s}} \langle C'', \sigma, n+1, \theta, \eta R, q\rangle} \text{(}\mathfrak{s}\text{–sched2)}$$

$$\frac{\langle C, \sigma, n, \theta, \eta, q\rangle \vdash \langle C', \sigma', n+1, \theta', \eta, q'\rangle}{\langle C, \sigma, n, \theta, \eta, q\rangle \vdash_{\mathfrak{s}} \langle C', \sigma', n+1, \theta', \eta, q'\rangle} \text{(}\mathfrak{s}\text{–sched3)}$$

Figure 3.3: Inference rules for defining the $\mathfrak{s}$–scheduled transition relation $\vdash_{\mathfrak{s}}$ between configurations, where $\mathfrak{s} \in \mathsf{Scheds}$ is a scheduler.

probability mass of $1/3$ is the probability that no final state is reached, i.e. the probability of nontermination.

Let us now bring nondeterminism back into the picture and see how we can describe the outcomes of executing general pGCL programs on given initial states in a systematic way, namely by extracting probability distributions over final program states from computation trees. For that, we first need a way to remove nondeterminism from the computation tree so that it becomes a purely probabilistic transition system, since otherwise there need not exist a unique probability distribution. We do this by means of restricting the number of successors in the $\vdash$ transition relation.

Definition 3.7 (Scheduled $\vdash$–Transitions):
A scheduler $\mathfrak{s}$ is a function

$$\mathfrak{s} \colon \mathsf{pGCL} \times \Sigma \to \{L, R\}$$

mapping pairs of programs and program states to either letter L or R. The set of all schedulers is denoted by Scheds.

For $\mathfrak{s} \in \mathsf{Scheds}$, the $\mathfrak{s}$–scheduled transition relation $\vdash_{\mathfrak{s}}$ is given by the inference rules in Figure 3.3.

Schedulers are a means of resolving nondeterminism in systems that feature both probabilistic and nondeterministic uncertainty [Var85; Put05]. Sched-

uled transitions behave just like unscheduled ones (see rule (𝔰–sched3)) with a single exception (see rules (𝔰–sched1) and (𝔰–sched2)): When program C executes a nondeterministic choice, the scheduled relation $\vdash_{\mathfrak{s}}$ selects a *single* successor configuration according to scheduler $\mathfrak{s}$, whereas the unscheduled relation $\vdash$ has to two successors in pari passu.

The single purpose of a scheduler is thus to *resolve nondeterministic choices*. The benefit is that this allows for defining a probability distribution over final states that is established by executing a probabilistic program on some initial state under a scheduler that resolves nondeterminism.

Definition 3.8 (Probability Distribution Semantics of pGCL): *Let C be a* pGCL *program, $\sigma \in \Sigma$ be an initial program state and $\mathfrak{s} \in$ Scheds be a scheduler. Then the distribution over final states established by executing C on input σ under scheduler $\mathfrak{s}$, denoted $[\![C]\!]^{\mathfrak{s}}_{\sigma}$, is a (sub)probability distribution over program states[4] given by*

$$[\![C]\!]^{\mathfrak{s}}_{\sigma}(\tau) \;=\; \sum_{\langle\downarrow,\tau,n,\theta,\eta,q\rangle\in K} q\,, \qquad \text{where}$$

$$K \;=\; \left\{ \langle\downarrow,\tau,n,\theta,\eta,q\rangle \;\middle|\; \langle C,\sigma,0,\varepsilon,\varepsilon,1\rangle \;\vdash^{*}_{\mathfrak{s}}\; \langle\downarrow,\tau,n,\theta,\eta,q\rangle \right\}.$$

Notice that in case of tame programs, the scheduler is entirely irrelevant and the programs naturally produce probability (sub)distributions. We thus omit the scheduler for tame programs and simply write $[\![C]\!]_{\sigma}$. We conclude this section with some examples of both nondeterministic and tame programs.

Example 3.9 (Probability Distributions of pGCL Programs):

A. Consider the program

$$\{x := 1 - x\} \,\square\, \{\texttt{skip}\}\,;$$
$$\{x := 0\}\,[1/2]\,\{x := 1\}\,,$$

some initial state σ, and *any* scheduler $\mathfrak{s}$. Then the probability distribution $[\![C]\!]^{\mathfrak{s}}_{\sigma}$ is given by

$$[\![C]\!]^{\mathfrak{s}}_{\sigma}(\tau) \;=\; \begin{cases} \frac{1}{2}, & \text{if } \tau = \sigma[x \mapsto 0] \text{ or } \tau = \sigma[x \mapsto 1] \\ 0, & \text{otherwise.} \end{cases}$$

Note that the probability distribution is unique, no matter what scheduler is imposed.

4 I.e. $[\![C]\!]^{\mathfrak{s}}_{\sigma} : \Sigma \to [0,1]$, such that $\sum_{\tau\in\Sigma} [\![C]\!]^{\mathfrak{s}}_{\sigma}(\tau) \leq 1$.

B. Consider the program

$$\{x := 0\}\,[1/2]\,\{x := 1\}\,;$$
$$\{x := 1 - x\} \,\square\, \{\texttt{skip}\}\,,$$

some initial state σ, and scheduler $\mathfrak{s}_{min}$ given by

$$\mathfrak{s}_{min}(C, \sigma') \;=\; \begin{cases} L, & \text{if } \sigma'(x) = 1 \\ R, & \text{otherwise.} \end{cases}$$

Then the probability distribution $[\![C]\!]^{\mathfrak{s}_{min}}_{\sigma}$ is given by

$$[\![C]\!]^{\mathfrak{s}}_{\sigma}(\tau) \;=\; \begin{cases} 1, & \text{if } \tau = \sigma'[x \mapsto 0] \\ 0, & \text{otherwise.} \end{cases}$$

C. Consider the program C_{geo} given by

```
while(c = 1){
    {c := 0} [1/2] {x := x + 1}
}
```

and some initial state σ with $\sigma(c) = 1$ and $\sigma(x) = 0$. Then the computation tree $\mathcal{T}^{C_{geo},\sigma}$ of executing C_{geo} on σ is shown in Figure 3.4. The resulting probability distribution $[\![C_{geo}]\!]_{\sigma}$ is given by

$$[\![C]\!]_{\sigma}(\tau) \;=\; \begin{cases} \frac{1}{2^{n+1}}, & \text{if } \tau = \sigma[c, x \mapsto 0, n], \text{ for } n \in \mathbb{N} \\ 0, & \text{otherwise.} \end{cases}$$

The program C thus establishes a geometric distribution on x whenever it is ran on an initial state σ with $\sigma(c) = 1$ and $\sigma(x) = 0$. Notice that schedulers are irrelevant and thus omitted since C_{geo} is tame.

3.3.3 *Markov Decision Process Semantics*

In Section 3.3.1, we presented a very basic computation tree semantics of pGCL programs in which configurations contained histories of choices that were made in the past. We also noticed that computation steps are independent from those histories (see Proposition 3.6). In this section, we present a semantics for pGCL in which those histories are omitted. The semantics we are about to present will be based on Markov decision processes (MDPs). We do not introduce them here but instead refer to Appendix B for basic

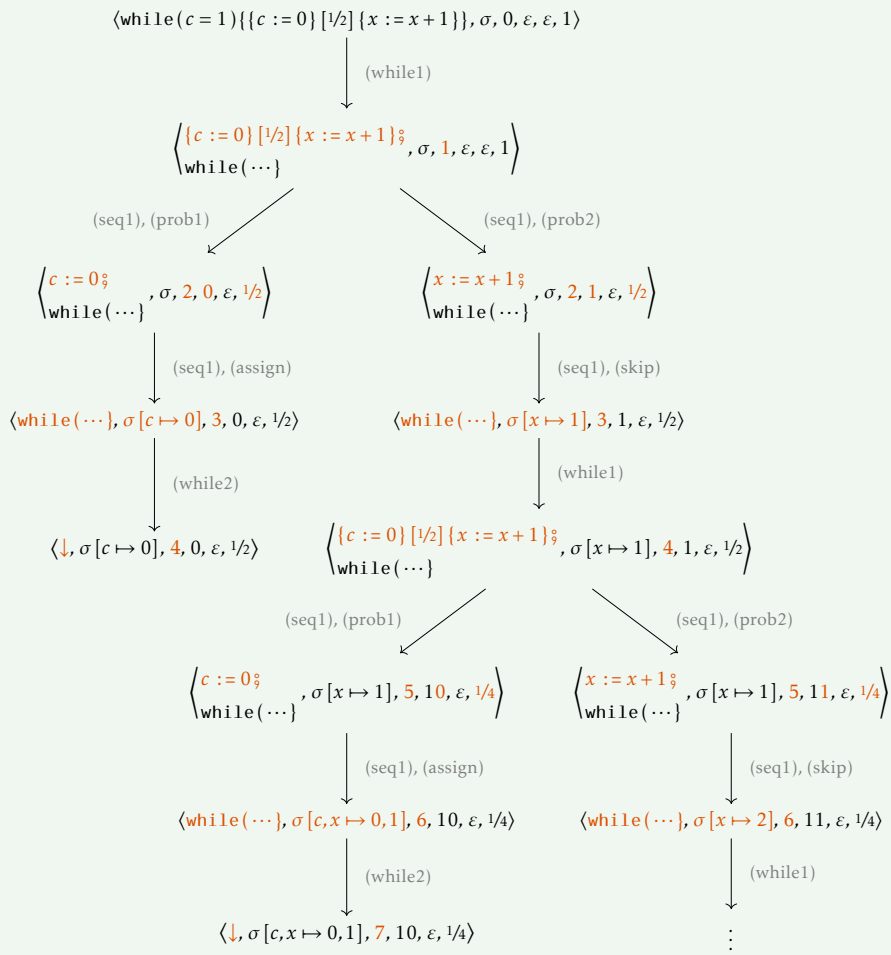


Figure 3.4: The computation tree $\mathcal{T}^{C_{geo},\sigma}$ yielded by executing program C_{geo} = `while(c=1){{c := 0} [1/2] {x := x + 1}}` on an initial state σ with $\sigma(c) = 1$ and $\sigma(x) = 0$. Edge labels indicate which $\vdash$–rules are used to derive this transition. Notice that $\mathcal{T}^{C_{geo},\sigma}$ is *infinite* since all configurations of the form $\langle \texttt{while}(\cdots), \sigma[x \mapsto n], 3n, 1^n, \varepsilon, 1/2^n \rangle$ for $n \in \mathbb{N}$ are reachable along a rightmost path.

s	α	s'	$P(s,\alpha)(s')$
$\langle C', \sigma' \rangle$	N	$\langle C'', \sigma'' \rangle$	$\begin{cases} q, & \text{if } \langle C', \sigma', 0, \varepsilon, \varepsilon, 1 \rangle \vdash \langle C'', \sigma'', 1, \theta, \varepsilon, q \rangle \\ 0, & \text{otherwise.} \end{cases}$
$\langle C', \sigma' \rangle$	L	$\langle C'', \sigma'' \rangle$	$\begin{cases} 1, & \text{if } \langle C', \sigma', 0, \varepsilon, \varepsilon, 1 \rangle \vdash \langle C'', \sigma', 1, \varepsilon, L, 1 \rangle \\ 0, & \text{otherwise.} \end{cases}$
$\langle C', \sigma' \rangle$	R	$\langle C'', \sigma'' \rangle$	$\begin{cases} 1, & \text{if } \langle C', \sigma', 0, \varepsilon, \varepsilon, 1 \rangle \vdash \langle C'', \sigma', 1, \varepsilon, R, 1 \rangle \\ 0, & \text{otherwise.} \end{cases}$
$\langle \downarrow, \sigma' \rangle$	N	$\langle \mathit{sink} \rangle$	1
$\langle \mathit{sink} \rangle$	N	$\langle \mathit{sink} \rangle$	1
— all other cases —			0

Table 3.1: Definition of the transition probability function of operational MDPs.

definitions, to Baier & Katoen for more details [BK08, Chapter 10], and to Puterman for a dedicated in–depth treatment [Put05].

A Markov decision process semantics for pGCL was presented by Gretz, Katoen, and McIver [GKM12; GKM14]. Earlier work on viewing programs as Markov decision processes was presented by Monniaux [Mon05]. The semantics we present in the following is basically the one of Gretz *et al.* and differs only in minor technical details (e.g. in that Gretz *et al.* do not consider random assignments).

Definition 3.10 (Operational Markov Decision Processes):
The operational MDP of executing program $C \in \text{pGCL}$ on input $\sigma \in \Sigma$ is the MDP (cf. Definition B.1*) $\mathcal{M}^{C,\sigma} = (S, \langle C, \sigma \rangle, \{L, R, N\}, P)$, where*

- $S = \{\langle C', \sigma' \rangle \mid \langle C, \sigma, 0, \varepsilon, \varepsilon, 1 \rangle \vdash^* \langle C', \sigma', n, \theta', \eta', q' \rangle\} \cup \{\langle \mathit{sink} \rangle\}$ *is a set of states,*
- $\langle C, \sigma \rangle$ *is the initial state,*
- $\{L, R, N\}$ *is the set of actions, and*
- P *is the transition probability function defined according to the rules in* Table 3.1.

A state of an operational MDP represents a *collection* of all computation tree configurations that share the same program and program state. A designated

$\langle \mathit{sink} \rangle$ state acts as a sink after reaching a state of the form $\langle \downarrow, \_\_ \rangle$ indicating termination of the computation. This sink is needed since in MDPs every state needs at least one successor state. The actions are given by the letters L, R, and N which stand for *L*eft, *R*ight, and *N*one, respectively. L and R indicate which branch is chosen when performing a nondeterministic choice whereas N is the default action when no nondeterministic choice is to be executed.

The probability $P(s, \alpha)(s')$ determined by the transition probability function P is the probability of making a transition from state s to state s' with action α. Let us very briefly go over the definition of the transition probability function in Table 3.1: The first rule deals with deterministic and probabilistic instructions (i.e. guard evaluations, probabilistic choices, deterministic assignments, and random assignments) in the self–evident way. The associated action is N since no nondeterministic choice is performed.

The next two rules cover nondeterministic choices. The transition probabilities are either 1 or 0 since no randomness is involved; the action must be either L or R in order to determine which branch is chosen.

The rule $P(\langle \downarrow, \sigma' \rangle, N)(\langle \mathit{sink} \rangle) = 1$ leads terminated executions into the designated sink state. Recall that a sink state is necessary since every MDP state has to have a successor state. Likewise, the sink state has to have a successor state which is again the sink state. This is captured by the rule $P(\langle \mathit{sink} \rangle, N)(\langle \mathit{sink} \rangle) = 1$.

The operational MDP $\mathcal{M}^{C,\sigma}$ is a potentially more compact representation of the computation tree $\mathcal{T}^{C,\sigma}$. An example where the representation is more compact (in fact: *finite instead of infinite*) is given by reconsidering Example 3.5 from an MDP point of view:

Example 3.11 (Operational MDPs of Probabilistic Programs): Reconsider the program C from Example 3.5 given by

```
while(c = 1){
    {c := 0} [1/2] {skip}
}
```

and consider some initial state σ with $\sigma(c) = 1$. Then the operational MDP of executing C on σ is shown in Figure 3.5. Notice that this MDP can be translated into a *Markov chain*, since the program C is tame, i.e. the program contains no nondeterministic choices.

We notice that the MDP of Figure 3.5 captures nicely the automorphism on the computation tree of Figure 3.2, namely by the back edge from state $\langle \texttt{skip}; \texttt{while}(\cdots\}, \sigma \rangle$ to the initial state of the MDP. This „folding“ of the computation tree into itself is what makes the MDP finite whereas the com-

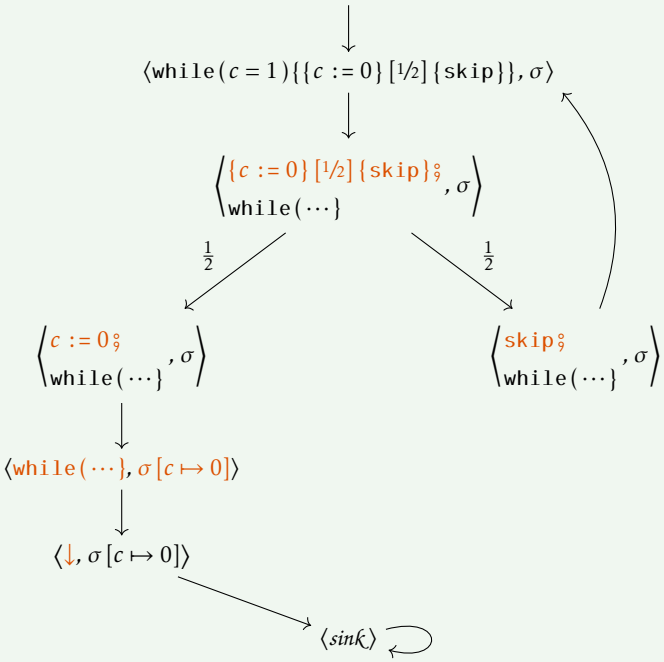


Figure 3.5: Operational MDP $\mathcal{M}^{C,\sigma}$ of executing the probabilistic program C = `while(`$c = 1$`){{`$c := 0$`} [1/2] {skip}}` on some state σ with $\sigma(c) = 1$. Unlabeled edges are transitions with probability 1. All transitions in this MDP are associated with action N.

putation tree was infinite. An advantage of having an MDP representation of a pGCL program's computation readily available is that it makes probabilistic programs amenable to *fully automated probabilistic model checking tools*, at least in case of finite operational MDPs. In case of infinite operational MDPs, *bounded model checking* is an appropriate technique. A first approach for bounded model checking of probabilistic programs which builds upon the Storm probabilistic model checker [Deh+17] was presented in [Jan+16].

Let us finally show how the MDP semantics is related to the distributions over final states presented in the previous section. We do so by constructing from an operational MDP an operational *Markov chain* (MC) that is induced by a given scheduler.

Definition 3.12 (Operational Markov Chains Induced by Schedulers):

A. *Let $\mathcal{M}^{C,\sigma} = (S, \langle C, \sigma\rangle, \{L, R, N\}, P')$ be the operational MDP of executing C on σ and let $\mathfrak{s} \in \mathsf{Scheds}$ be a scheduler. Then the operational MC of executing program $C \in \mathsf{pGCL}$ on input $\sigma \in \Sigma$ under scheduler $\mathfrak{s}$ is the MC (cf.* Definition B.3*) $\mathcal{M}^{C,\sigma}_{\mathfrak{s}} = (S, \langle C, \sigma\rangle, P)$, where for all $s, s' \in S$ we have*[5]

$$P(s)(s') \;=\; P'\Big(s, \mathfrak{s}(s)\Big)(s')\,.$$

B. *We denote by $\Pr_{\mathcal{M}^{C,\sigma}_{\mathfrak{s}}}(\Diamond s)$ the probability of eventually reaching state s in the operational MC $\mathcal{M}^{C,\sigma}_{\mathfrak{s}}$ (cf.* Definition B.4*).*

Schedulers for MDPs thus play the same role as schedulers in computation trees: their purpose is to resolve the nondeterminism in order to obtain a fully probabilistic transition system. Having means of resolving nondeterminism in MDPs as well as computation trees, we can now relate reachability probabilities in operational MCs to the distributions over final states presented in the previous section.

Proposition 3.13:
Let $\mathfrak{s} \in \mathsf{Scheds}$ be a scheduler, let $[\![C]\!]^{\mathfrak{s}}_{\sigma}$ be the distribution over final states established by executing C on input σ under scheduler $\mathfrak{s}$, and let $\mathcal{M}^{C,\sigma}_{\mathfrak{s}}$ be the operational MC of executing C on σ under scheduler $\mathfrak{s}$. Then for all final states $\tau \in \Sigma$,

$$[\![C]\!]^{\mathfrak{s}}_{\sigma}(\tau) \;=\; \Pr_{\mathcal{M}^{C,\sigma}_{\mathfrak{s}}}\Big(\Diamond\langle\downarrow, \tau\rangle\Big)\,.$$

Intuitively the above proposition states that the probability of reaching a certain final program state τ in a computation tree under scheduler $\mathfrak{s}$ is the

5 We are being a little bit sloppy on notation here: a scheduler $\mathfrak{s}$ is a function of type $\mathsf{pGCL} \times \Sigma \to \{L, R\}$ and thus takes as argument a *pair*. However, since an MDP state s for us is a pair from the set $\mathsf{pGCL} \times \Sigma$, we can safely write $\mathfrak{s}(s)$.

same as reaching the same final state in the operational MDP under the same scheduler $\mathfrak{s}$. This ties together all three semantical point of views that we have presented: computation trees, distributions over final states, and operational MDPs.

We conclude this section with a remark on the type of schedulers we employ. These are so–called *positional* or *history–independent*, i.e. they base their decision only on the current state of the MDP, not the history of visited states (cf. Definition B.2 B). The class of positional schedulers is sufficient to capture minimal and maximal reachability probabilities (see [Put05, Proposition 4.4.3.a]) as well as minimal and maximal expected rewards (see [Put05, Theorem 6.10.4]). The notion of expected rewards in MDPs (cf. Definition B.5) is needed for relating weakest preexpectation transformers to MDP semantics [GKM12; GKM14].

WEAKEST PREEXPECTATION REASONING 4

Lifting weakest precondition reasoning and anticipated value reasoning for (non)deterministic programs (see Chapter 2) to probabilistic programs *with nondeterminism* (see Chapter 3) is the subject matter of this chapter. We will present so–called *expectation transformer* calculi for quantitative reasoning about partial and total correctness of such programs. Furthermore, we study some basic properties of these transformers.

The idea of expectation transformers goes back to Kozen's seminal work on probabilistic propositional dynamic logic (PPDL) [Koz85]. PPDL is a modal logic for reasoning about (in our terminology) tame probabilistic programs, i.e. probabilistic programs *without* nondeterminism. Later, McIver & Morgan (re)incorporated nondeterministic choice and developed the weakest preexpectation calculus [MMS96; MM05]. We will present the calculus in the style of McIver & Morgan here, although it should be noted that — on fully probabilistic programs — their calculus is very closely related to PPDL (basically, the two formalisms coincide).

4.1 REASONING ABOUT EXPECTED VALUES

In an effort to enable formal reasoning about probabilistic programs, we now lift the notion of anticipated value reasoning (which subsumes reasoning about predicates; see Section 2.3.2) to *weakest preexpectation reasoning* (which will subsume reasoning about probabilities of events). As a first example, consider the program

$$\{x := 5\}\,[4/5]\,\{x := 10\}\,.$$

In contrast to a deterministic program, the variable x may have value 5 *or* 10 after termination of the program. Hence, there is *no single anticipated value of* x. That fact renders the whole concept of an anticipated value useless *as is* in the context of the above probabilistic program.

The situation is similar to a nondeterministic choice, were we also did not necessarily have a single anticipated value available. More detrimentally even, we had no information whatsoever on what branch is going to be executed. We therefore chose the minimal (demonic nondeterminism) or maximal (angelic nondeterminism) anticipated value of x (see Section 2.3.3). For the program above, this would be 5 or 10, respectively.

For probabilistic choices, the situation is in some sense better: We do have some information on the further course of the execution, namely the

probability with which each of the two branches is executed. While this still does not tell us *for sure* what is going to happen, it does provide us with *quantitative information* that we can sensibly incorporate into reasoning about probabilistic programs.

Imagine in the program above that x is some sort of payoff, penalty, or something alike. Instead of determining a minimal or maximal anticipated value of x, a generally accepted and arguably very important concept in the realm of probability theory is the notion of the *expected value* of x. For the case of payoffs, we would then obtain a mean or average payoff.

In order to determine the expected value of x in the program above, we need to average the anticipated value of x from the left and the right branch, which is $\mathsf{wp}\,[\![x := 5]\!]\,(x) = 5$ and $\mathsf{wp}\,[\![x := 10]\!]\,(x) = 10$, respectively. Weighting those anticipated values with the probabilities with which the branches are executed then gives us the expected value

$$\tfrac{4}{5} \cdot \mathsf{wp}\,[\![x := 5]\!]\,(x) + \tfrac{1}{5} \cdot \mathsf{wp}\,[\![x := 10]\!]\,(x) \;=\; \tfrac{4}{5} \cdot 5 + \tfrac{1}{5} \cdot 10 \;=\; 6\,.$$

The explanations above already provide most of the intuition we need for extending the anticipated value calculus to reasoning about expected values for probabilistic programs. In the following, we gradually develop the weakest preexpectation calculus which allows us to conduct this sort of reasoning in a systematic way.

4.1.1 *Weakest Preexpectations*

Similarly to anticipated value reasoning (see Section 2.3), we are given a program C, an initial state σ, and a function f mapping (final) program states to positive reals or infinity. Let us first — for now — leave nondeterminism out of the picture and consider only tame programs C.

For the function f we would like to know its *expected value* with respect to the distribution over final states obtained by executing C on σ (i.e. the distribution $[\![C]\!]_\sigma$, see Section 3.3.2). Such a function f can thus simply be thought of as a random variable. We follow a widespread terminology here[1] and refer to the class of random variables we use in our setting as to *expectations*:

Definition 4.1 (Expectations):

A. *The set of expectations, denoted* $\mathbb{E}$*, is defined to coincide with the set of anticipations (see* Definition 2.10*), i.e.*

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

Consequently, the complete lattice $(\mathbb{E}, \preceq)$*, its least element, and the construction of suprema is defined exactly as for anticipations,*

1 See e.g. [CNZ17], [Fen+17], [Cha+16], [CS14], [Coc14], [GKM14], [CS13], [Kat+10], [APM09], [MM05], [Mon05], or [MM99].

i.e. the order relation is given by

$$f_1 \preceq f_2 \qquad \text{iff} \qquad \forall \sigma \in \Sigma\colon \quad f_1(\sigma) \leq f_2(\sigma)\,;$$

the least element is

$$\lambda\sigma.\ 0\,,$$

which we overloadingly denote by 0*; and the supremum of a subset* $S \subseteq \mathbb{E}$ *is constructed pointwise by*

$$\sup S \;=\; \lambda\sigma.\ \sup_{f \in S} f(\sigma)\,.$$

We write $f \ll g$ *to indicate that* f *is everywhere smaller than* g*, i.e.*

$$f \ll g \qquad \text{iff} \qquad \forall \sigma \in \Sigma\colon \quad f(\sigma) < g(\sigma)\,.$$

B. *The set of bounded expectations, denoted* $\mathbb{E}_{\leq \exists b}$ *is defined as*[2]

$$\mathbb{E}_{\leq \exists b} \;=\; \{f \in \mathbb{E} \mid \exists\, b \in \mathbb{R}_{\geq 0}\colon\ f \preceq b\}\,.$$

$(\mathbb{E}_{\leq \exists b}, \preceq)$ *is a lattice with least element 0 (as above) but it is* not complete *since suprema are not guaranteed to exist.*[3]

C. *The set of one–bounded expectations, denoted* $\mathbb{E}_{\leq 1}$ *is defined as*

$$\mathbb{E}_{\leq 1} \;=\; \{f \in \mathbb{E} \mid f \preceq 1\}\,.$$

$(\mathbb{E}_{\leq 1}, \preceq)$ *is a complete lattice with least element 0 (as in* A. *above) and greatest element* $\lambda\sigma.\ 1$ *which we overloadingly denote by* 1*. Suprema are constructed as in* $\mathbb{E}$*.*

We remark that McIver & Morgan's oeuvre on weakest preexpectation reasoning relies in almost its entirety on *bounded* expectations [MM05; KM17] (in particular, see [MM05, p. 25 (especially Footnote 39) and Section 2.11]). Bounded expectations do *not* form a complete lattice and existence of least fixed points is not a consequence of the Kleene fixed point theorem but has to be proven by different means [MM05, Lemma 5.6.8].

We, on the other hand, take a more general view in which expectations may generally be *unbounded* and even evaluate to infinity. Indeed, we depend on these more general unbounded expectations because we will later use expectations to reason about expected runtimes, which in general cannot bounded by a constant but depend on the input.

2 For $b \in \mathbb{R}_{\geq 0}$, we write $f \preceq b$ to mean $f \preceq \lambda\sigma.\ b$.
3 E.g., the set $\{\lambda\sigma.\ 1, \lambda\sigma.\ 2, \lambda\sigma.\ 3, \ldots\} \subset \mathbb{E}_{\leq \exists b}$ has supremum $\lambda\sigma.\ \infty \in \mathbb{E}$ but $\lambda\sigma.\ \infty \notin \mathbb{E}_{\leq \exists b}$.

Reasoning about expected values. Analogously to anticipated value reasoning, we will refer to the expectation whose expected value we want to know as to a *postexpectation*. Given a postexpectation $f \in \mathbb{E}$ and a probabilistic program C, we would like to know a function $g \in \mathbb{E}$ that maps each (initial) state σ to the expected value of f after execution of C on input σ. We call this function g the

weakest preexpectation of C with respect to postexpectation f

and denote it by $\mathsf{wp}\,[\![C]\!]\,(f)$. The characterizing equation of a weakest preexpectation is given by

$$\mathsf{wp}\,[\![C]\!]\,(f) \;=\; \lambda\sigma.\; \int_{\Sigma} f \; d\,[\![C]\!]_\sigma\,, \tag{4.1}$$

where we denote by $\int_\Sigma h\, d\,\mu$ the expected value of expectation (read: random variable) h with respect to a distribution μ over the set of program states Σ.

Example 4.2 (Weakest Preexpectations):
Consider the program

$$\{x := x+5\}\,[4/5]\,\{x := 10\}\,.$$

Suppose we want to know the expected value of x, i.e. the weakest preexpectation of the above program with respect to postexpectation x. This weakest preexpectation is given by

$$\tfrac{4}{5}\cdot(x+5) + \tfrac{1}{5}\cdot 10 \;=\; \frac{4x}{5} + 6\,.$$

When executing the above program on initial state σ, the expected value of x is hence $4\sigma(x)/5 + 6$.

Reasoning about probabilities. An important special case is when the postexpectation given as $[F]$, where F is a predicate. In that case, we can think of F as an event and $\mathsf{wp}\,[\![C]\!]\,([F])(\sigma)$ is then the probability that executing C on input σ will terminate in a final state $\tau \models F$, or in other words: $\mathsf{wp}\,[\![C]\!]\,([F])(\sigma)$ is the probability that event F occurs after termination of C on input σ.

Example 4.3 (Probabilities of Events as Weakest Preexpectations):
Reconsider the program

$$\{x := x+5\}\,[4/5]\,\{x := 10\}$$

from Example 4.2 and take predicate (i.e. event) $F = (x{=}10)$. Then the weakest preexpectation of this program with respect to postexpectation $[F]$ is

$$\tfrac{4}{5} \cdot [x+5=10] + \tfrac{1}{5} \cdot [\mathsf{true}] \;=\; \frac{4 \cdot [x=5]+1}{5}\,.$$

Thus, for any initial state σ, the probability that x is 10 after executing the above program on σ is $(4\cdot 1+1)/5 = 1$ if $\sigma(x) = 5$, and $(4\cdot 0+1)/5 = 1/5$ otherwise.

Reasoning about Nondeterminism. We now bring nondeterminism back into the picture. As we have seen in Section 3.3.2, we have to resolve all nondeterminism occurring along the computation of a pGCL program in order to sensibly obtain a probability distribution over final states. Resolving nondeterminism was achieved by so–called schedulers that resolve all nondeterminism occurring in the computation tree. We thereby obtained a distribution $[\![C]\!]^{\mathfrak{s}}_{\sigma}$ relative to some scheduler $\mathfrak{s}$.

As we can see in Equation 4.1, for characterizing weakest preexpectations we need a probability distribution. For programs with nondeterminism, we will have to choose a scheduler in order to obtain a probability distribution. *A* sensible choice is a scheduler that *minimizes* the preexpectation. The characterizing equation of weakest preexpectations for full pGCL is thus given by

$$\mathsf{wp}\,[\![C]\!]\,(f) \quad = \quad \lambda\sigma.\;\; \inf_{\mathfrak{s}\in\mathsf{Scheds}} \int_{\Sigma} f \; d\,[\![C]\!]^{\mathfrak{s}}_{\sigma}\,. \tag{4.2}$$

The above minimizing exegesis of weakest preexpectations agrees with Dijkstra's original notion of weakest preconditions of (nonprobabilistic) nondeterministic programs. For anticipated value reasoning, we called this interpretation *demonic* nondeterminism.

Angelic nondeterminism, i.e. a *maximizing* exegesis, is in some cases an equally sensible choice: If we think about expected runtimes for instance, a maximizing scheduler cannot even be conceived as very *angelic* in the truest sense of the word, but indeed as a *demonic* worst–case. The characterizing equation of angelic weakest preexpectations for full pGCL is given by

$$\mathsf{awp}\,[\![C]\!]\,(f) \quad = \quad \lambda\sigma.\;\; \sup_{\mathfrak{s}\in\mathsf{Scheds}} \int_{\Sigma} f \; d\,[\![C]\!]^{\mathfrak{s}}_{\sigma}\,. \tag{4.3}$$

4.1.2 *Weakest Liberal Preexpectations*

For deterministic programs, the weakest liberal pre*condition* of a program C with respect to a postcondition (i.e. a predicate) F is a predicate G such that the execution of C on an initial state $\sigma \models G$ will either diverge or terminate in a state $\tau \models F$ (cf. Section 2.2.5). Weakest liberal pre*expectations* are the probabilistic analog to this concept:

For a predicate F, the weakest liberal pre*expectation* of C with respect to postexpectation $[F]$ is an expectation $g \in \mathbb{E}_{\leq 1}$ such that $g(\sigma)$ equals the *probability* that executing C on input σ will either diverge or terminate in a state $\tau \models F$. In other words: $g(\sigma)$ is the probability that event F occurs *if* C terminates on σ. More generally, for any $f \in \mathbb{E}_{\leq 1}$, the

weakest liberal preexpectation of C with respect to postexpectation f,

denoted by $\mathsf{wlp}\,[\![C]\!]\,(f)$, is an expectation in $\mathbb{E}_{\leq 1}$ such that the expected value of f after execution of C on an initial state σ plus the probability that C does not terminate on σ equals $\mathsf{wlp}\,[\![C]\!]\,(f)(\sigma)$, formally

$$\mathsf{wlp}\,[\![C]\!]\,(f) \quad = \quad \lambda\sigma.\ \inf_{\mathfrak{s}\in\mathsf{Scheds}} \int_{\Sigma} f \; d\,[\![C]\!]^{\mathfrak{s}}_{\sigma} \;+\; \left(1 - \int_{\Sigma} 1 \; d\,[\![C]\!]^{\mathfrak{s}}_{\sigma}\right), \tag{4.4}$$

where the infimum on the right–hand–side accounts for possible demonic nondeterminism occurring in the program.

EXAMPLE 4.4 (Weakest Liberal Preexpectations):

A. Consider the program

$$\{\texttt{diverge}\}\,[1/3]\,\{x := 10\}$$

and take predicate (read: event) $F = (x{=}10)$. Then the weakest liberal preexpectation of this program with respect to postexpectation $[F]$ is

$$\tfrac{1}{3}\cdot 1 + \tfrac{2}{3}\cdot 1 \;=\; 1.$$

B. Consider the program

$$\{\texttt{diverge}\}\,[1/3]\,\{x := x+5\}\,.$$

Then the weakest liberal preexpectation of the above program with respect to postexpectation $[F]$ from A. is

$$\tfrac{1}{3}\cdot 1 + \tfrac{2}{3}\cdot [x+5=10] \;=\; \frac{1+2\cdot[x=5]}{3}.$$

C. Consider the program

$$\begin{array}{l} c := 1 \\ \texttt{while}(c=1)\{ \\ \quad \{\texttt{diverge}\}\,[1/2]\,\{x := x+1\}\,; \\ \quad \{\texttt{skip}\}\,[1/2]\,\{c := 0\} \\ \} \end{array}$$

and take the event that x is even. Then the weakest liberal preexpectation of this program with respect to postexpectation $[x \text{ even}]$ is

$$\frac{2}{3} + \frac{4 \cdot [x \text{ odd}]}{15} + \frac{[x \text{ even}]}{15} .$$

4.1.3 *The Weakest Preexpectation Calculus*

So far, we have seen characterizations for (angelic) weakest (liberal) preexpectations, but we have not seen how to systematically determine them given a concrete program and postexpectation. It turns out that for (angelic) weakest preexpectations this can be done analogously to anticipated value reasoning (see Section 2.3) and for weakest liberal preexpectations this can be done analogously to weakest liberal precondition reasoning (see Section 2.2.5). Thus, we define continuation–passing style, backwards–moving expectation transformers as follows:

DEFINITION 4.5 (Expectation Transformers [Koz85; MM05]):
For $C \in$ pGCL we define the following expectation transformers:

A. *The weakest preexpectation transformer*

$$\mathsf{wp}\,[\![C]\!] \colon \quad \mathbb{E} \to \mathbb{E}$$

is defined according to the rules in Table 4.1.

B. *The weakest liberal preexpectation transformer*

$$\mathsf{wlp}\,[\![C]\!] \colon \quad \mathbb{E}_{\leq 1} \to \mathbb{E}_{\leq 1}$$

is defined according to the rules in Table 4.2.

C. *The angelic weakest preexpectation transformer*

$$\mathsf{awp}\,[\![C]\!] \colon \quad \mathbb{E} \to \mathbb{E}$$

is defined according to the rules obtained from Table 4.1 *by replacing every occurrence of* wp *by* awp *and the* min *by a* max.

D. *The angelic weakest liberal preexpectation transformer*

$$\mathsf{awlp}\,[\![C]\!] \colon \quad \mathbb{E}_{\leq 1} \to \mathbb{E}_{\leq 1}$$

is defined according to the rules obtained from Table 4.2 *by replacing every occurrence of* wlp *by* awlp *and the* min *by a* max.

C	$\mathsf{wp}[\![C]\!](f)$
`skip`	f
`diverge`	0
$x := E$	$f[x/E]$
$x :\approx \mu$	$\lambda\sigma.\ \int_{\mathsf{Vals}} \Big(\lambda v.\ f\big(\sigma[x \mapsto v]\big)\Big)\, d\,\mu_\sigma$
$C_1 \fatsemi C_2$	$\mathsf{wp}[\![C_1]\!]\big(\mathsf{wp}[\![C_2]\!](f)\big)$
$\texttt{if}\,(\varphi)\,\{C_1\}\,\texttt{else}\,\{C_2\}$	$[\varphi]\cdot \mathsf{wp}[\![C_1]\!](f) \;+\; [\neg\varphi]\cdot \mathsf{wp}[\![C_2]\!](f)$
$\{C_1\} \,\square\, \{C_2\}$	$\min\Big\{\mathsf{wp}[\![C_1]\!](f), \mathsf{wp}[\![C_2]\!](f)\Big\}$
$\{C_1\}\,[p]\,\{C_2\}$	$p\cdot \mathsf{wp}[\![C_1]\!](f) \;+\; (1-p)\cdot \mathsf{wp}[\![C_2]\!](f)$
$\texttt{while}(\varphi)\{C'\}$	$\mathsf{lfp}\, X.\ [\neg\varphi]\cdot f \;+\; [\varphi]\cdot \mathsf{wp}[\![C']\!](X)$

Table 4.1: The weakest preexpectation transformer.

E. *For* wp, *we call the function*

$$ {}^{\mathsf{wp}}_{\langle\varphi,C\rangle}\Phi_f(X) \;=\; [\neg\varphi]\cdot f \;+\; [\varphi]\cdot \mathsf{wp}[\![C]\!](X) $$

the wp*–characteristic function of* $\texttt{while}(\varphi)\{C\}$ *with respect to postexpectation* f*. We define the* wlp*–,* awp*–, and* awlp*–characteristic functions* ${}^{\mathsf{wlp}}_{\langle\varphi,C\rangle}\Phi_f$, ${}^{\mathsf{awp}}_{\langle\varphi,C\rangle}\Phi_f$, *and* ${}^{\mathsf{awlp}}_{\langle\varphi,C\rangle}\Phi_f$ *analogously. If either of* wp, wlp, awp, awlp, φ, C, *or* f *are clear from the context, we omit them from* Φ*.*

Notice that by the Kleene Fixed Point Theorem (Theorem A.5) we have

$$ \mathsf{wp}[\![\texttt{while}(\varphi)\{C\}]\!](f) \;=\; \mathsf{lfp}\, {}^{\mathsf{wp}}_{\langle\varphi,C\rangle}\Phi_f \;=\; \sup_{n\in\mathbb{N}} {}^{\mathsf{wp}}_{\langle\varphi,C\rangle}\Phi^n_f(0)\,, $$

where ${}^{\mathsf{wp}}_{\langle\varphi,C\rangle}\Phi^n_f$ denotes n-fold application of ${}^{\mathsf{wp}}_{\langle\varphi,C\rangle}\Phi_f$ to its argument. The analogous statement holds for awp. For wlp, since this is defined via a greatest fixed point, we have a dual statement, namely

$$ \mathsf{wlp}[\![\texttt{while}(\varphi)\{C\}]\!](f) \;=\; \mathsf{gfp}\, {}^{\mathsf{wlp}}_{\langle\varphi,C\rangle}\Phi_f \;=\; \inf_{n\in\mathbb{N}} {}^{\mathsf{wlp}}_{\langle\varphi,C\rangle}\Phi^n_f(1)\,. $$

Analogous statements hold for the angelic expectation transformers.

An immediate corollary about preexpectations of loops which can be derived by a close inspection of the characteristic function is the following:

C	$\mathsf{wlp}\,[\![C]\!]\,(f)$
`skip`	f
`diverge`	1
$x := E$	$f\,[x/E]$
$x :\approx \mu$	$\lambda\sigma.\ \int_{\mathsf{Vals}} \Big(\lambda v.\ f\big(\sigma\,[x \mapsto v]\big)\Big)\, d\,\mu_\sigma$
$C_1 \fatsemi C_2$	$\mathsf{wlp}\,[\![C_1]\!]\,\big(\mathsf{wlp}\,[\![C_2]\!]\,(f)\big)$
$\texttt{if}\,(\varphi)\,\{C_1\}\,\texttt{else}\,\{C_2\}$	$[\varphi]\cdot \mathsf{wlp}\,[\![C_1]\!]\,(f) \;+\; [\neg\varphi]\cdot \mathsf{wlp}\,[\![C_2]\!]\,(f)$
$\{C_1\}\,\square\,\{C_2\}$	$\min\big\{\mathsf{wlp}\,[\![C_1]\!]\,(f),\, \mathsf{wlp}\,[\![C_2]\!]\,(f)\big\}$
$\{C_1\}\,[p]\,\{C_2\}$	$p\cdot \mathsf{wlp}\,[\![C_1]\!]\,(f) \;+\; (1-p)\cdot \mathsf{wlp}\,[\![C_2]\!]\,(f)$
$\texttt{while}(\varphi)\{C'\}$	$\mathsf{gfp}\ X.\ [\neg\varphi]\cdot f \;+\; [\varphi]\cdot \mathsf{wlp}\,[\![C']\!]\,(X)$

Table 4.2: The weakest liberal preexpectation transformer.

COROLLARY 4.6 (Postexpectation Strengthening for Loops): *Let $C \in$ pGCL and $\mathcal{T} \in \{\mathsf{wp}, \mathsf{wlp}, \mathsf{awp}, \mathsf{awlp}\}$. Then*

$$\mathcal{T}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(f) \;=\; \mathcal{T}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,([\neg\varphi]\cdot f)\ ,$$

for an appropriate choice of $f \in \mathbb{E}$ or $f \in \mathbb{E}_{\leq 1}$ (depending on $\mathcal{T}$).

Proof. The $\mathcal{T}$–characteristic function of $\texttt{while}(\varphi)\{C\}$ with respect to post-expectation f is given by

$$\begin{aligned}&\lambda X.\ [\neg\varphi]\cdot f \;+\; [\varphi]\cdot \mathcal{T}\,[\![C]\!]\,(X)\\ &\;=\; \lambda X.\ [\neg\varphi]\cdot[\neg\varphi]\cdot f \;+\; [\varphi]\cdot \mathcal{T}\,[\![C]\!]\,(X)\ ,\end{aligned}$$

which is the $\mathcal{T}$–characteristic function with respect to $[\neg\varphi]\cdot f$. Since the characteristic functions coincide, so do their fixed points. Q.E.D.

Intuitively, the above corollary can be interpreted as the fact that a loop can only ever terminate in a state which satisfies the negation of the loop guard.

Notice that all rules for the anticipated value transformer in Table 2.3 are also found in Table 4.1. For the weakest preexpectation transformer, we have merely added rules for the probabilistic constructs. The conceptually easier one, namely the rule for probabilistic choice, reads

$$\mathsf{wp}\,[\![\{C_1\}\,[p]\,\{C_2\}]\!]\,(f) \;=\; p\cdot \mathsf{wp}\,[\![C_1]\!]\,(f) \;+\; (1-p)\cdot \mathsf{wp}\,[\![C_2]\!]\,(f)$$

The intuition behind this definition is straightforward: Since we cannot single out a value of f which is established by either C_1 or C_2, we simply average

these two values according to the probabilities with which C_1 and C_2 are executed, respectively, thus obtaining the *expected value* of f after executing C_1 with probability p and C_2 with probability $1-p$.

The rule for random assignments is technically more involved and reads

$$\mathsf{wp}\,[\![x :\approx \mu]\!]\,(f) \;=\; \lambda\sigma.\ \int_{\mathsf{Vals}} \Big(\lambda v.\ f\big(\sigma\,[x \mapsto v]\big)\Big)\, d\,\mu_\sigma\ .$$

The mechanics of the right–hand–side are as follows: Instead of averaging only over two options, we average now over updated versions of f according to a probability distribution. In more detail, the right–hand–side takes as input a state σ and averages the values of $f(\sigma\,[x \mapsto v])$ (i.e. f updated according to assignment $x := v$), where the values v are distributed according to probability distribution μ_σ.

Example 4.7 (Reasoning about Expected Values):
Reconsider the program

$$\{x := x+5\}\,[4/5]\,\{x := 10\}\ .$$

Suppose we want to know the expected value of x, i.e. the weakest preexpectation of the above program with respect to postexpectation x. We will reuse our annotation style from earlier (see Example 2.11), i.e.

$$\begin{array}{l} /\!/\!/\ g' \\ /\!/\!/\ g \\ C \\ /\!/\!/\ f \end{array}$$

expresses the fact that $g = \mathsf{wp}\,[\![C]\!]\,(f)$ and moreover that $g' = g$. We can then annotate the above program as shown in Figure 4.1 (read from bottom to top). When executing the above program on initial state σ, the expected value of x is hence $4\sigma(x)/5 + 6$.

Example 4.8 (Reasoning about Probabilities):
Reconsider the program from Example 4.7. Suppose we want to know the probability that x has value 10 after execution of that program. Then we can annotate this program as shown in Figure 4.2. When executing the above program on initial state σ with $\sigma(x) = 5$, then the probability that x equals 10 is $4/5 + 1/5 = 1$. Otherwise, it is $1/5$.

```
/// 4x/5 + 6
/// 4/5 · (x + 5) + 1/5 · 10
{
    /// x + 5
    x := x + 5
    /// x
} [4/5] {
    /// 10
    x := 10
    /// x
}
/// x
```

Figure 4.1: Weakest preexpectation annotations for Example 4.7.

```
/// 4/5 · [x = 5] + 1/5
/// 4/5 · [x = 5] + 1/5 · 1
{
    /// [x = 5]
    /// [x + 5 = 10]
    x := x + 5
    /// [x = 10]
} [4/5] {
    /// 1
    /// [true]
    /// [10 = 10]
    x := 10
    /// [x = 10]
}
/// [x = 10]
```

Figure 4.2: Weakest preexpectation annotations for Example 4.8.

EXAMPLE 4.9 (Weakest Liberal Preexpectations):
Reconsider the program

$$\{\texttt{diverge}\}\,[1/3]\,\{x := 10\}\,.$$

Suppose we want to know the probability that either the program terminates in a state where x has value 10 after execution of that program or the program diverges, i.e. the weakest *liberal* preexpectation of that program with respect to postexpectation $[x = 10]$. Then we can annotate this program as shown in Anticipated Values and Nontermination 4.9. The sought–after probability is thus 1.

EXAMPLE 4.10 (Weakest Liberal Preexpectations of While Loops):
Reconsider the program

$$\begin{array}{l} c := 1\,; \\ \texttt{while}(c = 1)\{ \\ \quad \{\texttt{diverge}\}\,[1/2]\,\{x := x+1\}\,; \\ \quad \{\texttt{skip}\}\,[1/2]\,\{c := 0\} \\ \}\,. \end{array}$$

Suppose we want to reason about the weakest liberal preexpectation of event „x is even“. The characteristic function of the loop with respect to postexpectation $[x \text{ even}]$ is given by

$$\begin{aligned} \Phi(X) \;=\; & [c \neq 1] \cdot [x \text{ even}] \\ & + \; [c = 1] \cdot \left(\frac{1}{2} + \frac{X[x/x+1] + X[c,x/0,x+1]}{4} \right). \end{aligned}$$

The first four iterations of the fixed point iteration for Φ are:

$$\begin{aligned} \Phi(1) \;&=\; [c \neq 1] \cdot [x \text{ even}] + [c = 1] \\ \Phi^2(1) \;&=\; [c \neq 1] \cdot [x \text{ even}] \;+\; [c = 1] \cdot \left(\frac{3}{4} + \frac{[x \text{ odd}]}{4} \right) \\ \Phi^3(1) \;&=\; [c \neq 1] \cdot [x \text{ even}] \;+\; [c = 1] \cdot \left(\frac{11}{16} + \frac{[x \text{ even}]}{16} + \frac{[x \text{ odd}]}{4} \right) \\ \Phi^4(1) \;&=\; [c \neq 1] \cdot [x \text{ even}] \\ &\qquad + \; [c = 1] \cdot \left(\frac{43}{64} + \frac{[x \text{ odd}]}{64} + \frac{[x \text{ even}]}{16} + \frac{[x \text{ odd}]}{4} \right) \end{aligned}$$

```
/// 1
/// 4/5 · 1 + 1/5 · 1
{
    /// 1
    diverge
    /// [x = 10]
} [4/5] {
    /// 1
    /// [true]
    /// [10 = 10]
    x := 10
    /// [x = 10]
}
/// [x = 10]
```

Figure 4.3: Weakest liberal preexpectation annotations for Example 4.9.

More detailed calculations are left as an exercise. After four iterations, we can slowly start seeing a somewhat complicated pattern for $n > 2$:

$$\Phi^n(1) \;=\; [c \neq 1]\cdot[x \text{ even}] \;+\; [c=1]\cdot\left(\frac{2^{n-1}+1}{4^{n-1}} + \sum_{i=0}^{\lfloor\frac{n-3}{2}\rfloor}\frac{[x \text{ even}]}{4^{2(i+1)}} + \sum_{i=0}^{\lfloor\frac{n-2}{2}\rfloor}\frac{[x \text{ odd}]}{4^{2i+1}}\right)$$

We omit proving the above pattern correct. By taking the limit (i.e. $n \to \omega$), we see that the sought–after weakest liberal preexpectation converges to

$$\begin{aligned}
&\mathsf{wp}\,[\![\texttt{while}(x>0)\{\ldots\}]\!]\,(z)\\
&= \mathsf{lfp}\,\Phi\\
&= \sup_{n\in\mathbb{N}} \Phi^n(0) \qquad \text{(by Theorem A.5)}\\
&= \sup_{n\in\mathbb{N}} [c\neq 1]\cdot[x \text{ even}] \;+\; [c=1]\cdot\left(\frac{2^{n-1}+1}{4^{n-1}} + \sum_{i=0}^{\lfloor\frac{n-2}{2}\rfloor}\frac{[x \text{ odd}]}{4^{2i+1}} + \sum_{i=0}^{\lfloor\frac{n-3}{2}\rfloor}\frac{[x \text{ even}]}{4^{2(i+1)}}\right)\\
&= [c\neq 1]\cdot[x \text{ even}] \;+\; [c=1]\cdot\left(\frac{2}{3} + \frac{4\cdot[x \text{ odd}]}{15} + \frac{[x \text{ even}]}{15}\right).
\end{aligned}$$

For the whole program, we can finally make these annotations:

$$\begin{aligned}
&/\!/\!/\ \tfrac{2}{3} + \tfrac{4\cdot[x \text{ odd}]}{15} + \tfrac{[x \text{ even}]}{15}\\
&c := 1\,;\\
&/\!/\!/\ [c\neq 1]\cdot[x \text{ even}] \;+\; [c=1]\cdot\left(\tfrac{2}{3} + \tfrac{4\cdot[x \text{ odd}]}{15} + \tfrac{[x \text{ even}]}{15}\right)\\
&\texttt{while}(c=1)\{\\
&\quad \{\texttt{diverge}\}\,[1/2]\,\{x := x+1\}\,;\\
&\quad \{\texttt{skip}\}\,[1/2]\,\{c := 0\}\\
&\}\\
&/\!/\!/\ [x \text{ even}]
\end{aligned}$$

We have thus proven

$$\mathsf{wlp}\,[\![\ldots]\!]\,([x \text{ even}]) \;=\; \frac{2}{3} + \frac{4\cdot[x \text{ odd}]}{15} + \frac{[x \text{ even}]}{15}.$$

This means that if we start the program in a state where x is odd, then there is a probability of $2/3 + 4/15 = 14/15$ that the program either not terminates or

terminates in a state where x is even. If we start the program in a state where x is even, this probability is $11/15$. Notice in particular that $14/15 + 11/15 = 5/3 > 1$.

4.1.4 *Connection to Operational Semantics*

Recall the characterizing equations of wp (Equation 4.2, p. 81), awp (Equation 4.3, p. 81), and wlp (Equation 4.4, p. 82). The next theorem states formally that those expectation transformers actually satisfy those equations:

Theorem 4.11 (Operational vs. Expectation Transformer Semantics): *Let $C \in \mathsf{pGCL}$, $f \in \mathbb{E}$, and $g \in \mathbb{E}_{\leq 1}$. Then:*

A. $$\mathsf{wp}\,[\![C]\!]\,(f) \;=\; \lambda\sigma.\ \inf_{\mathfrak{s}\in\mathsf{Scheds}} \int_{\Sigma} f\, d\,[\![C]\!]^{\mathfrak{s}}_{\sigma}$$

B. $$\mathsf{awp}\,[\![C]\!]\,(f) \;=\; \lambda\sigma.\ \sup_{\mathfrak{s}\in\mathsf{Scheds}} \int_{\Sigma} f\, d\,[\![C]\!]^{\mathfrak{s}}_{\sigma}$$

C. $$\mathsf{wlp}\,[\![C]\!]\,(f) \;=\; \lambda\sigma.\ \inf_{\mathfrak{s}\in\mathsf{Scheds}} \int_{\Sigma} f\, d\,[\![C]\!]^{\mathfrak{s}}_{\sigma} \;+\; \left(1 - \int_{\Sigma} 1\, d\,[\![C]\!]^{\mathfrak{s}}_{\sigma}\right)$$

D. $$\mathsf{awlp}\,[\![C]\!]\,(f) \;=\; \lambda\sigma.\ \sup_{\mathfrak{s}\in\mathsf{Scheds}} \int_{\Sigma} f\, d\,[\![C]\!]^{\mathfrak{s}}_{\sigma} \;+\; \left(1 - \int_{\Sigma} 1\, d\,[\![C]\!]^{\mathfrak{s}}_{\sigma}\right)$$

Proof. By induction on the structure of C.

Theorem 4.11 establishes a connection between the expectation transformers and the probability distribution over final states obtained by the operational semantics $[\![C]\!]^{\mathfrak{s}}_{\sigma}$ under some scheduler (see Definition 3.8). A connection between the distribution transformer semantics of Kozen [Koz79; Koz81] and a Markov process semantics was shown earlier by Sharir, Pnueli, and Hart [SPH84]. A formal connection between Markov chains and weakest preexpectations with an emphasis on invariants was studied more recently by Gretz, Katoen, and McIver [GKM12; GKM14].

4.2 HEALTHINESS CONDITIONS

Expectation transformers enjoy several properties like continuity, monotonicity, etc. In the literature, some of these properties are called *healthiness conditions* [MM05; Kei15; Hin+16] or *homomorphism properties* [BW98]. Informally speaking, healthiness conditions are a collection of properties that characterize those backward–moving predicate (or expectation) transformers that are the dual of a forward–moving state (or distribution) transformer that arises from an actual (probabilistic) program.

Many of the properties we present here will be used in our proofs. In their own right, they aid in concrete reasoning about probabilistic programs, for instance by forming a foundation for compositional reasoning.

4.2.1 *Continuity*

Continuity is perhaps one of the most fundamental properties that expectation transformers enjoy because it ensures for instance well–definedness of expectation transformer semantics of while loops. An expectation transformer $\mathcal{T} : \mathbb{E} \to \mathbb{E}$ is continuous iff for any chain of expectations $S = \{s_0 \preceq s_1 \preceq s_2 \preceq \ldots\} \subseteq \mathbb{E}$ we have

$$\mathcal{T}(\sup S) \;=\; \sup\, \mathcal{T}(S)\,;$$

see Definition A.2 for more details. All expectation transformers we have presented in this thesis are continuous:

> **Theorem 4.12 (Continuity of Expectation Transformers):**
> *Let C be a* pGCL *program. Then the associated expectation transformers* wp $[\![C]\!]$, wlp $[\![C]\!]$, awp $[\![C]\!]$, *and* awlp $[\![C]\!]$ *are continuous.*

Proof. By induction on the structure of C. Q.E.D.

The importance of continuity for well–defined semantics of loops can be sketched as follows: For any loop–free program C, continuity of wp $[\![C]\!]$ ensures that the characteristic function of the loop $\texttt{while}(\varphi)\{C\}$ (that has C as its loop body) is also continuous. This ensures by the Kleene fixed point theorem (Theorem A.5) that the characteristic function has a least fixed point, which in turn ensures that wp $[\![\texttt{while}(\varphi)\{C\}]\!]$ is well–defined. The fact that the transformer wp $[\![\texttt{while}(\varphi)\{C\}]\!]$ itself is also continuous ensures well–defined expectation transformer semantics of nested loops.

4.2.2 *Strictness*

Strictness is a healthiness condition that Dijkstra calls „Law of the Excluded Miracle“ [Dij75]. For his weakest precondition calculus, this law states that there exists no initial state from which the execution of a program C can terminate in a state satisfying the predicate false. In terms of wp, this law reads

$$\text{wp}\,[\![C]\!]\,(\text{false}) \;=\; \text{false}\,.$$

For weakest liberal preconditions a dual law (that we call *costrictness*) would state that for all initial states the execution of a program C either terminates (in some state state satisfying true) or does not terminate. In terms of wlp,

this can be expressed as

$$\mathsf{wlp}\,[\![C]\!]\,(\mathsf{true}) \;=\; \mathsf{true}\,.$$

In our quantitative setting, strictness and costrictness are defined as follows:

DEFINITION 4.13 (Strictness and Costrictness):
Let $C \in \mathsf{pGCL}$ and let $\mathcal{T} : \mathbb{E} \to \mathbb{E}$ be an expectation transformer. Then:

A. *$\mathcal{T}$ is called strict, iff*

$$\mathcal{T}(0) \;=\; 0\,.$$

B. *$\mathcal{T}$ is called costrict, iff*

$$\mathcal{T}(1) \;=\; 1\,.$$

Analogously to Dijkstra's predicate transformers, liberal expectation transformers are costrict and their nonliberal versions are strict:

THEOREM 4.14 (Strictness of Expectation Transformers):
Let $C \in \mathsf{pGCL}$. Then:

A. *$\mathsf{wp}\,[\![C]\!]$ and $\mathsf{awp}\,[\![C]\!]$ are strict.*

B. *$\mathsf{wlp}\,[\![C]\!]$ and $\mathsf{awlp}\,[\![C]\!]$ are costrict.*[4]

Proof. Strictness of wp and awp follows from feasibility of wp and awp, respectively; see Section 4.2.3. Q.E.D.

The quantitative version of strictness tells us that the expected value of the constantly 0 random variable after executing a program C is 0. Alternatively stated: the probability that C terminates in a state satisfying false is 0. Costrictness tells us that the probability to either terminate or not is 1.

4.2.3 *Feasibility*

The property that McIver & Morgan call *feasibility* states that preexpectations cannot become „too large" [MM05]. The notion of feasibility makes sense for bounded expectations $f \in \mathbb{E}_{\leq \exists b}$ only. Formally, it is stated as follows:

THEOREM 4.15 (Feasibility of Expectation Transformers[5]):
Let $C \in \mathsf{pGCL}$. Moreover, let $f \in \mathbb{E}_{\leq \exists b}$ be an expectation bounded by $b \in \mathbb{R}_{\geq 0}$, i.e. $f \preceq b$. Then

$$\mathsf{wp}\,[\![C]\!]\,(f) \;\preceq\; b \quad \text{and} \quad \mathsf{awp}\,[\![C]\!]\,(f) \;\preceq\; b\,.$$

4 For costrictness of wlp, see [MM05, Fact B.3.4 on p. 331].
5 See [MM05, Lemma 5.6.4 *and* p. 228].

Feasibility of wp implies its strictness and can thus be seen as a quantitative generalization of strictness. To see that feasibility implies strictness observe that 0 is a 0–bounded expectation and feasibility of wp states that

$$\mathsf{wp}\,[\![C]\!]\,(0) \;\preceq\; 0\,,$$

which implies $\mathsf{wp}\,[\![C]\!]\,(0) = 0$ by expectations being non–negative. An analogous argument applies to awp.

4.2.4 *Monotonicity*

The distinct feature of everything extant is its monotony.

— Vladimir Nabokov

Monotonicity is another fundamental property of expectation transformers. According to Back and von Wright, monotonicity is „the only healthiness criteria [sic] that has gone unquestioned“ [BW89]. An expectation transformer $\mathcal{T}$ is monotonic iff for any two expectations $f, g \in \mathbb{E}$, we have that

$$f \preceq g \quad \text{implies} \quad \mathcal{T}(f) \preceq \mathcal{T}(g)\,;$$

see Definition A.3 for more details. All expectation transformers we have presented so far are monotonic:

Theorem 4.16 (Monotonicity of Expectation Transformers):
Let $C \in$ pGCL. Then the associated expectation transformers $\mathsf{wp}\,[\![C]\!]$, $\mathsf{wlp}\,[\![C]\!]$, $\mathsf{awp}\,[\![C]\!]$, *and* $\mathsf{awlp}\,[\![C]\!]$ *are monotonic. Furthermore, all* wp–, wlp–, awp–, *and* awlp–*characteristic functions are monotonic.*

Proof. Every continuous function is monotonic, see Theorem A.4. Q.E.D.

Monotonicity is not just a healthiness condition but plays an important role in reasoning about programs. In the following we present two implications of the monotonicity property.

Compositionality. Monotonicity is useful for compositional reasoning in the following sense: Imagine two programs C_1 and C_2 and a postexpectation f such that

$$\mathsf{wp}\,[\![C_1]\!]\,(f) \;\preceq\; \mathsf{wp}\,[\![C_2]\!]\,(f)\,.$$

Then monotonicity ensures that if we put the components C_1 and C_2 into some context $C\,\fatsemi\,\ldots$, then we can be certain that

$$\mathsf{wp}\,[\![C \fatsemi C_1]\!]\,(f) \;\preceq\; \mathsf{wp}\,[\![C \fatsemi C_2]\!]\,(f)\,,$$

since $\mathsf{wp}\,[\![C \fatsemi C_i]\!]\,(f) = \mathsf{wp}\,[\![C]\!]\,\big(\mathsf{wp}\,[\![C_i]\!]\,(f)\big)$, for $i \in \{1, 2\}$.

Relation to the consequence rule. Monotonicity is closely related to the *consequence rule* of Hoare logic. This rule reads

$$\frac{G \implies G' \qquad \langle G' \rangle\, C\, \langle F' \rangle \qquad F' \implies F}{\langle G \rangle\, C\, \langle F \rangle}\ \text{(cons)}\,.$$

It weakens precondition G to G' and strengthens postcondition F to F' in order to prove validity of $\langle G \rangle\, C\, \langle F \rangle$ by proving validity of $\langle G' \rangle\, C\, \langle F' \rangle$.

The analogy to weakest preexpectation reasoning is as follows: In order to prove $g \preceq \mathsf{wp}\,[\![C]\!]\,(f)$ it suffices to

1. choose $g' \succeq g$,
2. choose $f' \preceq f$, and
3. prove $g' \preceq \mathsf{wp}\,[\![C]\!]\,(f')$,

since this gives

$$\begin{aligned} g &\preceq g' && \text{(by 1. above)} \\ &\preceq \mathsf{wp}\,[\![C]\!]\,(f') && \text{(by 3. above)} \\ &\preceq \mathsf{wp}\,[\![C]\!]\,(f)\,, && \text{(by 2. above and monotonicity, Theorem 4.16)} \end{aligned}$$

which in turn implies $g \preceq \mathsf{wp}\,[\![C]\!]\,(f)$. The *"consequence rule of weakest precondition reasoning"* thus reads

$$\frac{g \preceq g' \qquad g' \preceq \mathsf{wp}\,[\![C]\!]\,(f') \qquad f' \preceq f}{g \preceq \mathsf{wp}\,[\![C]\!]\,(f)}\ \text{(wp–cons)}\,.$$

4.2.5 *Linearity*

Linearity of expectation transformers plays a prominent role in the development of McIver & Morgan as they show that superlinearity[6] alone already characterizes their wp and thus implies monotonicity, strictness, continuity, and so on. However, as mentioned before, McIver & Morgan heavily rely on the fact that their expectations are bounded (see Definition 4.1 B.). For showing that superlinearity implies continuity they even need to restrict to a finite state space Σ [MM05, p. 148].

While this allows McIver & Morgan to nicely characterize all „healthy" expectation transformers by means of just a single healthiness condition, we cannot make the restriction of boundedness and do not wish to restrict to a finite state space. Another point is that we make use of the angelic awp

6 Note that McIver & Morgan use the term *sublinear* for *superlinear* transformers [MM05]. However, the super–/sub–nomenclature we use here is more in accordance with the standard mathematics terminology [Wikk; KM17].

transformer which is sub- instead of superlinear and while we saw that awp is monotonic, this fact does not follow from sublinearity.

Since our setting differs from that of McIver & Morgan, we will conduct our own linearity studies here. Linearity is made up of two properties: homogeneity and additivity. We study the former first:

Definition 4.17 (Positive Homogeneity):
Let $f \in \mathbb{E}$, $r \in \mathbb{R}_{\geq 0}$, and let $\mathcal{T} : \mathbb{E} \to \mathbb{E}$ be an expectation transformer. Then $\mathcal{T}$ is called positively homogeneous[7] iff

$$\mathcal{T}(r \cdot f) = r \cdot \mathcal{T}(f) .$$

We consider *positive* homogeneity (i.e. our scaling factor r is positive) instead of general homogeneity since we need to stay within the realm of expectations which are non–negative. Both nonliberal expectation transformers we have presented are positively homogeneous:

Theorem 4.18 (Positive Homogeneity of Expectation Transformers):
For any $C \in \mathsf{pGCL}$, $\mathsf{wp}\,[\![C]\!]$ and $\mathsf{awp}\,[\![C]\!]$ are positively homogeneous.

Proof. By induction on the structure of C.

Positive homogeneity of wp implies its strictness by the following reasoning:

$$\begin{aligned} \mathsf{wp}\,[\![C]\!]\,(0) &= \mathsf{wp}\,[\![C]\!]\,(2 \cdot 0) \\ &= 2 \cdot \mathsf{wp}\,[\![C]\!]\,(0) \qquad \text{(by positive homogeneity, Theorem 4.18)} \end{aligned}$$

which implies that $\mathsf{wp}\,[\![C]\!]\,(0) = 0$. The proof for awp is analogous.

Positive homogeneity and monotonicity together also suffice to prove for expectation transformers Markov's well–known inequality:

Theorem 4.19 (Markov's Inequality):
Let $C \in \mathsf{pGCL}$, $f \in \mathbb{E}$, and $a \in \mathbb{R}_{\geq 0}$ with $a > 0$. Then:

A. $\mathsf{wp}\,[\![C]\!]\,([f \geq a]) \preceq \dfrac{\mathsf{wp}\,[\![C]\!]\,(f)}{a}$

B. $\mathsf{awp}\,[\![C]\!]\,([f \geq a]) \preceq \dfrac{\mathsf{awp}\,[\![C]\!]\,(f)}{a}$

7 This property is called "scaling" by McIver and Morgan [MM05].

Proof. For A., consider the following:

$$
\begin{aligned}
& a \cdot [f \geq a] \preceq f \\
\text{implies} \quad & \mathsf{wp}\,[\![C]\!]\,(a \cdot [f \geq a]) \preceq \mathsf{wp}\,[\![C]\!]\,(f) \\
& \qquad\qquad \text{(by monotonicity, Theorem 4.16)} \\
\text{implies} \quad & a \cdot \mathsf{wp}\,[\![C]\!]\,([f \geq a]) \preceq \mathsf{wp}\,[\![C]\!]\,(f) \\
& \qquad\qquad \text{(by positive homogeneity, Theorem 4.18)} \\
\text{iff} \quad & \mathsf{wp}\,[\![C]\!]\,([f \geq a]) \preceq \frac{\mathsf{wp}\,[\![C]\!]\,(f)}{a} \qquad \text{(by } a > 0\text{)}
\end{aligned}
$$

The proof for awp is analogous. Q.E.D.

wlp and awlp are *not* positively homogenous as the following example shows:

$$\mathsf{wlp}\,[\![\texttt{diverge}]\!]\left(\tfrac{1}{2} \cdot 0\right) = 1 > \tfrac{1}{2} = \tfrac{1}{2} \cdot \mathsf{wlp}\,[\![\texttt{diverge}]\!]\,(0)$$

Neither wlp nor awlp satisfy Markov's inequality.

In order to study linearity, we now study additivity of our expectation transformers. Together with homogeneity this yields the notion of linearity.

DEFINITION 4.20 (Linearity of Expectation Transformers):
Let $f, g \in \mathbb{E}$, $r \in \mathbb{R}_{\geq 0}$, and $\mathcal{T} \colon \mathbb{E} \to \mathbb{E}$ be an expectation transformer. Then:

A. *$\mathcal{T}$ is called sublinear[8] iff*

$$\mathcal{T}(r \cdot f + g) \preceq r \cdot \mathcal{T}(f) + \mathcal{T}(g)\,.$$

B. *$\mathcal{T}$ is called superlinear iff*

$$r \cdot \mathcal{T}(f) + \mathcal{T}(g) \preceq \mathcal{T}(r \cdot f + g)\,.$$

C. *$\mathcal{T}$ is called linear iff*

$$\mathcal{T}(r \cdot f + g) = r \cdot \mathcal{T}(f) + \mathcal{T}(g)\,.$$

All our expectation transformers satisfy one of the above notions of linearity:

THEOREM 4.21 (Linearity of Expectation Transformers[9]):
Let $C \in$ pGCL. Then:

A. *$\mathsf{wp}\,[\![C]\!]$ is superlinear.*[10]

B. *$\mathsf{awp}\,[\![C]\!]$ is sublinear.*

8 Recall Footnote 6 on page 95.
9 See [Koz83; MM05].
10 Recall Footnote 6 on page 95.

Suppose moreover that C is tame. Then angelic and demonic expectation transformers coincide, and

C. $\mathsf{wp}\,[\![C]\!]$ *and* $\mathsf{awp}\,[\![C]\!]$ *are linear.*

D. $\mathsf{wlp}\,[\![C]\!]$ *and* $\mathsf{awlp}\,[\![C]\!]$ *are superlinear.*

Proof. As for A. and B., the proof is by induction on the structure of C.

As for C., linearity of wlp follows from the fact that wp and awp obviously coincide on tame programs. But since wp is superlinear and awp is sublinear, wp and awp have to be linear.

As for D., superlinearity of wlp follows from the connection of wlp and wp (see Corollary 4.26) as discussed in Section 4.3. Q.E.D.

Sublinearity of awp is not a mere theoretical observation. It has a concrete impact, for instance on the development of Kura *et al.* on reasoning about higher moments of expected runtimes [KUH19].

Superlinearity of wp implies monotonicity of wp by the following reasoning: Let f and g be two expectations such that $f \preceq g$. Then there exists an expectation $\epsilon \in \mathbb{E}$ such that $g = f + \epsilon$. By superlinearity of wp we then have

$$\begin{aligned} \mathsf{wp}\,[\![C]\!]\,(f) &\preceq \mathsf{wp}\,[\![C]\!]\,(f) + \mathsf{wp}\,[\![C]\!]\,(\epsilon) \\ &\preceq \mathsf{wp}\,[\![C]\!]\,(f+\epsilon) && \text{(by superlinearity, Theorem 4.21 A.)} \\ &= \mathsf{wp}\,[\![C]\!]\,(g) \end{aligned}$$

The above reasoning fails for awp as it is sublinear instead of superlinear and the inequality is hence in the wrong direction. Still, awp is monotonic.

A useful corollary for expectation *subtraction* is the following:

Corollary 4.22 (Linearity and Subtractions):
Let $C \in \mathsf{pGCL}$ and let $f, g \in \mathbb{E}$ such that $f \preceq g$, thus $g - f$ is a well–defined (non–negative) expectation. Then:

A. $\mathsf{wp}\,[\![C]\!]\,(g-f) \preceq \mathsf{wp}\,[\![C]\!]\,(g) - \mathsf{wp}\,[\![C]\!]\,(f)$.

B. $\mathsf{awp}\,[\![C]\!]\,(g-f) \succeq \mathsf{awp}\,[\![C]\!]\,(g) - \mathsf{awp}\,[\![C]\!]\,(f)$.

Suppose moreover that C is tame. Then:

C. $\mathsf{wp}\,[\![C]\!] = \mathsf{awp}\,[\![C]\!]$ *and* $\mathsf{wp}\,[\![C]\!]\,(g-f) = \mathsf{wp}\,[\![C]\!]\,(g) - \mathsf{wp}\,[\![C]\!]\,(f)$.

Proof. For wp consider the following:

$$\begin{aligned} \mathsf{wp}\,[\![C]\!]\,(g-f) &= \mathsf{wp}\,[\![C]\!]\,(g-f) + \mathsf{wp}\,[\![C]\!]\,(f) - \mathsf{wp}\,[\![C]\!]\,(f) \\ &\preceq \mathsf{wp}\,[\![C]\!]\,(g-f+f) - \mathsf{wp}\,[\![C]\!]\,(f) \\ &\qquad\qquad \text{(by superlinearity, Theorem 4.21 A.)} \\ &= \mathsf{wp}\,[\![C]\!]\,(g) - \mathsf{wp}\,[\![C]\!]\,(f) \end{aligned}$$

The reasoning for awp and for the case of tame programs is analogous. Q.E.D.

Thus, we see that for subtractions wp behaves sublinearly instead of superlinearly whereas awp behaves superlinearly instead of sublinearly.

We can make use of Corollary 4.22 to show that — in addition to monotonicity — superlinearity of wp also implies feasibility of wp: For showing this, let $f \in \mathbb{E}_{\leq \exists b}$ be an expectation bounded by $b \in \mathbb{R}_{\geq 0}$. Then $f \preceq b$ and

$$0 \preceq \mathsf{wp}[\![C]\!](b-f) \preceq \mathsf{wp}[\![C]\!](b) - \mathsf{wp}[\![C]\!](f) \qquad \text{(by Corollary 4.22 A.)}$$

$$\text{implies} \quad 0 \preceq \mathsf{wp}[\![C]\!](b) - \mathsf{wp}[\![C]\!](f)$$

$$\text{implies} \quad \mathsf{wp}[\![C]\!](f) \preceq \mathsf{wp}[\![C]\!](b)$$

Again, the above reasoning fails for awp as awp is sublinear instead of superlinear and the inequality is therefore in the wrong direction. Nevertheless, awp is feasible.

4.3 RELATING EXPECTATION TRANSFORMERS

The definitions of the different expectation transformers we studied in this chapter are quite similar and it is not surprising that the transformers are closely related. The most elementary and most obvious relationship between angelic (awp and awlp) and demonic preexpectations (wp and wlp) is that demonic preexpectations are never greater than angelic preexpectations:

Corollary 4.23 (Angelic Bound Demonic Preexpectations): *Let $C \in \mathsf{pGCL}$, $f \in \mathbb{E}$, and $g \in \mathbb{E}_{\leq 1}$. Then*

A. $\mathsf{wp}[\![C]\!](f) \preceq \mathsf{awp}[\![C]\!](f)$, *and*

B. $\mathsf{wlp}[\![C]\!](g) \preceq \mathsf{awlp}[\![C]\!](g)$.

The most elementary relationship between liberal (wlp and awlp) and nonliberal preexpectations (wp and awp) is that weakest preexpectations are never greater than weakest liberal preexpectations:

Corollary 4.24 (Liberal Bound Nonliberal Preexpectations): *Let $C \in \mathsf{pGCL}$ and $f \in \mathbb{E}_{\leq 1}$. Then*

A. $\mathsf{wp}[\![C]\!](f) \preceq \mathsf{wlp}[\![C]\!](f)$, *and*

B. $\mathsf{awp}[\![C]\!](f) \preceq \mathsf{awlp}[\![C]\!](f)$.

Proof. Follows immediately from the fact that nonliberal preexpectations are defined as a *least* fixed point whereas liberal preexpectations are defined as a *greatest* fixed point. Q.E.D.

Intuitively, we can understand Corollary 4.24 as the fact that a program is more likely to be partially correct as it is likely to be totally correct.

In addition to the above, we can make a more precise statement relating liberal and nonliberal preexpectations. In there, some care regarding nondeterminism must be taken — liberal transformers become nonliberal and angelic ones become demonic; and vice versa:

THEOREM 4.25 (Relationship between Expectation Transformers): *Let $C \in \mathsf{pGCL}$ be tame and let $f \in \mathbb{E}_{\leq 1}$. Then*

A. $\quad \mathsf{wp}\,[\![C]\!]\,(f) \;=\; 1 - \mathsf{awlp}\,[\![C]\!]\,(1-f)$

B. $\quad \mathsf{awp}\,[\![C]\!]\,(f) \;=\; 1 - \;\mathsf{wlp}\,[\![C]\!]\,(1-f)$

C. $\quad \mathsf{wlp}\,[\![C]\!]\,(f) \;=\; 1 - \;\mathsf{awp}\,[\![C]\!]\,(1-f)$

D. $\quad \mathsf{awlp}\,[\![C]\!]\,(f) \;=\; 1 - \;\mathsf{wp}\,[\![C]\!]\,(1-f)$

Proof. For proving A., consider the following:

$$
\begin{aligned}
&1 - \mathsf{awlp}\,[\![C]\!]\,(1-f) \\
&= 1 - \lambda\sigma.\ \sup_{\mathfrak{s}\in\mathsf{Scheds}} \int_\Sigma 1 - f \; d\,[\![C]\!]^{\mathfrak{s}}_\sigma \;+\; 1 - \int_\Sigma 1 \; d\,[\![C]\!]^{\mathfrak{s}}_\sigma \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by Theorem 4.11 D.)} \\
&= 1 - \lambda\sigma.\ \sup_{\mathfrak{s}\in\mathsf{Scheds}} \int_\Sigma 1 \; d\,[\![C]\!]^{\mathfrak{s}}_\sigma - \int_\Sigma f \; d\,[\![C]\!]^{\mathfrak{s}}_\sigma \;+\; 1 - \int_\Sigma 1 \; d\,[\![C]\!]^{\mathfrak{s}}_\sigma \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Big(\text{by linearity of } \textstyle\int\Big) \\
&= 0 - \lambda\sigma.\ \sup_{\mathfrak{s}\in\mathsf{Scheds}} - \int_\Sigma f \; d\,[\![C]\!]^{\mathfrak{s}}_\sigma \\
&= \lambda\sigma.\ \inf_{\mathfrak{s}\in\mathsf{Scheds}} \int_\Sigma f \; d\,[\![C]\!]^{\mathfrak{s}}_\sigma \\
&= \mathsf{wp}\,[\![C]\!]\,(f) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by Theorem 4.11 A.)}
\end{aligned}
$$

The proofs for B., C., and D. are analogous. **Q.E.D.**

Let us gain some intuition on the above by considering Theorem 4.25 A. and choosing as postexpectation a predicate $[\varphi]$. Then we get

$$\mathsf{wp}\,[\![C]\!]\,([\varphi]) \;=\; 1 - \mathsf{awlp}\,[\![C]\!]\,(1-[\varphi]) \;=\; 1 - \mathsf{awlp}\,[\![C]\!]\,([\neg\varphi])\ .$$

$\mathsf{wp}\,[\![C]\!]\,([\varphi])$ tries to minimize the probability of C terminating in a state satisfying φ. How can wp achieve that? It can either drive C towards diverging or terminating but satisfying $\neg\varphi$, i.e. the opposite of φ. So wp will maximize the probability of either of these two events. But this is precisely what $\mathsf{awlp}\,[\![C]\!]\,([\neg\varphi])$ does.

For tame programs, the angelic and demonic transformers coincide and the nonliberal transformers are linear. We hence get from linearity (Theorem 4.21 c.) and Theorem 4.25 c. the following corollary:

COROLLARY 4.26 ([Koz83]):
Let $C \in \mathsf{pGCL}$ be tame and let $f \in \mathbb{E}_{\leq 1}$. Then

$$\mathsf{wlp}\,[\![C]\!]\,(f) \;=\; \mathsf{wp}\,[\![C]\!]\,(f) + 1 - \mathsf{wp}\,[\![C]\!]\,(1)\ .$$

Thus, $\mathsf{wlp}\,[\![C]\!]\,(f)$ is expressible as the sum of $\mathsf{wp}\,[\![C]\!]\,(f)$ and the probability that C does *not* terminate, i.e. $1 - \mathsf{wp}\,[\![C]\!]\,(1)$. This connection between wp and wlp allows us to study linearity of wlp: We have

$$\begin{aligned}
&\mathsf{wlp}\,[\![C]\!]\,(r \cdot f + g) \\
&= 1 - \mathsf{wp}\,[\![C]\!]\,(1) + \mathsf{wp}\,[\![C]\!]\,(r \cdot f + g) && \text{(by Corollary 4.26)} \\
&= 1 - \mathsf{wp}\,[\![C]\!]\,(1) + r \cdot \mathsf{wp}\,[\![C]\!]\,(f) + \mathsf{wp}\,[\![C]\!]\,(g) \\
& && \text{(by linearity, Theorem 4.21 c.)} \\
&\geq\ r \cdot \mathsf{wp}\,[\![C]\!]\,(f) + \mathsf{wp}\,[\![C]\!]\,(g)
\end{aligned}$$

for any tame $C \in \mathsf{pGCL}$, $f, g \in \mathbb{E}_{\leq 1}$, and $r \in \mathbb{R}_{\geq 0}$, such that $r \cdot f + g \in \mathbb{E}_{\leq 1}$. Thus, wlp is in general not linear for tame programs but superlinear instead.

An even closer connection between $\mathsf{wp}\,[\![C]\!]$ and $\mathsf{wlp}\,[\![C]\!]$ can be established in case that C terminates almost–surely. We first note that

$$\mathsf{wp}\,[\![C]\!]\,(1)$$

is an expectation, such that $\mathsf{wp}\,[\![C]\!]\,(1)(\sigma)$ gives the *(minimal) probability that C terminates on input σ*. Dually, $\mathsf{awp}\,[\![C]\!]\,(1)(\sigma)$ gives the maximal probability of C terminating on σ.

Yet dually,

$$\mathsf{awlp}\,[\![C]\!]\,(0)$$

is an expectation such that $\mathsf{awlp}\,[\![C]\!]\,(0)(\sigma)$ gives the maximal probability that C diverges on σ, whereas $\mathsf{wlp}\,[\![C]\!]\,(0)(\sigma)$ gives the minimal probability of C terminating on σ.

If we know a predicate T, such that C terminates almost–surely (i.e. with probability 1) from every initial state satisfying T, we can express this by

$$[T] \;\preceq\; \mathsf{wp}\,[\![C]\!]\,(1)\ .$$

In fact, if we know such a predicate T, we can „sandwich“ weakest preexpectations by means of weakest liberal preexpectations:

THEOREM 4.27 ((Non)liberal Preexpectations under Termination):
Let $C \in \mathsf{pGCL}$, let $f \in \mathbb{E}_{\leq 1}$, let T be a predicate, and let C terminate from any state satisfying T, i.e. let $[T] \preceq \mathsf{wp}\,[\![C]\!]\,(1)$. Then

A. $[T] \cdot \mathsf{wlp}\,[\![C]\!]\,(f) \;\preceq\; \mathsf{wp}\,[\![C]\!]\,(f) \;\preceq\; \mathsf{wlp}\,[\![C]\!]\,(f)$, *and*

B. $[T] \cdot \mathsf{awlp}\,[\![C]\!]\,(f) \;\preceq\; \mathsf{awp}\,[\![C]\!]\,(f) \;\preceq\; \mathsf{awlp}\,[\![C]\!]\,(f)$

Proof. For A., consider the following: Let $f \ominus g$ be defined as $\max\{f-g, 0\}$ and consider the following:

$$\Big(\mathsf{wlp}\,[\![C]\!]\,(f) + \mathsf{wp}\,[\![C]\!]\,(1)\Big) \ominus 1 \;\preceq\; \mathsf{wp}\,[\![C]\!]\,((f+1)\ominus 1)$$
(by [MM05, Fact B.3.2 on p.331])

$$\text{implies}\quad \Big([T]\cdot\mathsf{wlp}\,[\![C]\!]\,(f) + [T]\cdot\mathsf{wp}\,[\![C]\!]\,(1)\Big) \ominus [T] \;\preceq\; \mathsf{wp}\,[\![C]\!]\,((f+1)\ominus 1)$$
(Multiply left–hand side by $[T]$)

$$\text{implies}\quad \Big([T]\cdot\mathsf{wlp}\,[\![C]\!]\,(f) + [T]\cdot[T]\Big) \ominus [T] \;\preceq\; \mathsf{wp}\,[\![C]\!]\,((f+1)\ominus 1)$$
(by assumption $[T] \preceq \mathsf{wp}\,[\![C]\!]\,(1)$)

$$\text{iff}\quad \Big([T]\cdot\mathsf{wlp}\,[\![C]\!]\,(f) + [T]\Big) \ominus [T] \;\preceq\; \mathsf{wp}\,[\![C]\!]\,((f+1)\ominus 1)$$

$$\text{iff}\quad [T]\cdot\mathsf{wlp}\,[\![C]\!]\,(f) \;\preceq\; \mathsf{wp}\,[\![C]\!]\,(f)$$

For B., we exploit Theorem 4.27 A. and Theorem 4.25 A. and B. as follows:

$$\begin{aligned}
&\mathsf{awp}\,[\![C]\!]\,(f)\\
&= 1 - \mathsf{wlp}\,[\![C]\!]\,(1-f) && \text{(by Theorem 4.25 B.)}\\
&\geq [T] - [T]\cdot\mathsf{wlp}\,[\![C]\!]\,(1-f)\\
&\geq [T]\cdot\Big(1 - [T]\cdot\mathsf{wlp}\,[\![C]\!]\,(1-f)\Big)\\
&\geq [T]\cdot\Big(1 - \mathsf{wp}\,[\![C]\!]\,(1-f)\Big) && \text{(by assumption and Theorem 4.27 A.)}\\
&= [T]\cdot\Big(1 - \big(1 - \mathsf{awlp}\,[\![C]\!]\,(1-(1-f))\big)\Big) && \text{(by Theorem 4.25 A.)}\\
&= [T]\cdot\mathsf{awlp}\,[\![C]\!]\,(f) && \text{Q.E.D.}
\end{aligned}$$

In the case of universally almost–surely terminating programs (i.e. programs that terminate almost–surely on *every* input), we even get that liberal and nonliberal expectation transformers coincide entirely, which implies that (on $\mathbb{E}_{\leq 1}$) least and greatest fixed points coincide:

COROLLARY 4.28:
Let $C \in \mathsf{pGCL}$, let $f \in \mathbb{E}_{\leq 1}$, and let C terminate universally almost–surely, i.e. let $\mathsf{wp}\,[\![C]\!]\,(1) = 1$. Then

A. $\mathsf{wlp}\,[\![C]\!]\,(f) \;=\; \mathsf{wp}\,[\![C]\!]\,(f)$, *and*

B. $\mathsf{awlp}\,[\![C]\!]\,(f) \;=\; \mathsf{awp}\,[\![C]\!]\,(f)$

Proof. For A., we have by Corollary 4.24 that $\mathsf{wp}\,[\![C]\!]\,(f) \preceq \mathsf{wlp}\,[\![C]\!]\,(f)$. It is left to show that $\mathsf{wlp}\,[\![C]\!]\,(f) \preceq \mathsf{wp}\,[\![C]\!]\,(f)$ holds which is an immediate

consequence from Theorem 4.27 A. by choosing $T = \text{true}$. The proof for the angelic transformers is analogous. Q.E.D.

We will see later in Chapter 5 that Theorem 4.27 and Corollary 4.28 have important consequences for reasoning about loops, as they makes reasoning about lower bounds of expected values considerably easier. Reasoning about the precondition of Corollary 4.28 however, namely almost–sure termination, is often difficult, as we will discuss in Chapter 6.

PROOF RULES FOR LOOPS

5

Reasoning about loops is one of the most — if not *the* most — difficult tasks in verification. For nonprobabilistic programs, this is usually done using loop invariants and loop variants. Roughly speaking, loop invariants allow proving partial correctness, meaning the algorithm is correct if it terminates. Loop variants, on the other hand, enable proving termination. Partial correctness and termination together give total correctness.

For probabilistic programs, neither correctness nor termination are strictly binary properties: Monte Carlo algorithms, for instance, typically trade off 100% correctness for runtime efficiency, thus giving correct answers to otherwise difficult problems only with high probability but make up for it with short expected runtime. In order to account for those quantitative aspects, techniques for reasoning about the correctness of randomized algorithms need to naturally also be of quantitative nature. Amongst others, Kozen, McIver & Morgan, Jones, and also ourselves have provided quantitative analogs to invariant–style reasoning about probabilistic loops. In this chapter, we survey those proof rules.

After recapping how to lift invariant–style reasoning from nonprobabilistic to probabilistic loops, we survey and discuss proof rules for proving bounds (both upper and lower) on weakest preexpectations and weakest liberal preexpectations. In particular, we discuss in some detail the problem of obtaining lower bounds on weakest preexpectations, i.e. lower bounds on least fixed points. All proof rules have been translated into our weakest preexpectation setting so that we can give a unified overview and comparison.

5.1 INVARIANTS

All proof rules we present in this chapter make in one way or another use of a probabilistic, or rather quantitative, notion of invariants. In order to transit from Boolean to quantitative reasoning, let us briefly recap invariant–style reasoning about partial correctness of *nonprobabilistic* while loops: Given a precondition G and a postcondition F, say we want to prove that if executing $\texttt{while}(\varphi)\{C\}$ on an initial state $\sigma \models G$ terminates, then it does so in a final state $\tau \models F$. For that, we have to find a predicate I such that

$$G \implies I \quad \text{and} \quad \neg\varphi \wedge I \implies F \quad \text{and} \quad \langle\, \varphi \wedge I \,\rangle\, C \,\langle\, I \,\rangle \text{ is valid,} \tag{5.1}$$

where we mean validity for partial correctness. Any I for which the Hoare triple $\langle\, \varphi \wedge I \,\rangle\, C \,\langle\, I \,\rangle$ is valid for partial correctness is called a *loop invariant*:

If the loop $\texttt{while}(\varphi)\{C\}$ is started in a state satisfying both loop guard φ and invariant I and one execution of the loop body C terminates from that state, then the execution of C terminates in a state that again satisfies I. Thus, satisfaction of I is *invariant under (guarded) iteration* of the loop body.[1]

Why do invariants do the trick? Because we can now apply the while rule for partial correctness of Hoare logic which reads as follows:

$$\frac{\langle \varphi \wedge I \rangle\, C\, \langle I \rangle}{\langle I \rangle\, \texttt{while}(\varphi)\{C\}\, \langle \neg\varphi \wedge I \rangle} \text{ (while–partial)} \tag{5.2}$$

We can now combine the while–partial rule above with the consequence rule of Hoare logic (see Section 4.2.4, Relation to the consequence rule) into a single proof tree and so obtain the full proof of partial correctness, namely

$$\frac{G \implies I \qquad \dfrac{\langle \varphi \wedge I \rangle\, C\, \langle I \rangle}{\langle I \rangle\, \texttt{while}(\varphi)\{C\}\, \langle \neg\varphi \wedge I \rangle} \qquad \neg\varphi \wedge I \implies F}{\langle G \rangle\, \texttt{while}(\varphi)\{C\}\, \langle F \rangle}\,,$$

which can be stated as a single inference rule:

$$\frac{G \implies I \qquad \langle \varphi \wedge I \rangle\, C\, \langle I \rangle \qquad \neg\varphi \wedge I \implies F}{\langle G \rangle\, \texttt{while}(\varphi)\{C\}\, \langle F \rangle} \text{ (while–partial2)}$$

In the realm of weakest precondition reasoning, the premises $\langle \varphi \wedge I \rangle\, C\, \langle I \rangle$ and $\neg\varphi \wedge I \implies F$ together are equivalent to

$$[I] \;\preceq\; {}^{\mathsf{wlp}}\Phi_{[F]}([I])\,,$$

where ${}^{\mathsf{wlp}}\Phi_{[F]}$ is the wlp–characteristic function of $\texttt{while}(\varphi)\{C\}$ with respect to postcondition $[F]$ (see Definition 4.5 E.). This can be seen by

$$\begin{aligned}
& \neg\varphi \wedge I \implies F \\
\text{iff}\quad & [\neg\varphi]\cdot[I] \;\preceq\; [F] \\
\text{iff}\quad & [\neg\varphi]\cdot[I] \;\preceq\; [\neg\varphi]\cdot[F] && \text{(by case distinction) (†)}
\end{aligned}$$

and

$$\begin{aligned}
& \langle \varphi \wedge I \rangle\, C\, \langle I \rangle \\
\text{iff}\quad & \varphi \wedge I \implies \mathsf{wlp}\,[\![C]\!]\,(I) \\
\text{iff}\quad & [\varphi]\cdot[I] \;\preceq\; \mathsf{wlp}\,[\![C]\!]\,([I]) \\
\text{iff}\quad & [\varphi]\cdot[I] \;\preceq\; [\varphi]\cdot\mathsf{wlp}\,[\![C]\!]\,([I]) && \text{(by case distinction)} \\
\text{iff}\quad & [\neg\varphi]\cdot[I] + [\varphi]\cdot[I] \;\preceq\; [\neg\varphi]\cdot[F] + [\varphi]\cdot\mathsf{wlp}\,[\![C]\!]\,([I]) && \text{(by (†) above)} \\
\text{iff}\quad & [I] \;\preceq\; {}^{\mathsf{wlp}}\Phi_{[F]}([I])\,. && \big(\text{by definition of } {}^{\mathsf{wlp}}\Phi_{[F]}\text{, Definition 4.5 E.}\big)
\end{aligned}$$

1 By "guarded iteration" we mean iterating the loop body only if the loop guard is true.

In the language of our weakest precondition calculi, the (while–partial2)–rule thus reads as follows:

$$\frac{[G] \preceq [I] \preceq {}^{\mathsf{wlp}}_{\langle\varphi,C\rangle}\Phi_{[F]}([I])}{[G] \preceq \mathsf{wlp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,([F])}$$

For our definition of quantitative invariants, we lift the above rule to weakest preexpectations in a straightforward way. Furthermore, we distinguish between super- and subinvariants.

DEFINITION 5.1 (Invariants):
Let Φ_f be the wp*–characteristic function of* $\texttt{while}(\varphi)\{C\}$ *with respect to postexpectation $f \in \mathbb{E}$ and let $I \in \mathbb{E}$. Then:*

A. *I is called a* wp*–superinvariant of* $\texttt{while}(\varphi)\{C\}$ *with respect to postexpectation f, iff*

$$\Phi_f(I) \preceq I\,.$$

B. *I is called a* wp*–subinvariant of* $\texttt{while}(\varphi)\{C\}$ *with respect to postexpectation f, iff*

$$I \preceq \Phi_f(I)\,.$$

C. *Super- and subinvariants for* wlp, awp, *and* awlp *are defined analogously by means of* wlp–, awp–, *and* awlp*–characteristic functions, respectively. Notice that* wlp– *and* awlp*–invariants are of type $\mathbb{E}_{\leq 1}$ rather than $\mathbb{E}$.*

The I we used to illustrate the while–rule for partial correctness (Rule 5.2 above) would be a wlp–subinvariant in the terminology of Definition 5.1.

Remark 5.2 (On Terminology in Related Literature). Our *subinvariants* correspond to *probabilistic invariants* in the terminology of McIver & Morgan (see [MM05, Definition 2.2.1, p. 39]) up to a slight technical difference: McIver & Morgan call I a probabilistic invariant iff

$$[\varphi]\cdot I \preceq \mathsf{wp}\,[\![C]\!]\,(I)\,,$$

which is implied by $I \preceq \Phi_f(I)$, but the converse implication is not true in general for arbitrary postexpectations f. However, McIver & Morgan do not consider arbitrary postexpectations f, but instead argue only about weakest preexpectations of loops with respect to postexpectation $[\neg\varphi]\cdot I$ and we have

$$[\varphi]\cdot I \preceq \mathsf{wp}\,[\![C]\!]\,(I) \quad \text{iff} \quad I \preceq \Phi_{[\neg\varphi]\cdot I}(I)\,,$$

as the following reasoning shows:

$$
\begin{aligned}
& I \preceq \Phi_{[\neg\varphi]\cdot I}(I) \\
\text{iff}\quad & [\neg\varphi]\cdot I + [\varphi]\cdot I \preceq [\neg\varphi]\cdot[\neg\varphi]\cdot I + [\varphi]\cdot \mathsf{wp}\,[\![C]\!]\,(I) \\
\text{iff}\quad & [\neg\varphi]\cdot I + [\varphi]\cdot I \preceq [\neg\varphi]\cdot I + [\varphi]\cdot \mathsf{wp}\,[\![C]\!]\,(I) \\
\text{iff}\quad & [\neg\varphi]\cdot I \preceq [\neg\varphi]\cdot I \quad \text{and} \quad [\varphi]\cdot I \preceq +[\varphi]\cdot \mathsf{wp}\,[\![C]\!]\,(I) \\
\text{iff}\quad & [\varphi]\cdot I \preceq +[\varphi]\cdot \mathsf{wp}\,[\![C]\!]\,(I)
\end{aligned}
$$

Our definition of subinvariants is therefore not a restriction compared to McIver & Morgan's probabilistic invariants.

Our *superinvariants* correspond to *supermartingales* in the terminology used by Chakarov & Sankaranarayanan [CS14], Fioriti & Hermanns [FH15], and Chatterjee *et al.* [CFG16; Cha+16; CF17; CNZ17; ACN18] with basically the same technical difference as above.

Generally speaking, sub– and superinvariants in our terminology can be conceived of, respectively, as sub– and supermartingales of the stochastic process that can naturally be associated to a probabilistic loop. △

Next, we introduce a concept we call ω–invariants. These are basically sequences of expectations that are invariants relative to each other. We will make use of those for reasoning about lower bounds on least fixed points and dually upper bounds on greatest fixed points.

DEFINITION 5.3 (ω–Invariants):

A. *Let Φ be the* wlp*–characteristic function of* `while`$(\varphi)\{C\}$ *with respect to postexpectation f and let $(I_n)_{n\in\mathbb{N}} \subset \mathbb{E}_{\leq 1}$ be a monotonically decreasing*[2] *sequence with $I_0 = 1$.*

Then $(I_n)_{n\in\mathbb{N}}$ is called a wlp*–ω–superinvariant of* `while`$(\varphi)\{C\}$ *with respect to postexpectation $f \in \mathbb{E}_{\leq 1}$, iff*

$$\forall\, n \in \mathbb{N}\colon \quad \Phi(I_n) \preceq I_{n+1}\,.$$

B. *Let Φ be the* wp*–characteristic function of* `while`$(\varphi)\{C\}$ *with respect to postexpectation f and let $(I_n)_{n\in\mathbb{N}} \subset \mathbb{E}$ be a monotonically increasing*[3] *sequence with $I_0 = 0$.*

Then $(I_n)_{n\in\mathbb{N}}$ is called a wp*–ω–subinvariant of* `while`$(\varphi)\{C\}$ *with respect to postexpectation f, iff*

$$\forall\, n \in \mathbb{N}\colon \quad I_{n+1} \preceq \Phi(I_n)\,.$$

C. awlp*–ω–superinvariants and* awp*–ω–subinvariants are analogously defined by means of* awlp*– and* awp*–characteristic functions.*

2 But not necessarily *strictly* decreasing.
3 But not necessarily *strictly* increasing.

Using such sequences to reason about the correctness of programs is to the best of our knowledge originally due to Jones [Jon90, p. 124]. In her thesis, she basically used what we here call wp–ω–subinvariants for a total–correctness logic. Audebaud & Paulin–Mohring later build upon Jones' ideas and use monotonically increasing sequences to reason about total correctness of randomized algorithms in Coq [APM09, Section 4.4].

5.2 BOUNDS ON EXPECTED VALUES

Bounds on expected values, i.e. bounds on preexpectations, are a key concept in reasoning about probabilistic programs. Several correctness properties can be expressed as either upper or lower bounds on preexpectations. For example, we have already seen that the probability of event A can be coded as the expected value of the event's characteristic function $[A]$. Verifying bounds on probabilities is also the main task of the model checking problem of probabilistic logics like PCTL [HJ94].

Reasoning loop–free programs is mostly straightforward. Weakest preexpectations can be computed in practice.[4] For while loops, the situation is more difficult: Weakest (liberal) preexpectations of loops are defined as fixed points and those are in general non–computable. All non–trivial approximations of the fixed points are non–computable as well (see Part III).

In this section, we thus describe proof rules that can aid in reasoning about weakest (liberal) preexpectations of loops. We first describe inductive proof rules that allow for reasoning about upper bounds on wp and awp and coinductive proof rules that are suitable for lower bounds on wlp and awlp. We also briefly discuss the problem of coinduction for lower bounds on wp. Thereafter, we describe what we call ω–rules for reasoning about lower bounds on wp and awp, and upper bounds on wlp and awlp. We then survey proof rules by McIver & Morgan for lower bounds on wp and finally show how any bound can potentially be tightened.

5.2.1 *Induction for Weakest Preexpectations*

Induction on natural numbers is a well–known proof principle which can be traced back to classical antiquity, e.g. Euclid's proof that the number of primes is infinite. The induction principle states that in order to prove that a predicate F is true for all natural numbers, it suffices to prove that both

A. $0 \models F$, and

B. $n \models F$ implies $n+1 \models F$

are true. We can reformulate induction over the natural numbers in the setting of continuous functions on complete lattices [Rot16, Section 2.1]: We

4 I.e. in case the postexpectation is computable.

choose the complete lattice $(\mathcal{P}(\mathbb{N}), \subseteq)$, the continuous function

$$\Phi(X) \;=\; \{0\} \cup \{n+1 \mid n \in X\}\,, \tag{5.3}$$

and conceive of the predicate F as a set $F \in \mathcal{P}(\mathbb{N})$. We can easily convince ourselves that $\mathbb{N}$ is the least fixed point of Φ and that checking A. and B. above together amounts to checking whether $\Phi(F) \subseteq F$. The induction principle for the natural numbers then tells us that

$$\Phi(F) \;\subseteq\; F \quad \text{implies} \quad \mathsf{lfp}\,\Phi \;\subseteq\; F\,. \tag{5.4}$$

Since $\mathsf{lfp}\,\Phi = \mathbb{N}$, thus $\mathbb{N} \subseteq F$, and $\mathbb{N}$ is the greatest element in $\mathcal{P}(\mathbb{N})$, we can conclude that $F = \mathbb{N}$ and thus F holds for all numbers.

Implication 5.4 above is a special case of a more general principle (see Lemma A.6): Let $(D, \sqsubseteq)$ be any complete lattice and let $\Phi\colon D \to D$ be any continuous self–map on D. Then

$$\forall\, d \in D\colon \quad \Phi(d) \;\sqsubseteq\; d \quad \text{implies} \quad \mathsf{lfp}\,\Phi \;\sqsubseteq\; d\,.$$

The above general principle is called *Park's Lemma*, *Scott induction* or simply *induction* [Rot16, Section 2]. Since weakest preexpectations are defined as least fixed points of continuous functions on complete lattices, we can make use of the induction principle in order to reason about upper bounds on weakest preexpectations:

THEOREM 5.4 (Induction for Upper Bounds on wp and awp[5]):
Let $I \in \mathbb{E}$ be a wp*–superinvariant of* $\texttt{while}(\varphi)\{C\}$ *with respect to post-expectation f (see* Definition 5.1 A.*). Then*

$$\mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(f) \;\preceq\; I\,.$$

The analogous result for awp *holds as well.*

Proof. This is an instance of Park's Lemma (see Lemma A.6): Simply choose complete lattice $(\mathbb{E}, \preceq)$ and continuous function ${}^{\mathsf{wp}}_{\langle\varphi,C\rangle}\Phi_f$. Q.E.D.

EXAMPLE 5.5 (Upper Bounds on wp):
Consider the program C_{geo}, given by

$$\begin{aligned}
&c := 1 \,\fatsemi\\
&\texttt{while}(c = 1)\{\\
&\qquad \{c := 0\}\,[1/2]\,\{x := x+1\}\\
&\}\,,
\end{aligned}$$

5 For induction for tame programs, see [Koz85, the *while rule* on p. 168]

and suppose we want to reason about an upper bound on the expected value of x after execution of C_{geo}. To this end, we propose the wp–superinvariant

$$I \;=\; x + [c = 1]$$

and check its wp–superinvariance by applying the wp–characteristic function

$$\Phi(X) \;=\; [c \neq 1] \cdot x + [c = 1] \cdot \tfrac{1}{2}\Big(X\,[c/0] + X\,[x/x+1]\Big)\,,$$

to I, which gives us

$$\begin{aligned}
\Phi(I) \;&=\; \Phi\Big(x + [c = 1]\Big) \\
&=\; [c \neq 1] \cdot x + [c = 1] \cdot \tfrac{1}{2}\Big(x + [0 = 1] + x + 1 + [c = 1]\Big) \\
&=\; x + [c = 1] \cdot \tfrac{1}{2}\Big(0 + 1 + 1\Big) \\
&=\; x + [c = 1] \\
&=\; I \;\preceq\; I\,.
\end{aligned}$$

Thus the induction rule (Theorem 5.4) gives us that

$$\mathsf{wp}\,[\![\texttt{while}\,(\ldots\}]\!]\,(x) \;\preceq\; x + [c = 1] \tag{†}$$

and hence we get

$$\begin{aligned}
\mathsf{wp}\,[\![C_{geo}]\!]\,(x) \;&=\; \mathsf{wp}\,[\![c := 1 \fatsemi \texttt{while}\,(\ldots\}]\!]\,(x) \\
&=\; \mathsf{wp}\,[\![c := 1]\!]\,(\mathsf{wp}\,[\![\texttt{while}\,(\ldots\}]\!]\,(x)) \\
&\preceq\; \mathsf{wp}\,[\![c := 1]\!]\,(x + [c = 1]) \\
&\qquad\qquad \text{(by † and monotonicity, Theorem 4.16)} \\
&=\; x + [1 = 1] \\
&=\; x + 1
\end{aligned}$$

and therefore $x + 1$ (evaluated in the initial state) is an upper bound on the expected value of x (evaluated in the final states) after executing C_{geo}.

5.2.2 *Coinduction for Weakest Liberal Preexpectations*

The principle of *coinduction* is the dual of the induction principle and reads as follows [Rot16, Section 2]: Let $(D, \sqsubseteq)$ be any complete lattice and let $\Phi\colon D \to D$ be any continuous function. Then

$$\forall\, d \in D\colon \quad d \;\sqsubseteq\; \Phi(d) \quad \text{implies} \quad d \;\sqsubseteq\; \mathsf{gfp}\,\Phi\,.$$

For our example of the natural numbers, coinduction is not very interesting, since $\mathbb{N}$ is not only the least but also the greatest fixed point of Φ as defined

in Equation 5.3. For a predicate $F \in \mathcal{P}(\mathbb{N})$, we thus get by coinduction

$$F \subseteq \Phi(F) \quad \text{implies} \quad F \subseteq \mathbb{N}\,,$$

which, however, does not provide any information on F as the right–hand–side of the implication is vacuously true for any $F \in \mathcal{P}(\mathbb{N})$.

The particular problem we encounter with Φ here is that $\mathbb{N}$ is not only the greatest fixed point of Φ but indeed the greatest element in $\mathcal{P}(\mathbb{N})$ altogether. This is not the situation, however, for weakest liberal preexpectations: Those are defined as greatest fixed points and they may very well be below 1 — the greatest element in $(\mathbb{E}_{\leq 1}, \preceq)$. We may thus make use of the coinduction principle to reason about lower bounds on weakest liberal preexpectations:

Theorem 5.6 (Coinduction for Lower Bounds on wlp[6]):
Let $I \in \mathbb{E}_{\leq 1}$ be a wlp*–subinvariant of* $\texttt{while}(\varphi)\{C\}$ *with respect to post-expectation f (see* Definition 5.1 b.*). Then*

$$I \preceq \mathsf{wlp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(f)\,.$$

The analogous result for awlp *holds as well.*

Proof. This is an instance of Park's Lemma (see Lemma A.6): Simply choose complete lattice $(\mathbb{E}_{\leq 1}, \preceq)$ and continuous function ${}^{\mathsf{wlp}}_{\langle\varphi,C\rangle}\Phi_f$. Q.E.D.

Example 5.7 (Lower Bounds on wlp):
Reconsider the program C, given by

$$\begin{array}{l} c := 1 \,;\\ \texttt{while}(c = 1)\{\\ \quad \{\texttt{diverge}\}\,[1/2]\,\{x := x+1\}\,;\\ \quad \{\texttt{skip}\}\,[1/2]\,\{c := 0\}\\ \}\,, \end{array}$$

and suppose we want to reason about a lower bound on the probability that x is even after execution of C (if C terminates at all). To this end, we propose

$$I \;=\; [c \neq 1]\cdot[x \text{ even}] \;+\; [c = 1]\cdot\left(\frac{2}{3} + \frac{4\cdot[x \text{ odd}]}{15} + \frac{[x \text{ even}]}{15}\right)$$

as wlp–subinvariant and check wlp–subinvariance by applying the wlp–characteristic function

6 See [MM05, Lemma 7.2.2, p. 185].

$$\Phi(X) \;=\; [c \neq 1]\cdot[x \text{ even}] + [c=1]\cdot\left(\frac{1}{2} + \frac{X[x/x+1]}{4} + \frac{X[c,x/0,x+1]}{4}\right),$$

to I, which gives us

$$\begin{aligned}
\Phi(I) \;&=\; [c \neq 1]\cdot[x \text{ even}] + [c=1]\cdot\left(\frac{1}{2} + \frac{I[x/x+1]}{4} + \frac{I[c,x/0,x+1]}{4}\right)\\
&=\; [c \neq 1]\cdot[x \text{ even}] + [c=1]\cdot\left(\frac{1}{2} + [c\neq 1]\cdot\frac{\cdots}{4}\right.\\
&\qquad + [c=1]\cdot\left(\frac{2}{3\cdot 4} + \frac{4\cdot[x+1 \text{ odd}]}{15\cdot 4} + \frac{[x+1 \text{ even}]}{15\cdot 4}\right)\\
&\qquad \left. + [0\neq 1]\cdot\frac{[x+1 \text{ even}]}{4} + [0=1]\cdot(\ldots)\right)\\
&=\; [c \neq 1]\cdot[x \text{ even}]\\
&\qquad + [c=1]\cdot\left(\frac{1}{2} + \frac{2}{3\cdot 4} + \frac{4\cdot[x \text{ even}]}{15\cdot 4} + \frac{[x \text{ odd}]}{15\cdot 4} + \frac{[x \text{ odd}]}{4}\right)\\
&=\; [c \neq 1]\cdot[x \text{ even}] + [c=1]\cdot\left(\frac{2}{3} + \frac{[x \text{ even}]}{15} + \frac{4\cdot[x \text{ odd}]}{15}\right)\\
&=\; I \;\preceq\; I\,.
\end{aligned}$$

Thus the coinduction rule (Theorem 5.6) gives us that

$$\begin{aligned}
&[c \neq 1]\cdot[x \text{ even}] \;+\; [c=1]\cdot\left(\frac{2}{3} + \frac{4\cdot[x \text{ odd}]}{15} + \frac{[x \text{ even}]}{15}\right) \qquad (\dagger)\\
&\qquad \preceq\; \mathsf{wp}\,[\![\texttt{while}\ (\ldots)\}]\!]\,([x \text{ even}])
\end{aligned}$$

and hence we get

$$\begin{aligned}
&\mathsf{wp}\,[\![C]\!]\,([x \text{ even}])\\
&\;=\; \mathsf{wp}\,[\![c := 1\,;\,\texttt{while}\ (\ldots)\}]\!]\,([x \text{ even}])\\
&\;=\; \mathsf{wp}\,[\![c := 1]\!]\,(\mathsf{wp}\,[\![\texttt{while}\ (\ldots)\}]\!]\,([x \text{ even}]))\\
&\;\geq\; \mathsf{wp}\,[\![c := 1]\!]\left([c \neq 1]\cdot[x \text{ even}] + [c=1]\cdot\left(\frac{2}{3} + \frac{4\cdot[x \text{ odd}]}{15} + \frac{[x \text{ even}]}{15}\right)\right)\\
&\qquad\qquad \text{(by } \dagger \text{ above and monotonicity, Theorem 4.16)}\\
&\;=\; [1 \neq 1]\cdot[\ldots] \;+\; [1=1]\cdot\left(\frac{2}{3} + \frac{4\cdot[x \text{ odd}]}{15} + \frac{[x \text{ even}]}{15}\right)\\
&\;=\; \frac{2}{3} + \frac{4\cdot[x \text{ odd}]}{15} + \frac{[x \text{ even}]}{15}
\end{aligned}$$

and therefore $2/3 + 4\cdot[x \text{ odd}]/15 + [x \text{ even}]/15$ (evaluated in the initial state) is a lower

bound on the probability that C either diverges or terminates in a state where x is even.

5.2.3 *No Coinduction for Weakest Preexpectations*

We have seen in the previous subsection that induction allows us to *get above* a least fixed point whereas coinduction allows to *get below* a greatest fixed point. Unfortunately, *getting below a least fixed point* — and dually: *getting above a greatest fixed point* — is not associated with such elegant proof principles as induction or coinduction. In particular, for a complete lattice $(D, \sqsubseteq)$ and a continuous function $\Phi\colon D \to D$, the supposedly evident rules

$$\forall\, d \in D\colon \quad d \sqsubseteq \Phi(d) \quad \text{implies} \quad d \sqsubseteq \mathsf{lfp}\,\Phi\,, \qquad \lightning$$

and

$$\forall\, d \in D\colon \quad \Phi(d) \sqsubseteq d \quad \text{implies} \quad \mathsf{gfp}\,\Phi \sqsubseteq d \qquad \lightning$$

are both *unsound*, not only in general but also in our particular use case of preexpectations as the following counterexample demonstrates:

Counterexample 5.8 (Unsoundness of Coinduction for wp):
Consider the program C, given by

$$\begin{aligned}&\texttt{while}(c = 1)\{\\ &\quad \{c := 0\}\,[1/2]\,\{x := x+1\}^{\circ}_{\circ}\\ &\quad k := k+1\\ &\}\,,\end{aligned}$$

and suppose we want to incorrectly reason about a lower bound on the expected value of x after execution of C by coinduction. The wp–characteristic function of the while loop with respect to postexpectation x is given by

$$\Phi(X) \;=\; [c \neq 1]\cdot x + [c=1]\cdot \tfrac{1}{2}\Big(X\,[k, c/k+1, 0] + X\,[k, x/k+1, x+1]\Big)\,.$$

We now propose *infinitely many* fixed points of Φ, namely for every $a > 0$

$$I_a \;=\; x + [c=1]\left(2^{k+a}+1\right)$$

is a fixed point of Φ, as one can easily check. However, for any $d < b$, we clearly have $I_d \prec I_b$. Thus, if we prove $I_b \preceq \Phi(I_b)$ we cannot have proven that I_b is a lower bound on the least fixed point of Φ, since I_d is a fixed point strictly smaller than I_b. In fact, none of the I_a's are the least fixed point of Φ. The intuitive reason is that the expected value of x is completely independent of k but k has an influence on the value that the I_a's assume. $\boxed{\lightning}$

It is important to note that *unsoundness of coinductive premises in order to obtain lower bounds on* wp *is absolutely not evident*. We will see later in Chapter 7, that for *deterministic* programs Frohn *et al.* have shown that one *can* prove *lower bounds on runtimes of programs* from wp–subinvariants, which Frohn *et al.* call *metering functions* [Fro+16b]. This allows to lower bound a least fixed point from a coinductive premise (i.e. from a premise of the form $d \sqsubseteq \Phi(d)$). Transferring the metering function method to probabilistic programs, however, unfortunately fails in a way similar to the above counterexample, as we will see later in this thesis.

5.2.4 *ω–Rules*

In light of our just described inability to obtain lower bounds on weakest preexpectations, and dually upper bounds on weakest liberal preexpectations, by simple means such as coinduction or induction, we now present two alternative proof rules for obtaining precisely such desired bounds. These proof rules will be conceptually less elegant and consequently more difficult to apply, as they make use of ω–invariants. In particular, it will be necessary to find the limit of such ω–invariants in order to actually gain some insights from applying these rules. That basically just shifts the problem of obtaining bounds into the realm of real analysis. The rule for lower bounds on weakest preexpectations (*getting below a least fixed point*) reads as follows:

THEOREM 5.9 (Lower Bounds on wp and awp from ω–Invariants[7]):

A. *Let $(I_n)_{n\in\mathbb{N}}$ be a* wp*–ω–subinvariant of* $\texttt{while}(\varphi)\{C\}$ *with respect to postexpectation f (see* Definition 5.3 B.*). Then*

$$\sup_{n\in\mathbb{N}} I_n \preceq \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(f)\,.$$

B. *Let $(I_n)_{n\in\mathbb{N}}$ be a* wlp*–ω–superinvariant of* $\texttt{while}(\varphi)\{C\}$ *with respect to postexpectation f (see* Definition 5.3 A.*). Then*

$$\mathsf{wlp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(f) \preceq \inf_{n\in\mathbb{N}} I_n\,.$$

C. *Analogous results for* awp *and* awlp *hold as well.*

Proof. We only prove A., because the proofs for B. and C. are analogous. Let Φ be the wp–characteristic function of $\texttt{while}(\varphi)\{C\}$ with respect to postexpectation f. We first prove by induction that

$$\forall\, n \in \mathbb{N}\colon \quad I_n \preceq \Phi^{n+1}(0)\,.$$

7 See also [Jon90, p. 124] and [APM09, Section 4.4].

For the induction base we have

$$I_0 \;=\; 0 \;\preceq\; \Phi(0)$$

trivially, since 0 is the least element in $\mathbb{E}$. For the induction step we assume induction hypothesis $I_n \preceq \Phi^{n+1}(0)$ and prove

$$\begin{aligned} I_{n+1} &\preceq \Phi(I_n) && \text{(by } (I_n)_{n\in\mathbb{N}} \text{ being a wp–}\omega\text{–subinv., Definition 5.3 B.)} \\ &\preceq \Phi\left(\Phi^{n+1}(0)\right) && \text{(by I.H. and monotonicity of } \Phi\text{, Theorem 4.16)} \\ &= \Phi^{n+2}(0)\,. \end{aligned}$$

Since $I_n \preceq \Phi^{n+1}(0)$ holds for all n and $\Phi^0(0) = 0$, we may take the supremum on both sides and conclude:

$$\begin{aligned} \sup_{n\in\mathbb{N}} I_n &\preceq \sup_{n\in\mathbb{N}} \Phi^{n+1}(0) \\ &= \sup_{n\in\mathbb{N}} \Phi^{n}(0) && \left(\text{by } \Phi^0(0) = 0 \text{ being the least element in } \mathbb{E}\right) \\ &= \mathsf{lfp}\,\Phi \\ &= \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(f) && \boxed{\text{Q.E.D.}} \end{aligned}$$

EXAMPLE 5.10 (Bounds from ω–rules):
Recall Example 2.6, Example 2.9, Example 2.11 B., and Example 4.10. In all of those examples we performed a fixed point iteration. The „patterns“ — as we called them — which we learned by inspecting the evolution of the first few iterations were in fact ω–invariants.

Proving the pattern correct would correspond to the induction on n in the ω–rule. Finding the limit of a pattern would correspond to finding a closed form for a sup or an inf, accordingly.

Let us briefly reflect on the usability of ω–rules. Recall that verification of loops by means of the induction and the coinduction rule (Theorem 5.4 and Theorem 5.6) was conceptually very simple. Informally, the steps we had to take are the following:

1. Find an appropriate invariant I.
2. Push I through the characteristic function of the loop once.
3. Check whether Step 2. took us down (for induction) or up (for coinduction) in the partial order $\preceq$.

Often, the „only“ difficulty that we encounter in practice is with Step 1: Finding an appropriate invariant (even though this can admittedly be *very* difficult in practice).

Verification of loops using ω–rules (Theorem 5.9) on the other hand is much more involved. In summary, the steps we have to take are as follows:

1. Find an appropriate ω–invariant, i.e. a *sequence* $(I_n)_{n\in\mathbb{N}}$.
2. Check that $(I_n)_{n\in\mathbb{N}}$ is indeed an ω–invariant, e.g. by *induction on n*:
   a) Push I_n through the characteristic function.
   b) Check whether performing Step a) took us above I_{n+1} (for wp) or below I_{n+1} (for wlp) in the partial order $\preceq$.
3. *Find the supremum* (for wp) or the *infimum* (for wlp) of $(I_n)_{n\in\mathbb{N}}$.

Steps 2.a) and 2.b) for the ω–rules basically correspond to Steps 2. and 3. for (co)induction. However, for ω–rules we have to perform an additional induction on the natural numbers.

The second — and probably more significant — extra effort we have to take is reasoning about the limits of the ω–invariants. For wp, for instance, one might very well argue that we may then just as well directly infer the supremum sequence $\Phi^n(0)$ in order to obtain the exact expected value. As a matter of fact, in my personal experience, we have never encountered a case where we found a wp–ω–subinvariant I_n that *truly underapproximated* $\Phi^n(0)$. Instead, we were always able to prove $I_n = \Phi^n(0)$. The difficulty with finding the supremum of the sequence, however, remains. Personally, I therefore believe that both the usability as well as the gain of ω–rules is very limited in practice.

Despite the extra difficulties that come with using ω–rules, a natural question that arises is whether an ω–rule for upper bounds on weakest preexpectations, and dually an ω–rule for lower bounds on weakest liberal preexpectations, could be of any advantage. Luckily, the following remark gives a negative answer to this question.

Remark 5.11 (Expendability of ω–rules for upper bounds on wp*).* Let us formulate the ω–rule for wp–reasoning: Let Φ be the wp–characteristic function of $\texttt{while}(\varphi)\{C\}$ with respect to postexpectation f and let $(I_n)_{n\in\mathbb{N}} \subset \mathbb{E}$ be a monotonically decreasing sequence. Then

$$\Phi(I_n) \preceq I_{n+1} \qquad \text{implies} \qquad \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(f) \preceq \inf_{n\in\mathbb{N}} I_n\,.$$

The soundness proof for this rule goes as follows:

$$\begin{aligned}
& \forall\, n \in \mathbb{N}\colon \quad \Phi(I_n) \preceq I_{n+1} \\
\text{implies} \quad & \inf_{n\in\mathbb{N}} \Phi(I_n) \preceq \inf_{n\in\mathbb{N}} I_{n+1} \\
\text{implies} \quad & \inf_{n\in\mathbb{N}} \Phi(I_n) \preceq \inf_{n\in\mathbb{N}} I_n \\
\text{implies} \quad & \Phi\Big(\inf_{n\in\mathbb{N}} I_n\Big) \preceq \inf_{n\in\mathbb{N}} I_n \qquad \text{(by continuity, Theorem 4.12)}
\end{aligned}$$

$$\text{implies} \qquad \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(f) \;\preceq\; \inf_{n\in\mathbb{N}} I_n$$

(by induction rule, Theorem 5.4)

As a byproduct of our proof, we have shown that $\inf_{n\in\mathbb{N}} I_n$ itself is a wp–superinvariant. Since, ultimately, we have to find the infimum $\check{I} = \inf_{n\in\mathbb{N}} I_n$ anyway in order to gain some insights from the ω–rule, we could have just as well applied the induction rule immediately to $\check{I}$ and could therefore have dispensed with the extra induction on n imposed by the ω–rule.

Dually to the above, an ω–rule for lower bounds on weakest liberal preexpectations is expendable as well. △

5.2.5 *Lower Bounds on* wp

There is a genuine and legitimate interest in reasoning about lower bounds on weakest preexpectations, namely when it comes to giving total correctness guarantees which amounts to *lower–bounding* the probability of total correctness. Yet, we saw that applying ω–rules is quite involved. McIver & Morgan came up with interesting total correctness rules that mitigate this unpleasant situation to the extent that their rules do not rely on ω–invariants. One of the most important rules on which a larger part of their oeuvre on proof rules for probabilistic loops builds upon reads as follows:

Theorem 5.12 ([MM05][8]):
Let $f \in \mathbb{E}_{\leq \exists b}$ be a ***bounded*** *postexpectation. Furthermore, let $I' \in \mathbb{E}_{\leq \exists b}$ be a bounded expectation such that expectation $I \in \mathbb{E}$ given by*

$$I \;=\; [\neg\varphi]\cdot f + [\varphi]\cdot I'$$

is a wp*–subinvariant of* $\texttt{while}(\varphi)\{C\}$ *with respect to f. Finally, let*

$$T \;=\; \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(1)\;.$$

be the termination probability of $\texttt{while}(\varphi)\{C\}$*. Then:*

A. *If $I = [G]$ for some predicate G, then*

$$T\cdot I \;\preceq\; \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(f)\;.$$

B. *If $[G] \preceq T$ for some predicate G, then*

$$[G]\cdot I \;\preceq\; \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(f)\;.$$

C. *If $\epsilon\cdot I \preceq T$ for some $\epsilon > 0$, then*

$$I \;\preceq\; \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(f)\;.$$

8 More specifically, this theorem combines Lemma 2.4.1 on p. 43, its relaxation described on p. 54, Lemma 7.7.6 on p. 203, Theorem 7.3.3 on p. 188, and Theorem B.2.2 on p. 329 in [MM05].

Intuitively, Theorem 5.12 provides lower bounds on $\mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(f)$ in the following scenarios:

1. If the invariant I is the indicator function of a predicate G, then I multiplied by the termination probability T is a lower bound.
2. If the termination probability T is lower–bounded by the indicator function of some predicate G, then the invariant I multiplied by that indicator function $[G]$ is a lower bound.
3. If the termination probability T is lower–bounded by some non–zero constant fraction ϵ of the invariant I, then the invariant I itself is a lower bound.

While at first glance Theorem 5.12 seems easier to apply than ω–rules, it has several drawbacks of its own: For one, it is only applicable to *bounded* expectations which renders reasoning about general expected values (as opposed to reasoning e.g. about probabilities) difficult, if not impossible.

Another major drawback of Theorem 5.12 is that it requires substantial knowledge about the termination probability $\mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(1)$. Reasoning about this probability is quite involved too, although we will later present proof rules (some more, some less involved) that can render reasoning about probabilistic termination feasible (see Chapter 6).

Despite the just mentioned difficulties of applying Theorem 5.12 in practice, expecially Theorem 5.12 c. is an important theoretical device for proving the correctness of several other proof rules. In particular, several of the termination rules in Chapter 6 ultimately build upon Theorem 5.12 c.

If by some means we already known that $\texttt{while}(\varphi)\{C\}$ terminates universally almost–surely, then for one–bounded expectations $f \in \mathbb{E}_{\leq 1}$ we know by Corollary 4.28 that $\mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(f)$ and $\mathsf{wlp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(f)$ coincide. Thus, in that case there exists only one fixed point and we hence obtain the following corollary:

COROLLARY 5.13 (Bounds on Almost–surely Terminating Loops): *Let the loop* $\texttt{while}(\varphi)\{C\}$ *terminate universally almost–surely, i.e.*

$$\mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(1) \;=\; 1$$

and let $I \in \mathbb{E}_{\leq 1}$. *Then:*

A. *If* I *is a* wp*–subinvariant of* $\texttt{while}(\varphi)\{C\}$ *with respect to postexpectation* f, *then*

$$I \;\preceq\; \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(f)\,.$$

B. *If* I *is a* wlp*–superinvariant of* $\texttt{while}(\varphi)\{C\}$ *with respect to postexpectation* f, *then*

$$\mathsf{wlp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(f) \;\preceq\; I\,.$$

Another rule by McIver & Morgan allows — interestingly — for reasoning about weakest preexpectations by means of wlp–subinvariants:

Theorem 5.14 ([MM05, Lemma 7.3.1 on p. 186])**:**
*Let $I' \in \mathbb{E}_{\leq 1}$ be a **one–bounded** expectation such that $I \in \mathbb{E}_{\leq 1}$ given by*

$$I \;=\; [\neg\varphi]\cdot f + [\varphi]\cdot I'$$

is a wlp*–subinvariant of* `while(`φ`){`C`}` *with respect to postexpectation f. Furthermore, let*

$$T \;=\; \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(1)\;,$$

and let $g \ominus h$ be defined as $\max\{g-h,\,0\}$, for any $g,h \in \mathbb{E}$. Then

$$(I+T)\ominus 1 \;\preceq\; \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(f)\;.$$

While this rule is methodologically interesting since it derives a total correctness property (a weakest preexpectation) from a partial correctness invariant (a wlp–invariant), it still has the potentially severe drawback that we need substantial knowledge about the termination probability of the loop at hand.

5.2.6 *Upper Bounds vs. Lower Bounds*

Generally speaking (and perhaps slightly over–simplified), we saw that reasoning about upper bounds of least fixed point (and dually reasoning about lower bounds of greatest fixed points) is easy, whereas reasoning about lower bounds of least fixed points (and dually reasoning about upper bounds of greatest fixed points) is more involved.

We will learn later in Part III that from a computational hardness perspective, the exact *opposite* to what we just stated should be expected. This constitutes a seemingly paradoxical situation to which to the best of our knowledge no *good* explanation is known.

An *unsatisfactory* explanation why lower bounds for weakest preexpectations are in fact computationally tractable is the following: Suppose we want to reason about a lower bound for $\mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(f)$ and Φ is the associated characteristic function. Then $(\Phi^n(0))_{n\in\mathbb{N}}$ is trivially an ω–invariant. But then for some fixed $k \geq 2$, the sequence

$$\left(0, \Phi(0), \Phi^2(0), \ldots, \Phi^{k-1}(0), \Phi^k(0), \Phi^k(0), \Phi^k(0), \ldots\right)_{n\in\mathbb{N}}\;,$$

i.e. the so to speak *forced stabilization* of $(\Phi^n(0))_{n\in\mathbb{N}}$ after k iterations, is also an ω–invariant with an easy–to–find (i.e. computable) limit: $\Phi^k(0)$.

This method is of course unsatisfactory, since we had to perform k iterations, i.e. applications of Φ, in order to obtain *some* lower bound. In fact, the

tighter a bound we want to obtain, the more effort we have to invest. *This is not the case for induction or coinduction.* So while the sequence $(\Phi^n(0))_{n\in\mathbb{N}}$ successively indeed enumerates all lower bounds, a major problem in probabilistic program verification remains open:

> **OPEN PROBLEM 1 (One–shot Verification of Lower Bounds on wp):** *Find a „one–shot" method as elegant as the induction or coinduction rule (*Theorems 5.4 *and* 5.6*), which, given a loop* $\texttt{while}(\varphi)\{C\}$*, a post-expectation* $f \in \mathbb{E}$*, and a* specific *hypothesis* $L \in \mathbb{E}$*, allows for checking whether* L *is in fact a lower bound on* $\mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(f)$.

In Section 6.2, we will present a rule that can be regarded as a partial solution to the above problem, namely for the special case of almost–sure termination, which amounts to proving that 1 is a (non–strict) lower bound on the termination probability. However, for lower bounds on arbitrary preexpectations, to the best of our knowledge, no sufficiently elegant method is known.

5.2.7 *Bound Refinement*

We saw that obtaining a bound on a weakest (liberal) preexpectation of a loop can be quite difficult. However, once we have obtained some bound — be it upper or lower — by any means (e.g. by application of one of the proof rules presented in the previous sections), we have a chance of refining and thereby tightening this bound fairly easily:

> **THEOREM 5.15 (Bound Refinement):** *Let* Φ *be the* wp*–characteristic function of* $\texttt{while}(\varphi)\{C\}$ *with respect to* f *and let* I *be an upper bound on* $\mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(f)$*, such that* $\Phi(I) \preceq I$.
>
> *Then* $\Phi(I)$ *is also an upper bound on* $\mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(f)$*. Moreover, whenever* $\Phi(I) \neq I$*, then* $\Phi(I)$ *is an even tighter upper bound on* $\mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(f)$ *than* I.
>
> *Dually, if* I *is a lower bound, such that* $I \preceq \Phi(I)$*, then* $\Phi(I)$ *is also a lower bound; and whenever* $\Phi(I) \neq I$*, then* $\Phi(I)$ *is an even tighter lower bound than* I.
>
> *Analogous results hold for* awp, wlp, *and* awlp *as well.*

Proof. Let I be an upper bound on $\mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(f)$. To see that $\Phi(I)$ is also an upper bound on $\mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(f)$, consider the following:

$$
\begin{aligned}
& \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(f) \;\preceq\; I \\
\text{iff}\quad & \mathsf{lfp}\;\Phi \;\preceq\; I \\
\text{implies}\quad & \Phi(\mathsf{lfp}\;\Phi) \;\preceq\; \Phi(I) \qquad \text{(by monotonicity, Theorem 4.16)}
\end{aligned}
$$

$$\begin{aligned} &\text{implies} \quad \mathsf{lfp}\ \Phi \preceq \Phi(I) && (\mathsf{lfp}\ \Phi \text{ is a fixed point of } \Phi)\\ &\quad\ \text{iff} \quad \mathsf{wp}\,[\![\mathtt{while}(\varphi)\{C\}]\!]\,(f) \preceq \Phi(I) \end{aligned}$$

By the assumption $\Phi(I) \preceq I$, $\Phi(I)$ is at least as tight an upper bound as I. Thus if $\Phi(I) \neq I$, $\Phi(I)$ must be an even tighter upper bound.

The reasoning for awp, wlp, awlp, and lower bounds is analogous. Q.E.D.

The particular bound refinement of Theorem 5.15 can of course be continued ad infinitum: For instance, if I is an upper bound on $\mathsf{wp}\,[\![\mathtt{while}(\varphi)\{C\}]\!]\,(f)$ with $\Phi(I) \preceq I$, then so is $\Phi(I)$ but also $\Phi^2(I)$, $\Phi^3(I)$, and so on. In fact, for increasing n, the sequence $\Phi^n(I)$ is decreasing and converges to a fixed point, more precisely the *greatest fixed point that is below (or equal to) I*. This is called the *Tarski–Kantorovich principle* [JGP00]. The so–obtained fixed point itself is then also an upper bound, thus

$$\mathsf{wp}\,[\![\mathtt{while}(\varphi)\{C\}]\!]\,(f) \preceq \inf_{n\in\mathbb{N}} \Phi^n(I)\,.$$

Dually, if I is a lower bound on $\mathsf{wp}\,[\![\mathtt{while}(\varphi)\{C\}]\!]\,(f)$ with $I \preceq \Phi(I)$, then so are $\Phi(I)$, $\Phi^2(I)$, $\Phi^3(I)$, and so on, and moreover

$$\sup_{n\in\mathbb{N}} \Phi^n(I) \preceq \mathsf{wp}\,[\![\mathtt{while}(\varphi)\{C\}]\!]\,(f)\,.$$

5.2.8 *Independent and Identically Distributed Loops*

We have learned in the previous sections that obtaining bounds — especially lower bounds —, on weakest preexpectations of while loops can be a very difficult task. Obtaining exact weakest preexpectations obviously cannot be any easier in principle. Under certain conditions, however, we are able to derive the *exact* weakest preexpectation of a while loop with respect to a given postexpectation. Informally, these conditions can be described as follows:

1. For each loop iteration, the probability to immediately terminate after that iteration is equal.

2. There is *no information* flow across different loop iterations with respect to any program variable that has an influence on the value of the postexpectation f.

In the following, we will make the above two conditions more formal. A central notion for achieving this formalization is the concept of a loop being f–independent identically distributed (f–i.i.d. for short):

DEFINITION 5.16 (f–i.i.d. Loops [Bat+18b]):
Let $f \in \mathbb{E}$ and $C \in \mathsf{pGCL}$.

A. *The set of variables occurring in f, denoted $\mathsf{Vars}(f)$, is defined as*

$$\mathsf{Vars}(f) \;=\; \{x \in \mathsf{Vars} \mid \exists v, v' \in \mathsf{Vals}\colon\; f[x/v] \;\neq\; f[x/v']\}\,.$$

B. *The set of variables modified by C, denoted $\mathsf{Mod}(C)$, is defined as the set of all variables $x \in \mathsf{Vars}$, such that x appears on the left–hand–side of an assignment occurring in C.*

C. *We say that C cannot influence f, denoted $C \not\Rightarrow f$, if the set of variables occurring in f is disjoint from the set of variables modified by C, i.e.*

$$C \not\Rightarrow f \qquad \text{iff} \qquad \mathsf{Mod}(C) \cap \mathsf{Vars}(f) \;=\; \emptyset\,.$$

D. *A loop $\texttt{while}(\varphi)\{C\}$ is called f–independent identically distributed (f–i.i.d. for short), iff*

$$C \not\Rightarrow \mathsf{wp}\,[\![C]\!]\,([\varphi]) \qquad \text{and} \qquad C \not\Rightarrow \mathsf{wp}\,[\![C]\!]\,([\neg\varphi]\cdot f)\,.$$

Notice that $\mathsf{Mod}(\ldots)$ is a *purely syntactic* notion. On the other hand, the definition of $\mathsf{Vars}(\ldots)$ has more of a *semantic flavor* as it speaks about a property of a potentially arbitrary function of type $\Sigma \to \mathbb{R}^{\infty}_{\geq 0}$. However, if we are given a closed form syntactic expression for the expectation f, we can at least overapproximate $\mathsf{Vars} f$ by the set of all variables that actually occurr in f, syntactically. Nevertheless, because of the semantic flavor of $\mathsf{Vars}(\ldots)$, the relation $\not\Rightarrow$ and the notion of f–i.i.d.–ness is not purely syntactic

The definition of f–i.i.d.–ness is very technical and providing an intuition for it is not an easy task. A more pleasant aspect about the definition is that in practice it can often be checked in a quite straightforward and even *automatable* manner, despite not being a purely syntactic notion [Bat+18b]. The most important aspect, however, is that for f–independent identically distributed loops we can obtain *exact* weakest preexpectations:

THEOREM 5.17 (Weakest Preexpectations of f–i.i.d. Loops [Bat+18b]):
Let $\texttt{while}(\varphi)\{C\}$ be f–i.i.d. Then the weakest preexpectation of the loop with respect to f is given by

$$\mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(f) \;=\; [\neg\varphi]\cdot f \;+\; [\varphi]\cdot \frac{\mathsf{wp}\,[\![C]\!]\,([\neg\varphi]\cdot f)}{1 - \mathsf{wp}\,[\![C]\!]\,([\varphi])}\,,$$

where we define $\%/0 = 0$.

Intuitively, as the expected value of f can be determined by just a single iteration of the loop body, the fraction appearing in Theorem 5.17 can be understood as the *conditional expected value* of f given that the loop terminates.

It is worthwhile to note that in order to apply Theorem 5.17 it is *not required* to find or guess in any way an invariant, ω–invariant, martingale, or alike. Instead, only f–i.i.d.–ness of f — the very postexpectation one is interested in — needs to be checked. Our theorem then immediately yields the *exact* sought–after preexpectation — not just a bound.

Finally, we would like to mention that Theorem 5.17 is obviously not a free–lunch–theorem: Checking f–i.i.d.–ness can potentially become a non–trivial and in general undecidable task. Also, once the expected value of postexpectation f depends in some way on the number of iterations a loop makes, i.e. once the loop performs some sort of *counting* and the value of the counter influences the value of f, the theorem fails to be applicable altogether. On the other hand, Theorem 5.17 has been successfully applied to reason about massively large Bayesian networks from the *Bayesian Network Repository* [Scu] with more than a thousand nodes [Bat+18b].

PROBABILISTIC TERMINATION 6

Termination is one of the most fundamental liveness properties of probabilistic programs and is naturally an active area of research [HSP83; SPH84; APZ03; BG05; BG06; SS11; EGK12; CS13; FH15; KK15b; CFG16; Cha+16; CNZ17; CF17; DLG17; ACN18; McI+18]. Already the very notion of *termination* is much more nuanced and subtle for probabilistic programs than it is for nonprobabilistic ones. Whereas a deterministic program either terminates on a given input with *certainty* or not at all, the following two forms of *probabilistic termination* are mainly considered in the literature:

- ✧ *Almost–sure termination:* Termination with probability 1.
- ✧ *Positive almost–sure termination:* Termination in finite expected time.

In this chapter, we survey and discuss proof rules for proving the different forms of probabilistic termination. In particular, we present a more recent proof rule for proving almost–sure termination of loops that do not necessarily terminate in finite expected time — a notoriously difficult task in probabilistic program verification.

Let us first develop the differences between different forms of termination. As a first example, consider the probabilistic program

$$\begin{array}{l}\texttt{while}(x>0)\{\\ \quad \{x:=x-1\}\,[1/2]\,\{x:=x-2\}\\ \}\,.\end{array}$$

This program terminates *universally certainly*, meaning that every possible computation path of the program terminates. And indeed, even though there are probabilistic choices and the time until termination depends on the outcome of the coin tosses, *every* possible computation path of this program on a given input terminates after at most $\max\{\lceil x\rceil, 0\}$, thus finitely many, loop iterations. The probabilistic nature of the program has thus no effect on termination itself, but only on the time until termination.

As a second example, consider the program

$$\begin{array}{l}\texttt{while}(x>0)\{\\ \quad \{x:=x-1\}\,[1/2]\,\{\texttt{skip}\}\\ \}\,.\end{array}$$

This program admits a single diverging run (namely the one in which infinitely often `skip` is executed). Even though the diverging path has proba-

bility 0, this path does exist and because of this, the program does not terminate certainly. The program does, however, terminate universally almost–surely, i.e. with probability 1. Moreover, the program terminates *universally positively almost–surely* as it needs on average $\max\{2\lceil x\rceil, 0\}$ loop iterations until it terminates, which for a given input x is a finite number. If we were to execute the program on an initial state with $x = 10$, we could thus expect to witness termination of the program within 20 loop iterations. Formally, positive almost–sure termination is defined as follows:

Definition 6.1 (Expected Runtimes and Positive A.–s. Termination):

A. *Let $C \in$ pGCL and let $\sigma \in \Sigma$ be a program state. Then the worst–case expected runtime of C on input σ is given by*[1]

$$\mathsf{ERT}\,[\![C]\!]_\sigma \;=\; \sup_{\mathfrak{s}\in\mathsf{Scheds}} \sum_{i=1}^{\omega} \left(1 \;-\; \sum_{\langle\downarrow,\tau,n,\theta,\eta,q\rangle\in K_{\mathfrak{s}}^{<i}} q\right), \qquad \text{where}$$

$$K_{\mathfrak{s}}^{<i} \;=\; \left\{ \langle\downarrow,\tau,n,\theta,\eta,q\rangle \;\middle|\; \langle C,\sigma,0,\varepsilon,\varepsilon,1\rangle \;\vdash_{\mathfrak{s}}^{*}\; \langle\downarrow,\tau,n,\theta,\eta,q\rangle,\; n<i \right\}.$$

B. *C terminates positively almost–surely on input σ iff its expected runtime on input σ is finite, i.e. $\mathsf{ERT}\,[\![C]\!]_\sigma < \infty$.*

C. *C terminates universally positively almost–surely iff C terminates positively almost–surely on* all *inputs, i.e. $\forall\, \sigma \in \Sigma\colon\ \mathsf{ERT}\,[\![C]\!]_\sigma < \infty$.*

The intuition for the formula for $\mathsf{ERT}\,[\![C]\!]_\sigma$ above is that we can express the expected value of a non–negative $\big(\mathbb{N}\cup\{\infty\}\big)$–valued random variable X as

$$\mathsf{EV}(X) \;=\; \sum_{i=1}^{\omega} \Pr(X \geq i)\,.$$

As we have no direct access to the probability that a probabilistic program runs for at least i steps, we compute 1 minus the probability that the program runs for less than i steps.

The terminology *positive almost–sure termination* was introduced by Bournez & Garnier [BG05]. Their inspiration for the term „positive“ came from Markov chain theory, more specifically from the distinction between *positively recurrent* states (states that are revisited with probability one and the expected time until a revisit is finite) and *null recurrent* states (states that are revisited with probability one but the expected time to revisit is infinite) [Put05, Section A.2, p. 588]. Adapting this line of thought, almost–surely terminating programs that do not terminate positively almost–surely could be called *null almost–surely terminating*. We consider such cases next.

As our third example, consider the program

1 Recall Definition 3.4 and Definition 3.7.

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

This program admits infinitely many diverging runs but their aggregated probability is 0. In contrast to the second example, however, this third program does *not* terminate within an expected finite number of loop iterations. Its expected runtime is infinite. Thus, the notion under which we can speak of termination of the above program is *weaker*: It terminates almost–surely, i.e. with probability 1. Formally, almost–sure termination of programs is defined as follows:

Definition 6.2 (Almost–sure Termination):
Let C be a pGCL *program and let $\sigma \in \Sigma$ be an initial program state. Then C terminates almost–surely on input σ iff*

$$\mathsf{wp}\,[\![C]\!]\,(1)(\sigma) \;=\; 1\,.$$

C terminates universally almost–surely iff C terminates almost–surely on all *inputs, i.e.*

$$\mathsf{wp}\,[\![C]\!]\,(1) \;=\; 1\,.^{2}$$

C terminates (universally) null almost–surely iff C terminates (universally) almost–surely but not (universally) positively almost–surely.

The example program above terminates *universally null almost–surely*, since it terminates with probability 1 but requiring infinite expected runtime. Intuitively, if we were to execute the program on a state with $x = 10$, we would expect the program to terminate, but we cannot expect to witness its termination within our lifespan.

Proving universal almost–sure termination of a program C amounts to proving that 1 is a (non–strict) *lower bound* on the termination probability of C, i.e. proving $1 \preceq \mathsf{wp}\,[\![C]\!]\,(1)$. Proving universal positive almost–sure termination of C amounts to proving a finite *upper bound* on the expected runtime of C (for a calculus for reasoning about expected runtimes, see Chapter 7). From our experience in Section 5.2 and in particular from our considerations in Section 5.2.6 we can expect positive almost–sure termination proofs to be easier in practice, since they constitute an upper bound proof. And indeed, we will see that the methodology for positive almost–sure termination is easier than the one for almost–sure termination proofs.

Besides certain, positive almost–sure, and almost–sure termination, another notion that is sometimes considered are so–called *tail bounds* or *tail*

2 Notice that the two 1's on the right hand sides of the two equations in this definition are of different type. The first 1 is the real number 1, whereas the second 1 is an expectation, namely $\lambda\sigma.\ 1$.

probabilities [CF17; CNZ17]. For a given program and input, tail bounds map each number $n \in \mathbb{N}$ to the probability that the program performs at least n computation steps on the given input. We will, however, not consider tail bounds in this thesis but instead focus on positive almost–sure and almost–sure termination, starting with the former.

6.1 POSITIVE ALMOST–SURE TERMINATION

For a nonprobabilistic loop $\texttt{while}(\varphi)\{C\}$, one way to prove termination is by use of *loop variants* [Flo67a, p. 30 et seqq.]. A loop variant is a mapping from program states to a well–founded set, i.e. a set together with an order relation in which no infinite descend is possible, such that iteration of the loop body strictly decreases the value of the variant. Existence of a loop variant then proves termination of the loop.

A particular form of loop variants are *ranking functions* [Dij75, p. 455]. A ranking function R maps program states to real numbers and satisfies the following two constraints for every state σ:

A. If $\sigma \models \varphi$, then execution of C on σ terminates in a state τ such that

$$R(\tau) \;\leq\; R(\sigma) - \epsilon\,,$$

for some fixed $\epsilon > 0$, and

B. if $R(\sigma) \leq 0$, then $\sigma \not\models \varphi$.

Constraint A. ensures that, from any state σ satisfying the loop guard, the execution of the loop body reaches a successor state whose ranking is at least by ϵ smaller than σ's ranking, thus ensuring a strict descent. Constraint B. ensures that if the ranking hits 0 or drops below, this falsifies the loop guard and thus causes the loop to terminate. Therefore, from any state σ, no infinite chain of successor states with ever decreasing ranking can be formed by iterated execution of the loop body without eventually falsifying the loop guard and thus terminating the loop, since the length of such a chain is bounded by $\lceil R(\sigma)/\epsilon \rceil$. This ensures certain termination of the loop within at most $\lceil R(\sigma)/\epsilon \rceil$ loop iterations.

For instance, for the program

$$\begin{array}{l}\texttt{while}(x > 0)\{\\ \quad x := x - 1\\ \}\end{array}$$

we can choose the ranking function $R = x$ or more formally

$$R \;=\; \lambda\sigma.\; \sigma(x)\,.$$

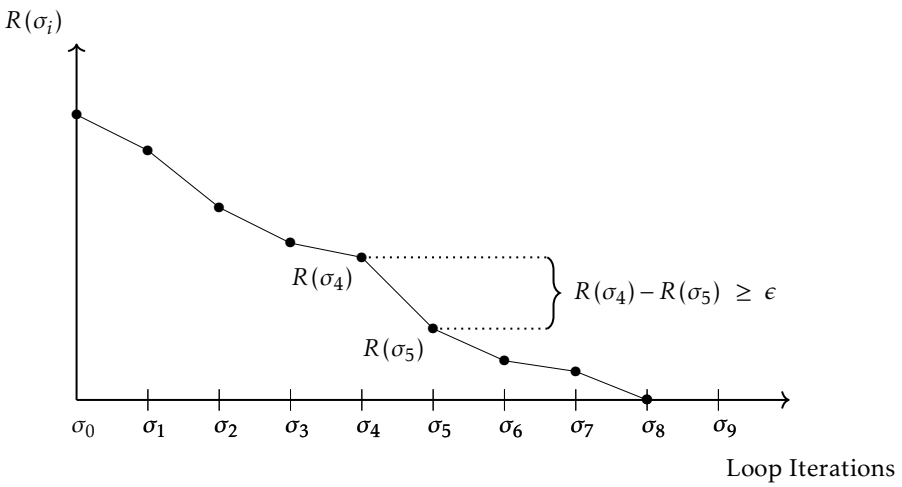


Figure 6.1: Evolution of the values of a ranking function R over the iterations of a loop. σ_0 is the initial state and $\sigma_1, \sigma_2, \sigma_3, \ldots$ are the states reached after $1, 2, 3, \ldots$ loop iterations, respectively. One iteration decreases the ranking by at least ϵ which guarantees eventually hitting 0 (or dropping below).

Then R is a ranking function as every iteration of the loop body decreases x by $\epsilon = 1$ and the loop body will not be executed again once $x \leq 0$.

For probabilistic programs, this reasoning fails. The loop body of

$$\texttt{while}(x > 0)\{ \quad \{x := x - 1\}\,[1/2]\,\{\texttt{skip}\} \quad \},$$

for instance, is not guaranteed to decrease x due to the possibility of executing `skip` instead of $x := x - 1$. However, every iteration of the loop body decreases x by $1/2$ *in expectation* and thus x is *ranking in expectation*.

Existence of a function that is ranking in expectation indeed suffices to prove positive almost–sure termination. Translated into our weakest preexpectation setting, we have the following theorem:

Theorem 6.3 (PAST from Ranking Superinvariants [CS13; FH15]): *Let* $\texttt{while}(\varphi)\{C\}$ *be a loop where the loop body* C *itself terminates universally* certainly.[3] *Furthermore, let* $I \in \mathbb{E}$ *be a ranking superinvariant*[4]

3 E.g. let C be loop–free.

4 Ranking superinvariants correspond to *ranking super martingales* in the terminology of [CS13].

of $\texttt{while}(\varphi)\{C\}$ *with respect to postexpectation 0, i.e.* $I \ll \infty$ *and there exist constants* ϵ *and* K *with* $0 < \epsilon < K$*, such that*

$$[\neg\varphi]\cdot I \;\leq\; K \quad \text{and} \quad [\varphi]\cdot K \;\ll\; [\varphi]\cdot I + [\neg\varphi]$$
$$\text{and} \quad \Phi_0(I) \;\leq\; [\varphi]\cdot(I-\epsilon)\,,$$

where Φ_0 *is the* awp*–characteristic function of* $\texttt{while}(\varphi)\{C\}$ *with respect to postexpectation* 0*.*

Then $\texttt{while}(\varphi)\{C\}$ *terminates universally positively almost–surely.*

Notice that we use awp because we want to *guarantee* that I is decreased in expectation by at least ϵ through one iteration of the loop body.

The two extra conditions involving the constant K are a technical necessity in order to avoid the need for our ranking superinvariants to map to negative values (as ranking functions do): The loop body should decrease I by ϵ in expectation, so I can drop by at most ϵ into the negative. We mitigate this by pulling everything up by $K > \epsilon$ and let a drop below K (instead of 0) indicate termination.

Theorem 6.3 is basically a reformulation of [CS13, Theorem 4] or [FH15, Theorem 5.6] but translated into our weakest preexpectation setting. It is also very similar and basically equivalent to Theorem 3 of [Kam+16], which we present in Chapter 7. The main difference is that [Kam+16] needs less preconditions and always uses $\epsilon = 1$, while still being complete.

Example 6.4 (PAST from Ranking Superinvariants):
Reconsider the program

$$\begin{array}{l}\texttt{while}(x>0)\{\\ \quad \{x := x-1\}\,[1/2]\,\{\texttt{skip}\}\\ \}\,,\end{array}$$

for which the awp–characteristic function with respect to 0 is given by

$$\begin{aligned}\Phi_0(X) \;&=\; [x\leq 0]\cdot 0 \;+\; [x>0]\cdot \mathsf{wp}\,[\![\{x := x-1\}\,[1/2]\,\{\texttt{skip}\}]\!]\,(X)\\ &=\; [x>0]\cdot\frac{1}{2}\Big(X[x/x-1]+X\Big)\end{aligned}$$

Then $I = [x \geq -1]\cdot x + 1$ is a ranking superinvariant with $K = 1$ and $\epsilon = 1/2$. To see that I is indeed a ranking superinvariant, consider

$$[x\leq 0]\cdot I \;=\; [x\leq 0]\cdot\Big([x\geq -1]\cdot x+1\Big) \;\leq\; 1 \;=\; K$$

and

$$\begin{aligned}
[x>0]\cdot K &= [x>0]\cdot 1\\
&\ll [x>0]\cdot(x+1)+[x\leq 0]\\
&\ll [x>0]\cdot\Big([x\geq -1]\cdot x+1\Big)+[x\leq 0]\\
&= [x>0]\cdot I+[x\leq 0]
\end{aligned}$$

and

$$\begin{aligned}
\Phi_0(I) &= [x>0]\cdot\frac{1}{2}\Big(I\,[x/x-1]+I\Big)\\
&= [x>0]\cdot\frac{1}{2}\Big([x-1\geq -1]\cdot(x-1)+1+[x\geq -1]\cdot x+1\Big)\\
&= [x>0]\cdot\frac{1}{2}\Big([x\geq 0]\cdot(x-1)+1+[x\geq -1]\cdot x+1\Big)\\
&= [x>0]\cdot\frac{1}{2}\Big(x-1+1+x+1\Big)\\
&= [x>0]\cdot\frac{1}{2}\Big(x-1+1+x+1\Big)\\
&= [x>0]\cdot\Big(x+1-\frac{1}{2}\Big)\\
&= [x>0]\cdot\Big([x>0]\cdot x+1-\frac{1}{2}\Big)\\
&= [x>0]\cdot(I-\epsilon)\,.
\end{aligned}$$

This proves that I is a ranking superinvariant which by Theorem 6.3 proves universal positive almost–sure termination of the loop under consideration.

A technically less involved, yet complete, method (no need for choosing K or ϵ) for proving a finite expected runtime (and thereby positive almost–sure termination) is presented in Chapter 7.

6.2 ALMOST–SURE TERMINATION

As mentioned earlier, proving almost–sure termination of null almost–surely terminating programs (i.e. programs that terminate with probability 1 but with infinite expected time until termination, cf. Definition 6.2), appears notoriously difficult, because it requires proving a lower bound on a least fixed point, namely that 1 is a (non–strict) lower bound on the termination probability. The lack of a finite upper bound on the expected runtime renders the coinductive proof technique of ranking supermartingales (Theorem 6.3) unavailable.

A new proof rule that does allow for proving almost–sure termination, even of null almost–surely terminating loops, is presented in Section 6.2.3. Although this method will clearly appear to be more involved than the rank-

ing supermartingale approach of Theorem 6.3, it often allows for relatively easy (sometimes even surprisingly easy) proofs of almost–sure termination.

Before we present the new proof rule, we recap some earlier rules by McIver & Morgan for proving almost–sure termination. These theorems, in particular a zero–one law for probabilistic termination, will form the bedrock on which the new proof rule is built.

6.2.1 *The Zero–one Law*

Zero–one laws in probability theory typically state that under certain conditions certain events occur either with probability 0 or 1, but this probability cannot lie properly in–between. Notable examples include the Borel–Cantelli Lemma [Bor09; Can17; Wikb], Kolmogorov's zero–one law [Wikh], or the Hewitt–Savage zero–one law [HS55; Wikf]. The law considered here is due to McIver & Morgan and is a zero–one law on the termination probability of probabilistic while loops. It reads as follows:

> **Theorem 6.5 (Zero–One Law of Probabilistic Termination[5]):**
> *Let I be a predicate such that $[I]$ is a* wp*–subinvariant of* $\texttt{while}(\varphi)\{C\}$ *with respect to postexpectation $[I]$. Furthermore, let $\epsilon > 0$ be a fixed constant such that*
>
> $$\epsilon \cdot [I] \;\preceq\; \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(1)\;.$$
>
> *Then*
>
> $$[I] \;\preceq\; \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,\Big([\neg\varphi \wedge I]\Big)\;.$$

Proof. We invoke Theorem 5.12 for obtaining lower bounds on preexpectations: For that, let $I' = [I]$ and $f = [\neg\varphi \wedge I]$. Then $[\neg\varphi]\cdot f + [\varphi]\cdot I'$ is a wp–subinvariant of $\texttt{while}(\varphi)\{C\}$ with respect to postexpectation f, since

$$\begin{aligned}
[I] \;&\preceq\; \Phi_0([I]) && \text{(by } [I] \text{ being a subinvariant with respect to 0)}\\
&=\; [\neg\varphi]\cdot 0 + [\varphi]\cdot \mathsf{wp}\,[\![C]\!]\,([I])\\
&\preceq\; [\neg\varphi]\cdot f + [\varphi]\cdot \mathsf{wp}\,[\![C]\!]\,([I]) && \text{(by } 0 \preceq f\text{)}\\
&=\; \Phi_f([I]) && \text{(by definition of } \Phi_f\text{)}
\end{aligned}$$

By assumption $\epsilon\cdot[I] \preceq \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(1)$, we have that

$$\epsilon\cdot\Big([\neg\varphi]\cdot f + [\varphi]\cdot I'\Big) \;=\; \epsilon\cdot[I] \;\preceq\; \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(1)$$

holds, so all preconditions of Theorem 5.12 c. are met and the theorem yields

$$[I] \;=\; [\neg\varphi]\cdot f + [\varphi]\cdot I'$$

5 This theorem subsumes [MM05, pp. 53 and 54] and [Hur03, Theorem 41].

$$\preceq \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(f) \qquad \text{(by Theorem 5.12 c.)}$$
$$= \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,([\neg\varphi \wedge I])\,. \qquad \boxed{\text{Q.E.D.}}$$

While the zero–one law of Theorem 6.5 allows for proving almost–sure termination relative to an invariant, we obtain as a special case (choose true as invariant) the following corollary for universal almost–sure termination:

COROLLARY 6.6:
Let $C \in \mathsf{pGCL}$ terminate universally almost–surely. Furthermore, let $\epsilon > 0$ be a fixed constant such that

$$\epsilon \preceq \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(1)\,.$$

Then $\texttt{while}(\varphi)\{C\}$ terminates universally almost–surely, i.e.

$$\mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(1) = 1\,.$$

6.2.2 *An Old Rule*

Building on the zero–one law for probabilistic termination (Theorem 6.5), McIver & Morgan have formulated a more practically oriented proof rule for proving almost–sure termination. Whereas the zero–one law needed as a precondition a lower bound ϵ on the *overall termination probability of a loop* (which is potentially as difficult to establish as almost–sure termination itself), the following rule makes use of a ranking function that is decreased with at least some constant probability by *one iteration* of the loop body. This fact is potentially much easier to check.

THEOREM 6.7 (AST from Bounded Integer Variants [MM05][6]):
Let I be a predicate such that $[I]$ is a wp*–subinvariant of $\texttt{while}(\psi)\{C\}$ with respect to postexpectation $[I]$. Furthermore, let $Z\colon \Sigma \to \mathbb{Z}$ such that*

A. *there exist constants $L, H \in \mathbb{Z}$ such that*

$$[\psi \wedge I] \preceq [L \leq Z \leq H], \qquad \text{and}$$

B. *there exists a constant $\epsilon \in (0, 1]$ such that*

$$\epsilon \cdot [\psi \wedge I] \preceq \lambda\sigma.\ \mathsf{wp}\,[\![C]\!]\,\Big([Z < Z(\sigma)]\Big)(\sigma)\,.$$

Then the loop $\texttt{while}(\psi)\{C\}$ terminates almost–surely from any initial state satisfying the invariant I, i.e.

$$[I] \preceq \mathsf{wp}\,[\![\texttt{while}(\psi)\{C\}]\!]\,(1)\,.$$

6 This theorem combines Lemma 2.7.1 on p. 55 and Lemma 7.5.1 on p. 191 in [MM05].

Proof. The full proof of Theorem 6.7 can be found in [MM05, p. 191 *et seq.*, proof of Lemma 7.5.1]. The key idea to exploit the zero–one law of probabilistic termination (Theorem 6.5). In order to understand the importance of that law for this rule, we rephrase here McIver & Morgan's sketch of the proof of Theorem 6.7 [MM05, p. 55 et seq., proof of Lemma 2.7.1]:

Recall that the variant Z is integer–valued, bounded from below by L, and bounded from above by H. Furthermore, the probability to *strictly* decrease Z (by at least 1, as Z is integer–valued) through one iteration of the loop body is at least ϵ from any starting state. Then after at most $H - L$ loop iterations, the value of the variant Z will have dropped to L or below with probability at least ϵ^{H-L}. This in turn implies falsification of the loop guard or violation of the invariant. But since satisfaction of the invariant is invariant under guarded iteration of the loop, violation of the invariant can be ruled out and so the loop terminates by falsification of the loop guard with probability at least ϵ^{H-L} from any initial state.

Since ϵ^{H-L} is a constant strictly larger than 0, we can appeal to the zero–one law of probabilistic termination (Theorem 6.5) which asserts that if the loop terminates from any state with at least some (universally) constant non–zero probability, then the loop terminates in fact almost–surely. We can thus conclude that the loop terminates not only with probability at least ϵ^{H-L} but in fact almost–surely from any initial state satisfying invariant I. Q.E.D.

While Theorem 6.7 allows for proving almost–sure termination by relatively simple means, its disadvantages are also evident: Integer–valuedness together with boundedness of the variant function clearly restricts its use cases. For instance, in order to prove almost–sure termination of a symmetric 1–dimensional random walk using Theorem 6.7, a substantial number of ad–hoc arguments are necessary and the termination proof becomes somewhat involved [MM05, Section 3.3]. Nevertheless, Theorem 6.7 can arguably be seen as an important precursor to the new rule which we present next.

6.2.3 *A New Rule*

I like your result.
Let's make it a joint paper
and I'll write the next one.

— Stefan Bergmann

Reconsider the symmetric 1–dimensional random walk modeled by the loop

$$\texttt{while}(x > 0)\{\ \{x := x - 1\}\,[1/2]\,\{x := x + 1\}\ \}.$$

We can easily convince ourselves that x is indeed a superinvariant of the loop, although it is *not ranking*.[7] In fact, the expected value of x is precisely x itself — in expectation the particle does not move. However, we can also easily convince ourselves that the probability that the particle moves a distance of 1 closer to 0 is $1/2$. As we will see, witnessing this fact indeed already suffices in order to prove almost–sure termination by our new proof rule.

Just like the proof rule of Theorem 6.7, the new rule involves a variant function which decreases by some amount with some probability through one iteration of the loop body. In contrast to Theorem 6.7, however, the variant function need neither be bounded nor integer–valued. In addition and also in contrast to Theorem 6.7, the minimum amount and probability of the variant's decrease need not be lower–bounded by some constants (for Theorem 6.7, those constants were 1 and some $\epsilon > 0$).

Before we state the new proof rule and give a detailed proof, I would sincerely like to acknowledge that the core idea of the new rule is entirely due to Annabelle McIver and Carroll Morgan, see [MM16] for their early sketch. My contribution was (a) to formalize the proof rule in terms of weakest pre-expectations and (b) give a rigorous soundness proof of the new proof rule. The version of the proof rule provided here differs slightly from the published version ([McI+18, Theorem 4.1]) both in its formulation as well as in its proof as I personally find the presentation at hand more natural.

THEOREM 6.8 (AST from Progressing Variants [McI+18]):
Let I be a predicate and moreover let

- $V : \Sigma \to \mathbb{R}_{\geq 0}$ *(for **v**ariant),*
- $p : \mathbb{R}_{\geq 0} \to (0, 1]$ *(for **p**robability) be antitone*[8]*,*
- $d : \mathbb{R}_{\geq 0} \to \mathbb{R}_{>0}$ *(for **d**ecrease) be antitone.*

Suppose further that the following conditions hold:

A. $[I]$ *is a* wp*–subinvariant of* `while(φ){C}` *with respect to* $[I]$*, i.e.*

$$[I] \;\leq\; {}^{\mathsf{wp}}\Phi_{[I]}\big([I]\big) \quad = \; [\neg\varphi]\cdot[I] \;+\; [\varphi]\cdot \mathsf{wp}\,[\![C]\!]\,([I])\,.$$

B. $V = 0$ *indicates termination, i.e.*

$$[\neg\varphi] \;=\; [V = 0]\,.$$

7 As mentioned earlier, this loop terminates *null* almost–surely. Thus, there cannot exist a ranking supermartingale for this loop as this would by Theorem 6.3 imply positive almost–sure termination of this loop.

8 Antitonicity is the dual notion to monotonicity [Wiki]: A function f is called antitone iff

$$a \leq b \quad \text{implies} \quad f(a) \geq f(b)\,.$$

c. *V is a* awp–*superinvariant of* `while`$(\varphi)\{C\}$ *with respect to* V*, i.e.*

$$V \geq \underset{\langle \varphi, C \rangle}{\mathsf{awp}} \Phi_V(V) \quad = [\neg\varphi] \cdot V + [\varphi] \cdot \mathsf{wp}\,[\![C]\!]\,(V)\,.$$

d. *V satisfies a progress condition, namely*[9]

$$p \circ V \cdot [\varphi] \cdot [I] \preceq \lambda\sigma.\ \mathsf{wp}\,[\![C]\!] \left(\left[V \leq V(\sigma) - d\Big(V(\sigma)\Big) \right] \right)(\sigma)\,.$$

Then the loop `while`$(\varphi)\{C\}$ *terminates almost–surely from any initial state satisfying the invariant I, i.e.*

$$[I] \preceq \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(1)\,.$$

The intuitive mechanics of the new proof rule is illustrated in Figure 6.2. Amount and probability of the variant V's decrease are neither fixed nor bounded bounded by the progress condition, but instead adjustable by *antitone* functions d and p, which take as inputs *not* the current state, but rather *the value of the variant in the current state*. The progress condition now ensures that if the current state is σ and the loop body will be iterated once more, then the probability to decrease V by at least $d(V(\sigma))$ is at least $(p \circ V)(\sigma) = p(V(\sigma))$. For any successor state in which the value of V has *decreased*, the amount and probability of decrease for the *next iteration* will both have *increased* due to antitonicity of p and d. In a nutshell and to put it very simply:

> The closer the loop comes to termination ($V = 0$),
> the more V is decreased by iteration of the loop body (antitone d)
> and the more likely becomes this decrease (antitone p).

Antitonicity of p and d rule out a sort of *Zeno behavior* where the variant does indeed strictly decrease but by an ever decreasing amount. This would allow for V to „converge" to a value strictly larger than 0, making it less and less likely to terminate and thus causing the loop to diverge.

Example 6.9 (Almost–sure Termination of the Random Walk):
Consider the symmetric 1–dimensional random walk, modeled by

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

For reasons of readability, let us suppose that x is of type $\mathbb{N}$. We choose

9 $p \circ V$ denotes functional composition (read: p after V), i.e. $p \circ V = \lambda\sigma.\ p\Big(V(\sigma)\Big)$, and binds stronger than multiplication.

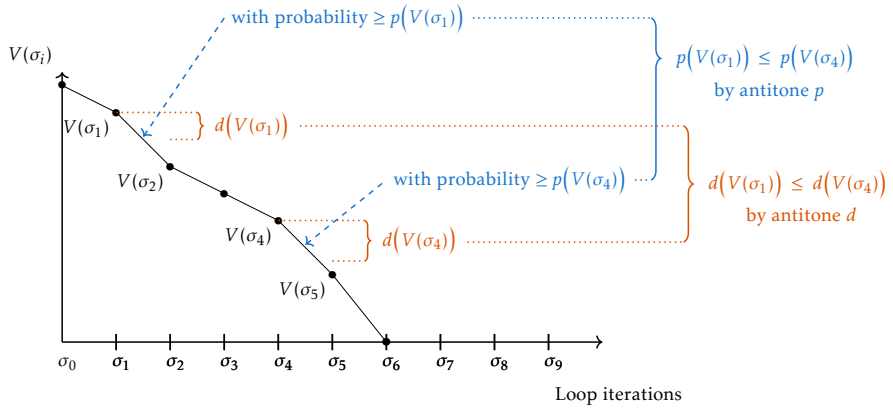


Figure 6.2: Evolution of the values of a variant V over the iterations of a probabilistic loop. σ_0 is the initial state and $\sigma_1, \sigma_2, \sigma_3, \ldots$ are states reached with non–zero probability after 1, 2, 3, … iterations, respectively. Iteration decreases the variant by an ever increasing (or constant) amount ($d(V(\sigma_i))$) with with ever increasing (or constant) probability ($p(V(\sigma_i))$).

$$I = \mathsf{true}, \qquad V = x, \qquad p = \frac{1}{2}, \qquad \text{and} \qquad d = 1\,.$$

as witnesses of almost–sure termination. p and d are constant and thus obviously antitone. true is a wp–subinvariant of any loop that terminates almost–surely. This is especially the case when the loop body itself is loop–free. $V = 0$ indicates termination since $V = 0$ iff $x \leq 0$ (because x is of type $\mathbb{N}$).

Next, we provide a detailed check that x is an awp–supermartingale:

$$\begin{aligned}
& {}^{\mathsf{awp}}_{\langle x>0, body\rangle}\Phi_x \preceq x \\
\text{iff}\quad & [x \leq 0]\cdot x + [x > 0]\cdot \mathsf{wp}\,[\![body]\!]\,(x) \preceq x \\
\text{iff}\quad & [x \leq 0]\cdot x + [x > 0]\cdot \frac{1}{2}\cdot(x-1+x+1) \preceq x \\
\text{iff}\quad & [x \leq 0]\cdot x + [x > 0]\cdot x \preceq x \\
\text{iff}\quad & x \preceq x
\end{aligned}$$

Finally, we check that the progress condition is satisfied:

$$\begin{aligned}
& p \circ V \cdot [\varphi]\cdot[I] \preceq \lambda\sigma.\ \mathsf{wp}\,[\![C]\!]\left(\left[V \leq V(\sigma) - d\big(V(\sigma)\big)\right]\right)(\sigma) \\
\text{iff}\quad & \frac{1}{2} \circ x \cdot [x > 0]\cdot[\mathsf{true}]
\end{aligned}$$

$$\leq \lambda\sigma.\ \mathsf{wp}\,[\![body]\!]\left(\left[x \leq x(\sigma) - (\lambda v.\ 1)\big(x(\sigma)\big)\right]\right)(\sigma)$$

$$\text{iff}\quad \frac{1}{2}[x>0] \leq \lambda\sigma.\ \mathsf{wp}\,[\![body]\!]\left([x \leq x(\sigma)-1]\right)(\sigma)$$

$$\text{iff}\quad \frac{1}{2}[x>0] \leq \lambda\sigma.\ \frac{1}{2}\cdot\Big([x-1 \leq x(\sigma)-1] + [x+1 \leq x(\sigma)-1]\Big)$$

$$\text{iff}\quad \frac{1}{2}[x>0] \leq \frac{1}{2}\cdot\Big([x-1 \leq x-1] + [x+1 \leq x-1]\Big)$$

$$\text{iff}\quad \frac{1}{2}[x>0] \leq \frac{1}{2}\cdot\Big(1+0\Big)$$

$$\text{iff}\quad \frac{1}{2}[x>0] \leq \frac{1}{2}$$

The last line is obviously true, thus concluding our proof, and thus we have proven almost–sure termination.

Notice that for the 1–dimensional symmetric random walk our termination witnesses were very simple functions, namely *constant* functions, and that checking the supermartingale property and the progress condition was quite simple. In particular, we did not have to reason ourselves about any limit whatsoever. *This is different for the book proof of almost–sure termination of the random walk* (see e.g. [Dur10, Theorem 4.2.3, p. 163]): There, one finds a formula for the termination probability and then proves ad–hoc that the *limit* is in fact 1.

Now that we have some intuition on the mechanics and we have seen an example on how to use the new rule in practice, we give a rigorous proof. In this proof, we will show precisely why we need a superinvariant and how the progress condition is used.

Proof (Theorem 6.8). Because the proof of Theorem 6.8 is somewhat involved, we will first give an outline of our proof strategy:

1. We fix an arbitrary constant $h \in \mathbb{R}_{>0}$ and prove that the modified loop $\texttt{while}(0 < V \leq h)\{C\}$ terminates almost–surely from any state satisfying I by exploiting Theorem 6.7.

   Notice that only the loop guard is changed from φ to $0 < V \leq h$. We have merely introduced a *cap* h on V and if V exceeds h, we force termination. Condition B. states that the original loop $\texttt{while}(\varphi)\{C\}$ terminates when V hits 0. Thus, if V hits 0, the modified loop terminates for the same reason as the original loop $\texttt{while}(\varphi)\{C\}$ would have terminated. Only if V exceeds h, then the modified loop *prematurely* terminates, whereas the original loop would still continue to be executed.

2. We prove that the supermartingale property on V implies that the modified loop $\texttt{while}(0 < V \leq h)\{C\}$ does not increase V in expectation, i.e. the expected value of V after execution of the modified loop on initial

state σ is bounded by $V(\sigma)$. Intuitively, the consequence of this is that the modified loop is more likely to terminate because of V hitting 0 than because of V exceeding h.

3. Using 1., we prove that the *non*termination probability of the original loop is bounded from above by the probability that execution of the modified loop terminates because of V exceeding h. By Markov's Inequality, the latter probability is bounded from above by the expected value of V divided by h. By 2., we then get that this fraction is itself bounded by V/h. Finally, we take the limit $h \to \infty$ to conclude that the nontermination probability of the original loop is bounded from above by 0 and thus the original loop terminates almost–surely.

Let us now conduct the proof. Let $h \in \mathbb{R}_{>0}$ be arbitrary but fixed, and let

$$\texttt{while}(0 < V \leq h)\{C\}$$

be the *modified loop*. Then we perform the proof steps we described above.

1. ***The modified loop*** $\texttt{while}(0 < V \leq h)\{C\}$ ***term. almost–surely.*** We prove that the modified loop with guard

$$\psi \;=\; (0 < V \leq h)$$

terminates almost–surely by applying Theorem 6.7 to that loop. Let us first prove that $[I]$ is a wp–subinvariant of $\texttt{while}(\psi)\{C\}$ with respect to $[I]$. We start our reasoning from condition A.:

$$\begin{aligned}
& [I] \text{ is a wp–subinvariant of } \texttt{while}(\varphi)\{C\} \text{ w.r.t. } [I] \\
\text{iff}\quad & [I] \;\preceq\; {}^{\mathsf{wp}}_{\langle \varphi, C\rangle}\Phi_{[I]}([I]) \\
& \qquad \text{(by definition of wp–subinvariance, Definition 5.1 B.)} \\
\text{iff}\quad & [I] \;\preceq\; [\neg\varphi]\cdot[I] + [\varphi]\cdot \mathsf{wp}\,[\![C]\!]\,([I]) \\
& \qquad \big(\text{by definition of } {}^{\mathsf{wp}}_{\langle \varphi, C\rangle}\Phi_{[I]}, \text{ Definition 4.5 E.}\big) \\
\text{implies}\quad & [\varphi]\cdot[I] \;\preceq\; [\varphi]\cdot \mathsf{wp}\,[\![C]\!]\,([I]) \qquad \text{(multiply both sides by } [\varphi]) \\
\text{iff}\quad & [V > 0]\cdot[I] \;\preceq\; [V > 0]\cdot \mathsf{wp}\,[\![C]\!]\,([I]) \\
& \qquad \text{(by } V = 0 \text{ indicating termination, condition B.)} \\
\text{implies}\quad & [V > 0]\cdot[V \leq h]\cdot[I] \;\preceq\; [V > 0]\cdot[V \leq h]\cdot \mathsf{wp}\,[\![C]\!]\,([I]) \\
\text{iff}\quad & [0 < V \leq h]\cdot[I] \;\preceq\; [0 < V \leq h]\cdot \mathsf{wp}\,[\![C]\!]\,([I]) \\
\text{iff}\quad & [\psi]\cdot[I] \;\preceq\; [\psi]\cdot \mathsf{wp}\,[\![C]\!]\,([I]) \qquad \text{(by definition of } \psi) \\
\text{iff}\quad & [\neg\psi]\cdot[I] + [\psi]\cdot[I] \;\preceq\; [\neg\psi]\cdot[I] + [\psi]\cdot \mathsf{wp}\,[\![C]\!]\,([I]) \\
\text{iff}\quad & [I] \;\preceq\; [\neg\psi]\cdot[I] + [\psi]\cdot \mathsf{wp}\,[\![C]\!]\,([I]) \\
\text{iff}\quad & [I] \;\preceq\; {}^{\mathsf{wp}}_{\langle \psi, C\rangle}\Phi_{[I]}([I]) \\
& \qquad \big(\text{by definition of } {}^{\mathsf{wp}}_{\langle \psi, C\rangle}\Phi_{[I]}, \text{ Definition 4.5 E.}\big)
\end{aligned}$$

$$\text{iff} \quad [I] \text{ is a wp–subinvariant of } \texttt{while}(\psi)\{C\} \text{ w.r.t. } [I]$$
(by definition of wp–subinvariance, Definition 5.1 B.)

Next, we have to choose for Theorem 6.7 a bounded integer–valued variant Z. Notice that we have with h an upper bound for the value of V. By antitonicity of p and d, we have with $p(h)$ and $d(h)$ *lower bounds* on the probability and the amount of V's decrease. We can thus use as integer variant Z the number of times that we can subtract $d(h)$ from V until we hit 0. The probability to decrease Z by at least 1 is then at least $p(h)$ — just as the probability to decrease V by at least $d(h)$. Formally, we discretize V as follows:

$$Z \;=\; \left\lceil \frac{V}{d(h)} \right\rceil$$

As bounds for Z we choose the lower bound $L = 0$ and upper bound $H = \lceil h/d(h) \rceil$. To see that Z is appropriately bounded (i.e. in the sense of Theorem 6.7), consider that $0 < V \leq h$ implies $0 \geq \lceil V/d(h) \rceil \leq \lceil h/d(h) \rceil$ and thus

$$[0 \leq V \leq h] \;\preceq\; \left[0 \leq \left\lceil \frac{V}{d(h)} \right\rceil \leq \left\lceil \frac{h}{d(h)} \right\rceil\right]$$
$$\text{implies} \quad [0 \leq V \leq h \wedge I] \;\preceq\; \left[0 \leq \left\lceil \frac{V}{d(h)} \right\rceil \leq \left\lceil \frac{h}{d(h)} \right\rceil\right]$$
$$\text{implies} \quad [0 < V \leq h \wedge I] \;\preceq\; \left[0 \leq \left\lceil \frac{V}{d(h)} \right\rceil \leq \left\lceil \frac{h}{d(h)} \right\rceil\right]$$
$$\text{iff} \quad [\psi \wedge I] \;\preceq\; [L \leq Z \leq H] \qquad \text{(by definition of } \psi, Z, L, \text{ and } H)$$

The last precondition we need in order to be able to apply Theorem 6.7 is that there exists an $\epsilon > 0$ such that

$$\epsilon \cdot [\psi \wedge I] \;\preceq\; \lambda\sigma.\ \mathsf{wp}\,[\![C]\!]\,([Z < Z(\sigma)])\ .$$

When choosing $\epsilon = p(h)$, we have by the progress condition D.:

$$p\big(V(\sigma)\big) \cdot [\varphi \wedge I] \;\preceq\; \lambda\sigma.\ \mathsf{wp}\,[\![C]\!]\,\Big(\big[V \leq V(\sigma) - d\big(V(\sigma)\big)\big]\Big)$$
$$\text{iff} \quad p\big(V(\sigma)\big) \cdot [V > 0 \wedge I] \;\preceq\; \lambda\sigma.\ \mathsf{wp}\,[\![C]\!]\,\Big(\big[V \leq V(\sigma) - d\big(V(\sigma)\big)\big]\Big)$$
(by $V = 0$ indicating termination, condition B.)
$$\text{implies} \quad p\big(V(\sigma)\big) \cdot [0 < V \leq h \wedge I]$$
$$\preceq\; \lambda\sigma.\ \mathsf{wp}\,[\![C]\!]\,\Big(\big[V \leq V(\sigma) - d\big(V(\sigma)\big)\big]\Big)$$
$$\text{iff} \quad p\big(V(\sigma)\big) \cdot [\psi \wedge I] \;\preceq\; \lambda\sigma.\ \mathsf{wp}\,[\![C]\!]\,\Big(\big[V \leq V(\sigma) - d\big(V(\sigma)\big)\big]\Big)$$
(by definition of ψ)

Since for all states σ with $V(\sigma) \leq h$ we have $p(h) \leq p(V(\sigma))$, we obtain:

$$\text{implies} \quad p(h) \cdot [\psi \wedge I] \;\preceq\; \lambda\sigma.\ \mathsf{wp}\,[\![C]\!]\,\Big(\big[V \leq V(\sigma) - d\big(V(\sigma)\big)\big]\Big)$$

$$\text{iff}\quad p(h)\cdot[\psi\wedge I]\ \preceq\ \lambda\sigma.\ \mathsf{wp}\,[\![C]\!]\left(\left[\frac{V}{d(h)}\leq\frac{V(\sigma)}{d(h)}-\frac{d(V(\sigma))}{d(h)}\right]\right)$$

Since ${}^{V}\!/_{d(h)}\leq {}^{V(\sigma)}\!/_{d(h)}-{}^{d(V(\sigma))}\!/_{d(h)}$ implies $\lceil {}^{V}\!/_{d(h)}\rceil\leq\lceil {}^{V(\sigma)}\!/_{d(h)}-{}^{d(V(\sigma))}\!/_{d(h)}\rceil$, we obtain by monotonicity of wp, Theorem 4.16:

$$\text{implies}\quad p(h)\cdot[\psi\wedge I]\ \preceq\ \lambda\sigma.\ \mathsf{wp}\,[\![C]\!]\left(\left[\left\lceil\frac{V}{d(h)}\right\rceil\leq\left\lceil\frac{V(\sigma)}{d(h)}-\frac{d(V(\sigma))}{d(h)}\right\rceil\right]\right)$$

For all states σ with $V(\sigma)\leq h$ we have that $d(h)\leq d(V(\sigma))$ by antitonicity of d. Thus $\lceil {}^{V}\!/_{d(h)}\rceil\leq\lceil {}^{V(\sigma)}\!/_{d(h)}-{}^{d(V(\sigma))}\!/_{d(h)}\rceil$ implies $\lceil {}^{V}\!/_{d(h)}\rceil\leq\lceil {}^{V(\sigma)}\!/_{d(h)}-1\rceil$ and we get by monotonicity of wp, Theorem 4.16:

$$\text{implies}\quad p(h)\cdot[\psi\wedge I]\ \preceq\ \lambda\sigma.\ \mathsf{wp}\,[\![C]\!]\left(\left[\left\lceil\frac{V}{d(h)}\right\rceil\leq\left\lceil\frac{V(\sigma)}{d(h)}-1\right\rceil\right]\right)$$

$$p(h)\cdot[\psi\wedge I]\ \preceq\ \lambda\sigma.\ \mathsf{wp}\,[\![C]\!]\left(\left[\left\lceil\frac{V}{d(h)}\right\rceil<\left\lceil\frac{V(\sigma)}{d(h)}\right\rceil\right]\right)$$

(by $\lceil {}^{V(\sigma)}\!/_{d(h)}-1\rceil<\lceil {}^{V(\sigma)}\!/_{d(h)}\rceil$)

$$\text{iff}\quad p(h)\cdot[\psi\wedge I]\ \preceq\ \lambda\sigma.\ \mathsf{wp}\,[\![C]\!]\,([Z<Z(\sigma)])\quad\text{(by definition of } Z)$$

We conclude by Theorem 6.7 that $\texttt{while}(0<V\leq h)\{C\}$ terminates almost–surely from any state satisfying I, i.e.

$$[I]\ \preceq\ \mathsf{wp}\,[\![\texttt{while}(0<V\leq h)\{C\}]\!]\,(1)\ . \tag{†}$$

2. ***The modified loop*** $\texttt{while}(0<V\leq h)\{C\}$ ***does not increase*** V ***in expectation.*** Since, by condition C., V is a awp–superinvariant of $\texttt{while}(\varphi)\{C\}$ with respect to postexpectation V, we know that

$$\begin{aligned}
&{}^{\mathsf{awp}}_{\langle\varphi,C\rangle}\Phi_V(V)\ \preceq\ V\\
\text{iff}\quad&[\neg\varphi]\cdot V\ +\ [\varphi]\cdot\mathsf{awp}\,[\![C]\!]\,(V)\ \preceq\ V\\
&\qquad\qquad\big(\text{by definition of } {}^{\mathsf{awp}}_{\langle\varphi,C\rangle}\Phi_V\text{, Definition 4.5 E.}\big)\\
\text{iff}\quad&[\varphi]\cdot\mathsf{awp}\,[\![C]\!]\,(V)\ \preceq\ V\\
\text{iff}\quad&[0<V]\cdot\mathsf{awp}\,[\![C]\!]\,(V)\ \preceq\ V\\
&\qquad\qquad\text{(by } V=0 \text{ indicating termination, condition B.)}\\
\text{implies}\quad&[0<V\leq h]\cdot\mathsf{awp}\,[\![C]\!]\,(V)\ \preceq\ V\\
\text{implies}\quad&\big([0=V]+[h<V]\big)\cdot V\ +\ [0<V\leq h]\cdot\mathsf{awp}\,[\![C]\!]\,(V)\ \preceq\ V\\
\text{iff}\quad&{}^{\mathsf{awp}}_{\langle 0<V\leq h,C\rangle}\Phi_V\ \preceq\ V\\
&\qquad\qquad\big(\text{by definition of } {}^{\mathsf{awp}}_{\langle 0<V\leq h,C\rangle}\Phi_V\text{, Definition 4.5 E.}\big)\\
\text{implies}\quad&\mathsf{awp}\,[\![\texttt{while}(0<V\leq h)\{C\}]\!]\,(V)\ \preceq\ V\qquad\qquad(\ddagger)\\
&\qquad\qquad\text{(by induction rule, Theorem 5.4)}
\end{aligned}$$

Thus we have concluded that the modified loop $\texttt{while}(0<V\leq h)\{C\}$ does not increase V in expectation.

3. ***The original loop*** $\texttt{while}(\varphi)\{C\}$ ***terminates almost–surely.*** We first prove that the original loop $\texttt{while}(0<V)\{C\}$ is more likely to terminate with $V=0$ than the modified loop $\texttt{while}(0<V\leq h)\{C\}$. This is intuitively clear, because whenever the modified loop exceeds h and thus terminates with $V\neq 0$, the original loop does not terminate and has still a chance of „returning“ and dropping down to 0. For a rigorous proof, consider the following for all $X\in\mathbb{E}$:

$$
\begin{aligned}
& [0<V\leq h] \;\leq\; [0<V] \\
\text{implies}\quad & [V=0] \;+\; [0<V\leq h]\cdot \mathsf{awp}\,[\![C]\!]\,(X) \\
& \qquad \leq\; [V=0] \;+\; [0<V]\cdot \mathsf{awp}\,[\![C]\!]\,(X) \\
\text{iff}\quad & \big([0=V]+[h<V]\big)\cdot[V=0] \;+\; [0<V\leq h]\cdot \mathsf{awp}\,[\![C]\!]\,(X) \\
& \qquad \leq\; [0=V]\cdot[V=0] \;+\; [0<V]\cdot \mathsf{awp}\,[\![C]\!]\,(X) \\
\text{iff}\quad & {}^{\mathsf{wp}}\Phi_{[V=0]}(X) \;\leq\; {}^{\mathsf{wp}}\Phi_{[V=0]}(X) \\
& \big(\text{by definition of } {}^{\mathsf{wp}}\Phi_{[V=0]} \text{ and } {}^{\mathsf{wp}}\Phi_{[V=0]}, \text{ Definition 4.5 E.}\big) \\
\text{implies}\quad & \mathsf{wp}\,[\![\texttt{while}(0<V\leq h)\{C\}]\!]\,([V=0]) \qquad (\ddagger\ddagger) \\
& \qquad \leq\; \mathsf{wp}\,[\![\texttt{while}(0<V)\{C\}]\!]\,([V=0])
\end{aligned}
$$

We are now in a position to gradually develop a lower bound on the termination probability of the original loop. For that, consider the following:

$$
\begin{aligned}
& \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(1) \\
&= \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,([\neg\varphi]) \\
&\qquad\qquad \text{(by postexpectation strengthening, Corollary 4.6)} \\
&= \mathsf{wp}\,[\![\texttt{while}(0<V)\{C\}]\!]\,([V=0]) \\
&\qquad\qquad \text{(by } V=0 \text{ indicating termination, condition B.)}
\end{aligned}
$$

Since the original loop $\texttt{while}(0<V)\{C\}$ is more likely to terminate with $V=0$ than the modified loop $\texttt{while}(0<V\leq h)\{C\}$, see ‡‡ above, we can lower–bound the above by:

$$
\begin{aligned}
&\geq \mathsf{wp}\,[\![\texttt{while}(0<V\leq h)\{C\}]\!]\,([V=0]) \\
&= 1-\mathsf{awlp}\,[\![\texttt{while}(0<V\leq h)\{C\}]\!]\,(1-[V=0]) \quad \text{(by Theorem 4.25 A.)} \\
&= 1-\mathsf{awlp}\,[\![\texttt{while}(0<V\leq h)\{C\}]\!]\,([0<V]) \\
&\geq [I]\cdot\big(1-\mathsf{awlp}\,[\![\texttt{while}(0<V\leq h)\{C\}]\!]\,([0<V])\big) \\
&= [I]\cdot\big(1-[I]\cdot\mathsf{awlp}\,[\![\texttt{while}(0<V\leq h)\{C\}]\!]\,([0<V])\big)
\end{aligned}
$$

Since the modified loop $\texttt{while}(0<V\leq h)\{C\}$ terminates from every state satisfying the invariant I, see † in 1., we conclude by Theorem 4.27 B. that $[I]\cdot\mathsf{awlp}\,[\![\texttt{while}\,(\ldots)\}]\!]\,([0<V])\leq \mathsf{awp}\,[\![\texttt{while}\,(\ldots)\}]\!]\,([0<V])$ and we can thus lower–bound the above by:

$$
\begin{aligned}
&\geq [I] \cdot \Big(1 - \mathsf{awp}\,[\![\texttt{while}(0 < V \leq h)\{C\}]\!]\,([0 < V])\Big) \\
&= [I] \cdot \Big(1 - \mathsf{awp}\,[\![\texttt{while}(0 < V \leq h)\{C\}]\!]\,\Big(\big([V = 0] + [V > h]\big) \cdot [0 < V]\Big)\Big) \\
&\qquad\qquad \text{(by postexpectation strengthening, Corollary 4.6)} \\
&= [I] \cdot \Big(1 - \mathsf{awp}\,[\![\texttt{while}(0 < V \leq h)\{C\}]\!]\,([V > h])\Big) \\
&\geq [I] \cdot \Big(1 - \mathsf{awp}\,[\![\texttt{while}(0 < V \leq h)\{C\}]\!]\,([V \geq h])\Big) \\
&\qquad\qquad \text{(by } [V > h] \leq [V \geq h] \text{ and monotonicity, Theorem 4.16)} \\
&\geq [I] \cdot \left(1 - \frac{\mathsf{awp}\,[\![\texttt{while}(0 < V \leq h)\{C\}]\!]\,(V)}{h}\right) \\
&\qquad\qquad \text{(by Markov's inequality, Theorem 4.19)} \\
&\geq [I] \cdot \left(1 - \frac{V}{h}\right) \qquad\qquad \Big(\text{by } \ddagger \text{ in } \boxed{2.}\Big)
\end{aligned}
$$

To summarize, we have until now established

$$[I] \cdot \left(1 - \frac{V}{h}\right) \;\preceq\; \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(1)\;.$$

Since this inequality holds for an arbitrary $h > 0$, we can take the limit $h \to \infty$ and thus obtain

$$
\begin{aligned}
& \lim_{h\to\infty} [I] \cdot \left(1 - \frac{V}{h}\right) \;\preceq\; \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(1) \\
\text{implies}\quad & [I] \cdot (1 - 0) \;\preceq\; \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(1) \\
\text{implies}\quad & [I] \;\preceq\; \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(1)\;,
\end{aligned}
$$

which finally proves that $\texttt{while}(\varphi)\{C\}$ terminates almost–surely from any initial state satisfying the invariant I. Q.E.D.

6.2.4 *Case Studies in Almost–sure Termination*

We now study a few more cases of almost–surely terminating loops and their termination proofs by means of Theorem 6.8 whose correctness we have just proved. We have already seen in Example 6.9 how easy it is to prove almost–sure termination of a symmetric 1–dimensional random walk. For some of the case studies we show in the following, it is much less obvious that they terminate almost–surely.

6.2.4.1 *The Demonically Symmetric Random Walk*

In order to demonstrate the capability of Theorem 6.8 to reason about nondeterminism and take loop invariants into account, we consider a while loop that contains both probabilistic and nondeterministic choice and terminates only from a certain set of states.

$$\texttt{while}(x \neq 0)\{ \quad \Big\{x := x-1\Big\}\,[1/2]\,\Big\{\{x := x+1\} \,\square\, \{\texttt{skip}\}\Big\} \quad \}$$

The execution of the loop is illustrated in Figure 6.3. The difference to the symmetric 1–dimensional random walk is that instead of incrementing x, the while loop above can also do nothing. The demonic behavior (in terms of termination) is of course to perform the increment. Furthermore, the loop guard is $x \neq 0$ instead of $x > 0$. Thus, the particle must hit exactly 0, which is only possible if x was initially an integer.

Apart from the integer issue, the motivation for this loop is the recursive procedure P inspired by an example of [Olm+16]; its definition is

$$P \;\rhd\; \Big\{\texttt{skip}\Big\}\,[1/2]\,\Big\{\,\texttt{call}\ P;\ \{\texttt{call}\ P\} \,\square\, \{\texttt{skip}\}\Big\}\,.$$

Above, we have rewritten this recursive program as a loop by viewing it as a random walk of a particle x whose position represents the height of the call stack. Intuitively, the loop keeps moving x in a random and demonic fashion until the particle hits the origin 0 (empty call stack, all procedure calls have terminated). For that, at each stage it either with probability 1/2 decrements the position of x by one (procedure call terminates after skip; call stack decremented by one), or with probability 1/2 it performs a demonic choice between incrementing the position of x by one (perform two consecutive procedure calls, then terminate; call stack in effect incremented by one ($+2-1=+1$)) or letting x remain at its position (perform one procedure call, then terminate; call stack in effect unchanged ($+1-1=0$)).

Proof of almost–sure termination. We choose the witnesses

$$I = (x \in \mathbb{N}), \quad V \;=\; [x \in \mathbb{N}]\cdot x + [x \notin \mathbb{N}], \quad d \;=\; 1, \quad \text{and} \quad p \;=\; \frac{1}{2}\,.$$

Intuitively, I, V, p, and d tell us that x decreases with probability at least 1/2 by at least 1 through one iteration of the loop body if initially x is an natural number unequal to zero.

Let us now check that all premises of Theorem 6.8 are satisfied: p and d are constant and thus obviously antitone. $V = 0$ indicates termination since $V = 0$ iff $x = 0$.

Next, we check in detail that $[I]$ is a wp–subinvariant with respect to $[I]$:

$$\begin{aligned}
&{}^{\mathsf{wp}}\Phi_{[I]}([I]) \\
&\quad= {}^{\mathsf{wp}}\Phi_{[x\in\mathbb{N}]}([x \in \mathbb{N}]) \\
&\quad= [x=0]\cdot[x\in\mathbb{N}] \;+\; [x\neq 0]\cdot \mathsf{wp}\,[\![body]\!]\,([x\in\mathbb{N}]) \\
&\quad= [x=0]\cdot[x\in\mathbb{N}]
\end{aligned}$$

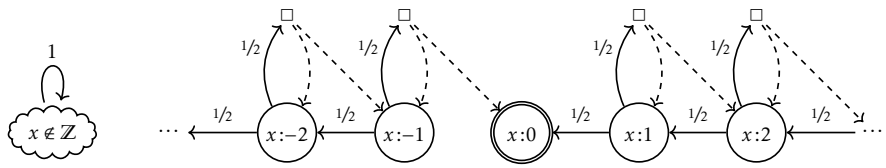


Figure 6.3: Execution of the demonically symmetric random walk. The □ nodes with the dashed arrows represent nondeterministic choices. The values of p and d are constantly 1/2 and 1, respectively. The fact that x is not integer–valued is invariant under iteration of the loop body and thus that set reaches itself with probability 1.

$$
\begin{aligned}
&+\ [x \neq 0] \cdot \tfrac{1}{2} \cdot \Big([x-1 \in \mathbb{N}] + \max\{ [x+1 \in \mathbb{N}],\, [x \in \mathbb{N}] \} \Big) \\
&= [x = 0] \cdot [x \in \mathbb{N}] \ +\ [x \neq 0] \cdot \tfrac{1}{2} \cdot \Big([x \in \mathbb{N}] + [x \in \mathbb{N}] \Big) \\
&= [x = 0] \cdot [x \in \mathbb{N}] \ +\ [x \neq 0] \cdot [x \in \mathbb{N}] \\
&= [x \in \mathbb{N}] \\
&= [I] \ \geq\ [I]
\end{aligned}
$$

We also check that x is an awp–superinvariant with respect to x:

$$
\begin{aligned}
&{}^{\mathsf{awp}}_{\langle x \neq 0, body \rangle} \Phi_V(V) \\
&= [x = 0] \cdot V \ +\ [x \neq 0] \cdot \mathsf{awp}\, [\![body]\!]\, ([x \in \mathbb{N}] \cdot x + [x \notin \mathbb{N}]) \\
&\preceq [x = 0] \cdot V \ +\ [x \neq 0] \cdot \mathsf{awp}\, [\![body]\!]\, ([x \in \mathbb{N}] \cdot x) \\
&\qquad\qquad +\ [x \neq 0] \cdot \mathsf{awp}\, [\![body]\!]\, ([x \notin \mathbb{N}]) \\
&\qquad\qquad\qquad\qquad \text{(sublinearity of awp, Theorem 4.21 B.)} \\
&= [x = 0] \cdot V \ +\ [x \neq 0] \cdot \tfrac{1}{2} \Big(\\
&\qquad\qquad [x-1 \in \mathbb{N}] \cdot (x-1) + \max\{ [x \in \mathbb{N}] \cdot x,\, [x+1 \in \mathbb{N}] \cdot (x+1) \} \\
&\qquad\qquad +\ [x-1 \notin \mathbb{N}] + \max\{ [x \notin \mathbb{N}],\, [x+1 \notin \mathbb{N}] \} \Big) \\
&= [x = 0] \cdot V \ +\ [x \neq 0] \cdot \tfrac{1}{2} \Big([x-1 \in \mathbb{N}] \cdot (x-1) + [x \in \mathbb{N}] \cdot (x+1) \\
&\qquad\qquad +\ [x-1 \notin \mathbb{N}] + [x \notin \mathbb{N}] \Big) \\
&= [x = 0] \cdot V \ +\ [x \neq 0] \cdot \tfrac{1}{2} \Big([x \in \mathbb{N}] \cdot (x-1) + [x \in \mathbb{N}] \cdot (x+1) \\
&\qquad\qquad +\ [x \notin \mathbb{N}] + [x \notin \mathbb{N}] \Big) \\
&= [x = 0] \cdot V \ +\ [x \neq 0] \cdot \tfrac{1}{2} \Big(2\,[x \in \mathbb{N}] \cdot x + 2\,[x \notin \mathbb{N}] \Big) \\
&= [x = 0] \cdot V \ +\ [x \neq 0] \cdot \Big([x \in \mathbb{N}] \cdot x + [x \notin \mathbb{N}] \Big) \\
&= [x = 0] \cdot V \ +\ [x \neq 0] \cdot V \ =\ V
\end{aligned}
$$

Lastly, we show that V, p, and d satisfy the progress condition:

$$
\begin{aligned}
& p \circ V \cdot [\varphi] \cdot [I] \;\leq\; \lambda\sigma.\ \mathsf{wp}\,[\![body]\!]\left(\left[V \leq V(\sigma) - d\Big(V(\sigma)\Big)\right]\right)(\sigma) \\
\text{iff}\quad & \tfrac{1}{2} \circ \Big(\ \cdots\ \Big) \cdot [x \neq 0] \cdot [x \in \mathbb{N}] \\
& \qquad \leq\; \lambda\sigma.\ \mathsf{wp}\,[\![body]\!]\,([V \leq V(\sigma) - 1])(\sigma) \\
\text{iff}\quad & \tfrac{1}{2} \cdot [x \in \mathbb{N}_{\geq 1}] \;\leq\; \lambda\sigma.\ \mathsf{wp}\,[\![body]\!]\,([V \leq V(\sigma) - 1])(\sigma) \\
\text{iff}\quad & \tfrac{1}{2} \cdot [x \in \mathbb{N}_{\geq 1}] \;\leq\; \frac{1}{2}\Big([V\,[x/x-1] \leq V - 1] + \max\{\ldots\}\Big) \\
\text{implied by}\quad & \tfrac{1}{2} \cdot [x \in \mathbb{N}_{\geq 1}] \;\leq\; \tfrac{1}{2}\,[V\,[x/x-1] \leq V - 1] \\
\text{iff}\quad & \tfrac{1}{2} \cdot [x \in \mathbb{N}_{\geq 1}] \;\leq\; [x \in \mathbb{N}_{\geq 1}] \cdot \tfrac{1}{2}\,[V\,[x/x-1] \leq V - 1] \\
\text{iff}\quad & \tfrac{1}{2} \cdot [x \in \mathbb{N}_{\geq 1}] \;\leq\; [x \in \mathbb{N}_{\geq 1}] \cdot \tfrac{1}{2}\,[x - 1 \leq x - 1] \\
& \qquad \text{(by careful analysis of } V\,[x/x-1] \text{ and } V \text{ given } x \in \mathbb{N}_{\geq 1}) \\
\text{iff}\quad & \tfrac{1}{2} \cdot [x \in \mathbb{N}_{\geq 1}] \;\leq\; [x \in \mathbb{N}_{\geq 1}] \cdot \tfrac{1}{2}
\end{aligned}
$$

The last inequality is obviously true. This shows that all preconditions of Theorem 6.8 are satisfied and as a consequence the demonically symmetric random walk terminates almost–surely from any initial state where x is integer–valued.

Coming back to our motivation, the procedure P' given by

$$P' \;\triangleright\quad \{\mathtt{skip}\}\,[1/2]\,\{\mathtt{call}\ P';\ \mathtt{call}\ P';\ \{\mathtt{call}\ P'\} \,\square\, \{\mathtt{skip}\}\}\ ,$$

i.e. potentially three consecutive procedure calls instead of just two procedure calls, interestingly is not almost–surely terminating: it terminates only with probability $(\sqrt{5}-1)/2 < 1$ [Olm+16].

6.2.4.2 *The Symmetric-in-the-Limit Random Walk*

While so far we have considered only constant probability and decrease functions, we now consider a while loop requiring a *non–constant* decrease function d. For that, consider the following while loop:[10]

$$
\begin{array}{l}
\mathtt{while}\,(x > 0)\,\{ \\
\quad q := x/2x+1; \\
\quad \{x := x - 1\}\,[q]\,\{x := x + 1\} \\
\}
\end{array}
$$

In order not to clutter the reasoning below, we assume that x is of type $\mathbb{N}$. The execution of the loop is illustrated in Figure 6.4.

Intuitively, the loop models an asymmetric random walk of a particle x, terminating when the particle hits the origin 0. In one iteration of the loop

10 This example is due to McIver & Morgan [MM16].

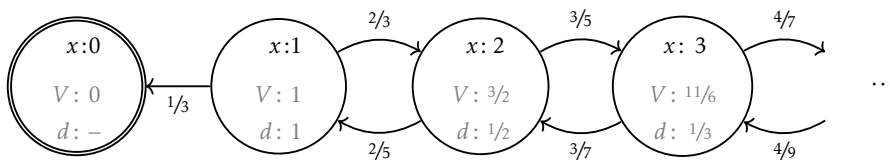


Figure 6.4: Execution of the symmetric-in-the-limit random walk. Inside the nodes we give the valuations of variable x as well as the values of the variant V and the decrease function d. The value of p is constantly 1/3.

body, the program either with probability $x/2x+1$ decrements the position of x by one, or with probability $x+1/2x+1$ increments the position of x by one. The further the particle x is away from 0, the more symmetric becomes the random walk since $x/2x+1$ approaches 1/2 asymptotically. Yet, it is not so obvious that this random walk indeed also terminates with probability 1.

Proof of almost–sure termination. We choose the witnesses

$$I = \mathsf{true}, \quad V = H_x, \quad d(v) = \begin{cases} \frac{1}{n}, & \text{if } v > 0 \text{ and } v \in (H_{n-1}, H_n] \\ 1, & \text{if } v = 0, \end{cases}$$

$$\text{and} \quad p(v) = \frac{1}{3},$$

where H_x is the x-th harmonic number.[11] Intuitively, these witnesses tell us that the variant V, i.e. the harmonic number of the value of x, decreases with probability at least 1/3 by at least $\frac{1}{x}$ through one loop iteration if initially $x > 0$.

Notice furthermore that while d measures precisely the potential decrease of V, the real probability to decrease V is monotonically *increasing* whereas Theorem 6.8 calls for an *antitone, thus monotonically decreasing*, p. The remedy here is that the decrease probability is globally lower–bounded by the constant 1/3 and thus an antitone probability function exists.

Let us now check that all premises of Theorem 6.8 are met: p is constant and thus obviously antitone. true is a wp–subinvariant of any loop that terminates almost–surely. This is especially the case when the loop body itself is loop–free. $V = 0$ indicates termination since $V = 0$ iff $x \leq 0$ (because x is assumed to be of type $\mathbb{N}$).

Next, we provide a detailed check that V is an awp–supermartingale:

$$\overset{\mathsf{awp}}{\langle x>0, body\rangle}\Phi_V(V)$$

11 Formally, $H_x = \sum_{n=1}^{x} \frac{1}{n}$. Notice that $H_0 = 0$.

$$
\begin{aligned}
&= \overset{\text{awp}}{\langle x>0, body\rangle}\Phi_{H_x}(H_x)\\
&= [x \le 0]\cdot H_x \;+\; [x>0]\cdot \text{awp}\,[\![body]\!]\,(H_x)\\
&= [x \le 0]\cdot H_x \;+\; [x>0]\cdot \left(\frac{x}{2x+1}\cdot H_{x-1} + \left(1-\frac{x}{2x+1}\right)\cdot H_{x+1}\right)\\
&= [x \le 0]\cdot H_x \;+\; [x>0]\cdot \left(\frac{x}{2x+1}\cdot \left(H_x - \frac{1}{x}\right) + \left(\frac{x+1}{2x+1}\right)\cdot \left(H_x + \frac{1}{x+1}\right)\right)\\
&= [x \le 0]\cdot H_x \;+\; [x>0]\cdot \left(\left(\frac{x}{2x+1} + \frac{x+1}{2x+1}\right)\cdot H_x - \frac{1}{2x+1} + \frac{1}{2x+1}\right)\\
&= [x \le 0]\cdot H_x \;+\; [x>0]\cdot H_x\\
&= H_x\\
&= V \;\preceq\; V
\end{aligned}
$$

Lastly, we show that V, p, and d satisfy the progress condition. For that, note that $d(H_n) = 1/n$ and consider the following:

$$
\begin{aligned}
& p \circ V \cdot [\varphi]\cdot[I]\\
&\qquad \le\; \lambda\sigma.\ \text{wp}\,[\![body]\!]\left(\left[V \le V(\sigma) - d\big(V(\sigma)\big)\right]\right)(\sigma)\\
\text{iff}\quad & \frac{1}{3}\circ H_x\cdot[x>0]\cdot[\text{true}]\\
&\qquad \le\; \lambda\sigma.\ \text{wp}\,[\![body]\!]\left(\left[H_x \le H_{x(\sigma)} - d\big(H_{x(\sigma)}\big)\right]\right)(\sigma)\\
\text{iff}\quad & \frac{1}{3}\cdot[x>0] \;\le\; \lambda\sigma.\ \text{wp}\,[\![body]\!]\left(\left[H_x \le H_{x(\sigma)} - \frac{1}{H_{x(\sigma)}}\right]\right)(\sigma)\\
\text{iff}\quad & \frac{1}{3}\cdot[x>0] \;\le\; \lambda\sigma.\ \left(\frac{x}{2x+1}\cdot\left[H_{x-1} \le H_{x(\sigma)} - \frac{1}{H_{x(\sigma)}}\right]\right.\\
&\qquad\qquad\qquad \left. +\;\left(1-\frac{x}{2x+1}\right)\cdot\left[H_{x+1} \le H_{x(\sigma)} - \frac{1}{H_{x(\sigma)}}\right]\right)(\sigma)\\
\text{iff}\quad & \frac{1}{3}\cdot[x>0] \;\le\; \frac{x}{2x+1}\cdot\left[H_{x-1} \le H_x - \frac{1}{H_x}\right]\\
&\qquad\qquad\qquad +\;\frac{x+1}{2x+1}\cdot\left[H_{x+1} \le H_x - \frac{1}{H_x}\right]\\
\text{implied by}\quad & \frac{1}{3}\cdot[x>0] \;\le\; [x>0]\cdot\frac{x}{2x+1}\cdot\left[H_{x-1} \le H_x - \frac{1}{H_x}\right]\\
\text{iff}\quad & \frac{1}{3}\cdot[x>0] \;\le\; [x>0]\cdot\frac{x}{2x+1}\cdot[\text{true}]\\
\text{iff}\quad & \frac{1}{3}\cdot[x>0] \;\le\; [x>0]\cdot\frac{x}{2x+1}
\end{aligned}
$$

The last line is true for all natural numbers $x > 0$. This shows that all preconditions of Theorem 6.8 are satisfied and as a consequence the symmetric-in-the-limit random walk terminates almost–surely.

Non–existence of an affine variant. For this program, note that our variant was *non–affine*, i.e. not of the form $a + bx + cq$. In fact, there exists *no affine variant* that satisfies the superinvariant property. Such affine variants are used e.g. by [CNZ17]. Any affine[12] variant V would be of the form

$$V \;=\; a + bx + cq\,,$$

for some (positive) coefficients a, b, c. [13] Now we attempt to check the superinvariant property for a variant of that form:

$$\begin{aligned}
&{}^{\mathsf{awp}}_{\langle x>0, body\rangle}\Phi_V(V)\\
&= {}^{\mathsf{awp}}_{\langle x>0, body\rangle}\Phi_{a+bx+cq}(a+bx+cq)\\
&= [x \le 0]\cdot x \;+\; [x>0]\cdot \mathsf{awp}\,[\![body]\!]\,(a+bx+cq)\\
&= [x \le 0]\cdot x \;+\; [x>0]\cdot\left(a - 2b\cdot\frac{x}{2x+1} + bx + b + c\cdot\frac{x}{2x+1}\right)\\
&\overset{!}{\le} a + bx + cq \;=\; V
\end{aligned}$$

For $x \le 0$ this is trivially satisfied. For $x > 0$, the above is satisfied iff

$$\begin{aligned}
& a - 2b\cdot\frac{x}{2x+1} + bx + b + c\cdot\frac{x}{2x+1} \;\le\; a + bx + cq\\
\text{iff}\quad & -2b\cdot\frac{x}{2x+1} + b + c\cdot\frac{x}{2x+1} \;\le\; cq\,,
\end{aligned}$$

which is only satisfiable for all possible valuations of q and $x > 0$ iff $b = c = 0$. Thus, if V is required to be affine, then V has to be constantly a, for $a \ge 0$. Indeed, a is a superinvariant. However, it is clear that the constant a cannot possibly indicate termination, i.e. clearly

$$[a = 0] \;\neq\; [x \le 0]\,.$$

Thus, there cannot exist an affine superinvariant that proves termination of symmetric–in–the–limit while loop.

6.2.4.3 *The Escaping Spline*

We now consider a while loop where we will make use of a non–constant probability function p. Consider the following while loop:[14]

$$\begin{aligned}
&\texttt{while}\,(x>0)\,\{\\
&\quad q := 1/_{x+1};\\
&\quad \{x := 0\}\,[q]\,\{x := x+1\}
\end{aligned}$$

12 Some authors call this a *linear* variant.

13 Coefficients need to be positive because otherwise $V \ge 0$ cannot be ensured. However, this is not crucial in this proof.

14 This example is due to McIver & Morgan [MM16].

}

Assume again that $x \in \mathbb{N}$. The execution of the loop is illustrated in Figure 6.5. Intuitively, the loop models a random walk of a particle x that terminates when the particle hits the origin 0. The random walk either with probability $1/x+1$ immediately terminates or with probability $x/x+1$ increments the position of x by one. This means that for each iteration where the loop does not terminate, it becomes even *more likely not to terminate in the next iteration*. Thus, the longer the loop runs, the less likely it will terminate since the probability to continue looping approaches 1 asymptotically. Yet this loop terminates almost–surely, as we will now prove.

Proof of almost–sure termination. We choose witnesses

$$I = \mathsf{true}, \quad V = x, \quad d(v) = 1, \quad \text{and} \quad p(v) = \frac{1}{v+1}.$$

Intuitively this tells us that x decreases with probability at least $1/x+1$ by at least 1 through one loop iteration if initially $x > 0$.

Notice that while the variant function measures precisely the potential decrease in each state, the actual decrease is monotonically *increasing* the further we move away from $x = 0$, whereas Theorem 6.8 calls for an *antitone*, *thus monotonically decreasing* decrease function. The remedy here is that the decrease is globally lower–bounded by 1 and thus a constant — and hence antitone — decrease function exists.

Let us now check that all premises of Theorem 6.8 are met: d is constant and thus obviously antitone. true is a wp–subinvariant of any loop that terminates almost–surely. This is especially the case when the loop body itself is loop–free. $V = 0$ indicates termination since $V = 0$ iff $x \leq 0$ (since $x \in \mathbb{N}$).

Next, we provide a detailed check that V is an awp–supermartingale:

$$\begin{aligned}
&{}^{\mathsf{awp}}_{\langle x>0, body\rangle}\Phi_V(V) \\
&= {}^{\mathsf{awp}}_{\langle x>0, body\rangle}\Phi_x(x) \\
&= [x \leq 0] \cdot x \;+\; [x > 0] \cdot \mathsf{awp}\,[\![body]\!]\,(x) \\
&= [x \leq 0] \cdot x \;+\; [x > 0] \cdot \left(\frac{1}{x+1} \cdot 0 + \left(1 - \frac{1}{x+1}\right) \cdot (x+1)\right) \\
&= [x \leq 0] \cdot x \;+\; [x > 0] \cdot \left(\frac{1}{x+1} \cdot 0 + \frac{x}{x+1} \cdot (x+1)\right) \\
&= [x \leq 0] \cdot x \;+\; [x > 0] \cdot x \\
&= x \\
&= V \;\leq\; V
\end{aligned}$$

Finally, we show that V, p, and d satisfy the progress condition:

$$p \circ V \cdot [\varphi] \cdot [I] \;\leq\; \lambda\sigma.\ \mathsf{wp}\,[\![body]\!]\left(\left[V \leq V(\sigma) - d\Big(V(\sigma)\Big)\right]\right)(\sigma)$$

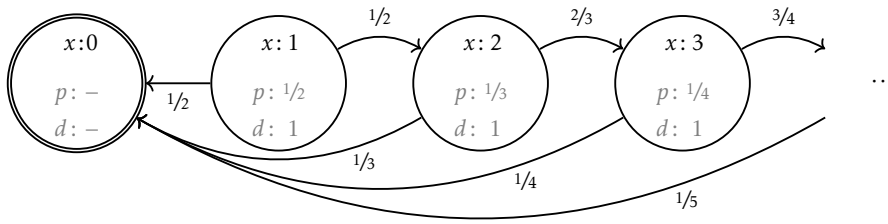


Figure 6.5: Execution of the escaping spline loop. The value of the variant V is equal to the value of the variable x in each state. Inside the nodes we give the valuations of variable x as well as the values of the probability function p and the decrease function d in each state.

$$
\begin{aligned}
\text{iff}\quad & \left(\lambda v.\ \frac{1}{v+1}\right) \circ x \cdot [x > 0] \cdot [\mathsf{true}] \\
& \quad \leq\ \lambda\sigma.\ \mathsf{wp}\,[\![body]\!]\left(\left[x \leq x(\sigma) - (\lambda v.\ 1)\big(x(\sigma)\big)\right]\right)(\sigma) \\
\text{iff}\quad & \frac{1}{x+1} \cdot [x > 0]\ \leq\ \lambda\sigma.\ \mathsf{wp}\,[\![body]\!]\,([x \leq x(\sigma) - 1])\,(\sigma) \\
\text{iff}\quad & \frac{1}{x+1} \cdot [x > 0] \\
& \quad \leq\ \lambda\sigma.\ \left(\frac{1}{x+1} \cdot [0 \leq x(\sigma) - 1] + \frac{x}{x+1} \cdot [x+1 \leq x(\sigma) - 1]\right)(\sigma) \\
\text{iff}\quad & \frac{1}{x+1} \cdot [x > 0]\ \leq\ \frac{1}{x+1} \cdot [0 \leq x - 1] + \frac{x}{x+1} \cdot [x+1 \leq x - 1] \\
\text{iff}\quad & \frac{1}{x+1} \cdot [x > 0]\ \leq\ \frac{1}{x+1} \cdot [x > 0] \cdot [0 \leq x - 1] + \frac{x}{x+1} \cdot [\mathsf{false}] \\
\text{iff}\quad & \frac{1}{x+1} \cdot [x > 0]\ \leq\ \frac{1}{x+1}
\end{aligned}
$$

This shows that all preconditions of Theorem 6.8 are satisfied and as a consequence the escaping spline loop terminates almost–surely.

Part II

ADVANCED WEAKEST PREEXPECTATION REASONING

In the second part part of this thesis, I present three different advancements of the classical weakest preexpectation calculus. The first one is for reasoning about *expected runtimes*. The second calculus enables weakest preexpectation reasoning for probabilistic programs *with conditioning*. The third calculus enables weakest preexpectation reasoning about *mixed–sign* random variables. For all three calculi, I present dedicated proof rules for reasoning about loops. I also discuss in which way they are novel and how they solve reasoning problems that could not so easily be handled using existing calculi.

EXPECTED RUNTIMES 7

In 1976, Michael Oser Rabin published his paper titled *Randomized Algorithms* in which he describes a method for solving the *closest–pair problem* in computational geometry [Rab76]. This work is today considered to be *the* seminal paper on randomized algorithms [Smi00]. While a naïve deterministic brute–force approach takes quadratic time, Rabin's randomized algorithm solves the closest–pair problem in *expected linear time*.

One year later, in 1977, Robert Martin Solovay and Volker Strassen presented a randomized primality test that decides in *polynomial time* whether a given number is either composite or probably prime, thus proving that primality testing is in the complexity class coRP[1] [SS77]. In 1992, Leonard Adleman and Ming–Deh Huang further reduced the complexity of primality testing to ZPP[2], thus proving that primality testing can be solved efficiently in expectation [AH92]. Turning an inefficient deterministic algorithm into a randomized algorithm that is either

A. certainly more efficient, yielding the correct result with high probability (*Monte Carlo algorithm: certainly fast and probably correct*), or

B. more efficient in expectation, yielding the correct result with certainty (*Las Vegas algorithm: probably fast and certainly correct*), or

C. more efficient in expectation, yielding the correct result with high probability (*Atlantic City algorithm: probably fast and probably correct)*

is a principal motivation of introducing randomization into the computation. Other prime examples are Freivalds' matrix multiplication [Fre79] or Hoare's randomized variant of quicksort with random pivot selection [Hoa62].

Besides describing randomized algorithms, providing precise encodings of complex probability distributions is another use for probabilistic programs.

1 coRP is the class of decision problems for which there is a randomized algorithm that [Gil77]

- *certainly* terminates in polynomial time,
- outputs *yes* if the correct answer is *yes* with probability 1, and
- outputs *no* if the correct answer is *no* with probability $\geq 1/2$.

Thus, if a coRP algorithm outputs *no*, this answer is always correct. On the other hand, if it outputs *yes*, this answer is correct only with high probability.

2 ZPP is the class of decision problems for which there are randomized algorithms that [Gil77]

- terminate in *expected polynomial time*, and
- *certainly* output the correct answer.

For this use case, too, expected runtimes are of paramount importance. They can here be interpreted as the expected time that is needed to obtain a single sample from the encoded probability distribution.

In general, the runtime of a probabilistic program is affected not only by the input but also by the internal randomness of the program. Technically speaking, the runtime is hence a random variable, i.e. it is t_1 with probability p_1, t_2 with probability p_2 and so on. Reasoning about expected runtimes of probabilistic programs is surprisingly subtle and full of nuances which underlines the desire for *formal methods* suited for reasoning about expected runtimes. We will develop such methods in this chapter.

After discussing some particular intricacies that make reasoning about expected runtimes difficult and after discussing why an obvious approach employing a runtime–counter ghost variable is *unsound*, we will develop the ert calculus — a weakest precondition style calculus specifically tailored to sound and complete reasoning about expected runtimes of probabilistic programs. We also discuss some basic properties of our ert transformers, such as continuity, monotonicity, and the relationship of expected runtime transformers to the expectation transformers studied in Chapter 4. We then present proof rules for reasoning about expected runtimes of while loops, which are expressed as least fixed points, and finally conclude with a case study and a discussion of related work.

7.1 HURDLES IN REASONING

Reasoning about the expected runtime of probabilistic programs is a difficult task, partly because such programs exhibit unexpected and sometimes counterintuitive behavior. In classical sequential programs, for instance, a single diverging program run yields the program to have an infinite runtime. For a randomized algorithm, on the other hand, it is perfectly fine to admit *infinite runs* while still having an expected, say polynomial, runtime.

For demonstration purposes, consider the simple program

$$\texttt{while}(x > 0)\{\ \{x := x - 1\}\,[1/2]\,\{\texttt{skip}\}\ \}$$

from Chapter 6. This program terminates within $\mathcal{O}(x)$ steps in expectation. However, the program does admit an infinite run, namely the one where infinitely often `skip` is executed (though this happens only with probability 0). The program

$$\texttt{while}(x > 0)\{\ \{x := 0\}\,[1/2]\,\{\texttt{skip}\}\ \}.$$

even terminates in $\mathcal{O}(1)$, i.e. *constantly* many, steps in expectation, while *still admitting an infinite run*.

The above examples show that in order to determine the worst–case (expected) runtime of a probabilistic program, it does not suffice to consider the length of the longest computation path, but instead the averaged length of (almost) *all* paths has to be accounted for. This circumstance stands in stark contrast to the case for deterministic programs.

Another problem occurs when it comes to reasoning about termination. If two deterministic programs C and C' each terminate after a finite number of steps on arbitrary inputs, then the sequential composition $C \fatsemi C'$ obviously also terminates after a finite number of steps on any arbitrary input.

For probabilistic programs this does not hold in general: Even universal positive almost–sure termination[3] is *not closed under sequential composition of programs*. Consider for instance the following two programs:

$$
\begin{aligned}
C_1 \rhd \quad & x := 1 \fatsemi \\
& y := 1 \fatsemi \\
& \texttt{while}(x > 0)\{ \\
& \qquad \{x := 0\}\,[1/2]\,\{y := y \cdot 2\} \\
& \}
\end{aligned}
\qquad\qquad
\begin{aligned}
C_2 \rhd \quad & \texttt{while}(y > 0)\{ \\
& \qquad y := y - 1 \\
& \}
\end{aligned}
$$

The program C_1 terminates within $\mathcal{O}(1)$ steps in expectation, whereas the program C_2 needs in expectation $\mathcal{O}(y)$ steps until it terminates. Individually, they thus both terminate universally positively almost–surely. Yet the expected value of y after executing the first program is ∞ and thus the sequential composition of the two programs, i.e. the program $C_1 \fatsemi C_2$, does *not* terminate universally positively almost–surely.

A last subtlety we would like to address is that expected runtimes are extremely sensitive to variations in the probabilities occurring in the program. Consider for instance the (possibly biased) 1–dimensional random walk

$$
\begin{aligned}
& \texttt{while}(x > 0)\{ \\
& \qquad \{x := x - 1\}\,[1/2 + \epsilon]\,\{x := x + 1\} \\
& \},
\end{aligned}
$$

where $0 \leq \epsilon \leq 1/2$. For $\epsilon = 0$, the random walk is symmetric and its expected runtime is *infinite*. However, for any arbitrarily small $0 < \epsilon$, the expected number of loop iterations drops from infinity to $1/2\epsilon \cdot x$, thus from infinite to *linear* expected runtime. This demonstrates formidably how sensitive expected runtimes and in particular positive almost–sure termination is to the probabilities occurring in the programs.

3 A program terminates universally positively almost–surely if it terminates on all inputs within an expected finite number of steps (see Definition 6.1 c.).

7.2 UNSOUNDNESS OF THE OBVIOUS APPROACH

Before we go on to develop a calculus specifically tailored to reasoning about expected runtimes, let us address the concerns of a hypothetical critic who might ask:

> *Why even develop a dedicated calculus?*
> *Why not annotate the program with a runtime–counting program variable and reason about the expected value of that variable?*

Besides that one may rightfully argue that the above is a somewhat inelegant way of approaching this problem, there is an even more convincing reason why reasoning about the expected value of a runtime counter is a bad idea: *it is unsound!* Consider for example the program

$$
\begin{aligned}
&x := 1 \,⨟\\
&\texttt{while}(x > 0)\{\\
&\qquad \{x := 0\}\,[1/2]\,\{\texttt{skip}\}\\
&\}\,.
\end{aligned}
$$

which we annotate with a runtime counter as follows:

$$
\begin{aligned}
&x := 1 \,⨟\\
&rtc := 0 \,⨟\\
&\texttt{while}(x > 0)\{\\
&\qquad rtc := rtc + 1 \,⨟\\
&\qquad \{x := 0\}\,[1/2]\,\{\texttt{skip}\}\\
&\}
\end{aligned}
$$

Here, the variable rtc counts only the number of loop iterations. The expected value of rtc after executing the above program is 2. In more detail, the wp–characteristic function of the while loop above with respect to post-expectation rtc is given by

$$\Phi(X) \;=\; [x \leq 0] \cdot rtc \;+\; [x > 0] \cdot \frac{1}{2} \cdot \Big(X\,[x, rtc/0, rtc+1] + X\,[rtc/rtc+1] \Big)\,.$$

By checking

$$\Phi(rtc + [x > 0] \cdot 2) \;\leq\; rtc + [x > 0] \cdot 2\,, \tag{†}$$

the induction rule (Theorem 5.4) yields that $rtc + [x > 0] \cdot 2$ is an upper bound on the expected value of rtc after executing the while loop. Prepending the loop with the initialization of x and rtc finally gives 2 as expected value of rtc for the whole program.

Consider now the program

$$
\begin{array}{l}
x := 1 \,⨟ \\
\texttt{while}(x > 0)\{ \\
\quad \{x := 0\}\,[1/2]\,\{\texttt{while}(\texttt{true})\{\texttt{skip}\}\} \\
\}\,.
\end{array}
$$

which we also annotate with a runtime counter as follows:

$$
\begin{array}{l}
x := 1 \,⨟ \\
rtc := 0 \,⨟ \\
\texttt{while}(x > 0)\{ \\
\quad rtc := rtc + 1 \,⨟ \\
\quad \{x := 0\}\,[1/2]\,\{\texttt{while}(\texttt{true})\{rtc := rtc + 1 \,⨟\, \texttt{skip}\}\} \\
\}
\end{array}
$$

Here, rtc also counts only loop iterations, but notice that since we have a nested loop we also have to account for the inner loop iterations. Now, for any postexpectation f, we have

$$\mathsf{wp}\,[\![\texttt{while}(\texttt{true})\{rtc := rtc + 1 \,⨟\, \texttt{skip}\}]\!]\,(f) \;=\; 0\,,$$

because this while loop terminates with probability 0. Thus, the inner loop $\texttt{while}(\texttt{true})\{rtc := rtc + 1 \,⨟\, \texttt{skip}\}$ is semantically equivalent to $\texttt{diverge}$ and hence the whole annotated program above is semantically equivalent to

$$
\begin{array}{l}
x := 1 \,⨟ \\
rtc := 0 \,⨟ \\
\texttt{while}(x > 0)\{ \\
\quad rtc := rtc + 1 \,⨟ \\
\quad \{x := 0\}\,[1/2]\,\{\texttt{diverge}\} \\
\}\,.
\end{array}
$$

The characteristic function of the while loop above with respect to postexpectation rtc is given by

$$\Psi(X) \;=\; [x \leq 0] \cdot rtc \;+\; [x > 0] \cdot \frac{1}{2} \cdot \Big(X\,[x, rtc/0, rtc + 1] + 0\Big)\,,$$

and we can clearly see that

$$\Psi(X) \;\preceq\; \Phi(X)\,, \tag{‡}$$

for all $X \in \mathbb{E}$. But this means that by

$$\Psi\Big(rtc + [x > 0] \cdot 2\Big) \overset{\text{by } ‡}{\preceq} \Phi\Big(rtc + [x > 0] \cdot 2\Big) \overset{\text{by } †}{\preceq} rtc + [x > 0] \cdot 2$$

the induction rule yields that $rtc + [x > 0] \cdot 2$ is also an upper bound on the expected value of *rtc* after executing the while loop with the `diverge` statement.

This is now very problematic, since after prepending the loop with the initializations for x and *rtc* we get that 2 is an upper bound on the expected value of the runtime counter, whereas the actual expected runtime is *infinite*, because the loop diverges with strictly positive probability (in fact the probability of divergence is $2/3$). This example rigorously demonstrates that the obvious but naïve approach of annotating a program with a runtime counter and reasoning about its expected value is in general *unsound* for reasoning about expected runtimes of probabilistic programs.

Of course, the critic may now argue that this situation would perhaps not occur, if the program at hand terminates with probability 1. But while nontermination of the inner loop was obvious in this example, this need not always be so obvious, and we learned in Chapter 6 that reasoning about almost–sure termination can be quite involved as no straightforward induction rule for proving almost–sure termination is available. The calculus we will develop in the following, on the other hand, does allow for *sound inductive reasoning about positive almost–sure termination*.

7.3 THE EXPECTED RUNTIME CALCULUS

Towards formal and systematic reasoning about expected runtimes on source code level, we propose a method similar to weakest preexpectation reasoning. Recall that we studied in Chapter 4 an expectation transformer (read: random variable transformer) wp that would associate with each program $C \in \mathsf{pGCL}$ a function

$$\mathsf{wp}\,[\![C]\!] : \quad \mathbb{E} \to \mathbb{E}\,,$$

where $\mathbb{E}$ was the set of random random variables mapping program states to non–negative reals or infinity. The transformer wp was defined in a way such that $\mathsf{wp}\,[\![C]\!]\,(f)(\sigma)$ gives the *expected value of f after executing C on input σ*. Consequently, $\mathsf{wp}\,[\![C]\!]\,(f)$ is a *function* mapping each initial state σ to the expected value of f after executing C on input σ.

We now lift this principle to reasoning about expected runtimes. Our goal is thus to associate to any program C a function that maps each state σ to the expected runtime of executing C on initial state σ. Whereas runtimes of deterministic programs take values in the natural numbers, expected runtimes of probabilistic programs can take values in the non–negative reals, even irrational, non–algebraic, and non–computable values. A runtime in the setting of probabilistic programs is thus again a function mapping (initial) program states to non–negative reals (interpreted as expected runtimes):

DEFINITION 7.1 (Runtimes [Kam+16; Kam+18]):

A. *The set of runtimes, denoted* $\mathbb{T}$*, is defined to coincide with the set of expectations (see* Definition 4.1*), i.e.*

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

Consequently, the complete lattice $(\mathbb{T}, \preceq)$*, its least element, and the construction of suprema is defined exactly as for expectations, i.e. the order relation is given by*

$$s \preceq t \qquad \text{iff} \qquad \forall\, \sigma \in \Sigma\colon \quad s(\sigma) \;\leq\; t(\sigma)\,;$$

the least element is

$$\lambda \sigma.\; 0\,,$$

which we overloadingly denote by 0*; and the supremum of a subset* $S \subseteq \mathbb{T}$ *is constructed pointwise by*

$$\sup S \;=\; \lambda \sigma.\; \sup_{t \in S} t(\sigma)\,.$$

We write $s \ll t$ *to indicate that* s *is everywhere smaller than* t*, i.e.*

$$s \ll t \qquad \text{iff} \qquad \forall\, \sigma \in \Sigma\colon \quad s(\sigma) \;<\; t(\sigma)\,.$$

For formal reasoning about expected runtimes, we will describe a runtime transformer ert that associates with each program $C \in \mathsf{pGCL}$ a function

$$\mathsf{ert}\,[\![C]\!]\colon \quad \mathbb{T} \to \mathbb{T}\,,$$

The transformer ert will be defined in a way such that $\mathsf{ert}\,[\![C]\!]\,(t)(\sigma)$ gives the expected time it takes to

1. execute C on input σ (yielding some final state τ), and then
2. let time $t(\tau)$ pass.

Consequently, $\mathsf{ert}\,[\![C]\!]\,(t)$ will be a function mapping each initial state σ to the respective expected time needed to execute C on σ and let time t pass after termination of C. Because t represents time that is spent *after* executing C and in an analogy to pre- and postanticipations and pre- and postexpectations, we call t a *postruntime* and consequently $\mathsf{ert}\,[\![C]\!]\,(t)$ a *preruntime*.

A function that maps to each initial state σ the expected time it takes to just execute C on σ (and be done afterwards) is given by

$$\mathsf{ert}\,[\![C]\!]\,(0)\,.$$

The reason we need a continuation t at all is that we want to allow for compositional reasoning of sequentially composed programs. This situation is

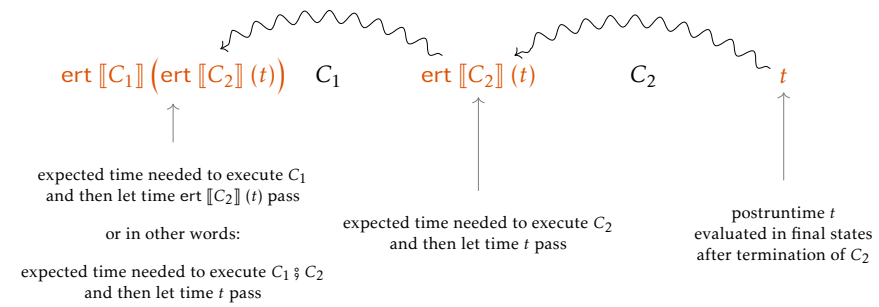


Figure 7.1: Continuation–passing style expected runtime transformer.

depicted in Figure 7.1. If we would like to reason about the expected runtime of $C_1 \fatsemi C_2$, then we want to express this as

$$\mathsf{ert}\,[\![C_1]\!]\left(\mathsf{ert}\,[\![C_2]\!]\,(0)\right).$$

Intuitively, this is nothing else than saying: The expected runtime of executing $C_1 \fatsemi C_2$ is equal to the expected time it takes to first execute C_1 and then let the expected time it takes to execute C_2 pass.

Just like the expectation transformers studied in Chapter 4, the ert transformer can be defined in a very systematic way, namely by induction on the structure of the program. Furthermore, we can observe that ert's definition is very close to the definition of the awp transformer.

Definition 7.2 (The Expected Runtime Transf. [Kam+16; Kam+18]): *For $C \in \mathsf{pGCL}$, the expected runtime transformer*

$$\mathsf{ert}\,[\![C]\!]\colon \quad \mathbb{T} \to \mathbb{T}$$

is defined according to the rules in Table 7.1.

We call the function

$$\Phi_t^{\mathsf{ert}}{}_{\langle \varphi, C\rangle}(X) \;=\; 1 \;+\; [\neg\varphi]\cdot t \;+\; [\varphi]\cdot \mathsf{ert}\,[\![C]\!]\,(X)$$

the ert*–characteristic function of* $\mathtt{while}(\varphi)\{C\}$ *with respect to postruntime t. If* ert*, φ, C, or t are clear from the context, we omit them from Φ.*

The rules for the ert transformer in Table 7.1 are very similar to those for the wp transformer in Table 4.1 and we thus assume familiarity of the reader with the latter and focus mostly on the differences. If you feel unfamiliar with the rules for wp, please refer to Chapters 2 and 4, in particular Section 2.3 (especially Section 2.3.2) and Section 4.1 (especially Section 4.1.3).

C	$\mathsf{ert}\,[\![C]\!]\,(t)$
`skip`	$1 + t$
`diverge`	∞
$x := E$	$1 + t[x/E]$
$x :\approx \mu$	$1 \;+\; \lambda\sigma.\; \int_{\mathsf{Vals}} \Big(\lambda v.\; t\big(\sigma\,[x \mapsto v]\big)\Big)\, d\,\mu_\sigma$
$C_1 \fatsemi C_2$	$\mathsf{ert}\,[\![C_1]\!]\,\big(\mathsf{ert}\,[\![C_2]\!]\,(t)\big)$
$\texttt{if}\,(\varphi)\,\{C_1\}\,\texttt{else}\,\{C_2\}$	$1 \;+\; [\varphi]\cdot \mathsf{ert}\,[\![C_1]\!]\,(t) \;+\; [\neg\varphi]\cdot \mathsf{ert}\,[\![C_2]\!]\,(t)$
$\{C_1\} \,\square\, \{C_2\}$	$1 \;+\; \max\big\{\mathsf{ert}\,[\![C_1]\!]\,(t), \mathsf{ert}\,[\![C_2]\!]\,(t)\big\}$
$\{C_1\}\,[p]\,\{C_2\}$	$1 \;+\; p\cdot \mathsf{ert}\,[\![C_1]\!]\,(t) \;+\; (1-p)\cdot \mathsf{ert}\,[\![C_2]\!]\,(t)$
$\texttt{while}(\varphi)\{C'\}$	$\mathsf{lfp}\; X.\, 1 \;+\; [\neg\varphi]\cdot t \;+\; [\varphi]\cdot \mathsf{ert}\,[\![C']\!]\,(X)$

Table 7.1: Rules for defining the expected runtime transformer ert.

The main difference between ert and wp is a $1 + \ldots$ occurring in most of the rules for ert. The `skip` statement, for instance, does not alter the program state. The associated weakest preexpectation transformer is thus defined as

$$\mathsf{wp}\,[\![\texttt{skip}]\!]\,(f) \;=\; f\,,$$

for any postexpectation f. But even if `skip` has no effect on the program state, we still assume that `skip` does have *some* effect, namely consuming one unit of time. Thus, we need

$$\mathsf{ert}\,[\![\texttt{skip}]\!]\,(t) \;=\; 1 + t\,.$$

units of time to execute `skip` and then let time t pass.

The most obvious difference is for `diverge` — the certainly diverging program. The associated weakest preexpectation transformer is defined as

$$\mathsf{wp}\,[\![\texttt{diverge}]\!]\,(f) \;=\; 0\,,$$

for any postexpectation f, since the resulting distribution over final states is the nulldistribution (no final state is reached at all) and the expected value of any f with respect to the nulldistribution is 0. On the other hand, we have

$$\mathsf{ert}\,[\![\texttt{diverge}]\!]\,(t) \;=\; \infty\,,$$

because the expected (and in fact certain) runtime of `diverge` is infinite.

Assignments behave similarly to `skip` in that they consume one unit of time, but in addition, they alter the program state: If we need, for instance,

x^2 units of time after an assignment $x := 5$ and the assignment itself consumes one unit of time, then the overall time needed to execute the assignment and then let time x^2 pass is

$$\mathsf{ert}\,[\![x := 5]\!]\left(x^2\right) \;=\; 1 + \left(x^2\right)[x/5] \;=\; 1 + 5^2 \;=\; 26\,.$$

Random assignments behave similarly to assignments: the overall time we need to execute the random assignment $x :\approx \mu$ and then let time t pass is 1 plus the expected value that t has after sampling a value from μ and assigning it to program variable x. This is expressed by the rule for random assignments. The rule for random assignments in Table 7.1 can thus be rewritten as

$$\begin{aligned}\mathsf{ert}\,[\![x :\approx \mu]\!]\,(t) \;&=\; 1 + \lambda\sigma.\ \int_{\mathsf{Vals}} \Big(\lambda v.\ t\big(\sigma\,[x \mapsto v]\big)\Big)\,d\,\mu_\sigma \quad \text{(see Table 7.1)}\\ &=\; 1 + \mathsf{wp}\,[\![x :\approx \mu]\!]\,(t)\,. \qquad\qquad\qquad\qquad\quad \text{(see Table 4.1)}\end{aligned}$$

For a more detailed explanation on the integral above, please refer to the explanation of $\mathsf{wp}\,[\![x :\approx \mu]\!]$ in Section 4.1.3 (right before Example 4.7).

The expected runtime transformer for probabilistic choice, defined as

$$\mathsf{ert}\,[\![\{\,C_1\,\}\,[p]\,\{\,C_2\,\}]\!]\,(t) \;=\; 1 \;+\; p \cdot \mathsf{ert}\,[\![C_1]\!]\,(t) \;+\; (1-p)\cdot \mathsf{ert}\,[\![C_2]\!]\,(t)\,,$$

also adds one unit of time for flipping the random coin with bias p and then averages the runtimes of the left and the right branch with weights p and $1-p$, respectively. Here (and at the rule for random assignments), ert enables reasoning about actual *expected runtimes* of probabilistic programs.

The expected runtime transformer for conditional choice, defined as

$$\begin{aligned}&\mathsf{ert}\,[\![\mathtt{if}\,(\varphi)\,\{\,C_1\,\}\,\mathtt{else}\,\{\,C_2\,\}]\!]\,(t)\\ &\quad=\; 1 \;+\; [\varphi]\cdot \mathsf{ert}\,[\![C_1]\!]\,(t) \;+\; [\neg\varphi]\cdot \mathsf{ert}\,[\![C_2]\!]\,(t)\,,\end{aligned}$$

is analogous to the one for probabilistic choice: It also adds one unit of time, here for evaluating the loop guard and conditionally jumping to the according branch which is to be executed next. Then it adds to that the runtimes of either the left or the right branch, depending on whether the guard φ evaluates to true or false.

Similarly to probabilistic or conditional choice, the nondeterministic choice transformer, defined as

$$\mathsf{ert}\,[\![\{\,C_1\,\}\,\square\,\{\,C_2\,\}]\!]\,(t) \;=\; 1 \;+\; \max\Big\{\mathsf{ert}\,[\![C_1]\!]\,(t), \mathsf{ert}\,[\![C_2]\!]\,(t)\Big\}\,,$$

also adds one unit of time for flipping the nondeterministic coin, but then chooses the pointwise maximum among the runtimes of the left and the right branch in order to model a demonic, i.e. worst–case, behavior. Reasoning with ert therefore means reasoning about the *worst–case expected runtime* of a program in the presence of nondeterministic choices.

The expected runtime transformer for the while loop $\texttt{while}(\varphi)\{C\}$ with respect to postruntime t is defined as a least fixed point, namely of the associated ert–characteristic function

$$\Phi_t(X) \;=\; 1 \;+\; [\neg\varphi]\cdot t \;+\; [\varphi]\cdot \mathsf{ert}\,[\![C]\!]\,(X)\;,$$

which informally captures the expected runtime of the „program“

```
if (φ){
    C;
    let time X pass
}else{
    let time t pass
}.
```

$\Phi_t\big(\Phi_t(X)\big)$ hence captures the expected runtime of the „program“

```
if (φ){
    C;
    let the time needed to execute the following program pass:
        if (φ){
            C;
            let time X pass
        }else{
            let time t pass
        }
}else{
    let time t pass
}.
```

By the Kleene Fixed Point Theorem (Theorem A.5) we have

$$\mathsf{lfp}\;\Phi_t \;=\; \sup\Big\{0,\quad \Phi_t(0),\quad \Phi_t\big(\Phi_t(0)\big),\quad \Phi_t\Big(\Phi_t\big(\Phi_t(0)\big)\Big),\quad \ldots\quad\Big\}\,.$$

From the above considerations, we obtain the intuition that if we plug for X the least fixed point of Φ_t, this captures precisely the expected time needed to iterate C as long as φ is true and then let time t pass. This is precisely what we would expect for the expected time needed to execute the loop $\texttt{while}(\varphi)\{C\}$ and then let time t pass.

Remark 7.3 (Our Runtime Model). Overall, we note that we assume a runtime model, where a `skip`, an assignment, a random assignment, evaluating the

guard of a conditional choice, evaluating the guard of a while loop, and flipping a nondeterministic coin, and flipping a random coin each consume one unit of time. Sequential composition of two programs, i.e. the ; operator itself, is assumed to not consume any time.

We would like to stress that *this runtime model is a design decision* for the sake of concreteness and simplicity. Our calculus can easily be adapted to capture alternative models, such as for instance the model where we count only the number of assignments in a program run, a model where the runtime of an assignment may depend on the complexity of the expression, or the model where only the number of loop iterations is of relevance.

Another design choice we made is that we consider *worst–case* expected runtimes with regard to nondeterministic choice. By using min instead of max in the rule for nondeterministic choice, we would obtain an angelic version of the ert calculus for reasoning about *best–case* expected runtimes. △

EXAMPLE 7.4 (ert–reasoning for Probabilistic Programs):
Consider the program $C_{7.4}$ given by

$$
\begin{aligned}
C_{7.4} \triangleright \quad & \{x := 2\}\,[1/2]\,\{x := 5\}\,; \\
& \texttt{if}\,(x > 3)\,\{\texttt{skip}\}\,\texttt{else}\,\{\texttt{skip}\,;\,\texttt{skip}\,;\,\texttt{skip}\}\,; \\
& \texttt{skip}
\end{aligned}
$$

and suppose we want to reason about the expected runtime of $C_{7.4}$, i.e. about ert $[\![C_{7.4}]\!]\,(0)$. Analogously to Chapter 4, we will use the annotation style

$$
\begin{aligned}
& /\!/\!/\ s' \\
& /\!/\!/\ s \\
& C \\
& /\!/\!/\ t
\end{aligned}
$$

to express the fact that $s = \text{ert}\,[\![C]\!]\,(t)$ and moreover that $s' = s$. It is thus more intuitive to read annotated programs from bottom to top, just like the ert transformer moves from the back to the front. Using this notation, we can annotate the program $C_{7.4}$ simply by applying the ert rules from Table 7.1 starting with 0 as postruntime as shown in Figure 7.2.

By these annotations, we have established ert $[\![C_{7.4}]\!]\,(0) = 6$. This tells us that from any initial state the execution of $C_{7.4}$ terminates within 6 computation steps in expectation.

```
/// 6
/// 1 + 1/2 · 6 + 1/2 · 4
{
    /// 6
    /// 2 + [2 > 3] · 2 + [2 ≤ 3] · 4
    x := 2
    /// 1 + [x > 3] · 2 + [x ≤ 3] · 4
} [1/2] {
    /// 4
    /// 2 + [5 > 3] · 2 + [5 ≤ 3] · 4
    x := 5
    /// 1 + [x > 3] · 2 + [x ≤ 3] · 4
};
/// 1 + [x > 3] · 2 + [x ≤ 3] · 4
if (x > 3) {
    /// 2
    skip
    /// 1
}else{
    /// 4
    skip
    /// 3
    skip
    /// 2
    skip
    /// 1
};
/// 1
skip
/// 0
```

Figure 7.2: Runtime annotations for Example 7.4.

7.4 SOUNDNESS AND COMPLETENESS

In the previous section, we have developed a calculus which is intended for reasoning about expected runtimes of probabilistic programs, but nowhere have we stated in which way this calculus *actually and formally* captures expected runtimes. In this section, we will thus state the relationship of the ert calculus to the notion of expected runtimes (Definition 6.1 A.) defined on the computation tree semantics of probabilistic programs introduced in Section 3.3.1. Furthermore, we will compare the ert calculus to a Floyd–Hoare–style logic developed by Hanne Riis Nielson for reasoning about runtimes of deterministic programs [Nie87].

7.4.1 *Relationship to Computation Tree Semantics*

For a given program C and initial state σ, we defined in Definition 6.1 A. operationally the (worst–case) expected runtime of a program C on input σ, denoted $\mathsf{ERT}\,[\![C]\!]_\sigma$, in terms of the corresponding computation tree of executing C on σ (cf. Definition 3.4). Recall that the configurations in the computation tree have a runtime counter which is increased with every computation step. This makes the runtime model that is assumed in the computation tree coincide with the runtime model that is assumed by the ert calculus.

It is important to note that $\mathsf{ERT}\,[\![C]\!]_\sigma$ indeed captures the expected runtime of C on σ and that this number is *not equal* to the expected value of a runtime–counting variable as discussed in Section 7.2. The intuitive reason is that $\mathsf{ERT}\,[\![C]\!]_\sigma$ captures the average length of the computation paths that occur *during* execution of C, whereas the expected value of a runtime–counting variable is with respect to the distribution over states reached *after* execution of C.

The ert calculus is sound and complete with respect to our operational computation tree model in the sense that expected runtimes defined on the computation tree are equal to expected runtimes obtained from the ert calculus:

THEOREM 7.5 (Operational Soundness and Completeness of ert): *Let $C \in \mathsf{pGCL}$ and $\sigma \in \Sigma$. Then $\mathsf{ert}\,[\![C]\!]\,(0)(\sigma)$ is the expected runtime of C on input σ, i.e.*

$$\mathsf{ERT}\,[\![C]\!]_\sigma \;=\; \mathsf{ert}\,[\![C]\!]\,(0)(\sigma)\,.$$

Proof. By induction on the structure of C. Q.E.D.

A similar result can be stated by comparing the runtimes obtained from the ert calculus to (maximal) expected rewards in operational reward MDPs (cf. Section 3.3.3), where all states (except for the final states) have reward 1.

Visiting a state (thereby collecting the reward of 1) then corresponds to consuming 1 unit of time. It can be shown that the results from ert–reasoning coincide with expected rewards in the operational MDP [Kam+16].

Operational MDPs can be employed to effectively perform bounded model checking for expected runtimes [Jan+16], whereas expected runtimes are not computable in general (cf. Part III). The disadvantage of working on an operational MDP, however, is that the initial program state σ needs to be fixed in general. ert, on the other hand, allows for *symbolic reasoning* on all initial states simultaneously.

7.4.2 *Relationship to Nielson's Hoare Logic for Runtimes*

In 1987, Hanne Riis Nielson presented a Floyd–Hoare–style logic for reasoning about runtimes of *deterministic* programs [Nie87]. Since our ert calculus can of course also be used to reason about deterministic programs and Hoare logics are closely related to weakest preconditions, it makes sense to compare the ert calculus to Nielson's Hoare logic. I would like to acknowledge that this comparison was mainly done by my colleague Christoph Matheja and I will therefore only briefly touch upon this comparison in this thesis.

Nielson's judgements, which we shall call *Nielson triples*, are of the form

$$\langle\, G \,\rangle\ C\ \langle\, t \Downarrow F \,\rangle$$

where $F \in \mathcal{P}(\Sigma)$ is a postcondition, $G \in \mathcal{P}(\Sigma)$ is a precondition and t is of type $\Sigma \to \mathbb{N}$. A *Nielson triple* $\langle\, G \,\rangle\ C\ \langle\, t \Downarrow F \,\rangle$ *is valid* iff there exists a constant $k \in \mathbb{N}$, such that from any initial state $\sigma \models G$ the program C terminates within at most $k \cdot t(\sigma)$ steps (i.e. in $\mathcal{O}(t)$ many steps) in a state $\tau \models F$.

Nielson also presents a proof system for proving validity of Nielson triples, which we shall call *Nielson logic*. Her rule for the `skip` statement, for instance, is an axiom in the proof system and reads

$$\frac{}{\langle\, F \,\rangle\ \texttt{skip}\ \langle\, 1 \Downarrow F \,\rangle}\ (\text{skip})\ .$$

So for any state satisfying precondition F, the program `skip` terminates in $\mathcal{O}(1)$ many steps in a state also satisfying F. In comparison to that, we have

$$\mathsf{ert}\,[\![\texttt{skip}]\!]\,(0)\ =\ 1+0\ =\ 1$$

For a more difficult example, consider her treatment of sequential composition. The according rule reads

$$\frac{\Big\langle\, G \wedge t_2' = u \,\Big\rangle\ C_1\ \Big\langle\, t_1 \Downarrow H \wedge t_2 \leq u \,\Big\rangle \qquad \Big\langle\, H \,\Big\rangle\ C_2\ \Big\langle\, t_2 \Downarrow F \,\Big\rangle}{\Big\langle\, G \,\Big\rangle\ C_1 \fatsemi C_2\ \Big\langle\, t_1 + t_2' \Downarrow F \,\Big\rangle}\ (\text{comp})\ ,$$

where u is an (implicitly) universally quantified fresh logical variable. The key to understanding this rule is that in a Nielson triple $\langle\, G \,\rangle\ C\ \langle\, t \Downarrow F \,\rangle$, the t

is evaluated in the initial state on which C is executed. The problem with sequential composition is now that t_2 is evaluated in the *intermediate state* on which C_2 is executed after termination of C_1, but we would like to obtain an expression on the entire time needed to execute $C_1 \fatsemi C_2$ evaluated in an *initial state* before executing C_1. The first premise

$$\langle\, G \wedge t_2' = u \,\rangle\, C_1 \,\langle\, t_1 \Downarrow H \wedge t_2 \leq u \,\rangle$$

ensures that starting from a state $\sigma \models G$, the execution of C_1 terminates within $t_1(\sigma)$ steps in an intermediate state π. The second premise

$$\langle\, H \,\rangle\, C_2 \,\langle\, t_2 \Downarrow F \,\rangle$$

ensures that from π, the execution of C_2 terminates within $t_2(\pi)$ steps in a final state $\tau \models F$. In total we thus need $t_1(\sigma) + t_2(\pi)$ steps to execute $C_1 \fatsemi C_2$.

So how can we measure t_2 in the initial state σ rather than in the intermediate state π? We have to *anticipate the value that t_2 will have after executing C_1 on σ* (cf. Section 2.3). This anticipated value is captured by t_2'. In Hoare logic, however, reasoning about anticipated values cannot be done without using universally quantified logical variables. This role is played by the u: it ensures that (an upper bound on) the value of t_2 measured in the intermediate state is expressible as t_2' measured in the initial state.

In contrast to Nielson's somewhat involved rule for sequential composition, ert reasoning for this case would simply read

$$\mathsf{ert}\,[\![C_1 \fatsemi C_2]\!]\,(0) \;=\; \mathsf{ert}\,[\![C_1]\!]\,\Big(\mathsf{ert}\,[\![C_2]\!]\,(0)\Big)\,.$$

The intuitive reason why Nielson's rule has to be so involved is that her logic does not allow for continuation passing on the level of runtimes: Nielson triples speak only about preruntimes but no postruntimes can be taken into account without encoding them into a logical postcondition.

As for the advantages of Nielson logic over our ert calculus, we note that ert does not allow for explicitly considering pre– or post*conditions*. It is, for instance, very well conceivable that the runtime of a program C is a very complicated expression, but that the runtime of C restricted only to those paths which start in some state $\sigma \models G$ and terminate in some state $\tau \models F$ is a very simple expression. Incorporating the possibility of explicitly restricting to given pre– and postconditions in ert is thus an interesting direction for future work.

With regard to soundness and completeness of the ert calculus relative to Nielson logic, we can state that our ert calculus conservatively extends Nielson's approach. More formally, we can state the following:

THEOREM 7.6 (ert vs. Nielson Logic[4] [Kam+16; Kam+18]):
Let $C \in$ pGCL be deterministic and $F, G \in \mathcal{P}(\Sigma)$. Then:

A. **Soundness:** *If $\langle G \rangle\, C\, \langle F \rangle$ is valid for total correctness, then*

$$\langle P \rangle\, C\, \langle \mathsf{ert}\,[\![C]\!]\,(0) \Downarrow F \rangle$$

is provable in Nielson logic.

B. **Completeness:** *If $\langle P \rangle\, C\, \langle t \Downarrow F \rangle$ is provable in Nielson logic, then there exists a constant $k \in \mathbb{N}$, such that for all initial states $\sigma \in \Sigma$*

$$\mathsf{ert}\,[\![C]\!]\,(0)(\sigma) \;\leq\; k \cdot t(\sigma)\,.$$

Intuitively, soundness of ert with respect to Nielson logic means that all ert judgements can be proven correct using Nielson logic. Completeness on the other hand states that for every runtime judgement provable in Nielson logic, we can make a runtime judgement at least as tight using the ert calculus.

7.5 HEALTHINESS CONDITIONS

Expected runtime transformers, just like the transformers we studied in Chapter 4, satisfy several so–called *healthiness conditions*, like continuity, monotonicity, etc., which can aid in concrete reasoning about probabilistic programs, for instance by forming a foundation for compositional reasoning. We will study some of these properties in this section.

7.5.1 *Continuity*

Just like for the expectation transformers we studied in Chapter 4, continuity perhaps the most fundamental property that runtime transformers enjoy because it ensures well–definedness of runtime transformers of while loops. A runtime transformer $\mathcal{T} : \mathbb{T} \to \mathbb{T}$ is called continuous iff for any chain of expectations $S = \{s_0 \preceq s_1 \preceq s_2 \preceq \ldots\} \subseteq \mathbb{T}$ we have

$$\mathcal{T}(\sup S) \;=\; \sup\, \mathcal{T}(S)\,;$$

see Definition A.2 for more details. The runtime transformers we have presented in this chapter are continuous:

THEOREM 7.7 (Continuity of ert [Kam+16; Kam+18]):
Let C be a pGCL *program. Then the associated expected runtime transformer* ert $[\![C]\!]$ *is continuous.*

Proof. By structural induction on C, see some Appendix for details. Q.E.D.

4 This theorem is mainly due to Christoph Matheja.

The importance of continuity for well–defined transformers of loops can be sketched as follows: For any loop–free program C, continuity of ert $[\![C]\!]$ ensures that the ert–characteristic function of the loop $\texttt{while}(\varphi)\{C\}$ (that has C as its loop body) is also continuous. This ensures by the Kleene fixed point theorem (Theorem A.5) that the characteristic function has a least fixed point, which in turn ensures that ert $[\![\texttt{while}(\varphi)\{C\}]\!]$ is well–defined. The fact that the transformer ert $[\![\texttt{while}(\varphi)\{C\}]\!]$ itself is also continuous ensures well–defined expected runtime transformers of nested loops.

7.5.2 *Cofeasibility and Preservation of ∞*

While *feasibility* of expectation transformers expresses that preexpectations cannot become *too large* (see Section 4.2.3), the *cofeasibility* (or constant propagation) property of runtime transformers states that preruntimes cannot become *too small*:

Theorem 7.8 (Cofeasibility of ert [Kam+16; Kam+18]):
Let $C \in \mathsf{pGCL}$, $t \in \mathbb{T}$ and $k \in \mathbb{R}^{\infty}_{\geq 0}$. Then

$$\mathsf{ert}\,[\![C]\!]\,(k+t) \;=\; k + \mathsf{ert}\,[\![C]\!]\,(t)\;.$$

Proof. By structural induction on C. Q.E.D.

Intuitively, if we definitely let at least some constant time k pass after executing C, then for executing C and then letting at least time k pass, we will in total also need at least time k. As a consequence of Theorem 7.8, we get

$$k \;\preceq\; t \qquad \text{implies} \qquad k \;\preceq\; \mathsf{ert}\,[\![C]\!]\,(t)\;,$$

wich rephrases Theorem 7.8 in a way that makes it look more like a *dual* of the original feasibility property (cf. Section 4.2.3).

A special case of cofeasibility is when we choose $k = \infty$. We then obtain the *preservation of* ∞ property:

Corollary 7.9 (Preservation of ∞ for ert [Kam+16; Kam+18]):
Let $C \in \mathsf{pGCL}$. Then

$$\mathsf{ert}\,[\![C]\!]\,(\infty) \;=\; \infty\;.$$

Intuitively this means that if after executing C we let infinitely much time pass, then prepending this with the execution of C cannot prevent that we will need infinite time. We can thus think of preservation as an analogon to Dijkstra's „Law of the Excluded Miracle" (cf. Section 4.2.2).

7.5.3 *Monotonicity*

Monotonicity is a fundamental property. A runtime transformer ert $[\![C]\!]$ is monotonic iff for any two runtimes $s, t \in \mathbb{T}$, we have that

$$s \preceq t \quad \text{implies} \quad \text{ert}\,[\![C]\!]\,(s) \;\preceq\; \text{ert}\,[\![C]\!]\,(t)\ ;$$

see Definition A.3. The ert transformers we have presented are monotonic:

THEOREM 7.10 (Monotonicity of ert [Kam+16; Kam+18]):
Let $C \in$ pGCL. Then the associated expected runtime transformer ert $[\![C]\!]$ *is monotonic. Furthermore, for any while loop and any postruntime, the associated* ert*–characteristic function is monotonic.*

Proof. Every continuous function is monotonic, see Theorem A.4. Q.E.D.

As for weakest preexpectations, monotonicity plays an important role for compositional reasoning: Imagine two programs C_1 and C_2 such that

$$\text{ert}\,[\![C_1]\!]\,(0) \;\preceq\; \text{ert}\,[\![C_2]\!]\,(0)\ ,$$

i.e. C_2 needs on average at least as long to execute as C_1. Then monotonicity ensures that if we put the components C_1 and C_2 into some context $C \fatsemi \ldots$, then we can be certain that

$$\text{ert}\,[\![C \fatsemi C_1]\!]\,(0) \;\preceq\; \text{ert}\,[\![C \fatsemi C_2]\!]\,(0)\ ,$$

since $\text{ert}\,[\![C \fatsemi C_i]\!]\,(0) = \text{ert}\,[\![C]\!]\,(\text{ert}\,[\![C_i]\!]\,(0))$, for $i \in \{1, 2\}$, and thus we can be certain that $C \fatsemi C_2$ needs on average at least as long to execute as $C \fatsemi C_1$.

7.5.4 *Affinity and Weakest Preexpectations*

We saw in the introduction to this chapter that using weakest preexpectations of runtime–counting variables for reasoning about expected runtimes is not sound, whereas using the ert calculus is sound. Nevertheless, we can also observe that $\text{ert}\,[\![C]\!]\,(t)$ is closely related to the expected value of t after executing C. Therefore, it seems natural that $\text{ert}\,[\![C]\!]\,(t)$ and $\text{wp}\,[\![C]\!]\,(t)$ are closely related, too. And indeed, at least for tame programs (recall Definition 3.1 E.) this connection can be made formal in the following way:

THEOREM 7.11 (Decomposition of ert [Olm+16; Kam+18]):
Let $C \in$ pGCL be tame and $t \in \mathbb{T}$. Then

$$\text{ert}\,[\![C]\!]\,(t) \;=\; \text{ert}\,[\![C]\!]\,(0) + \text{wp}\,[\![C]\!]\,(t)\ .$$

Proof. By structural induction on C. Q.E.D.

Thus, the expected time needed for executing C and then letting time t pass is the expected runtime of C plus the expected value of t. By linearity of wp (see Theorem 4.21 C.) we get that ert is an *affine map* for tame programs:

THEOREM 7.12 (Affinity of ert for Tame Programs):
Let $C \in \mathsf{pGCL}$ be tame, let $s, t \in \mathbb{T}$, and let $r \in \mathbb{R}^{\infty}_{\geq 0}$. Then

$$\mathsf{ert}\,[\![C]\!]\,(r \cdot s + t) \;=\; \mathsf{ert}\,[\![C]\!]\,(0) \;+\; r \cdot \mathsf{wp}\,[\![C]\!]\,(s) + \mathsf{wp}\,[\![C]\!]\,(t)\;.$$

Another interesting observation is that cofeasibility together with the relationship between ert and wp yield for tame programs a very short proof of the well–known fact that positive almost–sure termination (see Definition 6.1 B.) implies almost–sure termination (Definition 6.2):

THEOREM 7.13 (Positive A.–s. Termination implies A.–s. Termination):
Let $C \in \mathsf{pGCL}$ be tame and $\sigma \in \Sigma$ be any initial state. Then positive almost–sure termination of C on input σ implies almost–sure termination of C on input σ, i.e.

$$\mathsf{ert}\,[\![C]\!]\,(0)(\sigma) \;<\; \infty \quad \text{implies} \quad \mathsf{wp}\,[\![C]\!]\,(1)(\sigma) \;=\; 1\;.$$

Moreover, universal positive almost–sure termination of C implies universal almost–sure termination of C, i.e.

$$\mathsf{ert}\,[\![C]\!]\,(0) \;\ll\; \infty \quad \text{implies} \quad \mathsf{wp}\,[\![C]\!]\,(1) \;=\; 1\;.$$

Proof (adapted from* [Olm+16][5]*). Consider the following:

$$\begin{aligned}
& \mathsf{ert}\,[\![C]\!]\,(1) \;=\; \mathsf{ert}\,[\![C]\!]\,(1) \\
\text{iff}\quad & \mathsf{ert}\,[\![C]\!]\,(1) \;=\; \mathsf{ert}\,[\![C]\!]\,(0) + 1 && \text{(by cofeasibility, Theorem 7.8)} \\
\text{iff}\quad & \mathsf{ert}\,[\![C]\!]\,(0) + \mathsf{wp}\,[\![C]\!]\,(1) \;=\; \mathsf{ert}\,[\![C]\!]\,(0) + 1 \\
& \qquad \text{(by tameness of } C \text{ and decomposition of ert, Theorem 7.11)} \\
\text{implies}\quad & \mathsf{ert}\,[\![C]\!]\,(0)(\sigma) + \mathsf{wp}\,[\![C]\!]\,(1)(\sigma) \;=\; \mathsf{ert}\,[\![C]\!]\,(0)(\sigma) + 1 \\
\text{iff}\quad & \mathsf{wp}\,[\![C]\!]\,(1)(\sigma) \;=\; 1 && (\text{by } \mathsf{ert}\,[\![C]\!]\,(0)(\sigma) < \infty) \\
\text{iff}\quad & C \text{ terminates almost surely on } \sigma
\end{aligned}$$

The universal version of the theorem follows immediately from the above proof by requiring $\mathsf{ert}\,[\![C]\!]\,(0)(\sigma) < \infty$ for all $\sigma \in \Sigma$. Q.E.D.

7.6 PROOF RULES FOR LOOPS

REASONING about loops is one of the most — if not *the* most — difficult tasks in probabilistic program verification. We have seen in Chapter 5,

5 The idea for this short proof is mainly due to Federico Olmedo.

how invariants can help with this sort of reasoning. In particular, we saw that invariants precisely capture the principles of induction and coinduction.

In this section, we will show how we can reason about expected runtimes of loops by means of *runtime invariants*, which basically capture the same notion of invariance as for weakest preexpectations. We will present inductive methods for proving upper bounds on expected runtimes of loops as well as ω–rules for proving lower bounds. We will also discuss coinductive premises for obtaining *lower bounds* on runtimes of deterministic programs and how this method fails on probabilistic programs. Furthermore, we will discuss runtime–bound refinement as well as a method for obtaining *exact* expected runtimes of *independent and identically distributed* loops.

7.6.1 *Invariants*

The concept of invariants that we employ for the proof rules we present in this section is the same as for weakest preexpectation reasoning, see Section 5.1. The notion of a runtime invariant is defined as follows:

Definition 7.14 (Runtime Invariants [Kam+16; Kam+18]):
Let Φ_t be the ert*–characteristic function of* `while`$(\varphi)\{C\}$ *with respect to postruntime $t \in \mathbb{T}$ and let $I \in \mathbb{T}$. Then:*

A. *I is called a runtime invariant of* `while`$(\varphi)\{C\}$ *with respect to postruntime t, iff*

$$\Phi_t(I) \preceq I \,.$$

B. *I is called a runtime subinvariant of* `while`$(\varphi)\{C\}$ *with respect to postruntime t, iff*

$$I \preceq \Phi_t(I) \,.$$

Next, we introduce a concept we call *runtime ω–subinvariants*. These are basically sequences of expectations that are runtime subinvariants relative to each other. We will make use of those for reasoning about lower bounds on expected runtimes.

Definition 7.15 (Runtime ω–subinvariants [Kam+16; Kam+18]):
Let Φ_t be the ert*–characteristic function of* `while`$(\varphi)\{C\}$ *with respect to postruntime $t \in \mathbb{T}$ and let $(I_n)_{n\in\mathbb{N}} \subset \mathbb{T}$ be a monotonically increasing*[6] *sequence with $I_0 = 0$.*

Then $(I_n)_{n\in\mathbb{N}}$ is called a runtime ω–subinvariant of `while`$(\varphi)\{C\}$ *with respect to postexpectation t, iff*

$$\forall\, n \in \mathbb{N}\colon \quad I_{n+1} \preceq \Phi_t(I_n) \,.$$

6 But not necessarily *strictly* increasing.

7.6.2 *Induction*

Since expected runtimes are defined as least fixed points of continuous functions on complete lattices, we can make use of the induction principle that we discussed in Section 5.2.1 in order to reason about upper bounds on expected runtimes. Formally, the induction principle states that if $(D, \sqsubseteq)$ is a complete lattice and $\Phi\colon D \to D$ is a continuous self–map on D, then

$$\forall\, d \in D\colon \quad \Phi(d) \sqsubseteq d \quad \text{implies} \quad \mathsf{lfp}\,\Phi \sqsubseteq d\,.$$

Applied to the ert calculus, the induction principle immediately gives us the following proof rule:

THEOREM 7.16 (Induct. for Upper Bounds on ert [Kam+16; Kam+18]): *Let $I \in \mathbb{T}$ be a runtime invariant of $\texttt{while}(\varphi)\{C\}$ with respect to postruntime t (see Definition 7.14 A.). Then*

$$\mathsf{ert}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(t) \preceq I\,.$$

Proof. This is an instance of Park's Lemma (see Lemma A.6): Simply choose complete lattice $(\mathbb{T}, \preceq)$ and continuous function ${}^{\mathsf{ert}}_{\langle\varphi,C\rangle}\Phi_t$. Q.E.D.

Before we proceed with an example, we note that the induction rule is *complete*, since $\mathsf{ert}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(t)$ itself is trivially a runtime invariant of $\texttt{while}(\varphi)\{C\}$ with respect to postruntime t.

EXAMPLE 7.17 (Upper Bounds on ert):
Recall the program C from Section 7.1, given by

$$\begin{array}{l}\texttt{while}(x>0)\{\\ \quad \{x := x-1\}\,[1/2]\,\{\texttt{skip}\}\\ \}\,,\end{array}$$

where for simplicity we assume that x ranges over natural numbers only. Suppose we want to reason about its expected runtime using the ert calculus, i.e. we would like to reason about $\mathsf{ert}\,[\![C]\!]\,(0)$. To this end, we propose the runtime invariant

$$I \;=\; 1 + 6x$$

and check its invariance by applying the ert–characteristic function of the while loop with respect to postruntime 0, given by

$$\begin{aligned}\Phi(X) \;&=\; 1 \;+\; [x \le 0]\cdot 0 \;+\; [x>0]\cdot \mathsf{ert}\,[\![\{x := x-1\}\,[1/2]\,\{\texttt{skip}\}]\!]\,(X)\\ &=\; 1 \;+\; [x>0]\cdot \mathsf{ert}\,[\![\{x := x-1\}\,[1/2]\,\{\texttt{skip}\}]\!]\,(X)\end{aligned}$$

$$
\begin{aligned}
&= 1 \;+\; [x>0]\cdot\Big(1+\frac{1}{2}\cdot \mathsf{ert}\,[\![x := x-1]\!]\,(X)+\frac{1}{2}\cdot \mathsf{ert}\,[\![\mathtt{skip}]\!]\,(X)\Big)\\
&= 1 \;+\; [x>0]\cdot\Big(1+\frac{1}{2}\cdot\big(1+X\,[x/x-1]\big) \;+\; \frac{1}{2}\cdot\big(1+X\big)\Big)\\
&= 1 \;+\; [x>0]\cdot\Big(2+\frac{1}{2}\cdot\big(X\,[x/x-1]+X\big)\Big)\,,
\end{aligned}
$$

to $I = 1+6x$, which gives us

$$
\begin{aligned}
\Phi(I) \;&=\; \Phi(1+6x)\\
&= 1 \;+\; [x>0]\cdot\Big(2+\frac{1}{2}\cdot\big((1+6x)\,[x/x-1]+1+6x\big)\Big)\\
&= 1 \;+\; [x>0]\cdot\Big(2+\frac{1}{2}\cdot\big(1+6(x-1)+1+6x\big)\Big)\\
&= 1 \;+\; [x>0]\cdot\Big(2+\frac{1}{2}\cdot\big(12x-4\big)\Big)\\
&= 1 \;+\; [x>0]\cdot 6x\\
&= 1 \;+\; 6x \;=\; I \;\preceq\; I\,.
\end{aligned}
$$

Thus the induction rule for ert (Theorem 7.16) gives us that

$$\mathsf{ert}\,[\![\mathtt{while}\,(\ldots)\}]\!]\,(0) \;\preceq\; 1+6x\,.$$

and hence the loop needs on average at most $1+6x$ steps until it terminates.

As for an intuitive explanation, the loop needs to check whether $x>0$ at least once, hence the $1+$ ⋯. If $x>0$, then the loop is iterated $2x$ times in expectation and in each iteration 3 steps are performed: (1) flipping a fair coin, (2) either performing an assignment or a `skip`, and (3) rechecking the loop guard. In total, we thus get $1+2x\cdot 3 = 1+6x$ computation steps in expectation.

Our runtime invariants are closely related to the notion of *ranking functions* for deterministic programs (cf. Section 6.1 and [Fro+16b]). A ranking function R for a loop $\mathtt{while}(\varphi)\{C\}$ maps program states to real numbers and satisfies the following two constraints for every state σ:

A. If $\sigma \models \varphi$, then execution of C on σ terminates in a state τ such that

$$R(\tau) \;\preceq\; R(\sigma)-\epsilon\,,$$

for some fixed $\epsilon > 0$, and

B. if $\sigma \models \varphi$, then $R(\sigma) > 0$.

So from any state satisfying the loop guard, the execution of the loop body decreases the ranking by at least ϵ, and as long as the ranking is above 0, the loop guard is true. Thus, if the ranking hits 0 or drops below, this falsifies the loop guard and causes the loop to terminate.

We can note that the 0 in condition B. is arbitrary and we can also choose 1 as lower threshold instead of 0. We just have to fix some threshold such that if the ranking drops below that threshold, the loop terminates.

Using the threshold 1, we can translate the above two conditions into the setting of anticipated value reasoning (see Section 2.3), which then reads

A. $[\varphi]\,\mathsf{wp}\,[\![C]\!]\,(R) \preceq R - \epsilon$, for some fixed $\epsilon > 0$, and[7]

B. $[\varphi] \preceq [R > 1]$.

Runtime invariants resemble a very similar behavior: A runtime I is a runtime invariant of $\texttt{while}(\varphi)\{C\}$ with respect to postruntime 0, iff

$$1 + [\varphi] \cdot \mathsf{ert}\,[\![C]\!]\,(I) \preceq I\,. \tag{†}$$

This can be rewritten as

$$[\varphi] \cdot \mathsf{ert}\,[\![C]\!]\,(I) \preceq I - 1\,,$$

which looks closely related to condition A. above when fixing $\epsilon = 1$.

Furthermore, it follows from † that $1 \preceq I$ and therefore, by the *cofeasibility property*[8] of ert (see Theorem 7.8), we can also deduce that $1 \preceq \mathsf{ert}\,[\![C]\!]\,(I)$. From the latter, it follows that

$$[\varphi] \preceq [I > 1]\,.$$

This is because $\sigma \models \varphi$ implies

$$I \succeq 1 + \mathsf{ert}\,[\![C]\!]\,(I) \succeq 1 + 1 = 2\,.$$

In total we get that runtime invariants satisfy the following two conditions:

A. $[\varphi]\,\mathsf{ert}\,[\![C]\!]\,(I) \preceq I - 1$, and

B. $[\varphi] \preceq [I > 1]$.

Since ert and wp are closely connected (see Theorem 7.11), we can intuitively think of a runtime invariant as follows: From any state satisfying the loop guard, the execution of the loop body decreases the runtime invariant by at least $\epsilon = 1$ *in expectation*, and as long as the ranking is above 1, the loop guard is true. Thus, if the runtime invariant hits 1, this falsifies the loop guard and causes the loop to terminate.

Besides the fact that every runtime invariant satisfies the above two conditions, the converse is also true: Condition A. alone is equivalent to † and thus to the fact that I is a runtime invariant of $\texttt{while}(\varphi)\{C\}$ with respect to

7 We tacitly assume here that wp could handle functions R that may map into negative values.

8 This property states that $\mathsf{ert}\,[\![C]\!]\,(k + t) = k + \mathsf{ert}\,[\![C]\!]\,(t)$ for any constant $k \in \mathbb{R}^{\infty}_{\geq 0}$.

postruntime t. This demonstrates the close connection of runtime invariants and ranking functions.

The remarks above also demonstrate the close relationship of runtime invariants to *ranking supermartingales* for proving positive almost–sure termination (cf. Theorem 6.3): Ranking supermartingales are essentially ranking functions that decrease by ϵ *in expectation*. A runtime invariant can thus be thought of as a ranking supermartingale wich decreases in expectation by $\epsilon = 1$. We also notice that the constant $K > 0$ that was needed for the ranking supermartingale reasoning of Theorem 6.3 in order not to have ranking supermartingales map into negative numbers is not needed for runtime invariants, which renders runtime invariants conceptually easier. Since runtime invariants are complete for reasoning about expected runtimes[9], there is no case where a ranking supermartingale but no runtime invariant exits.

7.6.3 *Coinduction for Deterministic Programs*

The runtime invariants we introduced in Section 7.6.1 are in the spirit of inductive superinvariants as introduced in Section 5.1, whereas runtime subinvariants are in the spirit of coinductive subinvariants. Runtime subinvariants are closely related to the notion of *metering functions* for deterministic programs (see [Fro+16b]). A metering function M for a loop $\texttt{while}(\varphi)\{C\}$ maps program states to real numbers and satisfies the following two constraints for every state σ:

A. If $\sigma \models \varphi$, then execution of C on σ terminates in a state τ such that

$$R(\tau) \;\geq\; R(\sigma) - 1\,, \quad \text{and}$$

B. if $\sigma \not\models \varphi$, then $R(\sigma) \leq 0$.

So from any state satisfying the loop guard, the execution of the loop body decreases the metering by *at most* 1, and as as soon as the guard is false, the metering hits 0 or drops below. Thus, as long as the metering is larger than 0, the loop cannot terminate.

Similarly to the situation with ranking functions discussed in the previous section, we can again note that the 0 in condition B. is arbitrary and we can also choose 1 as lower threshold instead of 0. We just have to fix some value such that if the ranking drops below that value, the loop terminates.

Using threshold 1 instead of 0, we can translate conditions A. and B. above to the setting of anticipated value reasoning (see Section 2.3), which reads

A. $[\varphi]\,\mathsf{wp}\,[\![C]\!]\,(R) \;\geq\; R - 1$, and[10]

B. $[\neg\varphi] \;\leq\; [R \leq 1]$.

9 Though not in an algorithmic sense, since expected runtimes are in general not computable.
10 We tacitly assume here that wp could handle functions R that may map into negative values.

Runtime subinvariants resemble a very similar behavior: A runtime I is a runtime subinvariant of $\texttt{while}(\varphi)\{C\}$ with respect to postruntime 0, iff

$$1 + [\varphi] \cdot \mathsf{ert}\,[\![C]\!]\,(I) \;\geq\; I\,. \tag{†}$$

This can be rewritten as

$$[\varphi] \cdot \mathsf{ert}\,[\![C]\!]\,(I) \;\geq\; I - 1\,,$$

which looks closely related to condition A. above when fixing $\epsilon = 1$.

Furthermore, it follows from † that $1 \geq I(\sigma)$ if $\sigma \not\models \phi$, which can be expressed as $[\neg\varphi] \leq [I \leq 1]$. In total we get that runtime subinvariants satisfy the following two conditions:

A. $[\varphi]\,\mathsf{ert}\,[\![C]\!]\,(I) \;\geq\; I - 1$, and

B. $[\neg\varphi] \;\leq\; [I \leq 1]$.

Since ert and wp are closely connected (see Theorem 7.11), we can intuitively think of a runtime subinvariant as follows: From any state satisfying the loop guard, the execution of the loop body decreases the runtime invariant by at most $\epsilon = 1$ *in expectation*, and as soon as the loop guard becomes false, the subinvariant hits 1 or drops below.

Besides the fact that every runtime subinvariant satisfies the above two conditions, the converse is also true: Condition A. alone is equivalent to † and thus to the fact that I is a runtime subinvariant of $\texttt{while}(\varphi)\{C\}$ with respect to postruntime t. This demonstrates the close connection of runtime subinvariants and metering functions.

While we learned in Chapter 5 that subinvariants are in general not suitable for reasoning about lower bound on weakest preexpectations, it was proven by Frohn *et al.*, that if M is a metering function of a loop $\texttt{while}(\varphi)\{C\}$ then the loops needs *at least* $M(\sigma)$ iterations to terminate on input σ. Hence, metering functions do prove *lower bounds on runtimes of deterministic programs by means of a coinductive premise* [Fro+16b]. The same holds for runtime subinvariants in case of deterministic loops:

THEOREM 7.18 (Coinduction for Lower Bounds on ert[11]):
Let $\texttt{while}(\varphi)\{C\}$ *be a loop with deterministic loop body and let* $I \in \mathbb{T}$ *be a runtime subinvariant of* $\texttt{while}(\varphi)\{C\}$ *with respect to postruntime* 0 *(see* Definition 7.14 B.*). Then*

$$I \;\leq\; \mathsf{ert}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(0)\,.$$

Proof (inspired by* [Fro+16a]*). We proceed by induction on the number n of *guarded* loop iterations needed until termination. A single guarded loop

11 The idea for this theorem was developed together with Florian Frohn and Christoph Matheja.

iteration comprises of (1) checking whether the loop guard φ evaluates to true or false and (2a) either terminating the loop in case φ evaluates to false or (2b) executing the loop body C once in case φ evaluates to true.

Let σ be some arbitrary but fixed initial state. If $\texttt{while}(\varphi)\{C\}$ needs n guarded loop iterations until it terminates on σ, then

$$\Big(\mathsf{lfp}\ \Phi\Big)(\sigma) \;=\; \Big(\Phi^n(0)\Big)(\sigma)\,, \tag{‡}$$

where Φ is the ert–characteristic function of the loop with respect to post-runtime 0. We will proceed by induction on n to show that

$$I(\sigma) \;\preceq\; \Big(\Phi^n(0)\Big)(\sigma)\,.$$

A first case we have to consider is when $n = \omega$, i.e. the loop does *not* terminate on σ and the time needed to execute the loop is ∞. Then $I(\sigma)$ is obviously a lower bound on the units of time needed to execute $\texttt{while}(\varphi)\{C\}$ on σ, as $I \in \mathbb{T}$ and thus $I \preceq \infty$.

We now proceed with the actual induction: For the *induction base*, we have $n = 1$. This means that exactly one guarded loop iteration is performed, i.e. the loop must immediately terminates after checking the loop guard. This implies that the loop guard must have evaluated to false in the initial state σ, since otherwise at least one more guarded iteration would be performed. Since $\sigma \not\models \varphi$, we have

$$\Big(\Phi(0)\Big)(\sigma) \;=\; \Big(1 + [\varphi]\cdot \mathsf{ert}\,[\![C]\!]\,(0)\Big)(\sigma) \;=\; 1\,.$$

Furthermore,

$$\begin{aligned}
I(\sigma) \;&\preceq\; \Big(\Phi(I)\Big)(\sigma) && \text{(by subinvariance of } I\text{)}\\
&=\; \Big(1 + [\varphi]\cdot \mathsf{ert}\,[\![C]\!]\,(I)\Big)(\sigma) && \text{(by Definition of } \Phi\text{)}\\
&=\; 1\,. && \text{(by } \sigma \not\models \varphi\text{)}
\end{aligned}$$

Hence we get

$$I(\sigma) \;\preceq\; 1 \;=\; \Big(\Phi^n(0)\Big)(\sigma)\,,$$

which completes the proof for the induction base.

Now suppose — as our *induction hypothesis* — that whenever $\texttt{while}(\varphi)\{C\}$ terminates on some arbitrary but fixed initial state τ within n guarded loop iterations, then we have

$$I(\tau) \;\preceq\; \Big(\Phi^n(0)\Big)(\tau)\,.$$

As our *induction step*, assume that $\texttt{while}(\varphi)\{C\}$ terminates on some arbitrary but fixed initial state σ within $n+1$ guarded loop iterations. Then

$\texttt{while}(\varphi)\{C\}$ performs at least one guarded loop iteration and moreover $\sigma \models \varphi$. This single guarded iteration alone hence needs

$$1 + \mathsf{ert}\,[\![C]\!]\,(0)(\sigma) \;=\; \Big(1 + [\varphi]\cdot \mathsf{ert}\,[\![C]\!]\,(0)\Big)(\sigma)\,,$$

units of time. Suppose — without loss of generality — that the loop body C terminates on σ in state τ. Thereafter, the loop needs by assumption of the induction step n more guarded loop iterations until it terminates on τ. By induction hypothesis, the entire execution of $\texttt{while}(\varphi)\{C\}$ on σ takes *at least*

$$\Big(1 + [\varphi]\cdot \mathsf{ert}\,[\![C]\!]\,(0)\Big)(\sigma) + I(\tau)$$

units of time. We can now express $I(\tau)$ in σ by *anticipating the value of I in τ*. This gives (cf. Section 2.3)

$$I(\tau) \;=\; \Big(\mathsf{wp}\,[\![C]\!]\,(I)\Big)(\sigma)\,.$$

The entire execution of $\texttt{while}(\varphi)\{C\}$ on σ hence takes *at least*

$$\begin{aligned}
&\Big(1 + [\varphi]\cdot \mathsf{ert}\,[\![C]\!]\,(0)\Big)(\sigma) + I(\tau)\\
&= \Big(1 + [\varphi]\cdot \mathsf{ert}\,[\![C]\!]\,(0)\Big)(\sigma) + \Big(\mathsf{wp}\,[\![C]\!]\,(I)\Big)(\sigma)\\
&= \Big(1 + [\varphi]\cdot \mathsf{ert}\,[\![C]\!]\,(0) + \mathsf{wp}\,[\![C]\!]\,(I)\Big)(\sigma)\\
&= \Big(1 + [\varphi]\cdot \mathsf{ert}\,[\![C]\!]\,(I)\Big)(\sigma) \qquad \text{(by decomposition of ert, Theorem 7.11)}\\
&= \Big(\Phi(I)\Big)(\sigma) \qquad \text{(by definition of } \Phi)\\
&\geq I(\sigma) \qquad \text{(by subinvariance of } I)
\end{aligned}$$

units of time. We have thus proven by induction on n that

$$\begin{aligned}
I(\sigma) \;&\leq\; \Big(\Phi^n(0)\Big)(\sigma) \qquad \text{(by induction)}\\
&\leq\; \Big(\mathsf{lfp}\;\Phi\Big)(\sigma) \qquad \text{(by } \ddagger \text{ above)}
\end{aligned}$$

for all σ and hence

$$I \;\preceq\; \mathsf{lfp}\;\Phi \;=\; \mathsf{ert}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(0)\,. \qquad \text{Q.E.D.}$$

EXAMPLE 7.19 (Lower Bounds on ert for Deterministic Programs): Consider the program C, given by

$$\begin{array}{l}
\texttt{while}(x > 0)\{\\
\quad x := x - 1\\
\}\,,
\end{array}$$

where for simplicity we assume that x ranges over natural numbers only. Suppose we want to prove that C needs at least linear runtime. To this end, we propose the runtime subinvariant

$$I \;=\; x$$

and check its subinvariance by applying the ert–characteristic function of the while loop with respect to postruntime 0, given by

$$\begin{aligned}\Phi(X) \;&=\; 1 \;+\; [x \leq 0] \cdot 0 \;+\; [x > 0] \cdot \mathsf{ert}\,[\![x := x-1]\!]\,(X)\\ &=\; 1 \;+\; [x > 0] \cdot \mathsf{ert}\,[\![x := x-1]\!]\,(X)\\ &=\; 1 \;+\; [x > 0] \cdot \Big(1 + X\,[x/x-1]\Big)\end{aligned}$$

to I, which gives us

$$\begin{aligned}\Phi(I) \;&=\; \Phi\Big(x\Big)\\ &=\; 1 \;+\; [x > 0] \cdot \Big(1 + x\,[x/x-1]\Big)\\ &=\; 1 \;+\; [x > 0] \cdot (1 + x - 1)\\ &=\; 1 \;+\; [x > 0] \cdot x\\ &=\; 1 \;+\; x\\ &\geq\; x \;=\; I\,.\end{aligned}$$

Thus, the coinduction rule for ert (Theorem 7.18) gives us that

$$x \;\preceq\; \mathsf{ert}\,[\![\texttt{while}\,(\ldots)\}]\!]\,(0)\,.$$

and hence the loop needs at least x steps until it terminates.

Theorem 7.18 shows that for deterministic loops, we can use a *coinductive premise*, i.e. a premise of the form $I \preceq \Phi(I)$, in order to establish a lower bound on a least fixed point — in this particular case: on the runtime of a loop. As a consequence, this allows for basically *guessing* a runtime I, checking whether (a) $I \preceq \Phi(I)$ or (b) $\Phi(I) \preceq I$. In case of (a) I is a lower bound on the runtime and in case of (b) I is an upper bound. However, since $\preceq$ is only a partial order, nothing ensures that either case (a) or (b) occurs, i.e. I and $\Phi(I)$ might very well be *incomparable*.

7.6.4 *No Coinduction for Probabilistic Programs*

It is worthwhile to point out where the proof we gave for Theorem 7.18 fails in case of probabilistic loops. The point where this happens is at the very beginning of the proof where we stated that for deterministic programs, there

exists some $n \in \mathbb{N}$, such that

$$\big(\mathsf{lfp}\ \Phi\big)(\sigma) \;=\; \big(\Phi^n(0)\big)(\sigma)\,,$$

in case that the loop terminates on σ. This is *not true* for probabilistic programs: We may well have the situation that we need $n = \omega$, so that

$$\big(\mathsf{lfp}\ \Phi\big)(\sigma) \;=\; \big(\Phi^{\omega}(0)\big)(\sigma)\,,$$

even for a fixed initial state σ. Indeed, for probabilistic programs, the metering function approach is unfortunately unsound. The following counterexample shows that runtime subinvariants cannot provide lower bounds on expected runtimes in general:

COUNTEREXAMPLE 7.20 (Unsoundness of Coinduction for ert):
Consider the program C, given by

$$\begin{array}{l} \texttt{while}(c = 1)\{ \\ \qquad \{c := 0\}\,[1/2]\,\{x := x+1\} \\ \}\,, \end{array}$$

where we assume that x ranges over $\mathbb{N}$ for simplicity. Suppose we want to incorrectly reason about a lower bound on the expected runtime of C by coinduction. The ert–characteristic function of the while loop with respect to postruntime 0 is given by

$$\Phi(X) \;=\; 1 \;+\; [c = 1] \cdot \left(2 + \frac{1}{2}\Big(X\,[c/0] + X\,[x/x+1]\Big)\right).$$

We now propose *infinitely many* fixed points (thus also runtime subinvariants) of Φ, namely for every $a > 0$

$$I_a \;=\; x + [c = 1](6 + 2^{x+a})$$

is a fixed point of Φ, as one can easily check. However, for any $d < b$, we clearly have $I_d \prec I_b$. Thus, if we prove $I_b \preceq \Phi(I_b)$ we cannot have proven that I_b is a lower bound on the least fixed point of Φ, since I_d is a fixed point strictly smaller than I_b. In fact, none of the I_a's are the least fixed point of Φ. The intuitive reason is that the expected runtime of C is independent of x, but x has an influence on the value that the I_a's assume. ↯

Frohn *et al.*'s proof of soundness of the metering function approach [Fro+16a] goes by induction on the number of loop iterations, which are natural numbers for deterministic programs. Expected runtimes or expected numbers of loop iterations, however, need not be natural numbers, thus standard induction is not a viable method in the realm of expected runtimes. An important

	$I \preceq \Phi_t(I)$	$\Phi_t(I) \preceq I$
***C* deterministic**	$I \preceq \mathsf{ert}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(t)$	$\mathsf{ert}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(t) \preceq I$
***C* probabilistic**	— ? —	$\mathsf{ert}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(t) \preceq I$

Table 7.2: Rules for proving upper bounds and lower bounds on (expected) runtimes of deterministic and probabilistic loops.

direction for future work is thus to understand what method instead of standard induction on the natural numbers could help for obtaining proof rules for lower bounds on expected runtimes.

To summarize, we have the situation described in Table 7.2. If I is a *superinvariant*, then I is an *upper bound* on the (expected) runtime of the loop, regardless of whether the loop is deterministic or probabilistic. If I is a *subinvariant* and the loop is *deterministic*, then I is a *lower bound* on the loop's runtime. If I is a *subinvariant* and the loop is *probabilistic*, on the other hand, then we know *nothing* about the loop's expected runtime. Thus, the following problem in probabilistic program verification remains open:

> **OPEN PROBLEM 2 (One–shot Verification of Lower Bounds on ert):** *Find a „one–shot" method as elegant as the induction or coinduction rule (*Theorems 7.16 *and Frohn* et al.*'s metering functions), which, given a loop* $\texttt{while}(\varphi)\{C\}$ *and a* specific *hypothesis* $L \in \mathbb{T}$*, allows for checking whether* L *is in fact a lower bound on* $\mathsf{ert}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(0)$.

7.6.5 *ω–Rules*

As is the case for weakest preexpectations, reasoning about lower bounds of expected runtimes is difficult since no inductive or coinductive proof principle is available. For weakest preexpectation reasoning, we thus resorted to so–called ω–rules which employ ω–invariants (see Section 5.2.4). The same principle is applicable to expected runtime reasoning:

> **THEOREM 7.21 (Low. Bounds on ert from ω–Inv.** [Kam+16; Kam+18]**):** *Let* $(I_n)_{n\in\mathbb{N}} \subset \mathbb{T}$ *be a runtime* ω*–subinvariant of* $\texttt{while}(\varphi)\{C\}$ *with respect to postruntime* t *(see* Theorem 7.21*). Then*
>
> $$\sup_{n\in\mathbb{N}} I_n \preceq \mathsf{ert}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(t)\,.$$

Proof. Analogous to the proof of Theorem 5.9 A. Q.E.D.

Just like for weakest preexpectations, it is necessary to find the limit of such an ω–invariant in order to actually gain some insights from applying Theorem 7.21. That basically just shifts to problem of obtaining bounds into the realm of real analysis. For further remarks on the — in my personal opinion — poor usability and usefulness and on the expendability of ω–rules for upper bounds, see the remarks on ω–rules for weakest preexpectation reasoning in Section 5.2.4.

7.6.6 *Bound Refinement*

We saw in Section 5.2.7 that once we have obtained by some means some bound — be it upper or lower — on a preexpectation of a loop, we have a chance of refining and thereby tightening this bound fairly easily. Since this technique is rooted in fixed point theory and expected runtimes of loops are defined as least fixed points, the same technique can be applied to ert:

> THEOREM 7.22 **(Bound Refinement for ert** [Kam+16; Kam+18]**):**
> *Let Φ be the* ert*–characteristic function of* $\texttt{while}(\varphi)\{C\}$ *with respect to postruntime t and let I be an upper bound on* ert $[\![\texttt{while}(\varphi)\{C\}]\!](t)$*, such that $\Phi(I) \preceq I$.*
>
> *Then $\Phi(I)$ is also an upper bound on* ert $[\![\texttt{while}(\varphi)\{C\}]\!](t)$*. Moreover, whenever $\Phi(I) \neq I$, then $\Phi(I)$ is an even tighter upper bound on* ert $[\![\texttt{while}(\varphi)\{C\}]\!](t)$ *than I.*
>
> *Dually, if I is a lower bound, such that $I \preceq \Phi(I)$, then $\Phi(I)$ is also a lower bound; and whenever $\Phi(I) \neq I$, then $\Phi(I)$ is an even tighter lower bound than I.*

Proof. Analogous to the proof of Theorem 5.15. Q.E.D.

The particular bound refinement of Theorem 7.22 can of course be continued ad infinitum: For instance, if I is an upper bound on ert $[\![\texttt{while}(\varphi)\{C\}]\!](t)$ with $\Phi(I) \preceq I$, then so is $\Phi(I)$ but also $\Phi^2(I)$, $\Phi^3(I)$, and so on. In fact, for increasing n, the sequence $\Phi^n(I)$ is decreasing and *converges* to an upper bound on ert $[\![\texttt{while}(\varphi)\{C\}]\!](t)$ that is *below (or equal to) I*. For more details, see Section 5.2.7.

7.6.7 *Independent and Identically Distributed Loops*

We have learned in the previous sections that, similarly to the case for weakest preexpectations (see Chapter 5), obtaining bounds — especially lower bounds — on expected runtimes of while loops can be a very difficult task. Obtaining exact expected runtimes obviously cannot be any easier in principle. Under certain conditions, however, we are able to derive the *exact*

expected runtime of a while loop with respect to a given postruntime. Informally, these conditions can be described as follows:

1. For each loop iteration, the probability to immediately terminate after that iteration is equal.
2. There is *no information* flow across different loop iterations with respect to any program variable that has an influence on the value of the postruntime t.
3. For each loop iteration, the expected runtime of that iteration is equal.

A central notion for capturing conditions 1. and 2. above formally is the concept of a loop being t–independent identically distributed (t–i.i.d. for short); for more details on i.i.d.–ness, see Definition 5.16. Similarly to Theorem 5.17 for weakest preeexpectations of independent identically distributed loops, for t–independent identically distributed loops with almost–surely terminating loop body, we can obtain exact expected runtimes:

THEOREM 7.23 (Expected Runtimes of t–i.i.d. Loops [Bat+18b]):
Let $t \in \mathbb{T}$. Moreover, let the following hold:

A. $\texttt{while}(\varphi)\{C\}$ *is t–independent identically distributed.*

B. *The loop body C terminates almost–surely, i.e.* $\mathsf{wp}\,[\![C]\!]\,(1) = 1$.

C. *Each loop iteration of* $\texttt{while}(\varphi)\{C\}$ *takes equal expected runtime, i.e.* $C \not\Lightarrow \mathsf{ert}\,[\![C]\!]\,(0)$.

Then the expected time needed to first execute $\texttt{while}(\varphi)\{C\}$ *and then let time t pass, i.e.* $\mathsf{ert}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(t)$, *is given by*

$$\mathsf{ert}\,[\![\texttt{while}(\varphi)\{C\}]\!]\,(t) \;=\; 1 \;+\; [\neg\varphi]\cdot t \;+\; [\varphi]\cdot\frac{1+\mathsf{ert}\,[\![C]\!]\,([\neg\varphi]\cdot t)}{1-\mathsf{wp}\,[\![C]\!]\,([\varphi])}\,,$$

where we define $0/0 = 0$ *and* $a/0 = \infty$, *for* $a \neq 0$.

Intuitively, the fraction

$$\frac{1+\mathsf{ert}\,[\![C]\!]\,([\neg\varphi]\cdot t)}{1-\mathsf{wp}\,[\![C]\!]\,([\varphi])}$$

appearing in Theorem 7.23 can be understood as follows: If the loop guard is true, the expected runtime of a single (guarded) loop iteration[12] is given by

$$1+\mathsf{ert}\,[\![C]\!]\,([\neg\varphi]\cdot t)\;.$$

12 Recall that a guarded loop iteration comprises of (1) checking the loop guard and, in case the loop guard is true, (2) executing the loop body.

Furthermore, since $\mathsf{wp}\,[\![C]\!]\,([\varphi])$ is the probability to continue iteration, $1 - \mathsf{wp}\,[\![C]\!]\,([\varphi])$ is the probability to terminate in each iteration. Because of the termination probability being the same for each iteration, the loop establishes in effect a *geometric distribution* on the number of loop iterations. The expected number of loop iterations is hence given by the closed form for corresponding geometric series, namely by

$$\frac{1}{1 - \mathsf{wp}\,[\![C]\!]\,([\varphi])}\,.$$

The fraction appearing in Theorem 7.23 can thus be understood as the expected runtime of each loop iteration multiplied by the expected number of loop iterations.

Similarly to Theorem 5.17, it is worthwhile to note that in order to apply Theorem 7.23 it is *not required* to find or guess in any way an invariant, ω–invariant, martingale, or alike. Instead, only t–i.i.d.–ness of t — the very postruntime one is interested in — needs to be checked. Our theorem then immediately yields the *exact* sought–after preexpectation — not just a bound.

Finally, we would like to mention that Theorem 7.23 is also not a free–lunch–theorem: Checking t–i.i.d.–ness can potentially become a non–trivial and in general undecidable task. Also, once the expected value of postruntime t depends in some way on the number of iterations a loop makes, i.e. once the loop performs some sort of *counting* and the value of the counter influences the value of t, the theorem fails to be applicable altogether. On the other hand, Theorem 7.23 has been successfully applied to reasoning about expected sampling times for massively large Bayesian networks from the *Bayesian Network Repository* [Scu] exceeding a thousand nodes [Bat+18b]. In this work, it has been shown that expected sampling times of *millions of years* can be reasoned about within *less than a second*.

7.7 CASE STUDY: THE COUPON COLLECTOR

We now demonstrate the effectiveness of our our ert calculus by applying it to the well–known Coupon Collector's Problem [MU05]: Suppose there are N different types of coupons and we can buy one uniformly randomly sampled coupon at a time, i.e. each time we buy a random coupon, we get a coupon of type $i \in \{1, \ldots, N\}$ with probability $1/N$. Once we have collected at least one coupon of each type, we can trade them for a prize. The aim of the Coupon Collector's Problem is to determine the expected number of random coupons we have to buy in order to have collected at least one coupon of each type. The problem can be modeled by program C_{cc} below:[13]

13 For describing C_{cc} we use an array variable cp. We assume that the content of an array is encoded as a single number z and we abstract from this encoding. For instance the assignment $cp[i] := 1$

```
cp := [0, ..., 0]⨟
i := 1⨟
x := N⨟
while(x > 0){
    while(cp[i] ≠ 0){
        i :≈ Uniform[1...N]
    }⨟
    cp[i] := 1⨟
    x := x − 1
}
```

All cells in the array cp are initialized to 0 and whenever we obtain the first coupon of type i, the program C_{cc} sets $cp[i]$ to 1. The outer loop is iterated N times and in each iteration we collect a new — uncollected — coupon. The inner loop models the buying of new random coupons until an uncollected coupon is bought.

We begin the runtime analysis of C_{cc} by introducing some notation. Let C_{in} and C_{out} denote the inner and the outer loop, respectively. Furthermore, let

$$\#col \;=\; \sum_{i=1}^{N} [cp[i] \neq 0]$$

denote the number of coupons that have already been collected.

Analysis of the inner loop. In order to later analyze the runtime of the *outer* loop we need to find a runtime invariant I for the outer loop. In order to check invariance of I, we will have to push I through the loop body of the outer loop which itself contains the *inner* loop as a subprogram. It will thus be necessary at some point to calculate ert $[\![C_{in}]\!]\,(s)$, for some $s \in \mathbb{T}$, and we hence first analyze the runtime of the inner loop.

For analyzing the inner loop, we note that the body of the inner loop obviously terminates almost–surely (as it consists only of a single assignment). Furthermore, we observe that the inner loop is s–independent identically distributed for any s (see Section 7.6.7 for more details on i.i.d.–ness). The intuitive reason is that once i is sampled and replaced by some value in s, the variable i does not occur in s anymore. More formally, we can check that all preconditions of Theorem 7.23 are met. The loop body of the inner loop $i :\approx \texttt{Uniform}[1\ldots N]$ influences only the variable i, since it consists only of a single assignment and i is the variable that appears on the left–hand–side of that assignment. Consider now the following:

---

then corresponds to an arithmetic expression encoding an update of the number z in a way such that the array represented by z is updated at position i to value 1.

$$\mathsf{wp}\,[\![i :\approx \mathtt{Uniform}[1\ldots N]]\!]\,([cp[i]\neq 0]) \;=\; \frac{1}{N}\cdot\sum_{j=1}^{N}[cp[j]\neq 0] \;=\; \frac{\#col}{N} \quad (\star)$$

On the right–hand–side of $\star$, the variable i does not occur anymore and thus it is not influenced by the loop body of the inner loop. Next, we calculate

$$\begin{aligned}&\mathsf{wp}\,[\![i :\approx \mathtt{Uniform}[1\ldots N]]\!]\,([cp[i]\neq 0]\cdot s)\\ &= \frac{1}{N}\cdot\sum_{j=1}^{N}[cp[j]\neq 0]\cdot s[i/j]\;, \qquad (\star\star)\end{aligned}$$

for any arbitrary $s\in\mathbb{T}$. On the right–hand–side of $\star\star$, the variable i does not occur anymore, since in s it has been replaced by a constant j. Thus, the right–hand–side is not influenced by the loop body of the inner loop. $\star$ and $\star\star$ together constitute s–i.i.d.–ness of the inner loop with respect to any postruntime s.

The last precondition for Theorem 7.23 is that each loop iteration takes equal expected runtime. For this, we calculate

$$\mathsf{ert}\,[\![i :\approx \mathtt{Uniform}[1\ldots N]]\!]\,(0) \;=\; 1+\sum_{j=1}^{N}0\,[i/j] \;=\; 1\;,$$

which is a constant and can thus trivially not be influenced by the body of the inner loop. Since all preconditions of Theorem 7.23 are met, we can apply this theorem to obtain a closed form for ert of the inner loop with respect to an arbitrary postruntime s:

$$\begin{aligned}&\mathsf{ert}\,[\![C_{in}]\!]\,(s)\\ &= 1 \;+\; [cp[i]=0]\cdot s \;+\; [cp[i]\neq 0]\cdot\frac{2+\frac{1}{N}\sum_{j=1}^{N}\big([cp[j]=0]\cdot s[i/j]\big)}{1-\frac{\#col}{N}}\;.\end{aligned}$$

Analysis of the outer loop. Using our analysis of the inner loop, the expected runtime of the body of the outer loop with respect to an arbitrary postruntime $t\in\mathbb{T}$ is given by

$$\mathsf{ert}\,[\![C_{\text{in}}\,;\,cp[i] :=1\,;\,x:=x-1]\!]\,(t) \;=\; 2+\mathsf{ert}\,[\![C_{\text{in}}]\!]\,(t[x/x-1,cp[i]/1])\;.$$

Since the program C_{cc} terminates right after the execution of the outer loop C_{out}, we analyze the runtime of the outer loop C_{out} with respect to continuation 0, i.e. $\mathsf{ert}\,[\![C_{out}]\!]\,(0)$. To this end we propose

$$I \;=\; 1+\sum_{\ell=0}^{\omega}[x>\ell]\cdot\left(3+2\cdot\sum_{k=0}^{\omega}\left(\frac{\#col+\ell}{N}\right)^{k}\right)$$

$$-2\cdot[cp[i]=0]\cdot[x>0]\cdot\sum_{k=0}^{\omega}\left(\frac{\#col}{N}\right)^k$$

as runtime invariant of C_{out} with respect to postruntime 0. We omit here the tedious verification that I is indeed a runtime invariant of the outer loop (for more details, see [Kam+16]). After one has checked runtime invariance of I, Theorem 7.16 yields

$$\mathsf{ert}\,[\![C_{\text{out}}]\!]\,(0)\;\preceq\;I\,, \tag{‡}$$

Analysis of the overall program. To obtain the overall expected runtime of program C_{cc} we have to account for the initialization instructions before the outer loop. The calculations go as follows:

$$\begin{aligned}
&\mathsf{ert}\,[\![C_{cc}]\!]\,(0)\\
&=\;\mathsf{ert}\,[\![cp:=[0,\dots,0]\fatsemi i:=1\fatsemi x:=N\fatsemi C_{\text{out}}]\!]\,(0)\\
&=\;\mathsf{ert}\,[\![cp:=[0,\dots,0]\fatsemi i:=1\fatsemi x:=N]\!]\,(\mathsf{ert}\,[\![C_{\text{out}}]\!]\,(0))\\
&\preceq\;\mathsf{ert}\,[\![cp:=[0,\dots,0]\fatsemi i:=1\fatsemi x:=N]\!]\,(I)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\text{(by ‡ and monotonicity, Theorem 7.10)}\\
&=\;3+I\,[x,i,cp[1],\dots,cp[N]/N,1,0,\dots,0]\\
&=\;3+1+\sum_{\ell=0}^{\omega}[N>\ell]\cdot\left(3+2\cdot\sum_{k=0}^{\omega}\left(\frac{0+\ell}{N}\right)^k\right)-2\cdot[0=0]\cdot[N>0]\cdot\sum_{k=0}^{\omega}\left(\frac{0}{N}\right)^k\\
&=\;4+[N>0]\cdot\sum_{\ell=0}^{N-1}\left(3+2\cdot\sum_{k=0}^{\omega}\left(\frac{\ell}{N}\right)^k\right)-2\cdot1\cdot[N>0]\cdot0\\
&=\;4+[N>0]\cdot\left(3N+2\cdot\sum_{\ell=0}^{N-1}\left(\sum_{k=0}^{\omega}\left(\frac{\ell}{N}\right)^k\right)\right)\\
&=\;4+[N>0]\cdot\left(3N+2\cdot\sum_{\ell=0}^{N-1}\frac{1}{1-\frac{\ell}{N}}\right)\quad\text{(by closed form for geom. series)}\\
&=\;4+[N>0]\cdot\left(3N+2\cdot\sum_{\ell=0}^{N-1}\frac{N}{N-\ell}\right)\\
&=\;4+[N>0]\cdot\left(3N+2N\cdot\sum_{\ell=1}^{N-1}\frac{1}{\ell}\right)\\
&=\;4+[N>0]\cdot N\left(3+2\cdot\sum_{\ell=1}^{N-1}\frac{1}{\ell}\right)\\
&=\;4+[N>0]\cdot N\big(3+2\cdot\mathcal{H}_{N-1}\big)\,,
\end{aligned}$$

where $\mathcal{H}_{N-1}=1/1+1/2+1/3+\cdots+1/N-1$ denotes the $(N-1)$-st harmonic number. Since the harmonic numbers approach the natural logarithm in the limit, we

can conclude that the coupon collector program C_{cc} takes in expectation time $\mathcal{O}\big(N \cdot \log(N)\big)$ until termination.

7.8 OTHER RELATED WORK

Analyses of expected runtime of randomized algorithms are typically obtained by ad–hoc arguments exploiting classical probability and martingale theory [Fra98; MR95]. Typically, those „book proofs“ do not argue on source code level but instead abstract away from the code and argue on the level of stochastic processes. Our approach is different: We argue on source code level using a calculus that proves runtimes directly on the programs, *not on an abstraction* of them.

Apart from the numerous references to related work we have made in this chapter, the work that is perhaps closest to our ert calculus is by Celiku and McIver [CM05]. They provide a wp calculus for obtaining performance properties of probabilistic programs, including upper bounds on expected runtimes. Their focus is on refinement. Neither do they provide a soundness result of their approach nor do they consider lower bounds. Moreover, we believe that the ert calculus is simpler to work with in practice.

Arthan *et al.* [Art+09] provide a general framework for sound and complete Hoare–style logics, and show that a particular instantiation of their framework can be used to obtain upper bounds on the runtime of while loops. They do not consider probabilistic programs.

Deriving space and time consumption of deterministic programs has also been considered by Hehner [Heh98]. For deriving time bounds, Hehner proposes to use a runtime–counting ghost variable as we have studied in Section 7.2. We saw in that section that the ghost variable technique is unsound for reasoning about probabilistic programs in general.

Hickey and Cohen [HC88] automate the average–case analysis of deterministic programs by generating and solving a system of recurrence equations derived from the program that is to be analyzed. Average–case analysis considers the expected runtime of a *deterministic program* with respect to a random distribution of inputs. We, on the other hand, consider deterministic inputs for (inherently) *randomized algorithms*.

Berghammer and Müller–Olm [BM04] show how Hoare–style reasoning can be extended to obtain bounds on the closeness of results obtained using approximate algorithms to the optimal solution. In contrast to randomized algorithms, approximate algorithms *deterministically* give solutions that are close to the optimum in terms of some approximation factor.

Monniaux [Mon01] exploits abstract interpretation to automatically prove the probabilistic termination of programs using exponential bounds on the tail of the distribution. His technique yields upper bounds only and can be used to prove the soundness of experimental statistical methods to deter-

mine the expected runtime of probabilistic programs.

Brazdil *et al.* [Brá+15] study the runtime of probabilistic programs with unbounded recursion by modelling them as probabilistic pushdown automata. They show (using martingale theory) that for every pPDA the probability of performing a long run decreases exponentially (polynomially) in the length of the run, iff the pPDA has a finite (infinite) expected runtime. As opposed to program verification using the ert calculus, [Brá+15] considers reasoning at an operational level. This becomes problematic if the state space of the program becomes infinite and this infinity is not caused only by the call stack needed to implement recursion, but genuinely by infinitely many program states. The program then cannot be modeled by a pPDA, but this fact is in general undecidable.

Finally, we would like to mention that our calculus has been implemented in the interactive theorem prover Isabelle/HOL by Hölzl [Höl16]. Hölzl proved that our calculus is indeed sound and complete and that the theorems listed in this chapter are correct.

More recently, Ngo *et al.* developed a fully automatic approach for deriving polynomial runtime bounds [NCH18]. Ngo *et al.* use our ert calculus as an underlying theoretical framework for proving soundness of their approach.

As for future work, another direction — besides finding elegant proof methods for lower bounds, as mentioned earlier — would be to develop a calculus for reasoning about *expected space consumption*. This poses a challenge, because it is not clear (at least to us) how to formulate this as a least or greatest fixed point. Intuitively, the reason is that if a program does not terminate with a certain probability, it will still consume space in those cases, an this space can very well be finite.

CONDITIONING 8

Describing randomized algorithms for solving computationally difficult problems more efficiently on average is a classical use case of probabilistic programs. A more recent use case is found in machine learning: There, probabilistic programs conveniently describe complex probability distributions. The main goal is to have a mathematically rigorous description that is yet easily accessible to a working programmer [Gor+14]. A key ingredient in such descriptions is *conditioning*, which — as the name suggests — allows to describe conditional probability distributions.

As a simple example of such a conditional distribution, suppose that we would like to model that variable x is distributed according to a geometric distribution with parameter $1/2$, conditioned on the event that x is odd. In standard mathematical notation, the probability mass function

$$\mathsf{P}(x = k \mid x \text{ is odd}) = \begin{cases} \frac{3}{2^{k+1}}, & \text{if } k \text{ is odd,} \\ 0, & \text{if } k \text{ is even.} \end{cases}$$

describes the probability that x has value k, *given* that x is odd. Notice that the mathematical representation above completely hides both the *computational aspects* underlying a geometric distribution as well as the *event on which we want to condition*. As for the underlying computational aspect, an (unconditioned) geometric distribution can be described by the following *algorithm*: Keep flipping a fair coin until you throw, say, heads. *Count* in x the number of coin flips you had to perform in order to achieve that goal.

On the positive side, the above mathematical representation is *compact* and *explicit*. Questions, for instance, about probabilities of certain events or about the expected value of x can easily be answered. On the negative side, if we are merely presented with the above mathematical representation of the conditional distribution, it is rather difficult to extract the underlying algorithm that constructed that distribution. That in turn makes it rather difficult to understand the distribution and in particular difficult to adjust it: Say we wanted to adjust the parameter of the geometric distribution to $1/3$ instead of $1/2$. Then the resulting probability mass function is

$$\mathsf{P}'(x = k \mid x \text{ is odd}) = \begin{cases} \frac{2^k \cdot 5}{3^{k+2}}, & \text{if } k \text{ is odd,} \\ 0, & \text{if } k \text{ is even.} \end{cases}$$

However, it is not at all obvious how to obtain P' from P.

Another negative aspect of representing distributions by probability mass functions is that it is not so obvious *how to sample* from a distribution given only its probability mass function. Instead, general purpose sampling algorithms have to be employed. The probability of a specific sample being returned by the sampling algorithm corresponds to the probability specified by the probability mass function it gets as input, however often only up to some precision.

As an alternative, the probabilistic program

```
1: x := 0;
2: c := 1;
3: while(c = 1){
4:     x := x + 1;
5:     {c := 0} [1/2] {c := 1}
6: };
7: observe(x is odd)
```

describes the same conditional distribution P while not hiding the details of its construction: Both the repeated coin flips (Lines 3 to 6) as well as the event on which we condition (Line 7) are *explicitly typed out* in the above program. We can even explicitly see the individual coin flips (Line 5) and the counting (Line 4). We also notice that in the program representation it is completely obvious how to adjust the parameter of the geometric distribution from $1/2$ to $1/3$: We simply replace the $1/2$ by $1/3$ in the program.

As another positive aspect of the program representation, each probability distribution described by a probabilistic program comes with its own special purpose sampling algorithm: *the program itself!* By executing the program once, we effectively obtain one sample from the probability distribution that it implicitly describes.

While the program representation above is easy to understand and easy to sample from, the probability mass function it represents is not as explicitly given as with the probability mass function representation. A main task for probabilistic programs, in particular with conditioning, is thus *inference*, i.e. determining an „explicit representation of the probability distribution implicitly specified by the probabilistic program“ [Gor+14].

A workable approach to inference is determining the expected value of some function f after executing a probabilistic program [Gor+14]. We have already studied in Chapter 4 how this can be accomplished for programs without conditioning by means of the weakest preexpectation calculus. In this chapter, we will extend the weakest preexpectation calculus to reasoning about probabilistic programs *with conditioning*. We will show how previous approaches to the inference task are inferior when it comes to dealing with nontermination and present rules for reasoning about loops.

8.1 cpGCL — pGCL WITH CONDITIONING

In this section, we present syntax and operational semantics of the *probabilistic guarded command language with conditioning*. As for the syntax, we endow pGCL (cf. Section 3.2) with an additional `observe` statement, as we have already done tacitly in the introductory example.

Definition 8.1 (pGCL with Conditioning [Jan+15a; Olm+18]): *Recall* Definition 3.1 *which defines all notions related to the probabilistic guarded command language* pGCL. *The set of programs in probabilistic guarded command language with conditioning, denoted* cpGCL, *is given by the grammar*

$$
\begin{aligned}
C \longrightarrow\ & \texttt{skip} && \text{(effectless program)}\\
\mid\ & \texttt{diverge} && \text{(freeze)}\\
\mid\ & x := E && \text{(assignment)}\\
\mid\ & x :\approx \mu && \text{(random assignment)}\\
\mid\ & \texttt{observe}\,(\varphi) && \text{(conditioning)}\\
\mid\ & C \,\texttt{;}\, C && \text{(sequential composition)}\\
\mid\ & \texttt{if}\,(\varphi)\,\{C\}\,\texttt{else}\,\{C\} && \text{(conditional choice)}\\
\mid\ & \{C\}\,[p]\,\{C\} && \text{(probabilistic choice)}\\
\mid\ & \texttt{while}(\varphi)\{C\}\,, && \text{(while loop)}
\end{aligned}
$$

where $x \in \mathsf{Vars}$ is a program variable, E is an arithmetic expression over program variables, μ is a distribution expression, φ is a boolean expression over program variables guarding a choice or a loop, and p is a probability expression. Recall Definition 3.1 *for the meaning of the above technical terms.*

Note that cpGCL *programs are by definition tame, i.e. they contain no nondeterministic choices.*

Operationally, all instructions of a cpGCL program are executed exactly the same way as pGCL instructions (cf. Section 3.3). The obvious exception are `observe` statements, since these are not part of the pGCL language. The `observe` statements are executed as follows:

When an $\texttt{observe}\,(\varphi)$ instruction is encountered and the current program state is σ, it is checked whether σ satisfies the observation φ, i.e. whether $\sigma \models \varphi$. If so, the computation proceeds as if the $\texttt{observe}\,(\varphi)$ instruction was a `skip` instruction. If, however, $\sigma \not\models \varphi$, the computation terminates *unsuccessfully* in a designated observation violation state $\lightning$.

Formally, the operational behavior of a cpGCL program is given by extending the computation tree semantics of pGCL (see Definition 3.4). In particular, the SOS rules in Figure 3.1 are complemented by two rules for

the observe statement, namely:

$$\frac{\varphi(\sigma) \;=\; \text{true}}{\langle \texttt{observe}\,(\varphi), \sigma, n, \theta, \eta, q\rangle \;\vdash\; \langle \downarrow, \sigma, n+1, \theta, \eta, q\rangle} \text{ (observe1)}$$

$$\frac{\varphi(\sigma) \;=\; \text{false}}{\langle \texttt{observe}\,(\varphi), \sigma, n, \theta, \eta, q\rangle \;\vdash\; \langle \downarrow, \lightning, n+1, \theta, \eta, q\rangle} \text{ (observe2)}$$

For a given cpGCL program C and input σ, any computation path in the computation tree of executing C on input σ has *exactly one of three* forms:

A. The path terminates successfully in some final state $\tau \neq \lightning$.

B. The path terminates unsuccessfully in $\lightning$.

C. The path does not terminate but also does not violate any observation.

Executing C on σ thus gives rise to three disjoint sets of paths which we denote for illustrational purposes by A, B, and C, respectively. A diagrammatic depiction of this situation is shown in Figure 8.1.

Besides a computation tree semantics, we also described the semantics of pGCL programs as inducing a (sub)distribution over final states (see Section 3.3.2). Using the extension of the computation tree semantics described above, we obtain from a cpGCL program a (sub)distribution $[\![C]\!]_\sigma$ over $\Sigma \cup \{\lightning\}$ by applying Definition 3.8. However, for cpGCL programs, we would rather like to describe the *conditional distribution* $[\![C]\!]_\sigma\big|_{\neg\lightning}$ over terminal states, *conditioned* on the event $\neg$B, i.e. on the event that no observation violation occurred during computation [Gor+14]. Note that the event $\neg$B coincides with the event A $\cup$ C, i.e. the event that either the program terminates successfully or not at all. The conditional distribution $[\![C]\!]_\sigma\big|_{\neg\lightning}$ can be described by

$$[\![C]\!]_\sigma\big|_{\neg\lightning}(\tau) \;=\; \begin{cases} 0, & \text{if } \tau = \lightning \text{ and } [\![C]\!]_\sigma(\lightning) < 1 \\ \frac{[\![C]\!]_\sigma(\tau)}{1-[\![C]\!]_\sigma(\lightning)}, & \text{if } \tau \neq \lightning \text{ and } [\![C]\!]_\sigma(\lightning) < 1 \\ \text{undefined}, & \text{if } [\![C]\!]_\sigma(\lightning) = 1\,. \end{cases}$$

As we can see, the distribution $[\![C]\!]_\sigma\big|_{\neg\lightning}$ is a rather unwieldy object and sometimes even undefined, namely whenever we would have to deal with a division by zero. The expectation transformer based approach to reasoning about cpGCL programs which we present in the remainder of this chapter is more satisfactory in that aspect: It always gives well–defined, meaningful, and expected results, even in the problematic division–by–zero case.

8.2 CONDITIONAL EXPECTATION TRANSFORMERS

Extending weakest preexpectation reasoning for pGCL à la Chapter 4 to cpGCL is the subject matter of this section. This technique will allow us

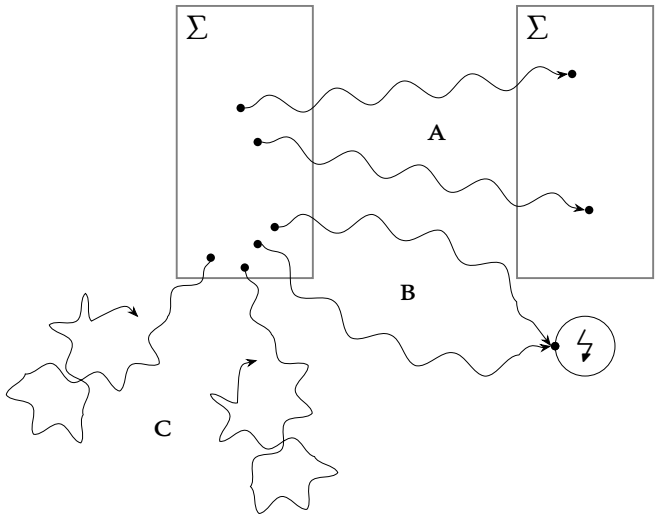


Figure 8.1: Executing a cpGCL program can lead to either one of three outcomes: (A) the program does not violate any observation along its computation and *terminates successfully* in some final state, (B) the program violates an observation along its computation and *terminates unsuccessfully* in the observation failure state $\lightning$, or (C) the program does not violate any observation along its computation but also *does not terminate*.

to reason about *conditional expected values* and *conditional probabilities*. As we did with pGCL, we will present two calculi: the *conditional weakest preexpectation calculus* for total correctness and the *conditional weakest liberal preexpectation calculus* for partial correctness.

8.2.1 *Conditional Weakest Preexpectations*

For a given postexpectation $f \in \mathbb{E}$ and program $C \in \mathsf{cpGCL}$, we are interested in the *conditional expected value of* f after successful termination of C on a given input σ, *given* that *no* observation that is encountered along the computation is violated. More precisely, we are interested in a mathematical object that maps every input σ to the respective conditional expected value. Put in terms of the three sets A, B, and C we described in Section 8.1, we are interested in the quantity

$$\frac{\mathsf{EV}\big(f \cdot [\neg \mathrm{B}]\big)}{\mathsf{Pr}(\neg \mathrm{B})} = \frac{\mathsf{EV}\big(f \cdot [\mathrm{A}] + f \cdot [\mathrm{C}]\big)}{\mathsf{Pr}(\mathrm{A} \cup \mathrm{C})} = \frac{\mathsf{EV}(f \cdot [\mathrm{A}])}{\mathsf{Pr}(\mathrm{A} \cup \mathrm{C})},$$

i.e. the expected value of f, given that no observation is violated. The last equality comes about because all paths in the set c are infinite and hence there exists no path in c that reaches a final state in which f could be evaluated with non–zero probability. The conditional expected value of f described above constitutes our notion of *conditional weakest preexpectations*. By choosing $f = [F]$, i.e. f is the indicator function of an event F, we see that reasoning about conditional weakest preexpectations subsumes reasoning about *conditional probabilities*.

Our general approach will be to calculate the numerator and denominator of the fraction above *separately*, in order not to run into trouble with problematic cases such as for instance „0/0“. Keeping a pair, we can just pair 0 and 0 without running into problems with undefinedness. We also notice that the numerator is a general expected value, whereas the denominator represents a probability. We will hence reason about pairs of expectations in $\mathbb{E} \times \mathbb{E}_{\leq 1}$. We call such pairs *conditional expectations*.

Definition 8.2 (Conditional Expectations [Jan+15a; Olm+18]):

A. *The set of conditional expectations, denoted $\mathbb{C}$, is defined as the set of pairs comprising of a general expectation in $\mathbb{E}$ and a one–bounded expectation in $\mathbb{E}_{\leq 1}$ (cf. Definition 4.1), i.e.*

$$\mathbb{C} \;=\; \mathbb{E} \times \mathbb{E}_{\leq 1} \,.$$

We denote a pair in $\mathbb{C}$ consisting of first component f and second component g by

$$\underline{f \,/\, g}$$

to indicate that it represents a fraction.

A complete lattice on $\mathbb{C}$ is induced by the partial order $\trianglelefteq$, given by

$$\underline{f \,/\, g} \;\trianglelefteq\; \underline{f' \,/\, g'} \qquad \text{iff} \qquad f \preceq f' \quad \text{and} \quad g \succeq g' \,.$$

Notice that the order on the second components is the reversed order of the first components. The least and the greatest element in the complete lattice $(\mathbb{C}, \trianglelefteq)$ is given by

$$\underline{0 \,/\, 1} \quad \text{and} \quad \underline{\infty \,/\, 0} \,,$$

respectively, where $0, 1, \infty \in \mathbb{E}$. The supremum of a subset $S \subseteq \mathbb{C}$ (with respect to the order $\trianglelefteq$) is given pointwise by

$$\sup_{\trianglelefteq} S \;=\; \underline{\sup_{\preceq} \left\{ f \;\middle|\; \underline{f \,/\, g} \in S \right\} \Big/ \inf_{\preceq} \left\{ g \;\middle|\; \underline{f \,/\, g} \in S \right\}} \,,$$

where the supremum and the infimum on the right–hand–side are understood with respect to the order $\preceq$.

B. *For* $\underline{f/g}, \underline{f'/g'} \in \mathbb{C}$ *and* $h \in \mathbb{E}_{\leq 1}$*, we define an addition* $\oplus$ *by*

$$\underline{f/g} \oplus \underline{f'/g'} = \underline{f + f'/g + g'}$$

and likewise a scalar multiplication $\odot$ *by*

$$h \odot \underline{f/g} = \underline{h \cdot f / h \cdot g}\,.$$

Notice that there is a crucial difference between f/g and $\underline{f/g}$: The expression f/g is a pointwise fraction, i.e.

$$\frac{f}{g} = \lambda\sigma.\ \frac{f(\sigma)}{g(\sigma)}\,,$$

which is potentially undefined, namely if $g(\sigma) = 0$. Moreover, we have

$$\frac{1}{1} = \frac{\frac{1}{2}}{\frac{1}{2}}\,.$$

On the other hand, a conditional expectation $\underline{f/g}$ is merely a *pair* of expectations, which is *interpreted* as a fraction:

$$\underline{f/g} \qquad \text{is interpreted as} \qquad \lambda\sigma.\ \begin{cases} \frac{f(\sigma)}{g(\sigma)}, & \text{if } g(\sigma) \neq 0 \\ \text{undefined}, & \text{if } g(\sigma) = 0. \end{cases}$$

However, formally, a conditional expectation is *not* a fraction. In particular,

$$\underline{1/1} \neq \underline{1/2 / 1/2}\,.$$

As conditional expectations are just pairs, $\underline{0/0}$ is a perfectly well–defined mathematical object, whereas the definedness of $0/0$ is at least controversial.

The partial order $\trianglelefteq$ on conditional expectations enables monotonic reasoning about total correctness. The order corresponds naturally to our „fractional interpretation“ in the sense that for all states σ we have that

$$\underline{f/g} \trianglelefteq \underline{f'/g'} \qquad \text{implies} \qquad \frac{f(\sigma)}{g(\sigma)} \leq \frac{f'(\sigma)}{g'(\sigma)}$$

in case that both fractions are defined (i.e. in case that $g'(\sigma) > 0$, which by $g \geq g'$ implies that $g(\sigma) > 0$). This means that overapproximations in the sense of the partial order $\trianglelefteq$ are indeed overapproximations of the sought–after conditional expected value.

For reasoning about conditional expected values yielded by cpGCL programs, we define an expectation transformer that acts on conditional expectations, i.e. on $\mathbb{C}$, as follows:

C	$\mathsf{cwp}\,[C]\left(\frac{f}{g}\right)$
`skip`	$\frac{f}{g}$
`diverge`	$\frac{0}{1}$
$x := E$	$\frac{f[x/E]}{g[x/E]}$
$x :\approx \mu$	$\frac{\lambda\sigma.\ \int_{\mathsf{Vals}} \left(\lambda v.\ f\left(\sigma[x \mapsto v]\right)\right) d\,\mu_\sigma}{\lambda\sigma.\ \int_{\mathsf{Vals}} \left(\lambda v.\ g\left(\sigma[x \mapsto v]\right)\right) d\,\mu_\sigma}$
`observe`(φ)	$[\varphi] \odot \frac{f}{g}$
$C_1 \fatsemi C_2$	$\mathsf{cwp}\,[\![C_1]\!]\left(\mathsf{cwp}\,[\![C_2]\!]\left(\frac{f}{g}\right)\right)$
`if`$(\varphi)\{C_1\}$`else`$\{C_2\}$	$[\varphi] \odot \mathsf{cwp}\,[\![C_1]\!]\left(\frac{f}{g}\right) \oplus [\neg\varphi] \odot \mathsf{cwp}\,[\![C_2]\!]\left(\frac{f}{g}\right)$
$\{C_1\}\,[p]\,\{C_2\}$	$p \odot \mathsf{cwp}\,[\![C_1]\!]\left(\frac{f}{g}\right) \oplus (1-p) \odot \mathsf{cwp}\,[\![C_2]\!]\left(\frac{f}{g}\right)$
`while`$(\varphi)\{C'\}$	$\mathsf{lfp}_{\trianglelefteq}\ \frac{X}{Y}.\ [\neg\varphi] \odot \frac{f}{g} \oplus [\varphi] \odot \mathsf{cwp}\,[\![C']\!]\left(\frac{X}{Y}\right)$

Table 8.1: The conditional weakest preexpectation transformer. The least fixed point for the while loop is understood in terms of the partial order $\trianglelefteq$.

DEFINITION 8.3 (Conditional wp [Jan+15a; Olm+18]):

A. *The conditional weakest preexpectation transformer*

$$\mathsf{cwp}\,[\![C]\!]\colon \quad \mathbb{C} \to \mathbb{C}$$

is defined according to the rules in Table 8.1.

B. *We call the function*

$$\overset{\mathsf{cwp}}{\langle\varphi,C\rangle}\Phi_{\frac{f}{g}}\left(\frac{X}{Y}\right) = [\neg\varphi] \odot \frac{f}{g} \oplus [\varphi] \cdot \mathsf{cwp}\,[\![C]\!]\left(\frac{X}{Y}\right)$$

the cwp–characteristic function of `while`$(\varphi)\{C\}$ *with respect to postexpectation* $\frac{f}{g}$*. If either of* cwp*,* φ*,* C*, or* $\frac{f}{g}$ *are clear from the context, we omit them from* Φ*.*

The rules for the cwp transformer basically calculate wp in the first component and wlp in the second component (cf. Section 4.1). For the `observe`(φ) statement, the indicator function of the observed evidence φ is multiplied to both components. In effect that reduces the expected value in the first component (the numerator) as well as the normalization factor in the second component (the denominator). We will study how these rules behave for loops in more detail later in this chapter.

If we want to use the cwp calculus to reason about the conditional expected value of f after executing program C, given that no observation is violated, we have to determine $\mathsf{cwp}\,[\![C]\!]\left(\frac{f}{1}\right)$. Below, we give an example:

EXAMPLE 8.4 (Conditional Weakest Preexpectation Reasoning [Olm+18]): Assume we want to compute the expected value of the expression $10+x$ after executing program C given by

$$
\begin{aligned}
&\{x := 0\}\,[1/2]\,\{x := 1\}\,;\\
&\texttt{if}\,(x = 1)\,\{\\
&\qquad \{y := 0\}\,[1/2]\,\{y := 2\}\\
&\}\texttt{else}\{\\
&\qquad \{y := 0\}\,[4/5]\,\{y := 3\}\\
&\}\,;\\
&\texttt{observe}\,(y = 0)
\end{aligned}
$$

That means, we have to reason about $\mathrm{cwp}\,[\![C]\!]\left(\frac{10+x}{1}\right)$. Reusing our annotation style from earlier in this thesis (see Example 2.11), i.e. we write

$$
\begin{aligned}
&/\!/\!/\ \frac{f''}{g''}\\
&/\!/\!/\ \frac{f'}{g'}\\
&C\\
&/\!/\!/\ \frac{f}{g}
\end{aligned}
$$

to expresses that $\frac{f'}{g'} = \mathrm{cwp}\,[\![C]\!]\left(\frac{f}{g}\right)$ and moreover that $\frac{f''}{g''} = \frac{f'}{g'}$, we can annotate the above program as shown in Figure 8.2 (read from bottom to top). The calculation of $\mathrm{cwp}\,[\![C]\!]\left(\frac{10+x}{1}\right)$ gives $\frac{27/4}{13/20}$. As for an *interpretation*, we can say that the conditional expected value of $10+x$, given that the observation is not violated, is for any initial state

$$
\frac{\frac{27}{4}}{\frac{13}{20}} = \frac{27\cdot 20}{13\cdot 4} = \frac{135}{13} \approx 10.38\,.
$$

8.2.2 *Conditional Weakest Liberal Preexpectations*

While we considered total correctness above, we now consider partial correctness: Given event F and program C, we ask: What is the *conditional probability* that C either diverges or terminates in a state satisfying F, *given* that no observation encountered along the computation is violated. Put in terms of the three sets A, B, and C (see Section 8.1), we are interested in

$$
\frac{\mathsf{Pr}\big((F \cup \mathrm{C}) \cap \neg \mathrm{B}\big)}{\mathsf{Pr}(\neg \mathrm{B})} = \frac{\mathsf{Pr}(F \cup \mathrm{C})}{\mathsf{Pr}(\mathrm{A} \cup \mathrm{C})}\,.
$$

$$/\!/\!/\ \underline{\tfrac{27}{4}\big/\tfrac{13}{20}}$$

$$/\!/\!/\ \underline{4\big/\tfrac{4}{10}} \oplus \underline{\tfrac{11}{4}\big/\tfrac{1}{4}}$$

$$/\!/\!/\ \tfrac{4}{10} \odot \underline{10\big/1} \oplus \tfrac{1}{4} \odot \underline{11\big/1}$$

$$/\!/\!/\ \tfrac{1}{2} \odot \tfrac{4}{5} \odot \underline{10\big/1} \oplus \tfrac{1}{2} \odot \tfrac{1}{2} \odot \underline{11\big/1}$$

$$/\!/\!/\ \tfrac{1}{2} \odot \Big([0=1] \odot \tfrac{1}{2} \odot \underline{10+0\big/1} \oplus [0\neq 1] \odot \tfrac{4}{5} \odot \underline{10+0\big/1}\Big)$$
$$\oplus\ \tfrac{1}{2} \odot \Big([1=1] \odot \tfrac{1}{2} \odot \underline{10+1\big/1} \oplus [1\neq 1] \odot \tfrac{4}{5} \odot \underline{10+1\big/1}\Big)$$

$\{x := 0\}\ [1/2]\ \{x := 1\}$;

$$/\!/\!/\ [x=1] \odot \tfrac{1}{2} \odot \underline{10+x\big/1} \oplus [x\neq 1] \odot \tfrac{4}{5} \odot \underline{10+x\big/1}$$

if $(x = 1)$ {

$$/\!/\!/\ \tfrac{1}{2} \odot \underline{10+x\big/1}$$

$$/\!/\!/\ \tfrac{1}{2} \odot [0=0] \odot \underline{10+x\big/1} \oplus \tfrac{1}{2} \odot [2=0] \odot \underline{10+x\big/1}$$

$\{y := 0\}\ [1/2]\ \{y := 2\}$

$$/\!/\!/\ [y=0] \odot \underline{10+x\big/1}$$

} else {

$$/\!/\!/\ \tfrac{4}{5} \odot \underline{10+x\big/1}$$

$$/\!/\!/\ \tfrac{4}{5} \odot [0=0] \odot \underline{10+x\big/1} \oplus \tfrac{1}{5} \odot [3=0] \odot \underline{10+x\big/1}$$

$\{y := 0\}\ [4/5]\ \{y := 3\}$

$$/\!/\!/\ [y=0] \odot \underline{10+x\big/1}$$

};

$$/\!/\!/\ [y=0] \odot \underline{10+x\big/1}$$

observe $(y = 0)$

$$/\!/\!/\ \underline{10+x\big/1}$$

Figure 8.2: Conditional weakest preexpectation annotations for Example 8.4.

The equality comes about because all computation paths that terminate successfully do not terminate in the $\lightning$ state. The conditional probability above constitutes our notion of *conditional weakest liberal preexpectations*.

Again, our approach will be to calculate the numerator and the denominator of the above fraction separately. We notice that both the numerator as well as the denominator represent probabilities. We will hence reason about pairs of expectations in $\mathbb{E}_{\leq 1} \times \mathbb{E}_{\leq 1}$. We call such pairs *one–bounded conditional expectations*.

DEFINITION 8.5 (One–bounded Cond. Expect. [Jan+15a; Olm+18]**):**

A. *The set of one–bounded conditional expectations, denoted $\mathbb{C}_{\leq 1}$, is defined as the set of conditional expectations, where both components are one–bounded expectations, i.e.*

$$\mathbb{C}_{\leq 1} \;=\; \mathbb{E}_{\leq 1} \times \mathbb{E}_{\leq 1}\,.$$

Obviously, we have $\mathbb{C}_{\leq 1} \subset \mathbb{C}$. We use the same notation for pairs in $\mathbb{C}_{\leq 1}$ as for pairs in $\mathbb{C}$. A complete lattice on $\mathbb{C}_{\leq 1}$ is induced by the partial order $\trianglelefteq$, given by

$$\underline{f \,/\, g} \;\trianglelefteq\; \underline{f' \,/\, g'} \qquad \text{iff} \qquad f \preceq f' \quad \text{and} \quad g \preceq g'\,.$$

Notice that (in contrast to the order $\trianglelefteq$ on $\mathbb{C}$) the order on the second components is the same as the order of the first components. The least and the greatest element in the complete lattice $(\mathbb{C}_{\leq 1}, \trianglelefteq)$ are given by

$$\underline{0 \,/\, 0} \quad \text{and} \quad \underline{1 \,/\, 1}\,,$$

respectively, where $0, 1 \in \mathbb{E}_{\leq 1}$. The supremum of a subset $S \subseteq \mathbb{C}_{\leq 1}$ (with respect to the order $\trianglelefteq$) is constructed pointwise by

$$\sup_{\trianglelefteq} S \;=\; \underline{\sup_{\preceq} \left\{ f \;\middle|\; \underline{f \,/\, g} \in S \right\} \Big/ \sup_{\preceq} \left\{ g \;\middle|\; \underline{f \,/\, g} \in S \right\}}\,,$$

where the two suprema on the right–hand–side are understood with respect to the partial order $\preceq$.

The partial order $\trianglelefteq$ will be used for defining conditional weakest liberal preexpectations as greatest fixed points. Unfortunately, the partical order $\trianglelefteq$ does not correspond as naturally to our „fractional interpretation“ as the partial order $\trianglelefteq$ on $\mathbb{C}$ did. This is because we have that

$$\underline{f \,/\, g} \;\trianglelefteq\; \underline{f' \,/\, g'} \quad \text{neither implies} \quad \frac{f(\sigma)}{g(\sigma)} \leq \frac{f'(\sigma)}{g'(\sigma)} \quad \text{nor} \quad \frac{f(\sigma)}{g(\sigma)} \geq \frac{f'(\sigma)}{g'(\sigma)}\,.$$

C	$\mathsf{cwlp}\,[\![C]\!]\left(\underline{f/g}\right)$
`skip`	$\underline{f/g}$
`diverge`	$\underline{1/1}$
$x := E$	$\underline{f[x/E]/g[x/E]}$
$x :\approx \mu$	$\underline{\lambda\sigma.\ \int_{\mathsf{Vals}}\left(\lambda v.\ f(\sigma[x\mapsto v])\right)d\,\mu_\sigma \,/\, \lambda\sigma.\ \int_{\mathsf{Vals}}\left(\lambda v.\ g(\sigma[x\mapsto v])\right)d\,\mu_\sigma}$
`observe`$\,(\varphi)$	$[\varphi]\odot\underline{f/g}$
$C_1\,;\,C_2$	$\mathsf{cwlp}\,[\![C_1]\!]\left(\mathsf{cwlp}\,[\![C_2]\!]\left(\underline{f/g}\right)\right)$
`if`$\,(\varphi)\,\{C_1\}\,$`else`$\,\{C_2\}$	$[\varphi]\odot\mathsf{cwlp}\,[\![C_1]\!]\left(\underline{f/g}\right)\ \oplus\ [\neg\varphi]\odot\mathsf{cwlp}\,[\![C_2]\!]\left(\underline{f/g}\right)$
$\{C_1\}\,[p]\,\{C_2\}$	$p\odot\mathsf{cwlp}\,[\![C_1]\!]\left(\underline{f/g}\right)\ \oplus\ (1-p)\odot\mathsf{cwlp}\,[\![C_2]\!]\left(\underline{f/g}\right)$
`while`$(\varphi)\{C'\}$	$\mathsf{gfp}_{\trianglelefteq}\ \underline{X/Y}.\ [\neg\varphi]\odot\underline{f/g}\ \oplus\ [\varphi]\odot\mathsf{cwlp}\,[\![C']\!]\left(\underline{X/Y}\right)$

Table 8.2: The conditional weakest liberal preexpectation transformer. The greatest fixed point for the while loop is understood in terms of the partial order $\trianglelefteq$.

Thus, the partial order $\trianglelefteq$ is not as suitable for monotonic reasoning about conditional probabilities. While the transformer we present shortly will indeed be $\trianglelefteq$–monotonic, this is hardly of any use because over– or underapproximating a result with respect to $\trianglelefteq$ does not necessarily give an over– or underapproximation of the sought–after conditional probability.

We now present the conditional weakest liberal preexpectation transformer that acts on $\mathbb{C}_{\leq 1}$ and enables reasoning about conditional probabilities.

Definition 8.6 (Conditional wlp [Jan+15a; Olm+18]):
The conditional weakest liberal preexpectation transformer

$$\mathsf{cwlp}\,[\![C]\!]\colon\quad \mathbb{C}_{\leq 1}\to\mathbb{C}_{\leq 1}$$

is defined according to the rules in Table 8.2.

The rules for the cwlp transformer basically calculate wlp in both components (cf. Section 4.1.2). As with cwp, for the `observe`$\,(\varphi)$ statement, the indicator function of φ is multiplied to both components, which in effect reduces the probability in the first component (the numerator) as well as the normalization factor in the second component (the denominator). We will study how cwlp behaves for loops in more detail in the next section.

If we want to use the cwlp calculus to reason about the conditional probability that an event F is established after executing program C, given that no observation is violated, we have to determine $\mathsf{cwlp}\,[\![C]\!]\left(\underline{[F]/1}\right)$.

8.3 CONDITIONING AND LOOPS

The interplay of conditioning and loops is a particularly intricate matter when attempting to give semantics to probabilistic programs with conditioning. Particular care must be taken, for instance, when conditioning *inside* a loop. However, even by syntactically forbidding conditioning inside loops we do not mitigate all problems.

8.3.1 *The* cwp *Interpretation for Total Correctness*

We first turn our attention to how our cwp transformer for total correctness behaves for loops. As an admittedly pointed — but on the other hand very demonstrative — example, consider the program C, given by

$$
\begin{aligned}
&x := 1 \,⨟\\
&\texttt{while}(x = 1)\{\\
&\quad \{x := 1\}\,[1/2]\,\{x := 0\}\\
&\quad \texttt{observe}\,(x = 1)\\
&\}\,.
\end{aligned}
$$

This program has exactly one diverging run, namely the one in which x is set to 1 infinitely often. This run occurs with probability 0. Inside the loop, x is set to 1 or 0 each with probability a half, but thereafter, we condition on the event that x was set to 1. In effect, we thus condition on the only diverging run, i.e. on an event that occurs with probability 0.

If we now ask, for instance, for the conditional probability that the above program terminates, given that no observation is violated, we would expect this to be the undefined fraction „0/0“. To see that our transformer indeed behaves as expected, let us reason about cwp $[\![C]\!]\left(\underline{1}/\underline{1}\right)$. Since the cwp transformer is backward–moving, we first need to study how our transformer behaves on the loop of program C. For that, we will perform the fixed point iteration for the cwp–characteristic function Φ of the loop with respect to postexpectation $\underline{1}/\underline{1}$, given by

$$
\Phi\left(\underline{X}/\underline{Y}\right) \;=\; [x \neq 1] \odot \underline{1}/\underline{1} \;\oplus\; [x = 1] \odot \tfrac{1}{2} \odot \underline{X[x/1]}/\underline{Y[x/1]}\,.
$$

Iterating Φ on the least element $\underline{0}/\underline{1}$ then gives:

$$
\begin{aligned}
\Phi\left(\underline{0}/\underline{1}\right) \;&=\; [x \neq 1] \odot \underline{1}/\underline{1} \;\oplus\; [x = 1] \odot \tfrac{1}{2} \odot \underline{0[x/1]}/\underline{1[x/1]}\\
&=\; [x \neq 1] \odot \underline{1}/\underline{1} \;\oplus\; [x = 1] \odot \underline{0}/\underline{\tfrac{1}{2}}
\end{aligned}
$$

$$
\begin{aligned}
\Phi^2\left(\underline{0}/\underline{1}\right) \;=\; &[x \neq 1] \odot \underline{1}/\underline{1}\\
&\oplus\; [x = 1] \odot \tfrac{1}{2} \odot \left([1 \neq 1] \odot \underline{1}/\underline{1} \;\oplus\; [1 = 1] \odot \underline{0}/\underline{\tfrac{1}{2}}\right)
\end{aligned}
$$

$$
\begin{aligned}
&= [x \neq 1] \odot \underline{1 \,/\, 1} \;\oplus\; [x = 1] \odot \underline{0 \,/\, \tfrac{1}{4}} \\
\Phi^3\left(\underline{0 \,/\, 1}\right) &= [x \neq 1] \odot \underline{1 \,/\, 1} \\
&\qquad \oplus\; [x = 1] \odot \tfrac{1}{2} \odot \left([1 \neq 1] \odot \underline{1 \,/\, 1} \;\oplus\; [1 = 1] \odot \underline{0 \,/\, \tfrac{1}{4}} \right) \\
&= [x \neq 1] \odot \underline{1 \,/\, 1} \;\oplus\; [x = 1] \odot \underline{0 \,/\, \tfrac{1}{8}} \\
&\;\;\vdots \\
\Phi^n\left(\underline{0 \,/\, 1}\right) &= [x \neq 1] \odot \underline{1 \,/\, 1} \;\oplus\; [x = 1] \odot \underline{0 \,/\, \tfrac{1}{2^n}}, \quad \text{for } n \geq 1 \\
&= \underline{[x \neq 1] \,/\, [x \neq 1] \;+\; [x = 1] \cdot \tfrac{1}{2^n}} \\
&\;\;\vdots \\
\Phi^\omega\left(\underline{0 \,/\, 1}\right) &= \underline{[x \neq 1] \,/\, [x \neq 1]}
\end{aligned}
$$

The last line is the conditional weakest preexpectation of the loop. Finally, we have to calculate

$$\mathsf{cwp}\,[\![x := 1]\!]\left(\underline{[x \neq 1] \,/\, [x \neq 1]}\right) \;=\; \underline{0 \,/\, 0}\,,$$

which is the conditional weakest preexpectation of the whole program C.

As we can see, we get the conditional expectation $\underline{0 \,/\, 0}$ as the result of our calculations, which corresponds exactly to what we would expect, namely „0/0“, except that our preexpectation is a a perfectly well–defined mathematical object. Our conditional preexpectation $\underline{0 \,/\, 0}$ tells us on the one hand that the probability of terminating and not violating any observations is 0, and on the other hand that the probability to not violate any observations is also 0.

In order to understand how the cwp transformer behaves for loops on a more abstract level, let us revisit our definition of the cwp transformer for loops and our definition of the order $\trianglelefteq$. By closer inspection, we can notice that for the fixed point iteration of the cwp–characteristic function we have

$$\underset{\langle \varphi, C \rangle}{\mathsf{cwp}}\Phi^n_{\underline{f \,/\, g}}\left(\underline{0 \,/\, 1}\right) \;=\; \underline{\underset{\langle \varphi, C \rangle}{\mathsf{wp}}\Phi^n_f(0) \,\Big/\, \underset{\langle \varphi, C \rangle}{\mathsf{wlp}}\Phi^n_g(1)}\,.$$

Thus, the first components of the chain

$$\underline{0 \,/\, 1} \;\trianglelefteq\; \underset{\langle \varphi, C \rangle}{\mathsf{cwp}}\Phi_{\underline{f \,/\, g}}\left(\underline{0 \,/\, 1}\right) \;\trianglelefteq\; \underset{\langle \varphi, C \rangle}{\mathsf{cwp}}\Phi^2_{\underline{f \,/\, g}}\left(\underline{0 \,/\, 1}\right) \;\trianglelefteq\; \cdots$$

give the ascending chain

$$0 \;\preceq\; \underset{\langle \varphi, C \rangle}{\mathsf{wp}}\Phi_f(0) \;\preceq\; \underset{\langle \varphi, C \rangle}{\mathsf{wp}}\Phi^2_f(0) \;\preceq\; \underset{\langle \varphi, C \rangle}{\mathsf{wp}}\Phi^3_f(0) \;\preceq\; \cdots$$

and the second components give the descending chain

$$1 \;\geq\; {}^{\mathsf{wlp}}_{\langle \varphi, C\rangle}\Phi_g(1) \;\geq\; {}^{\mathsf{wlp}}_{\langle \varphi, C\rangle}\Phi^2_g(1) \;\geq\; {}^{\mathsf{wlp}}_{\langle \varphi, C\rangle}\Phi^3_g(1) \;\geq\; \cdots\,.$$

For the above to make sense and be well–defined, we need to define both transformers $\mathsf{wp}\,[\![\texttt{observe}\,(\varphi)]\!]$ and $\mathsf{wlp}\,[\![\texttt{observe}\,(\varphi)]\!]$. We can do this by

$$\mathsf{wp}\,[\![\texttt{observe}\,(\varphi)]\!]\,(f) \;=\; [\varphi]\cdot f \;=\; \mathsf{wlp}\,[\![\texttt{observe}\,(\varphi)]\!]\,(f)\,.$$

When taking the limits of the latter two chains above, we see that our definition of cwp corresponds to a

$$\frac{\mathsf{wp}\,[\![C]\!]\,(f)}{\mathsf{wlp}\,[\![C]\!]\,(1)}$$

interpretation of conditional expected values, i.e. we indeed normalize only on the probability of not violating any observations and explicitly account for diverging runs that do not violate any observations.

Because of the difficulties that arise when trying to understand conditioning *within* loops, some authors have the opinion that conditioning within loops should be forbidden altogether. This, however, does not eradicate the need to exercise great caution when combining loops and conditioning. Consider for that the, again very unsubtle yet descriptive, program C', given by

$$\{x := 2\}\,[1/2]\,\{\texttt{diverge}\}\,.$$

This program has exactly one diverging run, namely the one in which `diverge` is executed. This run occurs with probability $1/2$. As the program is `observe`–free, the probability to not violate any observation is 1.

If we now for instance ask for the conditional expected value of x, given that no observation is violated, we would thus expect this to be

$$\frac{\overbrace{\tfrac{1}{2}\cdot 2}^{\text{EV of } x \text{ in non–diverging run}} \;+\; \overbrace{\tfrac{1}{2}\cdot 0}^{\text{EV of } x \text{ in diverging run}}}{\underbrace{1}_{\text{Probability of no observation violation}}} \;=\; \frac{1}{1} \;=\; 1\,.$$

Indeed, if we use the cwp transformer and calculate $\mathsf{cwp}\,[\![C']\!]\left(\underline{x}/\underline{1}\right)$, this yields

$$\mathsf{cwp}\,[\![C']\!]\left(\underline{x}/\underline{1}\right) \;=\; \tfrac{1}{2}\odot\left(\underline{2}/\underline{1}\oplus\underline{0}/\underline{1}\right) \;=\; \underline{1}/\underline{1}\,.$$

Since the program C' is `observe`–free, we would expect that the result from our cwp transformer is backward–compatible to what the standard wp transformer would yield. And indeed, the interpretation of $\underline{1}/\underline{1}$, namely as $1/1 = 1$, agrees with $\mathsf{wp}\,[\![C']\!]\,(x) = 1$.

8.3.2 *The Nori Interpretation*

While we just saw that our cwp transformer gives a reasonable, expected, and backward–compatible result when combining conditioning and loops, previous attempts to give semantics to probabilistic programs would have yielded a different result. The semantics of Nori *et al.* [Nor+14] follows a

$$\frac{\mathsf{wp}\,[\![C]\!]\,(f)}{\mathsf{wp}\,[\![C]\!]\,(1)}$$

interpretation (in fact: Nori *et al.* define the semantics of a probabilistic program with conditioning to be *the fraction* above), i.e. Nori *et al.* normalize on the probability of not violating any observations *and successfully terminating*. We call this the *Nori interpretation* of conditional expected values. Put in terms of the three sets A, B, and C we described in Section 8.1, the Nori interpretation expresses the quantity

$$\frac{\mathsf{EV}\big(f\cdot[\neg \mathrm{B}]\big)}{\mathsf{Pr}(\neg \mathrm{B}\cap \mathrm{A})} = \frac{\mathsf{EV}\big(f\cdot[\mathrm{A}]+f\cdot[\mathrm{C}]\big)}{\mathsf{Pr}\big((\mathrm{A}\cup \mathrm{C})\cap \mathrm{A}\big)} = \frac{\mathsf{EV}(f\cdot[\mathrm{A}])}{\mathsf{Pr}(\mathrm{A})}\,,$$

which as we can see does not renormalize to the event that no observation was violated, but instead to the potentially *less likely* event that the program terminates successfully.

For the above–mentioned program C', given by

$$\{x := 2\}\,[1/2]\,\{\texttt{diverge}\}\,,$$

the Nori interpretation would yield

$$\frac{\overbrace{\frac{1}{2}\cdot 2}^{\text{EV of } x \text{ in non–diverging run}} + \overbrace{\frac{1}{2}\cdot 0}^{\text{EV of } x \text{ in diverging run}}}{\underbrace{\frac{1}{2}}_{\substack{\text{Probability of no observation violation}\\ \text{and termination}}}} = \frac{1}{\frac{1}{2}} = 2$$

as the conditional expected value of x. The semantics of Nori *et al.* is hence not backward–compatible on `observe`–free programs, since

$$\mathsf{wp}\,[\![C']\!]\,(x) = 1 \neq 2\,.$$

Instead, the Nori semantics is only *conditionally backward–compatible*, namely whenever the program in question terminates almost–surely. We believe that this is undesirable. If we believe that conditioning should renormalize to *all* runs that do not violate any observation, the cwp interpretation should be preferred over the Nori interpretation.

Another undesirable fact is that the Nori interpretation impedes monotonic reasoning: Overapproximating $^{\mathsf{wp}[\![C]\!](f)}/_{\mathsf{wp}[\![C]\!](1)}$ calls for overapproximating $\mathsf{wp}[\![C]\!](f)$ and underapproximating $\mathsf{wp}[\![C]\!](1)$. The latter, however, is difficult, as we have extensively discussed through Sections 5.2.3 to 5.2.6.

Overapproximating the result of the cwp interpretation, on the other hand, is easier: Overapproximating $^{\mathsf{wp}[\![C]\!](f)}/_{\mathsf{wlp}[\![C]\!](1)}$ calls for overapproximating $\mathsf{wp}[\![C]\!](f)$ and underapproximating $\mathsf{wlp}[\![C]\!](1)$, both of which can be done by means of simple invariant–based techniques, see Sections 5.2.1 and 5.2.2. We discuss this further in Section 8.6.

As another descriptive example, consider the program `diverge`. We have

$$\mathsf{cwp}[\![\texttt{diverge}]\!]\left(\underline{f/1}\right) = \underline{0/1}\,.$$

The 0 comes from the fact that the program diverges and hence cannot produce any mass contributing to an expected value of f evaluated in a final state. The 1 on the other hand also comes from the fact that `diverge` is a shorthand for `while(true){skip}`, which is `observe`–free and thus the probability to not violate any observation while executing this loop is 1. Our approach thus yields a $0/1$–interpretation for the conditional expected value of f, given that no observation is violated, whereas the Nori interpretation would leave us with the problematic „0%“ case. Indeed, Nori *et al.* would in this case define the semantics of `diverge` to be the *undefined fraction* 0%.

8.3.3 *The* cwlp *Interpretation for Partial Correctness*

Besides total correctness, we also address partial correctness by means of the cwlp transformer. Nori *et al.* do not consider this. Again revisiting the program `diverge`, our cwlp transformer yields

$$\mathsf{cwlp}[\![\texttt{diverge}]\!]\left(\underline{[F]/1}\right) = \underline{1/1}\,.$$

The first 1 now comes from the fact that the probability that `diverge` either „establishes F“ or diverges while at the same time not violating any observations is 1. The second 1 comes just from the probability to not violate any observation. Our approach thus yields a $1/1$–interpretation for the conditional probability to either diverge or terminate successfully and establish event F, given that no observation is violated.

Again, in order to understand how the cwlp transformer behaves for loops on a more abstract level, let us revisit our definition of the cwlp transformer for loops and our definition of the order $\trianglelefteq$. By closer inspection, we can notice that the fixed point iteration of the cwlp–characteristic function gives

$$\underset{\langle\varphi,C\rangle}{\mathsf{cwlp}}\Phi^n_{\underline{f/g}}\left(\underline{0/1}\right) = \underline{\underset{\langle\varphi,C\rangle}{\mathsf{wlp}}\Phi^n_f(0)}\Big/\overline{\underset{\langle\varphi,C\rangle}{\mathsf{wlp}}\Phi^n_g(1)}\,.$$

Thus, the first component of the chain

$$\underline{1/1} \;\preceq\; {}^{\mathsf{cwlp}}_{\langle\varphi,C\rangle}\Phi_{\underline{f/g}}\left(\underline{1/1}\right) \;\preceq\; {}^{\mathsf{cwlp}}_{\langle\varphi,C\rangle}\Phi^2_{\underline{f/g}}\left(\underline{1/1}\right) \;\preceq\; \cdots$$

gives the descending chain

$$1 \;\geq\; {}^{\mathsf{wlp}}_{\langle\varphi,C\rangle}\Phi_f(1) \;\geq\; {}^{\mathsf{wlp}}_{\langle\varphi,C\rangle}\Phi^2_f(1) \;\geq\; {}^{\mathsf{wlp}}_{\langle\varphi,C\rangle}\Phi^3_f(1) \;\geq\; \cdots$$

and the second component gives the descending chain

$$1 \;\geq\; {}^{\mathsf{wlp}}_{\langle\varphi,C\rangle}\Phi_g(1) \;\geq\; {}^{\mathsf{wlp}}_{\langle\varphi,C\rangle}\Phi^2_g(1) \;\geq\; {}^{\mathsf{wlp}}_{\langle\varphi,C\rangle}\Phi^3_g(1) \;\geq\; \cdots\,.$$

When taking the limits of the latter two chains, we see that our definition of cwlp corresponds to a

$$\frac{\mathsf{wlp}\,[\![C]\!]\,(f)}{\mathsf{wlp}\,[\![C]\!]\,(1)}$$

interpretation of conditional weakest liberal preexpectations, i.e. we indeed consider the probability to either diverge or establish some event and normalize only on the probability of not violating any observations, explicitly accounting for diverging runs that do not violate any observations.

8.3.4 *A Fourth Interpretation*

So far, we have seen three possibilities to combine wp and wlp into a fraction. We have also studied under which circumstances they make sense and how the quantity they express should be interpreted. There exists a fourth possibility to combine wp and wlp into a fraction, namely

$$\frac{\mathsf{wlp}\,[\![C]\!]\,(f)}{\mathsf{wp}\,[\![C]\!]\,(1)}\,.$$

We have not studied that possibility yet — and for a good reason: This fourth interpretation is not meaningful. As for an illustration, consider the program

$$\{\texttt{skip}\}\,[1/2]\,\{\texttt{diverge}\}\,.$$

If we now ask, for instance, for the conditional *probability* of diverging or terminating, given that the program does not violate any observations and terminates, the above interpretation would yield

$$\frac{\overbrace{\tfrac{1}{2}\cdot 1}^{\text{Prob. of div. or term. in non–div. run}} + \overbrace{\tfrac{1}{2}\cdot 1}^{\text{Prob. of div. or term. in div. run}}}{\underbrace{\tfrac{1}{2}}_{\text{Probability of no observation violation and termination}}} \;=\; \frac{1}{\frac{1}{2}} \;=\; 2\,,$$

The cwp interpretation:	**The Nori interpretation:**
$\frac{\mathsf{wp}\,[\![C]\!]\,(f)}{\mathsf{wlp}\,[\![C]\!]\,(1)}$	$\frac{\mathsf{wp}\,[\![C]\!]\,(f)}{\mathsf{wp}\,[\![C]\!]\,(1)}$
(for general total correctness)	(for total correctness, *only for a.–s. terminating programs*)
The cwlp interpretation:	**The fourth interpretation:**
$\frac{\mathsf{wlp}\,[\![C]\!]\,(f)}{\mathsf{wlp}\,[\![C]\!]\,(1)}$	$\frac{\mathsf{wlp}\,[\![C]\!]\,(f)}{\mathsf{wp}\,[\![C]\!]\,(1)}$
(for partial correctness)	(*nonsensical, can yield probabilities* > 1)

Figure 8.3: The four possibilities of combining wp and wlp in order to make up a conditional expected value of f after executing program C.

which is a „probability“ larger than 1 and hence nonsensical.

Put in terms of the three sets A, B, and C we described in Section 8.1, the fourth interpretation expresses the quantity

$$\frac{\mathsf{Pr}\big((F \cup \mathrm{C}) \cap \neg \mathrm{B}\big)}{\mathsf{Pr}(\neg \mathrm{B} \cap \mathrm{A})} = \frac{\mathsf{Pr}(F \cup \mathrm{C})}{\mathsf{Pr}(\mathrm{A})}\,,$$

which does not describe a conditional probability.

An overview of all possible interpretations can be found in Figure 8.3. To summarize, all four interpretations agree on almost–surely terminating programs. This, however, is hard to ask from a programmer, as the halting problem is already undecidable for deterministic programs. For non–almost–surely terminating programs, the Nori interpretation is not backward–compatible to the standard wp calculus for `observe`–free programs, which is arguably undesirable. The fourth interpretation is not at all meaningful.

The cwp and the cwlp interpretation are suitable for reasoning about total and partial correctness of probabilistic programs with conditioning, regardless of the program's termination behavior. Moreover, the two transformers are backward–compatible with wp and wlp, respectively.

8.4 CONDITIONING AND NONDETERMINISM

Nondeterminism is a powerful means for underspecifying program behavior. By a nondeterministic choice, we can state that a program should behave either in this or that way, but definitely in one of the two specified

ways. However, while underspecified program behavior might be good for modeling purposes, a nondeterministic program *cannot be executed*, i.e. it does not describe a single algorithm but rather a set of algorithms.

Probabilistic programs, as understood in this section, describe algorithmic procedures that construct complex probability distributions. Nondeterminism does not quite fit into this picture. A probabilistic program with nondeterminism would give a set of probability distributions and may thus not be very relevant for modeling probability distributions. In fact, Gordon *et al.* state that „the use of nondeterminism as a modeling tool for representing unknown quantities in probabilistic programs is not common" [Gor+14].

Combining nondeterministic and probabilistic behavior is well–known to be problematic [Pan01; MOW04; Mis06; VW06; AR08; CS09; Bai+14]. For probabilistic programs with observations, we add yet another problem.

There are (at least) two ways to interpret nondeterministic choice: As an *adversary* or as *alternative implementations*. If we take upon the latter point of view, we take the stance that a nondeterministic choice $\{C_1\} \,\square\, \{C_2\}$ can be replaced by either C_1 or C_2, which is a standard assumption in program refinement [BW89]. Under this mild assumption, and even for loop–free programs, we can show that we cannot define a demonic expectation transformer *by induction on the program structure*, i.e. we cannot simply add a line to Table 8.1 in order to have a cwp rule for nondeterministic choice. We will not treat the proof of this result in this thesis as it is somewhat technical, but refer to [Jan+15a] and [Olm+18] for a detailed treatment.

8.5 HEALTHINESS CONDITIONS

Just like the classical expectation transformers we studied in Chapter 4, conditional expectation transformers too enjoy several properties like continuity, monotonicity, etc. (often called healthiness conditions, cf. Section 4.2). In the following, we present some of those.

8.5.1 *Continuity*

Continuity is perhaps one of the most fundamental properties that conditional expectation transformers enjoy because it ensures for instance well–definedness of the conditional expectation transformer semantics of while loops. A conditional expectation transformer $\mathcal{T} : \mathbb{C} \to \mathbb{C}$ is called $\trianglelefteq$–continuous iff for any chain of expectations $S = \{s_0 \trianglelefteq s_1 \trianglelefteq s_2 \trianglelefteq \ldots\} \subseteq \mathbb{C}$ we have

$$\mathcal{T}(\sup_{\trianglelefteq} S) \;=\; \sup_{\trianglelefteq} \mathcal{T}(S)\,;$$

see Definition A.2 for more details. $\blacktriangleleft$–continuity is defined analogously. Both conditional expectation transformers we have presented in this chapter are continuous with respect to their associated partial order.

THEOREM 8.7 (Continuity of cwp and cwlp [Olm+18]):
Let C be a cpGCL *program. Then:*

A. $\mathsf{cwp}\,[\![C]\!]$ *is* $\trianglelefteq$*–continuous.*

B. $\mathsf{cwlp}\,[\![C]\!]$ *is* $\blacktriangleleft$*–continuous.*

Proof. By induction on the structure of C. Q.E.D.

The importance of continuity for well–defined semantics of loops can be sketched as follows: For any loop–free program C, continuity of $\mathsf{cwp}\,[\![C]\!]$ ensures that the characteristic function of the loop $\texttt{while}(\varphi)\{C\}$ (that has C as its loop body) is also continuous. This ensures by the Kleene fixed point theorem (Theorem A.5) that the characteristic function has a least fixed point, which in turn ensures that $\mathsf{cwp}\,[\![\texttt{while}(\varphi)\{C\}]\!]$ is well–defined. The fact that the transformer $\mathsf{cwp}\,[\![\texttt{while}(\varphi)\{C\}]\!]$ itself is also continuous ensures well–defined expectation transformer semantics of nested loops.

8.5.2 *Decoupling*

In Section 8.3 have already hinted at the fact that cwp and cwlp can in some way be decoupled into wp and wlp. If we define

$$\begin{aligned}\mathsf{wp}\,[\![\texttt{observe}(\varphi)]\!]\,(f) &= [\varphi]\cdot f\\ \mathsf{wlp}\,[\![\texttt{observe}(\varphi)]\!]\,(g) &= [\varphi]\cdot g\,,\end{aligned}$$

then we can make this statement more formal:

THEOREM 8.8 (Decoupling of cwp and cwlp [Jan+15a; Olm+18]):
Let $C \in$ pGCL. Then:

A. $\mathsf{cwp}\,[\![C]\!]\left(\underline{f}\middle/\overline{g}\right) = \underline{\mathsf{wp}\,[\![C]\!]\,(f)}\big/\overline{\mathsf{wlp}\,[\![C]\!]\,(g)}$

B. $\mathsf{cwlp}\,[\![C]\!]\left(\underline{f}\middle/\overline{g}\right) = \underline{\mathsf{wlp}\,[\![C]\!]\,(f)}\big/\overline{\mathsf{wlp}\,[\![C]\!]\,(g)}$

Proof. By induction on the structure of C. Q.E.D.

8.5.3 *Strictness*

The strictness property of weakest preexpectation transformers states that

$$\mathsf{wp}\,[\![C]\!]\,(0) = 0\,;$$

see Theorem 4.14 A. More abstractly, the least element 0 in the complete lattice $(\mathbb{E}, \preceq)$ is mapped to itself. Dually, the costrictness property of weakest liberal preexpectation transformers states that

$$\mathsf{wlp}\,[\![C]\!]\,(1) = 1\,;$$

see Theorem 4.14 B. Again put more abstractly, the greatest element 1 in the complete lattice $(\mathbb{E}_{\leq 1}, \preceq)$ is mapped to itself.

Conditional expectation transformers are strict and costrict only in a more concrete sense. The following theorem holds:

THEOREM 8.9 ((Co)strictness of cwp and cwlp [Jan+15a; Olm+18]): *Let $C \in \mathsf{pGCL}$. Then:*

A. $\mathsf{cwp}\,[\![C]\!]\left(\frac{0}{1}\right) = \frac{0}{g}$, *where* $g = \mathsf{wlp}\,[\![C]\!]\,(1)$. *Thus, the interpretation of* $\mathsf{cwp}\,[\![C]\!]\left(\frac{0}{1}\right)$ *in any state* σ, *i.e. the fraction* $0/g(\sigma)$, *is either* 0 *or undefined.*

B. $\mathsf{cwlp}\,[\![C]\!]\left(\frac{1}{1}\right) = \frac{g}{g}$, *where* $g = \mathsf{wlp}\,[\![C]\!]\,(1)$. *Thus, the interpretation of* $\mathsf{cwlp}\,[\![C]\!]\left(\frac{1}{1}\right)$ *in any state* σ, *i.e. the fraction* $g(\sigma)/g(\sigma)$, *is either* 1 *or undefined.*

Proof. For proving the strictness of cwp in the sense of Theorem 8.9 A., consider the following:

$$\begin{aligned}
&\mathsf{cwp}\,[\![C]\!]\left(\tfrac{0}{1}\right)\\
&= \tfrac{\mathsf{wp}\,[\![C]\!]\,(0)}{\mathsf{wlp}\,[\![C]\!]\,(1)} && \text{(by decoupling, Theorem 8.8 A.)}
\end{aligned}$$

For proving the costrictness of cwlp in the sense of Theorem 8.9 B., consider the following:

$$\begin{aligned}
&\mathsf{cwlp}\,[\![C]\!]\left(\tfrac{1}{1}\right)\\
&= \tfrac{\mathsf{wlp}\,[\![C]\!]\,(1)}{\mathsf{wlp}\,[\![C]\!]\,(1)} && \text{(by decoupling, Theorem 8.8 B.)}
\end{aligned}$$

Q.E.D.

The above version of strictness tells us that the conditional expected value of the constantly 0 random variable after executing a program C, given that no observation is violated is either 0, or the probability to not valuate any observations is 0. Costrictness on the other hand tells us that the conditional probability to either terminate or not, given that no observation is violated is either 1, or the probability to not valuate any observations is 0.

Notice that cwp is not strict in the more abstract sense we have mentioned in the beginning, i.e. the least element in the underlying partial order is mapped to itself: In particular, the transformer $\mathsf{cwp}\,[\![\mathtt{observe}\,(\varphi)]\!]$ is not strict in that sense, since

$$\mathsf{cwp}\,[\![\mathtt{observe}\,(\varphi)]\!]\left(\tfrac{0}{1}\right) \;=\; [\varphi]\cdot\tfrac{0}{1} \;=\; \tfrac{0}{[\varphi]} \;\neq\; \tfrac{0}{1}\,.$$

Dually, the transformer cwlp is not costrict in the more abstract sense, i.e. the greatest element in the underlying partial order is mapped to itself: Again, the transformer $\mathsf{cwlp}\,[\![\mathtt{observe}\,(\varphi)]\!]$ is not costrict in that sense, since

$$\mathsf{cwlp}\,[\![\mathtt{observe}\,(\varphi)]\!]\left(\tfrac{1}{1}\right) \;=\; [\varphi]\cdot\tfrac{1}{1} \;=\; \tfrac{[\varphi]}{[\varphi]} \;\neq\; \tfrac{1}{1}\,.$$

8.5.4 *Conservativity*

A main advantage of the cwp interpretation of conditional weakest preexpectations (see Section 8.3.1) over the Nori interpretation (see Section 8.3.2) is that the cwp interpretation is backward compatible with classical weakest preexpectations (see Section 4.1.1) for programs that do not contain any observations. Dually, the cwlp interpretation (see Section 8.3.3) is backward compatible with classical weakest liberal preexpectations (see Section 4.1.2). Formally, we can state this as follows:

THEOREM 8.10 (Conservativity of cwp and cwlp [Jan+15a; Olm+18]): *Let $C \in \mathsf{cpGCL}$ be a an* `observe`*–free program. Then:*

A. $\mathsf{cwp}\,[\![C]\!]\left(\frac{f}{1}\right) = \frac{f'}{g'}$ implies $\frac{f'}{g'} = \mathsf{wp}\,[\![C]\!]\,(f)$.

B. $\mathsf{cwlp}\,[\![C]\!]\left(\frac{f}{1}\right) = \frac{f'}{g'}$ implies $\frac{f'}{g'} = \mathsf{wlp}\,[\![C]\!]\,(f)$.

Proof. For the proof of Theorem 8.10 A. consider

$$\begin{aligned}
&\mathsf{cwp}\,[\![C]\!]\left(\frac{f}{1}\right)\\
&= \frac{\mathsf{wp}\,[\![C]\!]\,(f)}{\mathsf{wlp}\,[\![C]\!]\,(1)} && \text{(by decoupling, Theorem 8.8 A.)}\\
&= \frac{\mathsf{wp}\,[\![C]\!]\,(f)}{1} && \text{(by costrictness, Theorem 4.14 B.)}
\end{aligned}$$

and

$$\frac{\mathsf{wp}\,[\![C]\!]\,(f)}{1} = \mathsf{wp}\,[\![C]\!]\,(f)\,.$$

For the proof of Theorem 8.10 B. consider

$$\begin{aligned}
&\mathsf{cwlp}\,[\![C]\!]\left(\frac{f}{1}\right)\\
&= \frac{\mathsf{wlp}\,[\![C]\!]\,(f)}{\mathsf{wlp}\,[\![C]\!]\,(1)} && \text{(by decoupling, Theorem 8.8 B.)}\\
&= \frac{\mathsf{wlp}\,[\![C]\!]\,(f)}{1} && \text{(by costrictness, Theorem 4.14 B.)}
\end{aligned}$$

and

$$\frac{\mathsf{wlp}\,[\![C]\!]\,(f)}{1} = \mathsf{wlp}\,[\![C]\!]\,(f)\,.$$ Q.E.D.

Note that conservativity holds for the Nori interpretation only if the program C terminates almost–surely, which can be nontrivial to assert (see Chapter 6).

8.5.5 *Feasibility*

The property that McIver & Morgan call *feasibility* states that preexpectations cannot become „too large“ [MM05]. The notion of feasibility makes sense for

bounded expectations $f \in \mathbb{E}_{\leq \exists b}$ only (cf. Section 8.5.5). A similar property holds for conditional expectations, too.

A first property we observe for that is that — informally speaking — whenever a conditional postexpectation is not of the form $a/0$, for $a > 0$, then the corresponding preexpectation is also not of that form. This means that if a conditional preexpectation results in a division by zero, it is always of the form „zero divided by zero". Formally, we have the following lemma:

Lemma 8.11:
Let $C \in$ pGCL. Then

$$\forall \sigma \in \Sigma\colon g(\sigma) = 0 \implies f(\sigma) = 0 \quad \text{and} \quad \mathsf{cwp}\,[\![C]\!]\left(\underline{f/g}\right) = \underline{f'/g'}$$
$$\text{implies} \qquad \forall \sigma \in \Sigma\colon g'(\sigma) = 0 \implies f'(\sigma) = 0\,.$$

Proof. By induction on the structure of C. Q.E.D.

The second property is more closely related to the notion of feasibility as studied in Section 4.2.3. Informally speaking, it states that if the interpretation f/g of a conditional postexpectation $\underline{f/g}$ is bounded by a constant b then the interpretation of the corresponding preexpectation is also bounded by b. Formally, we have the following theorem:

Theorem 8.12 (Feasibility of cwp):
Let $C \in$ pGCL. Then

$$\forall \sigma \in \Sigma\colon g(\sigma) > 0 \implies \frac{f(\sigma)}{g(\sigma)} \leq b \quad \text{and} \quad \mathsf{cwp}\,[\![C]\!]\left(\underline{f/g}\right) = \underline{f'/g'}$$
$$\text{implies} \qquad \forall \sigma \in \Sigma\colon g'(\sigma) > 0 \implies \frac{f'(\sigma)}{g'(\sigma)} \leq b\,.$$

Proof. By induction on the structure of C. The only case that is not straightforward is the case for probabilistic choice. Let

$$\mathsf{cwp}\,[\![C_1]\!]\left(\underline{f/g}\right) = \underline{f'/g'} \quad \text{and} \quad \mathsf{cwp}\,[\![C_2]\!]\left(\underline{f/g}\right) = \underline{f''/g''}$$

By the induction hypothesis, we have

$$\forall \sigma \in \Sigma\colon g'(\sigma) > 0 \implies \frac{f'(\sigma)}{g'(\sigma)} \leq b$$
$$\text{and} \quad \forall \sigma \in \Sigma\colon g''(\sigma) > 0 \implies \frac{f''(\sigma)}{g''(\sigma)} \leq b$$

$$\text{implies} \quad \forall \sigma \in \Sigma\colon g'(\sigma) > 0 \implies f'(\sigma) \leq g'(\sigma) \cdot b \qquad (\dagger)$$
$$\text{and} \quad \forall \sigma \in \Sigma\colon g''(\sigma) > 0 \implies f''(\sigma) \leq g''(\sigma) \cdot b\,.$$

For the induction step for probabilistic choice we have

$$\begin{aligned}
&\mathsf{cwp}\,[\![\{\,C_1\,\}\,[p]\,\{\,C_2\,\}]\!]\left(\underline{f/g}\right)\\
&= p\cdot \mathsf{cwp}\,[\![C_1]\!]\left(\underline{f/g}\right) + (1-p)\cdot \mathsf{cwp}\,[\![C_2]\!]\left(\underline{f/g}\right)\\
&= \underline{p\cdot f' + (1-p)\cdot f'' \,/\, p\cdot g' + (1-p)\cdot g''}\,.
\end{aligned}$$

We now fix a state σ in which we interpret the above conditional expectation. If $g'(\sigma) = 0$, then by Lemma 8.11 we also have $f'(\sigma) = 0$. In this case, we can immediately appeal to the induction hypothesis for C_2 as we then have

$$\begin{aligned}
\frac{p\cdot f'(\sigma) + (1-p)\cdot f''(\sigma)}{p\cdot g'(\sigma) + (1-p)\cdot g''(\sigma)} &= \frac{p\cdot 0 + (1-p)\cdot f''(\sigma)}{p\cdot 0 + (1-p)\cdot g''(\sigma)}\\
&= \frac{(1-p)\cdot f''(\sigma)}{(1-p)\cdot g''(\sigma)}\\
&= \frac{f''(\sigma)}{g''(\sigma)}\\
&\leq b\,. \qquad \text{(by I.H. on } C_2\text{)}
\end{aligned}$$

The reasoning is analogous if either $g''(\sigma) = 0$, $p = 0$, or $p = 1$.

In the remaining cases we reason as follows:

$$\begin{aligned}
&\frac{p\cdot f'(\sigma) + (1-p)\cdot f''(\sigma)}{p\cdot g'(\sigma) + (1-p)\cdot g''(\sigma)} \overset{!}{\leq} b\\
\text{iff}\quad & p\cdot f'(\sigma) + (1-p)\cdot f''(\sigma) \leq p\cdot g'(\sigma)\cdot b + (1-p)\cdot g''(\sigma)\cdot b\\
\text{implied by}\quad & p\cdot f'(\sigma) + (1-p)\cdot f''(\sigma) \leq p\cdot f'(\sigma) + (1-p)\cdot f''(\sigma) \quad (\text{by } \dagger)\\
\text{iff}\quad & 0 \leq 0 \qquad \text{Q.E.D.}
\end{aligned}$$

8.5.6 *Monotonicity*

Monotonicity is another fundamental property of conditional expectation transformers. A conditional expectation transformer $\mathcal{T}$ is called $\trianglelefteq$–monotonic iff for any two conditional expectations $\underline{f/g}, \underline{f'/g'} \in \mathbb{C}$, we have that

$$\underline{f/g} \trianglelefteq \underline{f'/g'} \quad \text{implies} \quad \mathcal{T}\left(\underline{f/g}\right) \trianglelefteq \mathcal{T}\left(\underline{f'/g'}\right);$$

see Definition A.3 for more details. $\trianglerighteq$–monotonicity is defined analogously. All conditional expectation transformers we have presented are monotonic with respect to their associated partial order:

THEOREM 8.13 (Monotonicity of cwp and cwlp [Jan+15a; Olm+18]): *Let $C \in$ pGCL. Then:*

1. cwp *is $\trianglelefteq$–monotonic.*

2. cwlp *is* $\blacktriangleleft$*–monotonic.*

Proof. Every continuous function is monotonic, see Theorem A.4. Q.E.D.

Monotonicity is not just a healthiness condition but plays an important role in reasoning about programs, namely for compositional reasoning: Imagine two programs C_1 and C_2 and a postexpectation f such that

$$\mathsf{cwp}\,[\![C_1]\!]\left(\underline{f}/\overline{1}\right) \;\trianglelefteq\; \mathsf{cwp}\,[\![C_2]\!]\left(\underline{f}/\overline{1}\right).$$

Then monotonicity ensures that if we put the components C_1 and C_2 into some context $C \fatsemi \ldots\ldots$, then we can be certain that

$$\mathsf{cwp}\,[\![C \fatsemi C_1]\!]\left(\underline{f}/\overline{1}\right) \;\trianglelefteq\; \mathsf{cwp}\,[\![C \fatsemi C_2]\!]\left(\underline{f}/\overline{1}\right),$$

since $\mathsf{cwp}\,[\![C \fatsemi C_i]\!]\left(\underline{f}/\overline{g}\right) = \mathsf{cwp}\,[\![C]\!]\left(\mathsf{cwp}\,[\![C_i]\!]\left(\underline{f}/\overline{1}\right)\right)$, for $i \in \{1, 2\}$. Note that by construction of $\trianglelefteq$ this implies that the conditional expected value of f, given that no observations fail, is higher after executing $C \fatsemi C_2$ than it is after executing $C \fatsemi C_1$. $\blacktriangleleft$–monotonicity of cwlp — unfortunately — does not come with such a meaningful interpretation, see Section 8.2.2.

8.5.7 *Linearity*

Linearity of expectation transformers plays a prominent role in McIver & Morgan's studies on the classical expectation transformer wp. Our cwp transformer is *not* linear in the sense that

$$\mathsf{cwp}\,[\![C]\!]\left(a \odot \underline{f}/\overline{g} \oplus \underline{f'}/\overline{g'}\right) \;\neq\; a \odot \mathsf{cwp}\,[\![C]\!]\left(\underline{f}/\overline{g}\right) \oplus \mathsf{cwp}\,[\![C]\!]\left(\underline{f'}/\overline{g'}\right). \quad \lightning$$

However, by decoupling of cwp and linearity of wp, we can see that cwp is linear in the *interpretation* of its result, i.e. we have [Jan+15a; Olm+18]:

$$\begin{aligned}
&\mathsf{cwp}\,[\![C]\!]\left(\underline{a \cdot f + g}/\overline{g'}\right) \\
&= \underline{\mathsf{wp}\,[\![C]\!]\,(a \cdot f + g)}/\overline{\mathsf{wlp}\,[\![C]\!]\,(g')} && \text{(by decoupling, Theorem 8.8 A.)} \\
&= \underline{a \cdot \mathsf{wp}\,[\![C]\!]\,(f) + \mathsf{wp}\,[\![C]\!]\,(g)}/\overline{\mathsf{wlp}\,[\![C]\!]\,(g')}\,. \\
& && \text{(by linearity of wp, Theorem 4.21 C.)}
\end{aligned}$$

8.6 PROOF RULES FOR LOOPS

As for weakest preexpectations and expected runtimes, reasoning about loops is a difficult tasks in probabilistic program verification. We have seen in Chapter 5, how invariants can help with this sort of reasoning. In particular, we saw that invariants precisely capture the principles of induction and coinduction.

In this section, we will show how we can reason about conditional preexpectations of loops by means of *conditional invariants*, which basically capture the same notion of invariance as for weakest preexpectations. We will present inductive methods for proving upper bounds on conditional weakest preexpectations of loops, ω–rules for proving lower bounds, and a method for refining obtained bounds.

8.6.1 *Invariants*

The concept of invariants that we employ for the proof rules we present in this section is the same as for weakest preexpectation reasoning, see Section 5.1. The notion of conditional invariants is defined as follows:

Definition 8.14 (Conditional Invariants [Olm+18]):
Let Φ be the cwp*–characteristic function of* `while`$(\varphi)\{C\}$ *with respect to postexpectation $\frac{f}{g} \in \mathbb{C}$. Then $\frac{I}{H} \in \mathbb{C}$ is called a conditional invariant of* `while`$(\varphi)\{C\}$ *with respect to conditional postexpectation $\frac{f}{g}$, iff*

$$\Phi\left(\frac{I}{H}\right) \trianglelefteq \frac{I}{H} .$$

8.6.2 *Induction for Conditional Weakest Preexpectations*

Since conditional weakest preexpectations are defined as least fixed points of continuous functions on complete partial orders, we can make use of the induction principle that we discussed in Section 5.2.1 in order to reason about upper bounds on conditional expected values. Formally, the induction principle states that if $(D, \sqsubseteq)$ is a complete partial order and $\Phi\colon D \to D$ is a continuous self–map on D. Then

$$\forall\, d \in D\colon \quad \Phi(d) \sqsubseteq d \quad \text{implies} \quad \mathsf{lfp}\,\Phi \sqsubseteq d .$$

Applied to the cwp calculus, the induction principle immediately gives us the following proof rule:

Theorem 8.15 (Induction for Upper Bounds on cwp [Olm+18]):
Let $\frac{I}{H} \in \mathbb{C}$ be a conditional superinvariant of `while`$(\varphi)\{C\}$ *with respect to postexpectation $\frac{f}{g}$ (see Definition 8.14). Then*

$$\mathsf{cwp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\left(\frac{f}{g}\right) \trianglelefteq \frac{I}{H} .$$

Proof. This is an instance of Park's Lemma (see Lemma A.6): Simply choose complete partial order $(\mathbb{C}, \trianglelefteq)$ and continuous function ${}^{\mathsf{cwp}}_{\langle\varphi,C\rangle}\Phi_{\frac{f}{g}}$. Q.E.D.

It is worthwhile to repeat here that the conclusion of Theorem 8.15, namely

$$\mathsf{cwp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\left(\frac{f}{g}\right) \trianglelefteq \frac{I}{H} .$$

is not only meaningful in the sense of the partial order $\trianglelefteq$ on conditional expectations, i.e. $\underline{I \,/\, H}$ approximates $\mathsf{cwp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\left(\underline{f \,/\, g}\right)$ from above. Also the *interpretation* $\underline{I \,/\, H}$ as an actual quotient approximates the interpretation of $\mathsf{cwp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\left(\underline{f \,/\, g}\right)$ as a quotient from above, i.e.

$$\Phi\left(\underline{I \,/\, H}\right) \trianglelefteq \underline{I \,/\, H} \quad \text{and} \quad \underline{f' \,/\, g'} \coloneqq \mathsf{cwp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\left(\underline{f \,/\, g}\right)$$
$$\text{implies} \qquad \frac{f'(\sigma)}{g'(\sigma)} \leq \frac{I(\sigma)}{H(\sigma)}\,,$$

for all σ with $g'(\sigma) > 0$. This demonstrates that conditional invariants are a useful notion for overapproximating the actual conditional expected value.

8.6.3 *Coinduction for Conditional Weakest Liberal Preexpectations*

Analogously to the coinduction rule for classical weakest liberal preexpectations (see Section 5.2.2), there is in theory a coinduction rule for obtaining lower bounds on conditional weakest liberal preexpectations since those are defined as greatest fixed points. However, the lower bound obtained from this theoretically existent coinduction rule would be a lower bound with respect to the partial order $\blacktriangleleft$. Unfortunately, this partial order does not give us an lower bound on the actual conditional probability that we seek for (see Section 8.2.2). We will thus not discuss the coinduction rule for cwlp any further as it is of little practical use.

8.6.4 *ω–Rules*

As is the case for weakest preexpectations or expected runtimes, reasoning about lower bounds of conditional weakest preexpectations is difficult since no inductive or coinductive proof principle is available. For weakest preexpectation reasoning and expected runtimes, we thus resorted to so–called ω–rules which employ ω–invariants (see Section 5.2.4 and Section 7.6.5). The same principle is applicable to conditional weakest preexpectations:

Theorem 8.16 (Lower Bounds on cwp from ω–Invariants [Kam+16]): *Let Φ be the* cwp*–characteristic function of* $\texttt{while}(\varphi)\{C\}$ *with respect to postexpectation $\underline{f \,/\, g}$ and let*

$$\underline{I_0 \,/\, H_0} \trianglelefteq \underline{I_1 \,/\, H_1} \trianglelefteq \underline{I_2 \,/\, H_2} \trianglelefteq \ldots\,,$$

with $\underline{I_0 \,/\, H_0} = \underline{0 \,/\, 1}$ be a monotonically increasing sequence of conditional expectations such that for all $n \in \mathbb{N}$

$$\underline{I_{n+1} \,/\, H_{n+1}} \trianglelefteq \Phi\left(\underline{I_n \,/\, H_n}\right).$$

Then

$$\sup_{n\in\mathbb{N}} \underline{I_n \,/\, H_n} \;\trianglelefteq\; \mathsf{cwp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\left(\underline{f \,/\, g}\right).$$

Proof. Analogous to the proof of Theorem 5.9 A. Q.E.D.

Just like for weakest preexpectations or expected runtimes, it is necessary to find the limit of such an ω–invariant in order to actually gain some insights from applying Theorem 7.21. That basically just shifts to problem of obtaining bounds into the realm of real analysis. For further remarks on the — in my personal opinion — poor usability and usefulness and on the expendability of ω–rules for upper bounds, see the remarks on ω–rules for weakest preexpectation reasoning in Section 5.2.4.

8.6.5 *Bound Refinement*

We saw in Section 5.2.7 that once we have obtained by some means some bound — be it upper or lower — on a preexpectation of a loop, we have a chance of refining and thereby tightening this bound fairly easily. Since this technique is rooted in fixed point theory and conditional weakest preexpectations of loops are defined as least fixed points, the same technique can be applied to cwp:

Theorem 8.17 (Bound Refinement for cwp):
Let Φ be the cwp*–characteristic function of* $\texttt{while}(\varphi)\{C\}$ *with respect to postexpectation* $\underline{f \,/\, g}$*. Moreover, let* $\underline{I \,/\, H}$ *be an upper bound on the preexpectation* $\mathsf{cwp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\left(\underline{f \,/\, g}\right)$*, such that* $\Phi(\underline{I \,/\, H}) \trianglelefteq \underline{I \,/\, H}$*.*

Then $\Phi(\underline{I \,/\, H})$ *is also an upper bound on* $\mathsf{cwp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\left(\underline{f \,/\, g}\right)$*. Moreover, whenever* $\Phi(\underline{I \,/\, H}) \neq \underline{I \,/\, H}$*, then* $\Phi(\underline{I \,/\, H})$ *is an even tighter upper bound on* $\mathsf{cwp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\left(\underline{f \,/\, g}\right)$ *than* $\underline{I \,/\, H}$*.*

Dually, if $\underline{I \,/\, H}$ *is a lower bound, such that* $\underline{I \,/\, H} \trianglelefteq \Phi(\underline{I \,/\, H})$*, then* $\Phi(\underline{I \,/\, H})$ *is also a lower bound; and whenever* $\Phi(\underline{I \,/\, H}) \neq \underline{I \,/\, H}$*, then* $\Phi(\underline{I \,/\, H})$ *is an even tighter lower bound than* $\underline{I \,/\, H}$*.*

Proof. Analogous to the proof of Theorem 5.15. Q.E.D.

The particular bound refinement of Theorem 8.17 can of course be continued ad infinitum: For instance, if $\underline{I \,/\, H}$ is an upper bound on the preexpectation $\mathsf{cwp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\left(\underline{f \,/\, g}\right)$ with $\Phi(\underline{I \,/\, H}) \trianglelefteq \underline{I \,/\, H}$, then so is $\Phi(\underline{I \,/\, H})$ but also $\Phi^2(\underline{I \,/\, H})$, $\Phi^3(\underline{I \,/\, H})$, and so on. In fact, for increasing n, the sequence $\Phi^n(\underline{I \,/\, H})$ is decreasing and *converges* to an upper bound on $\mathsf{cwp}\,[\![\texttt{while}(\varphi)\{C\}]\!]\left(\underline{f \,/\, g}\right)$ that is *below (or equal to)* $\underline{I \,/\, H}$. For more details, see Section 5.2.7.

8.7 FUTURE AND RELATED WORK

Assertions from classical programming languages correspond to the tests in Kozen's probabilistic propositional dynamic logic [Koz85] and are probably the most closely related concept to observations in probabilistic programming. Both `observe` (φ) and `assert` (φ) block all program executions violating φ. However, `observe` (φ) normalizes the unblocked executions with respect to the total probability mass of all non–violating executions. `assert` (φ), on the other hand, does not renormalize, yielding a subdistribution with a total probability mass of possibly less than one.

A different — more quantitative — interpretation of assertions in probabilistic programming is studied by Sampson *et al.* [Sam+14]. There, assertions are accompanied by a confidence value c and a probability value p. The meaning of such a quantitative assertion is that with confidence c, the assertion holds with probability (at least) p. Assertions in probabilistic programming have also been considered by Chakarov & Sankaranarayanan [CS13].

Beyond assertions, Bichsel *et al.* have extended our work by considering *exceptions* [BGV18]. These result in error states that are neither due to observation violation nor nontermination but rather due to other undesired program behavior, which can explicitly be witnessed. Furthermore, Bichsel *et al.* extend pGCL by a `score` statement, which allows to increase or decrease the probability of specific program executions. The `score` statement can be regarded as a generalization of the `observe` statement and in fact renders `observe` statements obsolete. In the following, we list other related work, sorted by topic, and point to directions for future work.

Measure vs. expectation transformers. Giving semantics to probabilistic programs can be done either in terms of forward moving measure transformers or in terms of backward moving expectation transformers. One of the first measure transformer semantics for probabilistic programs with conditioning was given by Borgström *et al.* [Bor+11]. We, on the other hand, have presented in this chapter an expectation transformer semantics. A semantics similar to ours has been provided by Nori *et al.* [Nor+14].

Nontermination. The main difference between our work and the work of Nori *et al.* is that our expectation transformers explicitly account for nonterminating program behavior, whereas Nori *et al.* normalize only to terminating runs. We presented a thorough comparison in Section 8.3.

Several works on probabilistic programs assume almost–sure or even *certain* termination, see e.g. [Bor+11; CMR13; Hur+14; Sam+14]. For some applications, restricting to almost–sure termination is understandable. In general, this restriction should not be made: For instance, Icard argues that cognitive models based on probabilistic programs should not be restricted to almost–surely terminating programs, for *both theoretical and practical rea-*

sons [Ica17]. Therefore, semantics of a general–purpose probabilistic programming language with conditioning should account for nontermination.

Computability. Inference for probabilistic programs is obviously undecidable, once we consider programs with loops (on the degree of undecidability, see Part III). For probabilistic programs with conditioning, the situation is even more subtle: Ackerman *et al.* have shown that one can construct two *computable* random variables X and Y, such that the conditional probability $\Pr(X \mid Y)$ is *not computable* [AFR11]. In other words: The operation of conditioning itself is what already introduces noncomputability.

In light of the above results, probabilistic inference in the presence of conditioning is arguably a difficult task, which renders overapproximations using our invariant–based rules even more useful.

Nondeterminism. In Section 8.4, we have sketched that we cannot come up with a conditional expectation transformer for nondeterministic programs that is constructed by induction on the structure of the program (for more details, see [Jan+15a; Olm+18]). Intuitively, an inductive expectation transformer can only look into the future (from a program execution time line point of view), but not into the past. For conditional expected values, however, looking into the past appears to be necessary, as, amongst many others, Chen & Sanders noticed [CS09]:

> *[S]tudies have revealed an unsuspected subtlety in the interaction between nondeterministic and probabilistic choices that can be summarised: the demon resolving the nondeterministic choice has memory of previous state changes, whilst the probabilistic choice is made spontaneously.*

This insight is also related to the fact that for conditional probabilities in Markov decision processes, memoryless schedulers (schedulers that on every visit to a state always take the same decision) are insufficient. Instead, history–dependent schedulers are needed, see [AR08; Bai+14].

Program transformation and slicing. Most program transformations for probabilistic programs, such as slicing [Hur+14] aim to accelerate the Markov Chain Monte Carlo analysis. In [Jan+15a] and [Olm+18], the two papers this chapter is based upon, two program transformations that eliminate and one that introduces `observe` statements are presented:

The first transformation „hoists“ the `observe` statements upwards through the program while updating the probabilistic choices. This technique is similar in spirit to [Nor+14]. The result of the hoisting process is a semantically equivalent `observe`–free program.

The second transformation basically recreates rejection sampling: It introduces one outer while loop around the original program. This while loop

executes the original program and sets a flag if an observation is violated during execution. If an observation has been violated, the outer loop reruns the original program. This process is repeated until no observation has been violated. The idea to rerun a program until all observations are passed is also used by [Bai+14] to automate the verification of conditioned temporal logic formulas in Markov models.

The second program transformation has been successfully applied in reasoning about expected sampling times of Bayesian networks [Bat+18b]. It was found that for certain large networks, obtaining a *single sample* can take *millions of years* in expectation using rejection sampling. Since rejection sampling is the *de facto* semantics for inference on most practical probabilistic programming languages, this connection shows that our cwp semantics is a real alternative to rejection sampling.

A third transformation is to replace an independent and identically distributed loop by an `observe` statement, i.e. in principle reversing the second transformation. This has strong resemblances with arguments made in textbooks on randomized algorithms, see e.g., [Sho09, Theorem 9.3.(iii)].

Random assignments from continuous distributions. The measure transformer semantics of Borgström *et al.* [Bor+11] includes sampling from continuous distributions like Gaussians, etc., and also the consequential possibility of conditioning on zero–probability events. However, Borgström *et al.* do not consider unbounded loops or unbounded recursion. We believe it would be a promising direction for future work to develop an expectation transformer semantics that can cope with both sampling from continuous distributions and possible nontermination. Such an endeavor will most likely involve in some way the disintegration theorem [Wike]. This theorem is already subject to intensive research on semantics of probabilistic programs [SR17; CJ17; Koz18].

Conditional expected runtimes. A second direction for future work we propose is to marry the ert calculus from Chapter 7 for reasoning about expected runtimes and the cwp calculus from this chapter in order to obtain a calculus for reasoning about *conditional expected runtimes*. This would enable reasoning about expected runtimes of randomized algorithms restricted to certain situations of interest.

Questions like these can lead to extremely counterintuitive situations. As a simple example, consider how often in expectation we need to throw a die in order to get a 6, *given* that all throws yield an even number. Surprisingly, the answer is 1.5, which is twice as fast as throwing a 3–sided die with numbers 2, 4, and 6 [Jin18].

MIXED–SIGN EXPECTATIONS 9

The vast majority of expectation–based techniques for reasoning about probabilistic programs, including all techniques presented so far in this thesis, make an important — though restrictive — assumption: the postexpectations, i.e. the random variables whose expected value we are interested in, map program states to the *non–negative* reals, cf. Chapter 5 and the various references therein. In other words, those approaches cannot handle *mixed–sign* postexpectations, i.e. expectations that can potentially map into both the positive and negative reals. McIver & Morgan even explicitly forbid mixed–sign expectations altogether and argue [MM05, pp. 70]:

> *For mixed–sign or unbounded expectations* […] *well–definedness is not assured: such cases must be treated individually.* […] *That is, although* [a program] *prog itself may be well defined, the* [weakest] *preexpectation wp.prog.*$(-2)^n$ *is not — and that is a good reason for avoiding mixed signs in general.*

A workaround is to assume bounded negative values [MM01], but this is not always possible and thus provides no general solution.

At first sight, avoiding mixed–sign expectations looks like a minor technical restriction. In practice it is not: For instance, program variables may become negative during program execution, having a negative impact of f's value. As another example, analyzing the efficiency of data structures such as randomized splay trees [AK02] is typically done using amortized analysis. Such an analysis is similar to expected runtime analysis, but is concerned with the cost averaged over a sequence of operations. In the *accounting method* and in the *potential method* in amortized analysis, a decrease in potential (or credit) „pays for“ particularly expensive operations whereas increases model cheap operations. The amortized cost during the execution of a probabilistic routine may thus become arbitrarily negative. Finally, we mention that negative expectations or even negative probabilities have applications in quantum computing and finance [Dir42; Hau04; BM12].

Previously to [KK17b] and [KK17c], on which this chapter is based on, expectation–based approaches could not handle the aforementioned scenarios off–the–shelf. A workaround is to perform a *Jordan decomposition* of f into

$$f \;=\; {}^{+}f - {}^{-}f \;,$$

where ${}^{+}f = \max\{f, 0\}$ and ${}^{-}f = \min\{f, 0\}$ are both non–negative expectations, and analyze ${}^{+}f$ and ${}^{-}f$ individually using the classical wp calculus.

This, however, can easily become quite involved, for example when trying to reason about the expected value of x after execution of

$$
\begin{aligned}
&c := 1 \,⨾\\
&\texttt{while}(c = 1)\{\\
&\qquad \{c := 0\} \,[1/2]\, \{x := -x - \mathsf{sign}(x)\}\\
&\}\,.
\end{aligned}
$$

In every iteration, a fair coin is flipped to decide whether to terminate the loop or change the sign of x and increase its distance to 0, followed by a recursive execution of the entire loop. Intuitively, this program computes a variant of a geometric distribution on x where the sign alternates with increasing absolute value of x. The expected value of x after execution of the above program is given by

$$\frac{x}{3} - \frac{\mathsf{sign}(x)}{9}\,.$$

A detailed comparison between tackling this analysis by the methods presented in this chapter and a Jordan–decomposition–based approach is provided in [KK17c, Appendix B]. To summarize the comparison: using Jordan–decomposition takes considerably more effort.

Despite the existence of a mathematical theory of signed random variables, there are good reasons why they are avoided in current expectation–based approaches: the notion of expectation needs to be revisited, and a complete lattice on these adapted expectations — the de facto gold standard of giving semantics to loops — is required. It turns out that this is not trivial.

In this chapter, we present a sound weakest preexpectation calculus for probabilistic programs that directly acts on mixed–sign expectations f without decomposing them. In particular, *our semantics is always defined regardless of whether classical preexpectations* [Koz85; MM05; Heh11] *exist or not*. We start by redefining what an expectation that can be negative in fact is. The crux of our approach is to keep track of the *integrability* of the mixed–sign random variable f by accompanying f with a non–negative (but possibly infinite) expectation g that bounds $|f|$. Notice that we *do not* require f to be integrable as we want our semantics to be well–defined regardless of whether f is integrable or not.

We will start by showing some instances that exemplify the issues that would occur with mixed–sign postexpectations when naively letting the classical wp calculus act on mixed–sign expectations. Thereafter, we present a new notion of mixed–sign expectations called *integrability–witnessing expectations*, which incorporate the aforementioned bookkeeping for the integrability of the expectations and then develop a weakest preexpectation calculus which acts on integrability–witnessing expectations. As with other calculi

presented in this thesis, we also present some basic properties of the new calculus and rules for reasoning about loops.

In order not to clutter the presentation and in order to put a focus on the issues that arise from mixing positive and negative values, we consider in this entire chapter *only tame programs*, cf. Definition 3.1 E.

9.1 CONVERGENCE AND DEFINEDNESS ISSUES

The classical expectation transformers we have studied in Chapter 4 act on expectations that map program states to *positive* real numbers or infinity. If we wish to reason about a postexpectation f that may also assume *negative* values, the corresponding preexpectation $\mathsf{wp}\,[\![C]\!]\,(f)$, i.e. the expected value of f after termination of C, might not be defined for several reasons. In the following, we present two problematic cases.

Indefinite divergence. As a first example, we adopt a counterexample from McIver & Morgan [MM05]: Consider the mixed–sign postexpectation

$$f \;=\; (-2)^x\,.$$

The expected value of f after executing the program C_{geo}, given by

$$\begin{aligned} C_{geo} \rhd \quad & x := 1 \,\mathfrak{g}\\ & c := 1 \,\mathfrak{g}\\ & \texttt{while}\,(c = 1)\,\{\\ & \qquad \{\,c := 0\,\}\,[1/2]\,\{\,x := x+1\,\}\\ & \}\,, \end{aligned}$$

on an arbitrary initial state is described by the series[1]

$$S \;=\; \sum_{i=1}^{\omega} \frac{(-2)^i}{2^i} \;=\; -1+1-1+1-1+\cdots\,,$$

which is indefinitely divergent, i.e. it neither converges to any real value nor does it tend to $+\infty$ or $-\infty$. Furthermore, the summands of this series can be reordered in such ways that the series tends to $+\infty$ or that it tends to $-\infty$. In any case, there exists *no meaningful* and in particular *no unique* expected value of f and thus no classical weakest preexpectation $\mathsf{wp}\,[\![C_{geo}]\!]\,(f)$.

If we were to *naively* apply the classical weakest preexpectation calculus anyway, we would first obtain a preexpectation for the loop, by constructing the corresponding wp–characteristic function

$$\Phi(X) \;=\; [c \neq 1]\cdot(-2)^x \;+\; [c=1]\cdot\frac{1}{2}\cdot\Big(X\,[c/0] + X\,[x/x+1]\Big)$$

1 $\sum_{v\in\mathsf{Vals}} \big[\!\big[C_{geo}\big]\!\big]_{\sigma}(v)\cdot f(v) \;=\; \sum_{i=1}^{\omega}\frac{(-2)^i}{2^i}$, for any initial state σ, cf. Definition 3.8.

and then doing fixed point iteration, i.e. iteratively apply Φ to 0. In doing so, we get the sequence

$$\begin{aligned}
\Phi(0) &= [c \neq 1] \cdot (-2)^x \\
\Phi^2(0) &= [c \neq 1] \cdot (-2)^x + [c = 1] \cdot \frac{(-2)^x}{2} \\
\Phi^3(0) &= [c \neq 1] \cdot (-2)^x + [c = 1] \cdot \left(\frac{(-2)^x}{2} + \frac{(-2)^{x+1}}{4} \right) \\
\Phi^4(0) &= [c \neq 1] \cdot (-2)^x + [c = 1] \cdot \left(\frac{(-2)^x}{2} + \frac{(-2)^{x+1}}{4} + \frac{(-2)^{x+2}}{8} \right)
\end{aligned}$$

and so on. Notice, that the sequence $\left(\Phi^n(0)\right)_{n \in \mathbb{N}}$ is *not* monotonically increasing, so iteratively applying Φ to 0 does not yield an ascending chain. If we nevertheless took the formal limit of this sequence—naively assuming it exists—, we would get

$$\Phi^\omega(0) = [c \neq 1] \cdot (-2)^x + [c = 1] \cdot \sum_{i=0}^{\omega} \frac{(-2)^{x+i}}{2^{i+1}} .$$

Finally, we have to apply the wp transformers of the assignments preceding the while loop to $\Phi^\omega(0)$, i.e. we have to calculate

$$\begin{aligned}
&\mathsf{wp}\, [\![x := 1 \mathbin{⨟} c := 1]\!] \left(\Phi^\omega(0) \right) \\
&= \mathsf{wp}\, [\![x := 1 \mathbin{⨟} c := 1]\!] \left([c \neq 1] \cdot (-2)^x + [c = 1] \cdot \sum_{i=0}^{\omega} \frac{(-2)^{x+i}}{2^{i+1}} \right) \\
&= \mathsf{wp}\, [\![x := 1]\!] \left([1 \neq 1] \cdot (-2)^x + [1 = 1] \cdot \sum_{i=0}^{\omega} \frac{(-2)^{x+i}}{2^{i+1}} \right) \\
&= \mathsf{wp}\, [\![x := 1]\!] \left(\sum_{i=0}^{\omega} \frac{(-2)^{x+i}}{2^{i+1}} \right) \\
&= \sum_{i=0}^{\omega} \frac{(-2)^{1+i}}{2^{1+i}} \\
&= -1 + 1 - 1 + 1 - 1 + \cdots .
\end{aligned}$$

The above is not well–defined and hence we see that the standard weakest preexpectation calculus cannot be applied to this example as is.

Non–absolute convergence. As a second example, consider the expected value of the mixed–sign postexpectation

$$f' = \frac{(-2)^x}{x+1}$$

after executing the above program C_{geo}, where we assume that x ranges over the natural numbers. It is described by the series[2]

$$S' \;=\; \sum_{i=1}^{\omega} \frac{(-2)^i}{2^i \cdot (i+1)} \;=\; -\frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \frac{1}{6} + \cdots .$$

The partial sums of the above series converge to $\ln(2) - 1$.

Again, if we were to naively apply the classical weakest preexpectation calculus, we would first obtain a preexpectation for the loop by constructing the corresponding wp–characteristic function

$$\Psi(X) \;=\; [c \neq 1] \cdot \frac{(-2)^x}{x+1} \;+\; [c = 1] \cdot \frac{1}{2} \cdot \Big(X\,[c/0] + X\,[x/x+1] \Big)$$

and then do fixed point iteration, i.e. iteratively apply Ψ to 0. This yields

$$\begin{aligned}
\Psi(0) \;&=\; [c \neq 1] \cdot \frac{(-2)^x}{x+1} \\
\Psi^2(0) \;&=\; [c \neq 1] \cdot \frac{(-2)^x}{x+1} \;+\; [c = 1] \cdot \frac{(-2)^x}{2(x+1)} \\
\Psi^3(0) \;&=\; [c \neq 1] \cdot \frac{(-2)^x}{x+1} \;+\; [c = 1] \cdot \left(\frac{(-2)^x}{2(x+1)} + \frac{(-2)^{x+1}}{4(x+2)} \right) \\
\Psi^4(0) \;&=\; [c \neq 1] \cdot \frac{(-2)^x}{x+1} \;+\; [c = 1] \cdot \left(\frac{(-2)^x}{2(x+1)} + \frac{(-2)^{x+1}}{4(x+2)} + \frac{(-2)^{x+2}}{8(x+3)} \right)
\end{aligned}$$

and so on. Notice that, again, the sequence $\big(\Psi^n(0)\big)_{n \in \mathbb{N}}$ is *not* monotonically increasing, so iteratively applying Ψ to 0 does not yield an ascending chain. If we nevertheless take a formal limit of this sequence — again just assuming it exists —, we get

$$\Psi^{\omega}(0) \;=\; [c \neq 1] \cdot \frac{(-2)^x}{x+1} \;+\; [c = 1] \cdot \sum_{i=0}^{\omega} \frac{(-2)^{x+i}}{2^{1+i} \cdot (x+i+1)} \,.$$

Finally, we have to apply the wp transformers of the assignments preceding the while loop to $\Phi^{\omega}(0)$, i.e. we have to calculate

$$\begin{aligned}
&\mathsf{wp}\,[\![x := 1 \,;\, c := 1]\!] \left(\Psi^{\omega}(0) \right) \\
&=\; \mathsf{wp}\,[\![x := 1 \,;\, c := 1]\!] \left([c \neq 1] \cdot \frac{(-2)^x}{x+1} \;+\; [c = 1] \cdot \sum_{i=0}^{\omega} \frac{(-2)^{x+i}}{2^{1+i} \cdot (x+i+1)} \right) \\
&=\; \mathsf{wp}\,[\![x := 1]\!] \left([1 \neq 1] \cdot \frac{(-2)^x}{x+1} \;+\; [1 = 1] \cdot \sum_{i=0}^{\omega} \frac{(-2)^{x+i}}{2^{1+i} \cdot (x+i+1)} \right)
\end{aligned}$$

2 $\sum_{v \in \mathsf{Vals}} \big[\![C_{geo}]\!]_{\sigma}(v) \cdot f'(v) = \sum_{i=1}^{\omega} \frac{(-2)^i}{2^i \cdot (i+1)}$, for any initial state σ, cf. Definition 3.8.

$$
\begin{aligned}
&= \mathsf{wp}\,[\![x := 1]\!]\left(\sum_{i=0}^{\omega} \frac{(-2)^{x+i}}{2^{1+i}\cdot(x+i+1)}\right)\\
&= \sum_{i=0}^{\omega} \frac{(-2)^{1+i}}{2^{1+i}\cdot(1+i+1)}\\
&= -\frac{1}{2}+\frac{1}{3}-\frac{1}{4}+\frac{1}{5}-\frac{1}{6}+\cdots,
\end{aligned}
$$

which converges to $\ln(2)-1$.

The reason that this example is nevertheless problematic is that even though the series does converge, it does not converge *absolutely*. We say that a series

$$\sum_{i=0}^{\omega} a_i \quad \text{is } \textit{absolutely convergent} \qquad \text{iff} \qquad \sum_{i=0}^{\omega} |a_i| \quad \text{converges.}$$

If a series is absolutely convergent, then the series is also *unconditionally convergent*, meaning that the partial sums of the series converge to a unique value *regardless of how the summands are ordered*. If, however, a series converges non–absolutely, then the well–known Riemann Series Theorem states that its summands can always be reordered in such a way that the series converges to an arbitrary value or that it tends to $+\infty$ or that it tends to $-\infty$.

Theorem 9.1 (Riemann Series Theorem [Rie67]):
Let $\sum_{i=0}^{\omega} a_i$ be a convergent but not absolutely convergent series, i.e.

$$\lim_{n\to\omega} \sum_{i=0}^{n} a_i = c \in \mathbb{R} \qquad \text{and} \qquad \lim_{n\to\omega} \sum_{i=0}^{n} |a_i| = +\infty\,.$$

Then for any $r \in \mathbb{R} \cup \{+\infty, -\infty\}$ there exists a permutation of the natural numbers $\pi\colon \mathbb{N} \to \mathbb{N}$, such that

$$\lim_{n\to\omega} \sum_{i=0}^{n} a_{\pi(i)} = r\,.$$

The behavior of non–absolutely convergent series under reordering is highly undesirable in connection with the notion of expected values. This is because the outcomes of random events are only assigned a probability, and in particular there exists no natural ordering of the summands in which their weighted masses should be summed up to an expected value. Absolute convergence is thus a desirable principle we want our preexpectations to be based upon. Next, we investigate how to incorporate the notion of absolute convergence into a new notion for mixed–sign expectations.

9.2 INTEGRABILITY–WITNESSING EXPECTATIONS

To overcome the issue of non–absolute convergence, in the field of probability theory the expected value $\int f \, d\mu$ of a mixed–sign random variable f with respect to a probability distribution μ is only defined if

$$\int |f| \, d\mu < \infty ,$$

i.e. if the expected value of the absolute value of f with respect to distribution μ is finite. In that case, f is called *integrable*. A good reason to demand integrability is that, e.g. integrability of a discrete random variable f is equivalent to absolute convergence of the series representing $\int f \, d\mu$, which in turn makes the series converge independently of the summand ordering.

Our goal is to formally incorporate the bookkeeping whether f is integrable or not into the objects on which a mixed–sign weakest preexpectation calculus acts in order to obtain a sound calculus for reasoning about mixed–sign postexpectations. The very first step on our path to achieving this goal is to alter our expectation space to allow for random variables to evaluate to both positive and negative reals.

Definition 9.2 (Mixed–sign Expectations [KK17b]):
The set of mixed–sign expectations, denoted $\mathbb{E}_{\pm}$, is defined as

$$\mathbb{E}_{\pm} = \{ f \mid f \colon \Sigma \to \mathbb{R} \} .$$

Notice that, in contrast to the definition of non–negative expectations (see Definition 4.1 A.), we have dropped the ∞ element from the codomain of an expectation, since if a random variable is integrable, then its expected value is finite anyway.

Next, we present our integrability bookkeeping approach. The idea is to keep a pair of expectations f and g, denoted $(f \mathrel{\reflectbox{$\ll$}} g)$, where f is the mixed–sign expectation we are actually interested in and g is an ordinary *non–negative* expectation that bounds $|f|$ and hence acts as an integrability witness. We call such a pair an *integrability–witnessing pair*. Preexpectations will later be computed for both components simultaneously.

Definition 9.3 (Integrability–witnessing Pairs [KK17b]):
The set of integrability–witnessing pairs, denoted $\mathbb{P}$, is defined as

$$\mathbb{P} = \left\{ (f \mathrel{\reflectbox{$\ll$}} g) \;\middle|\; f \in \mathbb{E}_{\pm},\, g \in \mathbb{E},\, |f| \leq g \right\} .$$

We define an addition of two integrability–witnessing pairs by

$$(f \mathrel{\reflectbox{$\ll$}} g) + (f' \mathrel{\reflectbox{$\ll$}} g') = (f + f' \mathrel{\reflectbox{$\ll$}} g + g') ,$$

and a multiplication by a mixed–sign expectation $h \in \mathbb{E}_{\pm}$ *by*

$$h \cdot (f \mathrel{\reflectbox{$\ddots$}} g) \;=\; (h \cdot f \mathrel{\reflectbox{$\ddots$}} |h| \cdot g)\,.$$

A visual intuition on integrability–witnessing pairs is provided in Figure 9.1. The X–axis represents the set of program states which is idealized to a linear representation. The Y–axis represents the extended real number line. Roughly in the middle of the graphs, the bounding expectation g escapes to ∞. f_2 does too, but we do not require the first components of integrability–witnessing pairs to be bounded at those points where the second component itself is unbounded, which is why $(f_2 \mathrel{\reflectbox{$\ddots$}} g)$ is a valid integrability–witnessing pair. On the other hand, $(f_3 \mathrel{\reflectbox{$\ddots$}} g)$ is *not* a valid integrability–witnessing pair, because g does not bound $|f_3|$ in the left part of the graph.

Next, we would like to define an order on integrability–witnessing pairs. We would like to compare pairs componentwise, i.e. $(f \mathrel{\reflectbox{$\ddots$}} g)$ should be less or equal $(f' \mathrel{\reflectbox{$\ddots$}} g')$ if both $f \leq f'$ and $g \leq g'$. This would naturally lift the partial order $\leq$ on $\mathbb{E}$ to $\mathbb{P}$. There is, however, a catch:

Recall that the intuition behind keeping a pair $(f \mathrel{\reflectbox{$\ddots$}} g)$ is that whenever the expected value of g is finite, then the expected value of $|f|$, too, is finite by monotonicity of the expected value operator. If the expected value of g is not finite, however, then the expected value of f cannot be ensured to be defined. (In particular, if $g = |f|$, then the expected value of f should definitely be undefined.) Therefore, if g' is the preexpectation of g and for a state $\sigma \in \Sigma$ we have $g'(\sigma) = \infty$, then we should not care about the preexpectation of f in state σ since definedness cannot be ensured for that state. This consideration should be reflected in our order on $\mathbb{P}$: For states where the second component evaluates to ∞, the first component should not be compared. This gives rise to the following definition:

Definition 9.4 (The Quasiorder $\lesssim$ on $\mathbb{P}$ [KK17b]):
A quasiorder on the set $\mathbb{P}$ *of integrability–witnessing pairs is given by*

$$(f \mathrel{\reflectbox{$\ddots$}} g) \;\lesssim\; (f' \mathrel{\reflectbox{$\ddots$}} g')$$

iff for all $\sigma \in \Sigma$,

$$g'(\sigma) < \infty \qquad \text{implies} \qquad f(\sigma) \leq f'(\sigma) \quad \text{and} \quad g(\sigma) \leq g'(\sigma)\,.$$

In contrast to a partial order which is reflexive, transitive and antisymmetric, a quasiorder is not required to be antisymmetric. Notice that, indeed, $\lesssim$ is only a quasiorder since the two integrability–witnessing pairs $(f \mathrel{\reflectbox{$\ddots$}} \infty)$ and $(f' \mathrel{\reflectbox{$\ddots$}} \infty)$ with $f \neq f'$ satisfy

$$(f \mathrel{\reflectbox{$\ddots$}} \infty) \;\lesssim\; (f' \mathrel{\reflectbox{$\ddots$}} \infty) \quad \text{and} \quad (f \mathrel{\reflectbox{$\ddots$}} \infty) \;\gtrsim\; (f' \mathrel{\reflectbox{$\ddots$}} \infty)$$

but not $(f \mathrel{\reflectbox{$\ddots$}} \infty) = (f' \mathrel{\reflectbox{$\ddots$}} \infty)$. This shows that $\lesssim$ is *not antisymmetric*.

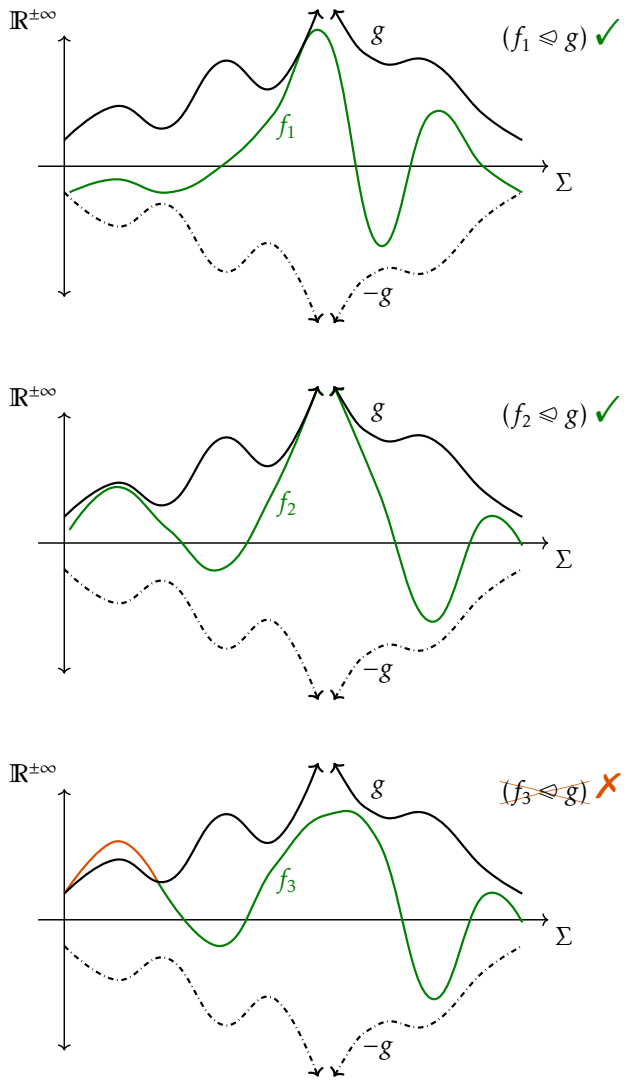


Figure 9.1: Two examples (top and middle) and one non–example (bottom) of integrability–witnessing pairs. The X–axis is a „linearized“ visualization of the program state space Σ. The Y–axis is the extended real number line.

On the other hand, two integrability–witnessing pairs $(f \mathrel{\triangleleft} g)$ and $(f' \mathrel{\triangleleft} g')$, for which $f(\sigma) \neq f'(\sigma)$ holds only for those states in which $g(\sigma) = \infty = g'(\sigma)$, should be considered *equivalent*, even though they are not equal. This is because for states σ in which $g(\sigma) = \infty = g'(\sigma)$, the evaluations of $f(\sigma)$ and $f'(\sigma)$ should be ignored since integrability is not ensured. Consequently, we need a notion of equivalence of integrability–witnessing pairs:

Definition 9.5 (Integrability–witnessing Expectations [KK17b]): *The quasiorder $\lesssim$ induces a canonical [AJ94] equivalence relation $\approx$, given by $\approx = \lesssim \cap \gtrsim$, i.e.*

$$(f \mathrel{\triangleleft} g) \approx (f' \mathrel{\triangleleft} g')$$

iff for all $\sigma \in \Sigma$,

$$\begin{aligned} g(\sigma) \neq \infty \quad &\text{or} \quad g'(\sigma) \neq \infty \\ \text{implies} \quad & f(\sigma) = f'(\sigma) \quad \text{and} \quad g(\sigma) = g'(\sigma)\,. \end{aligned}$$

We denote by $\wr f \mathrel{\triangleleft} g \wr$ the equivalence class of $(f \mathrel{\triangleleft} g)$ under $\approx$, i.e.

$$\wr f \mathrel{\triangleleft} g \wr = \left\{ (f' \mathrel{\triangleleft} g') \in \mathbb{P} \;\middle|\; (f' \mathrel{\triangleleft} g') \approx (f \mathrel{\triangleleft} g) \right\},$$

and call such an equivalence class an integrability–witnessing expectation. We denote by $\mathbb{IE}$ the set of integrability–witnessing expectations (which is the set of all equivalence classes of $\approx$), i.e.

$$\mathbb{IE} = \mathbb{P}/_{\approx}\,.$$

A visual intuition on integrability–witnessing expectations is provided in Figure 9.2. Roughly in the middle of the graphs, say at point[3] σ_m, the bounding expectation g escapes to ∞, i.e. we have $g(\sigma_m) = \infty$. f_1, f_2 and f_3 all coincide almost everywhere, except for at point σ_m. However, since g is unbounded at this point, the values of the first components are irrelevant and hence $(f_1 \mathrel{\triangleleft} g)$, $(f_2 \mathrel{\triangleleft} g)$, and $(f_3 \mathrel{\triangleleft} g)$ are all member of the *same* integrability–witnessing expectation, i.e.

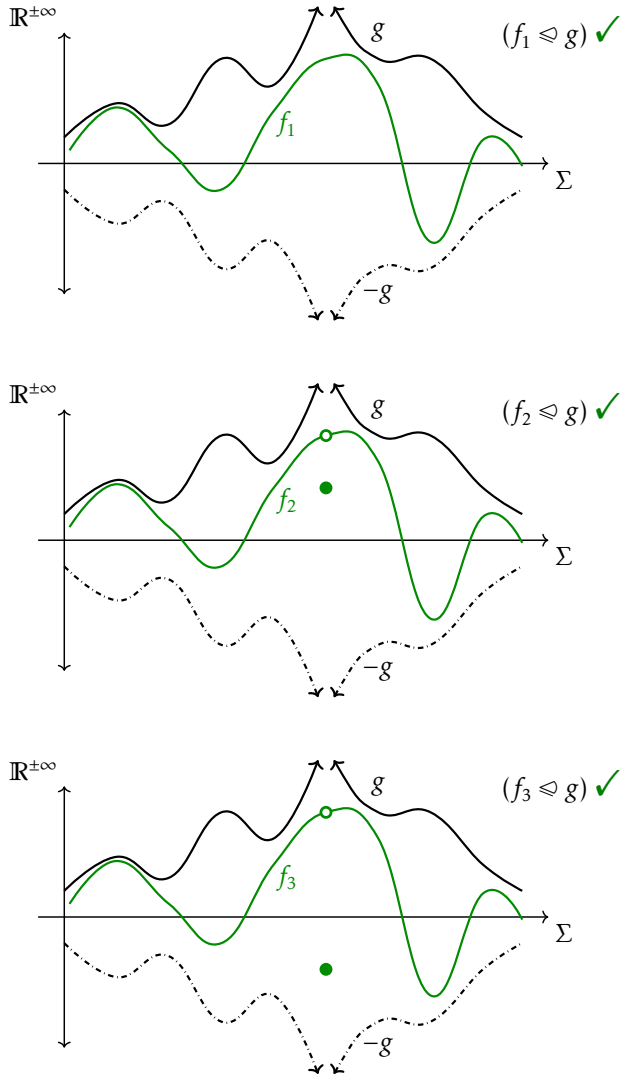
$$\wr f_1 \mathrel{\triangleleft} g \wr = \wr f_2 \mathrel{\triangleleft} g \wr = \wr f_3 \mathrel{\triangleleft} g \wr$$

As for a different intuition, $\wr f \mathrel{\triangleleft} g \wr$ can be thought of as a particular pair $(f \mathrel{\triangleleft} g)$ such that g maps each state either to a non–negative real number or ∞ and f maps each state, that is not mapped to ∞ by g, to a real number.[4]

Notice that we call the equivalence classes and not the pairs „expectations" as we consider $\mathbb{IE}$ and not $\mathbb{P}$ to be a suitable domain to calculate preexpectations on and thus we consider $\mathbb{IE}$ to be the mixed–sign counterpart to $\mathbb{E}$. Next, we define a *partial order on the equivalence classes*:

3 Recall that a „point" in this context is a state $\sigma \in \Sigma$.

4 This intuition was suggested by an anonymous reviewer of an earlier version of [KK17b] and, personally, I found it quite well–put and insightful.



$$\wr f_1 \trianglelefteq g \wr = \wr f_2 \trianglelefteq g \wr = \wr f_3 \trianglelefteq g \wr$$

Figure 9.2: Three different integrability–witnessing pairs. All three are in the same equivalence class, i.e. the corresponding integrability–witnessing expectations are equal.

Definition 9.6 (The Partial Order on $\mathbb{IE}$ [KK17b]):
The quasiorder $\precsim$ on the set $\mathbb{P}$ of integrability–witnessing pairs induces a canonical [AJ94] partial order $\lessapprox$ on the set $\mathbb{IE}$ of integrability–witnessing expectations by

$$\wr f_1 \circledcirc g_1 \int \lessapprox \wr f_2 \circledcirc g_2 \int \quad \text{iff} \quad (f_1 \circledcirc g_1) \precsim (f_2 \circledcirc g_2)\,.$$

As for an intuitive interpretation of this partial order, we note that if

$$\wr f_1 \circledcirc g_1 \int \lessapprox \wr f_2 \circledcirc g_2 \int$$

holds, then for all $(f_1' \circledcirc g_1') \in \wr f_1 \circledcirc g_1 \int$, $(f_2' \circledcirc g_2') \in \wr f_2 \circledcirc g_2 \int$, and all states σ in which $g_2(\sigma) < \infty$ holds, we have

$$f_1'(\sigma) \;=\; f_1(\sigma) \;\leq\; f_2(\sigma) \;=\; f_2'(\sigma)\,.$$

Thus if integrability in σ is witnessed, the first components compare in σ, which is the comparison we are actually interested in.

The lattice $(\mathbb{IE}, \lessapprox)$ is complete in the sense that every *non–empty* subset $S \subseteq \mathbb{IE}$ has a supremum given by

$$\sup S \;=\; \wr \hat{f} \circledcirc \hat{g} \int, \quad \text{where}$$

$$\hat{g}(\sigma) \;=\; \sup\Big\{ g(\sigma) \in \mathbb{R}^{\infty}_{\geq 0} \;\Big|\; (f \circledcirc g) \in \wr f \circledcirc g \int \in S \Big\}$$

$$\hat{f}(\sigma) \;=\; \begin{cases} \sup\{ f(\sigma) \in \mathbb{R} \mid (f \circledcirc g) \in \wr f \circledcirc g \int \in S \}, & \text{if } \hat{g}(\sigma) < \infty, \\ 0, & \text{otherwise.}^{5} \end{cases}$$

An unfortunate fact about the lattice $(\mathbb{IE}, \lessapprox)$ is that it has *no least element*. In particular, even though the element

$$\wr 0 \circledcirc 0 \int$$

will play an important role in our later development, it is *not* a least element of $\mathbb{IE}$ since, for example

$$\wr 0 \circledcirc 0 \int \;\not\lessapprox\; \wr -1 \circledcirc 1 \int\,.$$

This fact prevents us from applying the Kleene Fixed Point Theorem (see Theorem A.5) — as we did in Section 4.1.3 — in our later development.

In the next section, we investigate a weakest preexpectation calculus acting on integrability–witnessing expectations, i.e. the calculus will feature expectation transformers of type $\mathbb{IE} \to \mathbb{IE}$.

5 Notice that the 0 for this case is an arbitrary choice of a value in $\mathbb{R}$ since any $(\hat{f}' \circledcirc \hat{g})$, where $\hat{f}'(\sigma) \neq 0$ for any $\sigma \in \Sigma$ with $\hat{g}(\sigma) = \infty$, is in the same equivalence class as $(\hat{f} \circledcirc \hat{g})$.

9.3 EXPECTATION TRANSFORMERS

On our way to developing a weakest preexpectation calculus acting on integrability–witnessing expectations, we first observe that certain operations on an integrability–witnessing pair $(f \trianglelefteq g)$ preserve $\approx$–equivalence and thus lifting such operations to the integrability–witnessing expectation $⟅f \trianglelefteq g⟆$ can be done by performing the operation on the representative $(f \trianglelefteq g)$ and then taking the equivalence class of the resulting pair.

For instance, the assignment $x := E$ preserves $\approx$–equivalence, because if $(f \trianglelefteq g) \approx (f' \trianglelefteq g')$ holds, then for all $\sigma \in \Sigma$ we have

$$\begin{aligned} &g(\sigma) < \infty \quad \text{or} \quad g'(\sigma) < \infty \\ &\qquad \text{implies} \qquad f(\sigma) = f'(\sigma) \quad \text{and} \quad g(\sigma) = g'(\sigma)\,. \end{aligned}$$

But then this is in particular true for all updated states which are of the form $\sigma[x \mapsto \sigma(E)]$ and thus $\approx$–equivalence is preserved by the assignment, i.e.

$$\begin{aligned} &(f \trianglelefteq g) \;\approx\; (f' \trianglelefteq g') \\ &\qquad \text{implies} \qquad (f[x/E] \trianglelefteq g[x/E]) \;\approx\; (f'[x/E] \trianglelefteq g'[x/E])\,. \end{aligned}$$

Moreover this allows for defining a transformer

$$\mathsf{iwp}\,[\![x := E]\!]\,⟅f \trianglelefteq g⟆ \;=\; ⟅f[x/E] \trianglelefteq g[x/E]⟆\,.$$

Furthermore, one can show that both our addition of integrability–witnessing pairs as well as our scalar multiplication of integrability–witnessing pairs by mixed–sign expectations also preserve $\approx$–equivalence, i.e. we have

$$\begin{aligned} ⟅f \trianglelefteq g⟆ + ⟅f' \trianglelefteq g'⟆ \;&=\; ⟅f + f' \trianglelefteq g + g'⟆\,, \quad \text{and} \\ h \cdot ⟅f \trianglelefteq g⟆ \;&=\; ⟅h \cdot f \trianglelefteq |h| \cdot g⟆\,. \end{aligned}$$

This puts us in a position to formally define a weakest preexpectation transformer acting on $\mathbb{IE}$:

Definition 9.7 (Integrability–witnessing Expect. Transf. [KK17b]):

1. *For all tame programs $C \in \mathsf{pGCL}$, the weakest integrability–witnessing preexpectation transformer*

$$\mathsf{iwp}\,[\![C]\!] : \mathbb{IE} \to \mathbb{IE}\,,$$

is defined according to the rules in Table 9.1.

2. *We call the function*

$${}^{\mathsf{iwp}}_{\langle \varphi, C\rangle}\Phi_{⟅f \trianglelefteq g⟆}⟅X \trianglelefteq Y⟆ \;=\; [\neg\varphi] \cdot ⟅f \trianglelefteq g⟆ + [\varphi] \cdot \mathsf{iwp}\,[\![C]\!]\,⟅X \trianglelefteq Y⟆$$

the iwp–characteristic function of $\texttt{while}(\varphi)\{C\}$ with respect to postexpectation $⟅f \trianglelefteq g⟆$. Whenever either iwp, *φ*, *C*, *or $⟅f \trianglelefteq g⟆$ are clear from the context, we omit them from Φ.*

C	$\mathsf{iwp}\,[\![C]\!]\,\wr f \preccurlyeq g \wr$
`skip`	$\wr f \preccurlyeq g \wr$
`diverge`	$\wr 0 \preccurlyeq 0 \wr$
$x := E$	$\wr f[x/E] \preccurlyeq g[x/E] \wr$
$x :\approx \mu$	$\left\wr \lambda\sigma.\ \int_{\mathsf{Vals}} \left(\lambda v.\ f(\sigma[x \mapsto v])\right) d\mu_\sigma \ \preccurlyeq\ \lambda\sigma.\ \int_{\mathsf{Vals}} \left(\lambda v.\ g(\sigma[x \mapsto v])\right) d\mu_\sigma \right\wr$
$C_1 \fatsemi C_2$	$\mathsf{iwp}\,[\![C_1]\!]\left(\mathsf{iwp}\,[\![C_2]\!]\,\wr f \preccurlyeq g \wr\right)$
$\mathtt{if}\,(\varphi)\,\{C_1\}\,\mathtt{else}\,\{C_2\}$	$[\varphi]\cdot \mathsf{iwp}\,[\![C_1]\!]\,\wr f \preccurlyeq g \wr \ +\ [\neg\varphi]\cdot \mathsf{iwp}\,[\![C_2]\!]\,\wr f \preccurlyeq g \wr$
$\{C_1\}\,[p]\,\{C_2\}$	$p\cdot \mathsf{iwp}\,[\![C_1]\!]\,\wr f \preccurlyeq g \wr \ +\ (1-p)\cdot \mathsf{iwp}\,[\![C_2]\!]\,\wr f \preccurlyeq g \wr$
$\mathtt{while}(\varphi)\{C'\}$	$\lim_{n\to\omega}\ {}^{\mathsf{iwp}}_{\langle\varphi,C'\rangle}\Phi^n_{\wr f \preccurlyeq g \wr}\wr 0 \preccurlyeq 0 \wr$

$$ {}^{\mathsf{iwp}}_{\langle\varphi,C'\rangle}\Phi_{\wr f\preccurlyeq g\wr}\wr X \preccurlyeq Y \wr \ =\ [\neg\varphi]\cdot\wr f \preccurlyeq g\wr + [\varphi]\cdot \mathsf{iwp}\,[\![C]\!]\,\wr X \preccurlyeq Y \wr $$

Table 9.1: The weakest preexpectation transformer acting on $\mathbb{IE}$.

Let us briefly go over some of the rules in Table 9.1 in order to get an intuition: Just like $\mathsf{wp}\,[\![\mathtt{skip}]\!]$, $\mathsf{iwp}\,[\![\mathtt{skip}]\!]$ is an identity since `skip` does not modify the program state. $\mathsf{iwp}\,[\![\mathtt{diverge}]\!]\,\wr f \preccurlyeq g \wr$ returns the integrability–witnessing expectation $\wr 0 \preccurlyeq 0 \wr$ which is in fact the singleton set $\{(0 \preccurlyeq 0)\}$. This is meaningful, since the probability of termination of `diverge` is 0, and so no mass whatsoever can be contributed to the expected value of any expectation $f \in \mathbb{E}_\pm$ or $g \in \mathbb{E}$ after termination of `diverge`. $\mathsf{iwp}\,[\![x := E]\!]\,\wr f \preccurlyeq g \wr$ takes a representative $(f \preccurlyeq g) \in \wr f \preccurlyeq g \wr$, performs the assignment $x := E$ on both components to obtain $(f[x/E] \preccurlyeq g[x/E])$ and then returns the corresponding equivalence class $\wr f[x/E] \preccurlyeq g[x/E] \wr$. As described earlier, assignments preserve $\approx$–equivalence, so doing the update on the representative is a sound and sufficient course of action.

Before we turn our attention to the definitions of iwp for while loops, let us illustrate the effects of the iwp transformer by means of an Example:

Example 9.8 (Truncated Alternating Geometric Distribution [KK17b]): Consider the probabilistic program $C_{alttrunc}$:

$$
\begin{aligned}
C_{alttrunc} \rhd \quad & \{\mathtt{skip}\}\,[1/2]\,\{ \\
& \qquad x := -x - \mathsf{sign}(x) \fatsemi \{\mathtt{skip}\}\,[1/2]\,\{x := -x - \mathsf{sign}(x)\} \\
& \}
\end{aligned}
$$

It establishes on x a truncated variant of a geometric distribution where in addition the sign of x alternates. Suppose we want to know the expected value of x after termination of $C_{alttrunc}$. The according integrability–witnessing postexpectation for obtaining an answer to this question is $(\!| x \mathrel{\reflectbox{$\leqslant$}} |x| |\!)$. Notice that in this example, the need for mixed–sign random variables arises *not* from some artificially constructed mixed–sign postexpectation but *directly from the program code*. In order to reason about the expected value of x after termination, we calculate iwp $[\![C_{alttrunc}]\!]\,(\!| x \mathrel{\reflectbox{$\leqslant$}} |x| |\!)$. Reusing our annotation style from earlier in this thesis (see Example 2.11), i.e. we write

$$
\begin{array}{l}
/\!/\!/\ (\!| f'' \mathrel{\reflectbox{$\leqslant$}} g'' |\!) \\
/\!/\!/\ (\!| f' \mathrel{\reflectbox{$\leqslant$}} g' |\!) \\
C \\
/\!/\!/\ (\!| f \mathrel{\reflectbox{$\leqslant$}} g |\!)
\end{array}
$$

to expresses the fact that $(\!| f' \mathrel{\reflectbox{$\leqslant$}} g' |\!) = \mathsf{cwp}\,[\![C]\!]\,(\!| f \mathrel{\reflectbox{$\leqslant$}} g |\!)$ and moreover the fact that $(\!| f'' \mathrel{\reflectbox{$\leqslant$}} g'' |\!) = (\!| f' \mathrel{\reflectbox{$\leqslant$}} g' |\!)$, we can annotate the program $C_{alttrunc}$ as shown in Figure 9.3 (read from bottom to top).

Let us now interpret the results of the program annotation: The first observation we can make is that the expected value of x is defined after execution of $C_{alttrunc}$, since in every initial state we have

$$|x| + \frac{3 \cdot [x \neq 0]}{4} \;<\; \infty\,.$$

The second observation we can make is that this expected value is for every initial state given by

$$\frac{2x + \mathsf{sign}(x)}{4}\,,$$

which is to be evaluated in the initial state in which $C_{alttrunc}$ is started. In particular, the above expression gives the correct expected value, *regardless* of whether the program is started with a positive, negative, or zero variable valuation for x.

9.3.1 *Preexpectations of While Loops*

While the calculation of iwp in the above example was straightforward (although tedious) as the program $C_{alttrunc}$ is loop–free, iwp of while loops is defined using a limit construct. For that, we first need to formally define what a limit of a sequence of integrability–witnessing expectations, i.e. a *limit of a sequence of equivalence classes*, is.

$$/\!/\!/ \left(\frac{2x + \mathsf{sign}(x)}{4} \trianglelefteq |x| + \frac{3 \cdot [x \neq 0]}{4} \right)$$

$$/\!/\!/ \tfrac{1}{2} \cdot (\!| x \trianglelefteq |x| |\!) + \tfrac{1}{2} \cdot \left(\frac{\mathsf{sign}(x)}{2} \trianglelefteq |x| + \frac{3 \cdot [x \neq 0]}{2} \right)$$

{

$/\!/\!/ (\!| x \trianglelefteq |x| |\!)$

skip

$/\!/\!/ (\!| x \trianglelefteq |x| |\!)$

} [1/2] {

$$/\!/\!/ \left(\frac{\mathsf{sign}(x)}{2} \trianglelefteq |x| + \frac{3 \cdot [x \neq 0]}{2} \right)$$

$$/\!/\!/ \left(\frac{\mathsf{sign}(-(-x - \mathsf{sign}(x)))}{2} \trianglelefteq |-x - \mathsf{sign}(x)| + \cdots \right)$$

$x := -x - \mathsf{sign}(x)$;

$$/\!/\!/ \left(\frac{\mathsf{sign}(-x)}{2} \trianglelefteq |x| + \frac{[x \neq 0]}{2} \right)$$

$/\!/\!/ \tfrac{1}{2} \cdot (\!| x \trianglelefteq |x| |\!) + \tfrac{1}{2} \cdot (\!| -x - \mathsf{sign}(x) \trianglelefteq |x| + [x \neq 0] |\!)$

{

$/\!/\!/ (\!| x \trianglelefteq |x| |\!)$

skip

$/\!/\!/ (\!| x \trianglelefteq |x| |\!)$

} [1/2] {

$/\!/\!/ (\!| -x - \mathsf{sign}(x) \trianglelefteq |x| + [x \neq 0] |\!)$

$/\!/\!/ (\!| -x - \mathsf{sign}(x) \trianglelefteq |-x - \mathsf{sign}(x)| |\!)$

$x := -x - \mathsf{sign}(x)$

$/\!/\!/ (\!| x \trianglelefteq |x| |\!)$

}

$/\!/\!/ (\!| x \trianglelefteq |x| |\!)$

}

$/\!/\!/ (\!| x \trianglelefteq |x| |\!)$

Figure 9.3: integrability–witnessing weakest preexpectation annotations for the program $C_{alttrunc}$ of Example 9.8.

DEFINITION 9.9 (Limits of Sequences in $\mathbb{IE}$ [KK17b]):
Let $\big(\wr f_n \preccurlyeq g_n \wr\big)_{n\in\mathbb{N}} \subset \mathbb{IE}$ be a sequence in $\mathbb{IE}$. Then

$$\wr f \preccurlyeq g \wr \quad \textit{is a limit of} \quad (\wr f_n \preccurlyeq g_n \wr)_{n\in\mathbb{N}}\,,$$

if there exists a sequence $((f'_n \preccurlyeq g'_n))_{n\in\mathbb{N}}$ of representatives (meaning for all $n \in \mathbb{N}$, $(f'_n \preccurlyeq g'_n) \in \wr f_n \preccurlyeq g_n \wr$), with

$$g(\sigma) \;=\; \lim_{n\to\omega} g'_n(\sigma)\,, \quad \text{and}$$

$$f(\sigma) \;=\; \begin{cases} \lim\limits_{n\to\omega} f'_n(\sigma)\,, & \textit{if } g(\sigma) < \infty\,, \\ 0\,, & \textit{otherwise,}^{6} \end{cases}$$

where ∞ is assumed to be a valid limit for $g'_n(\sigma)$.

The intuition behind this definition is that a limit of a sequence in $\mathbb{IE}$ is a pointwise limit (in each state $\sigma \in \Sigma$).

If a limit exists, we note the following: For each integrability–witnessing pair in any $\approx$–equivalence class, the second component is unique. Thus the sequence $(g'_n)_{n\in\mathbb{N}}$ is uniquely determined by $(g_n)_{n\in\mathbb{N}}$.

Now, if $\lim_{n\to\omega} g_n(\sigma) = \infty$, then the limit in state σ does not depend on the sequence $(f'_n)_{n\in\mathbb{N}}$ and is uniquely determined. If on the other hand $\lim_{n\to\omega} g_n(\sigma) < \infty$, then for almost all g_i we have $g_i(\sigma) < \infty$ and thus also almost all f'_i are uniquely determined by f_i. All in all this leads to the fact that if a limit of $(\wr f_n \preccurlyeq g_n \wr)_{n\in\mathbb{N}}$ exists, then we can reason about the existence by means of the sequence of representatives $((f_n \preccurlyeq g_n))_{n\in\mathbb{N}}$.

The iwp transformer of while loops is defined as the limit of a sequence of integrability–witnessing expectations, but in order to speak of *the* limit, such limits must be unique if they exist. This is ensured by the following theorem:

THEOREM 9.10 (Uniqueness of Limits in $\mathbb{IE}$ [KK17b]):
Let $(\wr f_n \preccurlyeq g_n \wr)_{n\in\mathbb{N}} \subseteq \mathbb{IE}$ and let a limit of that sequence exist. Then that limit is unique, i.e. if

$$\wr f \preccurlyeq g \wr \textit{ and } \wr f' \preccurlyeq g' \wr \quad \textit{both being a limit of} \quad \wr f_n \preccurlyeq g_n \wr$$
$$\text{implies} \qquad \wr f \preccurlyeq g \wr \;=\; \wr f' \preccurlyeq g' \wr\,.$$

Proof. Suppose for a contradiction that $\wr f \preccurlyeq g \wr \neq \wr f' \preccurlyeq g' \wr$ are both a limit of the sequence $(\wr f_n \preccurlyeq g_n \wr)_{n\in\mathbb{N}} \subseteq \mathbb{IE}$. Recall that we can reason about such a limit entirely by the sequence of representatives $((f_n \preccurlyeq g_n))_{n\in\mathbb{N}}$. Because of $\wr f \preccurlyeq g \wr \neq \wr f' \preccurlyeq g' \wr$ we have $(f \preccurlyeq g) \not\approx (f' \preccurlyeq g')$. Hence, there must exist a

6 Notice that this 0 is again an arbitrary choice of a value in $\mathbb{R}$ since any $(f' \preccurlyeq g)$, where $f'(\sigma) \neq 0$ for any $\sigma \in \Sigma$ with $g(\sigma) = \infty$, is in the same equivalence class as $(f \preccurlyeq g)$. See also Footnote 5.

state σ such that

$$g(\sigma) < \infty \quad \text{or} \quad g'(\sigma) < \infty$$
$$\text{and} \qquad g(\sigma) \neq g'(\sigma) \quad \text{or} \quad f(\sigma) \neq f'(\sigma)\,.$$

But if that were the case, then for that state σ either

$$\lim_{n\to\omega} g_n(\sigma) \;=\; g(\sigma) \;\neq\; g'(\sigma) \;=\; \lim_{n\to\omega} g_n(\sigma), \quad \text{or}$$
$$\lim_{n\to\omega} f_n(\sigma) \;=\; f(\sigma) \;\neq\; f'(\sigma) \;=\; \lim_{n\to\omega} f_n(\sigma)$$

should hold, both of which is a contradiction to the fact that limits of real numbers are unique if they exist. Therefore, the assumption $\llparenthesis f \preccurlyeq g \rrparenthesis \neq \llparenthesis f' \preccurlyeq g' \rrparenthesis$ cannot be true and the limit of $(\llparenthesis f_n \preccurlyeq g_n \rrparenthesis)_{n\in\mathbb{N}}$ must be unique. Q.E.D.

Due to the limit's uniqueness, we are now in a position to write

$$\lim_{n\to\omega} \llparenthesis f_n \preccurlyeq g_n \rrparenthesis \;=\; \llparenthesis f \preccurlyeq g \rrparenthesis\,,$$

if a limit exists and $\llparenthesis f \preccurlyeq g \rrparenthesis$ is *the* limit of $\lim_{n\to\omega} \llparenthesis f_n \preccurlyeq g_n \rrparenthesis$.

Using the limit construct, the iwp transformer of the loop $\texttt{while}(\varphi)\{C'\}$ is defined as the limit of iteratively applying the iwp–characteristic function of $\texttt{while}(\varphi)\{C'\}$, given by

$$\overset{\mathsf{iwp}}{\langle \varphi, C' \rangle}\Phi_{\llparenthesis f \preccurlyeq g \rrparenthesis}\llparenthesis X \preccurlyeq Y \rrparenthesis \;=\; [\neg\varphi]\cdot\llparenthesis f \preccurlyeq g \rrparenthesis + [\varphi]\cdot \mathsf{iwp}\,[\![C']\!]\,\llparenthesis X \preccurlyeq Y \rrparenthesis\,,$$

to $\llparenthesis 0 \preccurlyeq 0 \rrparenthesis$. Formally, we have defined in Table 9.1

$$\mathsf{iwp}\,[\![\texttt{while}(\varphi)\{C'\}]\!]\,\llparenthesis f \preccurlyeq g \rrparenthesis \;=\; \lim_{n\to\omega}\; \overset{\mathsf{iwp}}{\langle \varphi, C' \rangle}\Phi^n_{\llparenthesis f \preccurlyeq g \rrparenthesis}\llparenthesis 0 \preccurlyeq 0 \rrparenthesis\,,$$

where $\overset{\mathsf{iwp}}{\langle \varphi, C' \rangle}\Phi^n_{\llparenthesis f \preccurlyeq g \rrparenthesis}$ denotes the n-fold application of $\overset{\mathsf{iwp}}{\langle \varphi, C' \rangle}\Phi_{\llparenthesis f \preccurlyeq g \rrparenthesis}$ to its argument. This is somewhat similar to the wp–semantics for non–negative expectations, where we basically have

$$\mathsf{wp}\,[\![\texttt{while}(\varphi)\{C'\}]\!]\,(f) \;=\; \lim_{n\to\omega}\; \overset{\mathsf{wp}}{\langle \varphi, C' \rangle}\Phi^n_f(0)$$

since the Kleene Fixed Point Theorem gives

$$\begin{aligned}
&\mathsf{wp}\,[\![\texttt{while}(\varphi)\{C'\}]\!]\,(f)\\
&= \mathsf{lfp}\;\overset{\mathsf{wp}}{\langle \varphi, C' \rangle}\Phi_f\\
&= \sup_n\; \overset{\mathsf{wp}}{\langle \varphi, C' \rangle}\Phi^n_f(0) && \text{(Kleene Fixed Point Theorem, Theorem A.5)}\\
&= \lim_{n\to\omega}\; \overset{\mathsf{wp}}{\langle \varphi, C' \rangle}\Phi^n_f(0)\,, && \left(\overset{\mathsf{wp}}{\langle \varphi, C' \rangle}\Phi^n_f(0) \text{ increases monotonically in } n\right)
\end{aligned}$$

and ensures existence of this limit. This, however, works only because 0 is the least element in the complete lattice $(\mathbb{E}, \preceq)$. Because of monotonicity of

${}^{\mathsf{wp}}_{\langle \varphi, C' \rangle}\Phi_f$ (see Theorem 4.16), we automatically obtain the ascending chain

$$0 \preceq {}^{\mathsf{wp}}_{\langle \varphi, C' \rangle}\Phi_f(0) \preceq {}^{\mathsf{wp}}_{\langle \varphi, C' \rangle}\Phi_f^2(0) \preceq {}^{\mathsf{wp}}_{\langle \varphi, C' \rangle}\Phi_f^3(0) \preceq \cdots$$

for which a supremum exists by completeness of the underlying lattice.

In contrast to that, $\llparenthesis 0 \preccurlyeq 0 \rrparenthesis$ is not the least element in the partial order $(\mathbb{IE}, \lessapprox)$ and therefore, the sequence

$$\left({}^{\mathsf{iwp}}_{\langle \varphi, C' \rangle}\Phi^n_{\llparenthesis f \preccurlyeq g \rrparenthesis}\llparenthesis 0 \preccurlyeq 0 \rrparenthesis \right)_{n \in \mathbb{N}}$$

is not necessarily an ascending chain. It is because of that, that the Kleene Fixed Point Theorem fails to be applicable in the context of integrability–witnessing expectations. We have to ensure the existence of the limit defining the semantics of while loops by other means. Obviously, it is desired that this limit always exists in order for iwp to be a well–defined transformer for *all possible programs* together with *all possible postexpectations*, and indeed, we can establish the following result:

> **THEOREM 9.11 (Well–definedness of iwp for While Loops [KK17b]):** *Let $C' \in \mathsf{pGCL}$ be tame and let $\llparenthesis f \preccurlyeq g \rrparenthesis \in \mathbb{IE}$. Then the limit*
>
> $$\mathsf{iwp}\,[\![\mathtt{while}(\varphi)\{C'\}]\!]\,\llparenthesis f \preccurlyeq g \rrparenthesis = \lim_{n \to \omega} {}^{\mathsf{iwp}}_{\langle \varphi, C' \rangle}\Phi^n_{\llparenthesis f \preccurlyeq g \rrparenthesis}\llparenthesis 0 \preccurlyeq 0 \rrparenthesis$$
>
> *exists and hence the* iwp *transformer of any while loop with respect to any postexpectation is well–defined.*

The core idea for proving this theorem is adopted from a well–known proof proving that every absolutely convergent series is also convergent. Let us go over this particular proof: If a series

$$S_{a_i} = \sum_{i=0}^{\infty} a_i \quad \text{converges absolutely}$$

this means that

$$S_{|a_i|} = \sum_{i=0}^{\infty} |a_i| \quad \text{converges to some value } a,$$

which implies that it does so unconditionally and monotonically since all summands are positive. This, in turn, implies that

$$S_{2\cdot|a_i|} = \sum_{i=0}^{\infty} 2 \cdot |a_i|$$

converges unconditionally and monotonically to $2 \cdot a$. Since

$$0 \;\leq\; a_i + |a_i| \;\leq\; 2 \cdot |a_i|$$

holds, we obtain

$$0 \;\leq\; \sum_{i=0}^{\infty} |a_i| + a_i \;\leq\; \sum_{i=0}^{\infty} 2 \cdot |a_i| \;=\; 2 \cdot a\,.$$

By that, we can see that the series

$$S_{|a_i|+a_i} \;=\; \sum_{i=0}^{\infty} |a_i| + a_i \quad \text{is bounded.}$$

Furthermore, since $|a_i| + a_i$ must be positive, the partial sums of the series $S_{|a_i|+a_i}$ are monotonically increasing and therefore

$$S_{|a_i|+a_i} \;=\; \sum_{i=0}^{\infty} |a_i| + a_i \quad \text{converges unconditionally.}$$

Since S_{a_i} is the difference of two unconditionally convergent series, namely

$$S_{a_i} \;=\; S_{|a_i|+a_i} - S_{|a_i|}$$

the series S_{a_i} must also converge. This basic idea of

$$\text{„express} \quad \sum a_i \quad \text{as} \quad \sum |a_i| + a_i - \sum |a_i|\text{“}$$

in case that the latter two infinite sums converge, is the underlying principle of the following proof of Theorem 9.11.

Proof (Theorem 9.11). First, we need to show by induction on the nesting depth of while loops and by induction on n that

$${}^{\mathsf{iwp}}_{\langle \varphi, C' \rangle}\Phi^{n}_{[\![f \leqslant g]\!]}([\![0 \leqslant 0]\!]) \;=\; \left({}^{\mathsf{wp}}_{\langle \varphi, C' \rangle}\Phi^{n}_{|f|+f}(0) - {}^{\mathsf{wp}}_{\langle \varphi, C' \rangle}\Phi^{n}_{|f|}(0) \;\leqslant\; {}^{\mathsf{wp}}_{\langle \varphi, C' \rangle}\Phi^{n}_{g}(0) \right)$$

holds for all n and any C'. This induction is straightforward and thus omitted here. It is then left to show that the limit of the right–hand–side exists for $n \to \omega$. We can see that the second component of that sequence converges monotonically towards

$$\sup_{n \in \mathbb{N}} {}^{\mathsf{wp}}_{\langle \varphi, C' \rangle}\Phi^{n}_{g}(0) \;=\; \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C'\}]\!]\,(g)\,.$$

Then for any state σ for which $\mathsf{wp}\,[\![\texttt{while}(\varphi)\{C'\}]\!]\,(g)(\sigma) < \infty$ holds, we have

$$\begin{aligned} &\sup_{n \in \mathbb{N}} {}^{\mathsf{wp}}_{\langle \varphi, C' \rangle}\Phi^{n}_{|f|+f}(0)(\sigma) \\ &\;=\; \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C'\}]\!]\,(|f|+f)(\sigma) \end{aligned}$$

$$
\begin{aligned}
&\leq \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C'\}]\!]\,(2\cdot|f|)(\sigma) && \text{(by monotonicity, Theorem 4.16)}\\
&\leq \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C'\}]\!]\,(2\cdot g)(\sigma) && \text{(by monotonicity, Theorem 4.16)}\\
&\leq 2\cdot \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C'\}]\!]\,(g)(\sigma) && \text{(by linearity, Theorem 4.21)}\\
&< 2\cdot\infty = \infty\,, \quad \text{and}
\end{aligned}
$$

$$
\begin{aligned}
&\sup_{n\in\mathbb{N}} {}^{\mathsf{wp}}_{\langle\varphi,C'\rangle}\Phi^n_{|f|}(0)(\sigma)\\
&= \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C'\}]\!]\,(|f|)(\sigma)\\
&\leq \mathsf{wp}\,[\![\texttt{while}(\varphi)\{C'\}]\!]\,(g)(\sigma) && \text{(by monotonicity, Theorem 4.16)}\\
&< \infty\,.
\end{aligned}
$$

Hence, the limit for both ${}^{\mathsf{wp}}_{\langle\varphi,C'\rangle}\Phi^n_{|f|+f}(0)(\sigma)$ and ${}^{\mathsf{wp}}_{\langle\varphi,C'\rangle}\Phi^n_{|f|}(0)(\sigma)$ exists, thus also the limit for ${}^{\mathsf{wp}}_{\langle\varphi,C'\rangle}\Phi^n_{|f|+f}(0)(\sigma) - {}^{\mathsf{wp}}_{\langle\varphi,C'\rangle}\Phi^n_{|f|}(0)(\sigma)$ exists, and therefore

$$
\lim_{n\to\omega}\left(\!\!\left[{}^{\mathsf{wp}}_{\langle\varphi,C'\rangle}\Phi^n_{|f|+f}(0) - {}^{\mathsf{wp}}_{\langle\varphi,C'\rangle}\Phi^n_{|f|}(0) \;\trianglelefteq\; {}^{\mathsf{wp}}_{\langle\varphi,C'\rangle}\Phi^n_{g}(0) \right]\!\!\right)_{n\in\mathbb{N}}
$$

exists, too. Q.E.D.

Let us revisit the two examples we presented in Section 9.1, i.e. the program

$$
\begin{aligned}
C_{geo} \rhd \quad & x := 1 \,;\\
& c := 1 \,;\\
& \texttt{while}(c = 1)\{\\
& \qquad \{c := 0\}\,[1/2]\,\{x := x+1\}\\
& \}\,,
\end{aligned}
$$

together with postexpectations $f = (-2)^x$ and $f' = {}^{(-2)^x}\!/_{x+1}$, respectively. In the iwp calculus, the respective weakest preexpectations are indeed well–defined, namely they are given by

$$
\mathsf{iwp}\,[\![C_{geo}]\!]\,[\![(-2)^x \trianglelefteq |(-2)^x|]\!] \;=\; [\![0 \trianglelefteq \infty]\!]
$$

$$
\text{and} \qquad \mathsf{iwp}\,[\![C_{geo}]\!]\left[\!\!\left[\frac{(-2)^x}{x+1} \;\trianglelefteq\; \left|\frac{(-2)^x}{x+1}\right| \right]\!\!\right] \;=\; [\![0 \trianglelefteq \infty]\!]\,.
$$

So the preexpectations of these two examples are well–defined and therefore these examples are not at all pathological in our presented calculus.

9.3.2 *Soundness of the* iwp *Calculus*

An important property of the iwp calculus that we have not established so far is its soundness, meaning that if we can establish

$$\mathsf{iwp}\,[\![C]\!]\,(\!|f \mathrel{\reflectbox{$\leqslant$}} g|\!) = (\!|f' \mathrel{\reflectbox{$\leqslant$}} g'|\!) \quad \text{and} \quad g'(\sigma) < \infty\,,$$

then $f'(\sigma)$ is in fact the expected value of f after termination of C on initial state σ. For that, we first generalize the fact that we have established in the proof of Theorem 9.11:

Lemma 9.12 ([KK17b]):
Let $(\!|f \mathrel{\reflectbox{$\leqslant$}} g|\!) \in \mathbb{IE}$ *and* $\mathsf{iwp}\,[\![C]\!]\,(\!|f \mathrel{\reflectbox{$\leqslant$}} g|\!) = (\!|f' \mathrel{\reflectbox{$\leqslant$}} g'|\!)$ *with* $\mathsf{wp}\,[\![C]\!]\,(g)(\sigma) < \infty$.
Then $f'(\sigma) = \mathsf{wp}\,[\![C]\!]\,(|f| + f)(\sigma) - \mathsf{wp}\,[\![C]\!]\,(|f|)(\sigma)$.

Proof. By induction on the structure of C, using the first equation from the proof of Theorem 9.11. Q.E.D.

In standard probability theory any mixed–sign random variable f can be decomposed into a positive part

$${}^{+}f \;=\; \lambda\sigma.\, \max\{0, f(\sigma)\} \in \mathbb{E}$$

and a negative part

$${}^{-}f \;=\; \lambda\sigma.\, -\min\{0, f(\sigma)\} \in \mathbb{E}\,,$$

with $f = {}^{+}f - {}^{-}f$. This decomposition is called *Jordan decomposition*. Notice that ${}^{+}f$ and ${}^{-}f$ are both *non–negative* expectations. The expected value of f is then defined as the expected value of ${}^{+}f$ minus the expected value of ${}^{-}f$, i.e. as

$$\int_{\Sigma} f \, d\mu \;=\; \int_{\Sigma} {}^{+}f \, d\mu \;-\; \int_{\Sigma} {}^{-}f \, d\mu\,,$$

if both $\int_{\Sigma} {}^{+}f \, d\mu < \infty$ and $\int_{\Sigma} {}^{-}f \, d\mu < \infty$. Using the Jordan decomposition of f, we can now establish the following lemma:

Lemma 9.13 ([KK17b]):
Let $(\!|f \mathrel{\reflectbox{$\leqslant$}} g|\!) \in \mathbb{IE}$, $\mathsf{iwp}\,[\![C]\!]\,(\!|f \mathrel{\reflectbox{$\leqslant$}} g|\!) = (\!|f' \mathrel{\reflectbox{$\leqslant$}} g'|\!)$ *with* $\mathsf{wp}\,[\![C]\!]\,(g)(\sigma) < \infty$,
and $f = {}^{+}f - {}^{-}f$. *Then* $f'(\sigma) + \mathsf{wp}\,[\![C]\!]\,({}^{-}f)(\sigma) = \mathsf{wp}\,[\![C]\!]\,({}^{+}f)(\sigma)$.

Proof. Consider the following:

$$\begin{aligned}
& f'(\sigma) + \mathsf{wp}\,[\![C]\!]\,({}^{-}f)(\sigma) \;=\; \mathsf{wp}\,[\![C]\!]\,({}^{+}f)(\sigma) \\
\text{iff}\quad & \mathsf{wp}\,[\![C]\!]\,(|f| + f)(\sigma) - \mathsf{wp}\,[\![C]\!]\,(|f|)(\sigma) + \mathsf{wp}\,[\![C]\!]\,({}^{-}f)(\sigma) \\
& \quad = \mathsf{wp}\,[\![C]\!]\,({}^{+}f)(\sigma) && \text{(by Lemma 9.12)} \\
\text{iff}\quad & \mathsf{wp}\,[\![C]\!]\,(|f| + f)(\sigma) + \mathsf{wp}\,[\![C]\!]\,({}^{-}f)(\sigma) \\
& \quad = \mathsf{wp}\,[\![C]\!]\,({}^{+}f)(\sigma) + \mathsf{wp}\,[\![C]\!]\,(|f|)(\sigma) \\
& && \text{(by } \mathsf{wp}\,[\![C]\!]\,(|f|)(\sigma) \leq \mathsf{wp}\,[\![C]\!]\,(g)(\sigma) < \infty) \\
\text{iff}\quad & \mathsf{wp}\,[\![C]\!]\,(|f| + f + {}^{-}f)(\sigma) \;=\; \mathsf{wp}\,[\![C]\!]\,({}^{+}f + |f|)(\sigma) \\
& && \text{(by linearity, Theorem 4.16)}
\end{aligned}$$

$$\begin{aligned}
&\text{iff} \quad \mathsf{wp}\,[\![C]\!]\,(|f| + {}^{+}f)(\sigma) \;=\; \mathsf{wp}\,[\![C]\!]\,({}^{+}f + |f|)(\sigma) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\text{by } f = {}^{+}f - {}^{-}f \text{ iff } {}^{+}f = f + {}^{-}f) \\
&\text{iff} \quad \text{true}
\end{aligned}$$

Q.E.D.

The soundness of the iwp transformer is a special case of Lemma 9.13.

THEOREM 9.14 (Soundness of iwp [KK17b]):
Let $f \in \mathbb{E}_{\pm}$ with $\mathsf{wp}\,[\![C]\!]\,(|f|)(\sigma) < \infty$ and let $\mathsf{iwp}\,[\![C]\!]\,⟅f \leqslant |f|⟆ = ⟅f' \leqslant g'⟆$. Then $f'(\sigma)$ is the expected value of f after termination of the tame program C on state σ, i.e. if $f = {}^{+}f - {}^{-}f$, then

$$f'(\sigma) \;=\; \mathsf{wp}\,[\![C]\!]\,({}^{+}f)(\sigma) - \mathsf{wp}\,[\![C]\!]\,({}^{-}f)(\sigma)\,.$$

Proof. In principle by setting $g = |f|$ in Lemma 9.13. Q.E.D.

9.4 HEALTHINESS CONDITIONS

Integrability–witnessing expectation transformers are closely connected with the classical expectation transformers we studied in Chapter 4 and hence share several of their properties. These can aid in reasoning about probabilistic programs, for instance by forming a foundation for compositional reasoning. We will study some of these properties in this section.

9.4.1 *Strictness*

Strictness is what Dijkstra calls „Law of the Excluded Miracle“ [Dij75]. The strictness property of weakest preexpectation transformers states that

$$\mathsf{wp}\,[\![C]\!]\,(0) \;=\; 0\,;$$

see Theorem 4.14 A. More abstractly, the least element 0 in the complete lattice $(\mathbb{E}, \preceq)$ is mapped to itself.

In our integrability–witnessing setting, there is no least element in $\mathbb{IE}$ and hence the question whether the iwp transformer is strict in this more abstract sense is nonsensical. Nevertheless, the iwp transformer maps the „zero element“ $⟅0 \leqslant 0⟆$ to itself.

THEOREM 9.15 (Strictness of iwp):
Let $C \in \mathsf{pGCL}$ be tame. Then $\mathsf{iwp}\,[\![C]\!]\,⟅0 \leqslant 0⟆ = ⟅0 \leqslant 0⟆$.

Proof. By induction on the structure of C. Q.E.D.

The above version of strictness tells us that — as expected — the expected value of the constantly 0 random variable after executing a program C is 0.

9.4.2 *Feasibility*

The property that McIver & Morgan call *feasibility* states that preexpectations cannot become „too large“ [MM05] (cf. Section 4.2.3). The notion of feasibility makes sense for bounded expectations f only and states that preexpectations cannot become „too large“, formally

$$f \preceq b, \text{ for some } b \in \mathbb{R}_{\geq 0} \quad \text{implies} \quad \mathsf{wp}\,[\![C]\!]\,(f) \preceq b\,.$$

In a mixed–sign setting, a natural generalization of feasibility would read

$$a \preceq f \preceq b, \text{ for some } a,b \in \mathbb{R} \quad \text{and} \quad \mathsf{iwp}\,[\![C]\!]\,\big(\!\!\big[f \trianglelefteq |f| \big]\!\!\big) = \big(\!\!\big[f' \trianglelefteq g' \big]\!\!\big)$$
$$\text{implies} \qquad g' \ll \infty \quad \text{and} \quad a \preceq f' \preceq b\,.$$

However, there is a simple counterexample: Consider $f = -1$ which is obviously upper– and lower–bounded by $a = b = -1$. But for the freezing program `diverge`, we have

$$\mathsf{iwp}\,[\![\texttt{diverge}]\!]\,\big(\!\!\big[-1 \trianglelefteq |-1| \big]\!\!\big) = \big(\!\!\big[0 \trianglelefteq 0 \big]\!\!\big)\,,$$

and while indeed $0 \ll \infty$, we find that $-1 \preceq 0 \not\preceq -1$. Therefore, strictness does not generalize to integrability–witnessing preexpectations, at least when generalizing as described above.

9.4.3 *Monotonicity*

Perhaps the single most useful property of the iwp transformer is monotonicity, as that is what enables compositional reasoning: It enables to continue reasoning soundly using overapproximations. For more details, see Section 4.2.4. Our integrability–witnessing transformer is monotonic with respect to the partial order $\lesssim$:

Theorem 9.16 (Monotonicity of iwp [KK17b]):
The iwp *transformer is* monotonic *with respect to* $\lesssim$*, i.e. for all tame programs* $C \in \mathsf{pGCL}$ *and postexpectations* $\big(\!\!\big[f \trianglelefteq g \big]\!\!\big), \big(\!\!\big[f' \trianglelefteq g' \big]\!\!\big) \in \mathbb{IE}$*,*

$$\big(\!\!\big[f \trianglelefteq g \big]\!\!\big) \lesssim \big(\!\!\big[f' \trianglelefteq g' \big]\!\!\big)$$
$$\text{implies} \qquad \mathsf{iwp}\,[\![C]\!]\,\big(\!\!\big[f \trianglelefteq g \big]\!\!\big) \lesssim \mathsf{iwp}\,[\![C]\!]\,\big(\!\!\big[f' \trianglelefteq g' \big]\!\!\big)\,.$$

Proof. By induction on the structure of C. All cases are straightforward, except for the while loop. Let $\big(\!\!\big[f \trianglelefteq g \big]\!\!\big) \lesssim \big(\!\!\big[f' \trianglelefteq g' \big]\!\!\big)$. Now consider

$$\mathsf{iwp}\Phi^{n}_{\langle \varphi, C' \rangle \big(\!\!\big[f \trianglelefteq g \big]\!\!\big)}\big(\!\!\big[0 \trianglelefteq 0 \big]\!\!\big) = \Big(\!\!\Big[\mathsf{wp}\Phi^{n}_{\langle \varphi, C' \rangle |f|+f}(0) - \mathsf{wp}\Phi^{n}_{\langle \varphi, C' \rangle |f|}(0) \trianglelefteq \mathsf{wp}\Phi^{n}_{\langle \varphi, C' \rangle g}(0) \Big]\!\!\Big)$$

from the proof of Theorem 9.11. Given that fact, the proof boils down to showing by induction on n that the inequality

$$\begin{aligned} &{}^{\mathsf{wp}}_{\langle \varphi, C' \rangle}\Phi^{n}_{|f|+f}(0)(\sigma) - {}^{\mathsf{wp}}_{\langle \varphi, C' \rangle}\Phi^{n}_{|f|}(0)(\sigma) \\ &\quad \leq {}^{\mathsf{wp}}_{\langle \varphi, C' \rangle}\Phi^{n}_{|f'|+f'}(0)(\sigma) - {}^{\mathsf{wp}}_{\langle \varphi, C' \rangle}\Phi^{n}_{|f'|}(0)(\sigma)\,, \end{aligned} \tag{†}$$

holds if $\mathsf{wp}\,[\![\texttt{while}(\varphi)\{C'\}]\!]\,(g')(\sigma) < \infty$ (and therefore by monotonicity also $\mathsf{wp}\,[\![\texttt{while}(\varphi)\{C'\}]\!]\,(g)(\sigma) < \infty$) holds. In that case, both

$$ {}^{\mathsf{wp}}_{\langle \varphi, C' \rangle}\Phi^{n+1}_{|f|}(0)(\sigma) \;\leq\; {}^{\mathsf{wp}}_{\langle \varphi, C' \rangle}\Phi^{n+1}_{g}(0)(\sigma) \;\leq\; \left(\mathsf{lfp}\, {}^{\mathsf{wp}}_{\langle \varphi, C' \rangle}\Phi_{g}\right)(\sigma) \;<\; \infty $$

and

$$ {}^{\mathsf{wp}}_{\langle \varphi, C' \rangle}\Phi^{n+1}_{|f'|}(0)(\sigma) \;\leq\; {}^{\mathsf{wp}}_{\langle \varphi, C' \rangle}\Phi^{n+1}_{g'}(0)(\sigma) \;\leq\; \left(\mathsf{lfp}\, {}^{\mathsf{wp}}_{\langle \varphi, C' \rangle}\Phi_{g'}\right)(\sigma) \;<\; \infty $$

holds, and we can thus rewrite inequality (†) as

$$\begin{aligned} &{}^{\mathsf{wp}}_{\langle \varphi, C' \rangle}\Phi^{n}_{|f|+f}(0)(\sigma) + {}^{\mathsf{wp}}_{\langle \varphi, C' \rangle}\Phi^{n}_{|f'|}(0)(\sigma) \\ &\quad \leq {}^{\mathsf{wp}}_{\langle \varphi, C' \rangle}\Phi^{n}_{|f'|+f'}(0)(\sigma) + {}^{\mathsf{wp}}_{\langle \varphi, C' \rangle}\Phi^{n}_{|f|}(0)(\sigma)\,, \end{aligned}$$

and prove this above statement instead by induction on n. For the induction step, consider the following:

$$\begin{aligned} &{}^{\mathsf{wp}}_{\langle \varphi, C' \rangle}\Phi^{n+1}_{|f|+f}(0)(\sigma) + {}^{\mathsf{wp}}_{\langle \varphi, C' \rangle}\Phi^{n+1}_{|f'|}(0)(\sigma) \\ &\quad \leq {}^{\mathsf{wp}}_{\langle \varphi, C' \rangle}\Phi^{n+1}_{|f'|+f'}(0)(\sigma) + {}^{\mathsf{wp}}_{\langle \varphi, C' \rangle}\Phi^{n+1}_{|f|}(0)(\sigma) \end{aligned}$$

iff

$$\begin{aligned} &\Big([\neg\varphi]\cdot(|f| + f + |f'|)\Big)(\sigma) \\ &\qquad + \Big([\varphi]\cdot \mathsf{wp}\,[\![C']\!]\,\Big({}^{\mathsf{wp}}_{\langle \varphi, C' \rangle}\Phi^{n}_{|f|+f}(0) + {}^{\mathsf{wp}}_{\langle \varphi, C' \rangle}\Phi^{n}_{|f'|}(0)\Big)\Big)(\sigma) \\ \leq\; &\Big([\neg\varphi]\cdot(|f'| + f' + |f|)\Big)(\sigma) \\ &\qquad + \Big([\varphi]\cdot \mathsf{wp}\,[\![C']\!]\,\Big({}^{\mathsf{wp}}_{\langle \varphi, C' \rangle}\Phi^{n}_{|f'|+f'}(0) + {}^{\mathsf{wp}}_{\langle \varphi, C' \rangle}\Phi^{n}_{|f|}(0)\Big)\Big)(\sigma) \\ &\qquad \text{(by definition of wp–characteristic function and linearity of wp)} \end{aligned}$$

The above follows from the induction hypothesis on n and from monotonicity of wp, Theorem 4.16. Q.E.D.

9.4.4 *Linearity*

Linearity of the classical wp transformers plays a prominent role in the development of McIver & Morgan on bounded expectations. We have studied linearity in our slightly more general model using unbounded expectations in Section 4.2.5. Not surprisingly, our integrability–witnessing expectation transformers are linear as well:

THEOREM 9.17 (Linearity of iwp):
Let $C \in \mathsf{pGCL}$ be tame, $⟅f \mathrel{⩽} g⟆, ⟅f' \mathrel{⩽} g'⟆ \in \mathbb{IE}$, and $r \in \mathbb{R}$. Then

$$\begin{aligned}&\mathsf{iwp}\,[\![C]\!]\left(r \cdot ⟅f \mathrel{⩽} g⟆ + ⟅f' \mathrel{⩽} g'⟆\right)\\ &\quad = r \cdot \mathsf{iwp}\,[\![C]\!]\,⟅f \mathrel{⩽} g⟆ + \mathsf{iwp}\,[\![C]\!]\,⟅f' \mathrel{⩽} g'⟆\,.\end{aligned}$$

Proof. Follows from Lemma 9.12 and linearity of wp. Q.E.D.

9.5 PROOF RULES FOR LOOPS

WHEREAS reasoning about non–loopy programs is mostly straightforward, reasoning about the iwp of a loop is more complicated as it involves reasoning about limits of integrability–witnessing expectation sequences. To help overcoming this difficulty, we present now an invariant–based approach that allows for overapproximating those limits.

We have already seen that the fact

$$\mathsf{iwp}\Phi^n_{⟨\varphi,C'⟩}{}_{⟅f \mathrel{⩽} g⟆}⟅0 \mathrel{⩽} 0⟆ = ⟅{}^{\mathsf{wp}}\Phi^n_{⟨\varphi,C'⟩}{}_{|f|+f}(0) - {}^{\mathsf{wp}}\Phi^n_{⟨\varphi,C'⟩}{}_{|f|}(0) \mathrel{⩽} {}^{\mathsf{wp}}\Phi^n_{⟨\varphi,C'⟩}{}_{g}(0)⟆$$

from the proof of Theorem 9.11 was vital to proving monotonicity of iwp. It will also allow us to reason about integrability–witnessing preexpectations through reasoning about standard weakest preexpectations, which is simpler since we have an easy–to–apply invariant rule for these.

If we take a closer look at the sequence

$$\left(⟅{}^{\mathsf{wp}}\Phi^n_{⟨\varphi,C'⟩}{}_{|f|+f}(0) - {}^{\mathsf{wp}}\Phi^n_{⟨\varphi,C'⟩}{}_{|f|}(0) \mathrel{⩽} {}^{\mathsf{wp}}\Phi^n_{⟨\varphi,C'⟩}{}_{g}(0)⟆\right)_{n\in\mathbb{N}}$$

we see that in order to overapproximate the limit of that sequence, we can

1. overapproximate $\lim_{n\to\omega}\ {}^{\mathsf{wp}}\Phi^n_{⟨\varphi,C'⟩}{}_{g}(0)$,
2. overapproximate $\lim_{n\to\omega}\ {}^{\mathsf{wp}}\Phi^n_{⟨\varphi,C'⟩}{}_{|f|+f}(0)$, and
3. underapproximate $\lim_{n\to\omega}\ {}^{\mathsf{wp}}\Phi^n_{⟨\varphi,C'⟩}{}_{|f|}(0)$.

Notice that these over- and underapproximations are over- and under–approximations of classical weakest preexpectations. Furthermore, recall that by Theorem 5.4 and Theorem 5.9 A. we have invariant rules for those approximations. This immediately leads us to the following proof rule for loops:

THEOREM 9.18 (Upper Bounds on iwp [KK17b]):
Let $⟅f \mathrel{⩽} g⟆ \in \mathbb{IE}$, $C' \in \mathsf{pGCL}$ be tame, and moreover let

A. *$I \in \mathbb{E}$ be a wp–superinvariant of $\mathtt{while}(\varphi)\{C\}$ with respect to post-expectation $|f| + f$, cf. Definition 5.1 A.,*

B. $(H_n)_{n\in\mathbb{N}} \subseteq \mathbb{E}$ *be a* wp*–*ω*–subinvariant of* $\texttt{while}(\varphi)\{C\}$ *with respect to postexpectation* $|f|$*, cf. Definition 5.3* B.*, and*

C. $G \in \mathbb{E}$*, with* $G(\sigma) < \infty$*, be a* wp*–superinvariant of* $\texttt{while}(\varphi)\{C\}$ *with respect to postexpectation* g*.*

Then $\quad \mathsf{iwp}\,[\![\texttt{while}(\varphi)\{C'\}]\!]\,(\!|f \trianglelefteq g|\!) \;\gtrapprox\; \left(\!\!\left| I - \sup_{n\in\mathbb{N}} H_n \trianglelefteq 2\cdot G \right|\!\!\right)$.

By similar considerations, we can find a dual theorem for lower bounds:

THEOREM 9.19 (Lower Bounds on iwp [KK17c]):
Let $(\!|f \trianglelefteq g|\!) \in \mathbb{IE}$*,* $C' \in \mathsf{pGCL}$ *be tame, and moreover let*

A. $(H_n)_{n\in\mathbb{N}} \subseteq \mathbb{E}$ *be a* wp*–*ω*–subinvariant of* $\texttt{while}(\varphi)\{C\}$ *with respect to postexpectation* $|f| + f$*, cf. Definition 5.3* B.*,*

B. $I \in \mathbb{E}$ *be a* wp*–superinvariant of* $\texttt{while}(\varphi)\{C\}$ *with respect to postexpectation* $|f|$*, cf. Definition 5.1* A.*, and*

C. $G \in \mathbb{E}$*, with* $G(\sigma) < \infty$*, be a* wp*–superinvariant of* $\texttt{while}(\varphi)\{C\}$ *with respect to postexpectation* g*.*

Then $\quad \left(\!\!\left| \sup_{n\in\mathbb{N}} H_n - I \trianglelefteq 2\cdot G \right|\!\!\right) \;\lessapprox\; \mathsf{iwp}\,[\![\texttt{while}(\varphi)\{C'\}]\!]\,(\!|f \trianglelefteq g|\!)$.

Notice that we have to use $2\cdot G$ in the second component of the over- and underapproximation of $\mathsf{iwp}\,[\![\texttt{while}(\varphi)\{C'\}]\!]\,(\!|f \trianglelefteq g|\!)$ in the above two theorems. This is just to ensure that the second component really bounds the absolute value of the first component. Using G instead might not yield a proper member of $\mathbb{IE}$. Notice that using $2\cdot G$ does not effect the integrability–witnessing property of the second component.

EXAMPLE 9.20 (Towards Amortized Expected runtime Analysis [KK17b]):
Suppose we want to perform an amortized analysis of a randomized data structure by means of a potential function. Suppose further that a certain operation *Op* first increases the potential by 1 and thereafter keeps flipping a coin until the first heads. With every flip of tails though, the potential is decreased by 3. We can model this situation by means of the following probabilistic program:

$$
\begin{aligned}
C_{Op} \rhd \quad & \Delta := \Delta + 1 \,; \\
& c := 1 \,; \\
& \texttt{while}(c = 1)\{ \\
& \qquad \{c := 0\}\,[1/2]\,\{\Delta := \Delta - 3\} \\
& \}
\end{aligned}
$$

Here Δ represents the change in the potential function. Notice that the change in potential might very well be positive (in fact with probability 1/2) as well as negative, so both possibilities have to be accounted for.

Even though an application of the operation *Op* might increase the potential, we now want to prove that an application of *Op decreases* the potential *in expectation*. This amounts to proving that the preexpectation of Δ evaluated in any initial state σ with $\sigma(\Delta) = 0$ is negative. For that, we need to calculate

$$\begin{aligned}&\mathsf{iwp}\,[\![C_{Op}]\!]\,(\!|\Delta \mathrel{\circledcirc} |\Delta|\,|\!)\\ &= \mathsf{iwp}\,[\![\Delta := \Delta + 1 \fatsemi c := 1 \fatsemi \mathtt{while}(c = 1)\{\ldots\}]\!]\,(\!|\Delta \mathrel{\circledcirc} |\Delta|\,|\!)\end{aligned}$$

So the first thing we need to do is to reason about the preexpectation of the while loop. Appealing to Theorem 9.18, we propose following invariants:

$$\begin{aligned}I &= \Delta + [c \neq 1]\cdot|\Delta| + [c = 1]\cdot\left(\sum_{i=0}^{\omega}\frac{|\Delta - 3\cdot i|}{2^{i+1}} - 3\right),\\ H_n &= [c \neq 1]\cdot|\Delta| + [c = 1]\cdot\sum_{i=0}^{n}\frac{|\Delta - 3\cdot i|}{2^{i+1}}, \qquad \text{and}\\ G &= [c \neq 1]\cdot|\Delta| + [c = 1]\cdot\sum_{i=0}^{\omega}\frac{|\Delta - 3\cdot i|}{2^{i+1}}\end{aligned}$$

Indeed, one can verify that these loop invariants satisfy the preconditions of Theorem 9.18. Furthermore, we observe that $\sup_{n\in\mathbb{N}} H_n = G$ holds. Applying Theorem 9.18 therefore yields

$$\mathsf{iwp}\,[\![\mathtt{while}(c = 1)\{\ldots\}]\!]\,(\!|\Delta \mathrel{\circledcirc} |\Delta|\,|\!) \;\lessapprox\; (\!|I - G \mathrel{\circledcirc} 2\cdot G\,|\!)\,.$$

Because G and I are absolutely convergent for any valuation of Δ (e.g. by the ratio test), we can calculate $I - G = \Delta - [c = 1]\cdot 3$, and so we get

$$\mathsf{iwp}\,[\![\mathtt{while}(c = 1)\{\ldots\}]\!]\,(\!|\Delta \mathrel{\circledcirc} |\Delta|\,|\!) \;\lessapprox\; (\!|\Delta - [c = 1]\cdot 3 \mathrel{\circledcirc} 2\cdot G\,|\!)\,.$$

Since iwp is monotonic (see Theorem 9.16), we can now safely continue our reasoning with the overapproximation $(\!|\Delta - [c = 1]\cdot 3 \mathrel{\circledcirc} 2\cdot G\,|\!)$ and calculate

$$\begin{aligned}&\mathsf{iwp}\,[\![\Delta := \Delta{+}1 \fatsemi c := 1]\!]\,(\!|\Delta - [c = 1]\cdot 3 \mathrel{\circledcirc} 2G\,|\!)\\ &= \left(\!\!\left|\, \Delta{-}2 \mathrel{\circledcirc} \sum_{i=0}^{\omega}\frac{|\Delta{+}1{-}3i|}{2^{i+1}} \,\right|\!\!\right).\end{aligned}$$

By that, we get in total an overapproximation of the sought–after preexpec-

tation iwp $[\![C_{Op}]\!]\ (\Delta \trianglelefteq |\Delta|)$. If we instantiate the second component of that overapproximation in an initial state σ with $\sigma(\Delta) = 0$, we get

$$\sum_{i=0}^{\omega} \frac{|0+1-3\cdot i|}{2^{i+1}} \;=\; 3 \;<\; \infty\,.$$

So the expected value at σ was integrable and thus it makes sense to evaluate the first component in σ (which is what we are really interested in). This gives $0-2=-2$ and thus executing Op decreases the potential *in expectation* by at least 2.

Note that the above analysis would not have been possible using either the deduction rules of PPDL [Koz85] or the invariant–based approaches presented in Section 5.2 off–the–shelf. Instead, a tailor–made argument would be needed for reasoning about the mixed–sign postexpectation Δ.

9.6 FUTURE AND RELATED WORK

An exception to the widespread and generally condoned neglect of unbounded mixed–sign expectations is Kozen's PPDL [Koz85] as it provides an expectation transformer semantics for probabilistic programs with respect to general measurable postexpectations f and thus does *not* forbid mixed–sign expectations altogether. PPDL's proof rule for *reasoning* about while loops, however, again requires f to be non–negative [Koz85, Section 4, page 168: the "while rule"]. This proof rule is hence unfit for reasoning about mixed–sign expectations. In fact, three out of four rules of the deduction system of PPDL that deal with iteration (and therefore with loops) require the postexpectation to be non–negative and are hence not applicable to reasoning about mixed–sign postexpectations f [Koz85, Section 4: Rules (8), (9), and the "while rule"]. The only exception to this is a rule that allows for upper bounding the *preexpectation* by a *non–negative function*, even if f is mixed–sign [Koz85, Section 4: Rule (10)]. This rule, however, is insufficient for upper–bounding the preexpectation by a negative value, which in practice can be desirable and is possible in our calculus, see Example 9.20.

Another drawback of PPDL is that reasoning even about simple programs and properties can become quite involved, requiring a fairly high degree of mathematical reasoning, i.e. to say that PPDL requires a lot of reasoning inside the program semantics while the approach of McIver & Morgan and the approach we present in this chapter constitutes more of a syntactic reasoning on the source code level. For example, [Koz85, Section 7] gives a circa two–page proof sketch of the expected runtime of a "simple random walk" carried out in PPDL. It requires a fair amount of domain–specific knowledge about integers and combinatorics and is thus not easily amenable to automation. A

full proof of the expected runtime in the wp calculus à la McIver & Morgan requires only a fraction of the effort (see [KK17c, Appendix A]).

As for future research directions, automations of wp–style proofs in theorem provers such as Isabelle/HOL have been developed [Hur02; Coc14]. A partial automation of the ert calculus from Chapter 7 has been carried out by Hölzl [Höl16]. The wp–style calculus for mixed–sign expectations we present here is closely related to these wp–style calculi and so we believe that existing automation techniques are likely to carry over easily.

Another promising direction for future work is, as already touched upon earlier in this chapter, *amortized expected runtime analysis*. In particular, it would be interesting to combine the ert calculus for reasoning about expected runtimes from Chapter 7 with the iwp calculus from this chapter and incorporate a potential function, in order to obtain a calculus for reasoning about amortized expected runtimes. A first step towards this was taken recently by Ngo *et al.* [NCH18].

Part III

COMPUTATIONAL HARDNESS

Did you implement this?

— Joël Ouaknine[7]

While the purpose of first two parts of this thesis was to develop methods for *reasoning* about probabilistic programs, I present in this third part results on the *computational hardness* of probabilistic program analysis. I first briefly recap the notion of the *arithmetical hierarchy*. Thereafter, I present results on the computational hardness of *approximating weakest preexpectations* and *deciding probabilistic termination*. Finally, I present hardness results on *approximating variance and covariance*.

7 On two occasions when I gave a presentation on the contents of Chapter 11 and Chapter 12.

THE ARITHMETICAL HIERARCHY 10

For every sensible question there is an answer;
for every answer there is a sensible question.

— Borut Robič [Rob15]

How undecidable is the halting problem? Is deciding termination of probabilistic programs in some mathematically rigorous sense *even harder*, given the fact that the halting problem of deterministic programs is already undecidable? How would we even sensibly capture *nuances of computational complexity* for undecidable decision problems, which evade any computable decision–making in the first place?

A solution to the just described classification task presents itself in the form of the *arithmetical hierarchy* — a non–collapsing double–stranded hierarchy for classifying sets of natural numbers according to the least amount of *syntactic complexity* required for defining them. What is this syntactic complexity that we speak of here? It will be the number of *quantifier alternations* needed to define a set using a formula in the language of first–order Peano arithmetic. The arithmetical hierarchy, originally independently due to Stephen Cole Kleene and Andrzej Mostowski, is defined as follows:

Definition 10.1 (Arithmetical Hierarchy [Kle43; Mos47; Odi92]**):**

A. *For every $n \in \mathbb{N}$, the class Σ_n^0 is defined as*[1]

$$\Sigma_n^0 = \Big\{ \mathcal{A} \;\Big|\; \mathcal{A} = \{ x \;|\; \exists y_1 \, \forall y_2 \, \exists y_3 \, \forall y_4 \cdots : \; (x, y_1, \ldots, y_n) \in \mathcal{R} \}, \\ \mathcal{R} \text{ is a decidable relation} \Big\}.$$

Multiple consecutive quantifiers of the same type *can be contracted into* one *quantifier of that type, so the number n refers to the number of necessary* quantifier alternations *rather than to the actual number of quantifiers occurring in a defining formula. In other words, the Σ_n^0 sets are definable by a formula with $n-1$ quantifier alternations, where the first quantifier is an* existential *one.*

1 The last quantifier is universal if n is even and existential if n is odd.

Note that we allow the values of the quantified variables to be drawn from a computable domain other than $\mathbb{N}$ that could be encoded in the natural numbers such as $\mathbb{Q}$, the set of syntactically correct programs, etc.

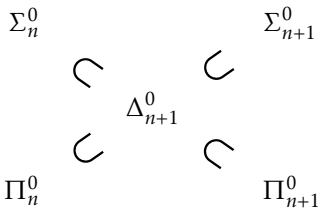


Figure 10.1: Strict inclusion relations between classes in the arithmetical hierarchy. Note that additionally $\Sigma^0_n \neq \Pi^0_n$ and $\Sigma^0_{n+1} \neq \Pi^0_{n+1}$ holds.

B. *For every $n \in \mathbb{N}$, the class Π^0_n is defined as*[2]

$$\Pi^0_n = \Big\{ \mathcal{A} \;\Big|\; \mathcal{A} = \{ x \;\big|\; \forall y_1 \exists y_2 \forall y_3 \exists y_4 \cdots : \ (x, y_1, \ldots, y_n) \in \mathcal{R} \}, \\ \mathcal{R} \text{ is a decidable relation} \Big\}.$$

Again, multiple consecutive quantifiers of the same type can be contracted. In other words, the Π^0_n sets are definable by a formula with $n-1$ quantifier alternations, where the first quantifier is a universal *one.*

C. *For every $n \in \mathbb{N}$, the class Δ^0_n is defined as*

$$\Delta^0_n = \Sigma^0_n \cap \Pi^0_n .$$

In other words, the Δ^0_n sets are definable by two *formulas with $n-1$ quantifier alternations, namely by one formula starting with an existential quantifier as well as by another formula starting with a universal quantifier.*

D. *A set $\mathcal{A}$ is called arithmetical, iff $\mathcal{A}$ is a member of Γ^0_n, for some $\Gamma \in \{\Sigma, \Pi, \Delta\}$ and $n \in \mathbb{N}$.*

E. *The inclusion diagram depicted in* Figure 10.1 *holds for every $n \geq 1$, thus the arithmetical sets form a strict non–collapsing hierarchy.*

F. *The classes $\Sigma^0_0 = \Pi^0_0 = \Delta^0_0 = \Delta^0_1$ all coincide and form precisely the class of the decidable sets. Σ^0_1 forms the class of the computably enumerable sets and Δ^0_2 the class of the limit–computable sets.*

A schematic depiction of the arithmetical hierarchy is shown in Figure 10.2.

2 The last quantifier is existential if n is even and universal if n is odd.

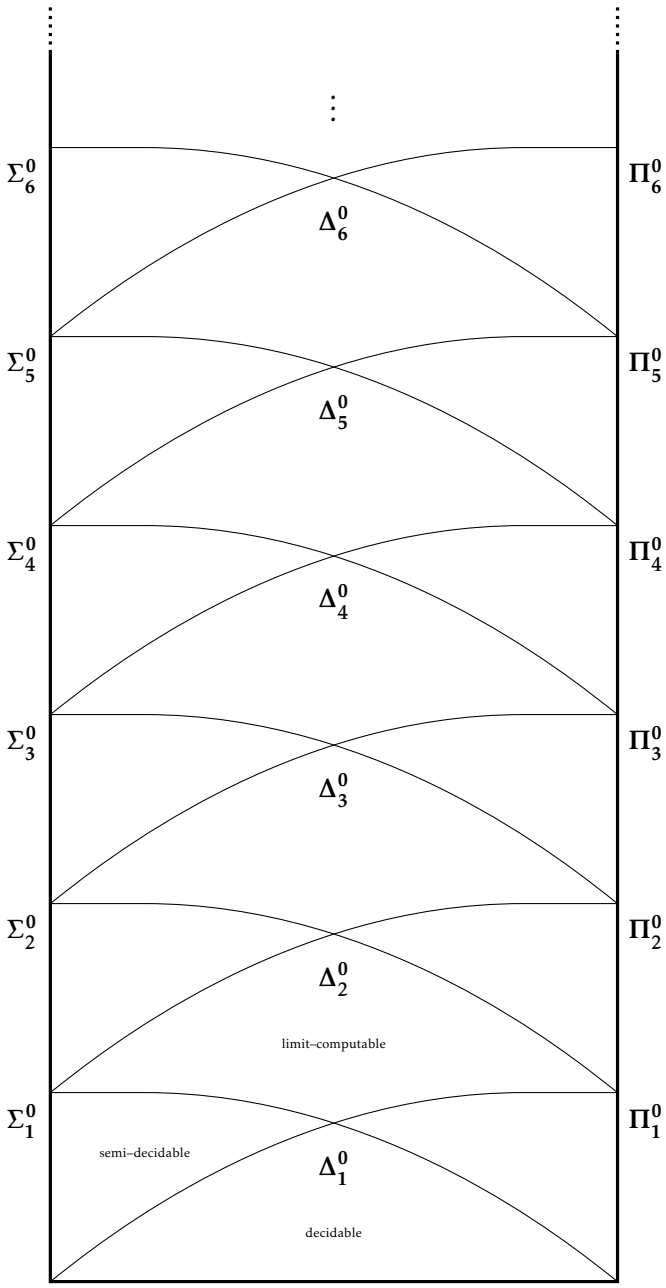


Figure 10.2: The infinite, double–stranded, non–collapsing arithmetical hierarchy.

The arithmetical hierarchy is of utter utility: Besides establishing a strong connection between computability and logic, stating precisely at which level in the arithmetical hierarchy a decision problem lies amounts to giving a measure of just „how unsolvable“ the decision problem is [Dav58].

In order to make mathematically rigorous statements of the form

„Problem $\mathcal{A}$ is at least as hard to solve as problem $\mathcal{B}$.“

we make use of two notions originally introduced by Emil Leon Post, nowadays called *many–one reducibility* and *many–one completeness*.

Definition 10.2 (Reducibility and Completeness [Pos44; Odi92]):
Let $\mathcal{A}$ and $\mathcal{B}$ be arithmetical sets and let X and Y be some appropriate universes such that $\mathcal{A} \subseteq X$ and $\mathcal{B} \subseteq Y$. $\mathcal{A}$ is called many–one reducible (or simply reducible) to $\mathcal{B}$, denoted

$$\mathcal{A} \leq_{\mathrm{m}} \mathcal{B} ,$$

iff there exists a computable function $r\colon X \to Y$, such that

$$\forall x \in X\colon \qquad x \in \mathcal{A} \quad \text{iff} \quad r(x) \in \mathcal{B} .$$

If r is a function such that r reduces $\mathcal{A}$ to $\mathcal{B}$, we denote this by

$$r\colon \mathcal{A} \leq_{\mathrm{m}} \mathcal{B} .$$

Note that $\leq_{\mathrm{m}}$ is transitive.

For $\Gamma \in \{\Sigma, \Pi, \Delta\}$, a set $\mathcal{A}$ is called many–one complete for Γ^0_n (or simply Γ^0_n–complete) iff both

A. *$\mathcal{A}$ is a member of Γ^0_n, and*

B. *$\mathcal{A}$ is Γ^0_n–hard, meaning $\mathcal{C} \leq_{\mathrm{m}} \mathcal{A}$, for any set $\mathcal{C} \in \Gamma^0_n$.*

The double–strandedness of the arithmetical hierarchy gives rise to the following duality between many–one complete sets and their complements.

Corollary 10.3:
Let $\mathcal{A} \subseteq X$ and let $\overline{\mathcal{A}} = X \setminus \mathcal{A}$ be the complement of $\mathcal{A}$. Then:

1. *If $\mathcal{A}$ is Σ^0_n–complete, then $\overline{\mathcal{A}}$ is Π^0_n–complete.*
2. *If $\mathcal{A}$ is Π^0_n–complete, then $\overline{\mathcal{A}}$ is Σ^0_n–complete.*

An important fact about Σ^0_n– and Π^0_n–complete sets is that they are in some sense the most complicated sets in Σ^0_n and Π^0_n, respectively: Formally, we have the following theorem:

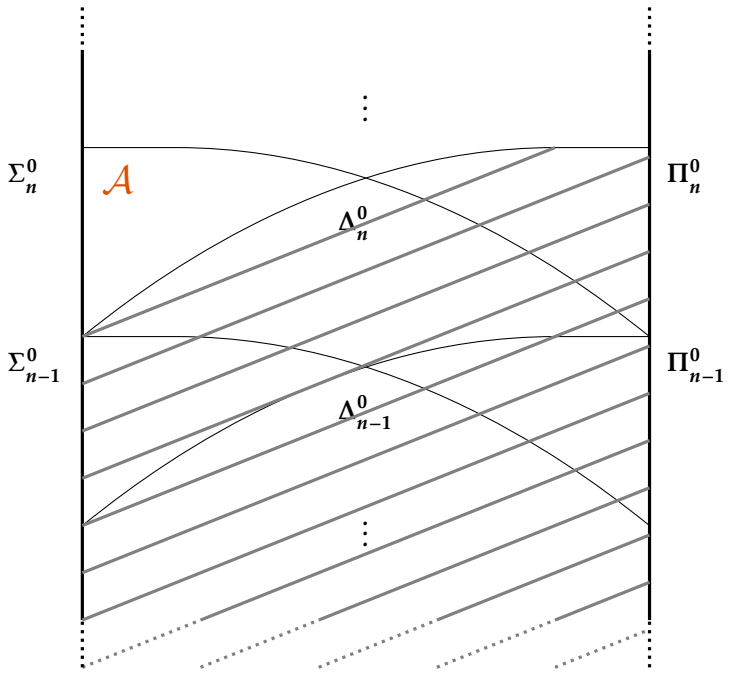


Figure 10.3: $\mathcal{A}$ is Σ^0_n–complete and hence sits properly at level n of the arithmetical hierarchy. In particular, $\mathcal{A}$ cannot be placed within the shaded area.

Theorem 10.4 ([Dav58])**:**

A. *If $\mathcal{A}$ is Σ^0_n–complete, then $\mathcal{A} \in \Sigma^0_n \setminus \Pi^0_n$*

B. *If $\mathcal{A}$ is Π^0_n–complete, then $\mathcal{A} \in \Pi^0_n \setminus \Sigma^0_n$.*

Theorem 10.4 implies in particular that if $\mathcal{A}$ is Σ^0_n–complete or Π^0_n–complete, then $\mathcal{A}$ is neither a member of Σ^0_{n-1} nor of Π^0_{n-1}, but $\mathcal{A}$ really „sits at level n“ in the arithmetical hierarchy. A graphical depiction of this situation is provided in Figure 10.3.

Many well–known and natural problems are complete for some level of the arithmetical hierarchy. Arguably one of the most prominent problems is the halting problem for deterministic programs, most prominently studied by Alan Mathison Turing [Tur37] and Alonzo Church [Chu36]. Below, we give a definition of the problem as well as a classification in terms of a level in the arithmetical hierarchy.

DEFINITION 10.5 (The Halting Problem):

A. *For a program $C \in \mathsf{pGCL}$, the set Σ_C is defined as*

$$\Sigma_C = \{\sigma \mid \sigma \colon V \to \mathsf{Vals}, \quad V \text{ is the set of program variables occurring in } C\},$$

i.e. Σ_C is the set of variable valuations that give only those variables a valuation, that actually occur in the program C. We call Σ_C the set of valid inputs for C. Note in particular that the individual V's are finite *for each C and thus Σ_C is* countable*, whereas the set Σ (see* Definition 2.1 B.*) is not countable.*

In order to pair programs with valid inputs, we define the set

$$\mathsf{GCL} \otimes \Sigma = \{(C, \sigma) \mid C \in \mathsf{GCL}, \quad \sigma \in \Sigma_C\}.$$

B. *The halting problem is the problem whether a deterministic program terminates on a given valid input. The according problem set $\mathcal{H} \subset \mathsf{GCL} \otimes \Sigma$ is defined as*

$$(C, \sigma) \in \mathcal{H} \qquad \text{iff}$$
$$\exists\, k\, \exists\, \tau \colon \quad \langle C, \sigma, 0, \varepsilon, \varepsilon, 1\rangle \vdash^k \langle \downarrow, \tau, k, \varepsilon, \varepsilon, 1\rangle,$$

i.e. σ is a valid input for C and there exists a number k and a state τ, such that the program C terminates on input σ within k computation steps in final state τ[3]*, see Definition 3.4.*

C. *The complement of the halting problem is the problem whether a program does* not *terminate on a given valid input. It is given by*

$$\overline{\mathcal{H}} = (\mathsf{GCL} \otimes \Sigma) \setminus \mathcal{H}.$$

THEOREM 10.6 (Hardness of the Halting Problem [Odi92; Odi99]):

A. *$\mathcal{H}$ is Σ^0_1–complete.*

B. *$\overline{\mathcal{H}}$ is Π^0_1–complete.*

The halting problem is the problem of whether a given program terminates on a *specific* valid input. Its *universal* version is the problem of whether a given program terminates on *all* valid inputs. Below, we also define the universal version and classify it in terms of a level in the arithmetical hierarchy.

3 Without having to make any probabilistic or nondeterministic choices (because the program is deterministic), hence the two ε's and the 1 on the right–hand–side of the $\vdash^k$. We reuse the definitions we made for probabilistic programs (cf. Definition 3.4), though, because they apply to deterministic programs as well and we have them readily available.

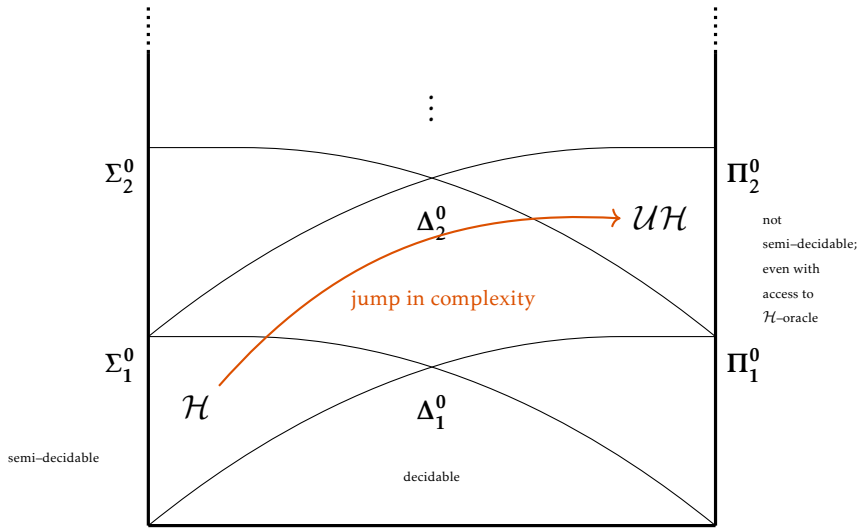


Figure 10.4: The jump in complexity when moving from the halting problem for a specific input to the universal halting problem for all inputs.

Definition 10.7 (The Universal Halting Problem):

A. *The universal halting problem is the problem whether a deterministic program terminates on all possible valid inputs. The according problem set* $\mathcal{UH} \subset \mathsf{GCL}$ *is defined as*

$$C \in \mathcal{UH} \quad \text{iff} \quad \forall \sigma \in \Sigma_C\colon \quad (C, \sigma) \in \mathcal{H}\,.$$

B. *The complement of the universal halting problem is the problem whether there exists an input on which a program does not terminate. It is given by*

$$\overline{\mathcal{UH}} = \mathsf{GCL} \setminus \mathcal{UH}\,.$$

Theorem 10.8 (Hardness of the Universal Halting Problem [Odi99]):

A. $\mathcal{UH}$ *is* Π_2^0*–complete.*

B. $\overline{\mathcal{UH}}$ *is* Σ_2^0*–complete.*

We observe that — as one would naturally expect — we have a complexity jump from $\mathcal{H}$ to $\mathcal{UH}$, namely from Σ_1^0 to Π_2^0, i.e. a jump one level up and to the „other strand“ of the hierarchy, see Figure 10.4. In other words, it is

strictly harder to decide whether a program halts on all inputs than it is to decide whether a program halts on a specific input.

Since we will later need to climb up to the third level of the arithmetical hierarchy in order to classify certain types of probabilistic program termination, we introduce another complete arithmetical problem, originally studied by Hartley Rogers [Rog59], that sits at level three of the arithmetical hierarchy: the problem of whether the set of valid inputs on which a deterministic program *diverges* is finite.

THEOREM 10.9 (The Cofiniteness Problem [Rog59; Odi99]):

A. *The cofiniteness problem is the problem of deciding whether the set of valid inputs on which a deterministic program C terminates is cofinite.*[4] *The according problem set $\mathcal{COF} \subset \mathsf{GCL}$ is given by*

$$C \in \mathcal{COF} \quad \text{iff} \quad \{\sigma \in \Sigma_C \mid (C, \sigma) \in \mathcal{H}\} \text{ is cofinite.}^{5}$$

$\mathcal{COF}$ is Σ_3^0–complete.

B. *The complement of the cofiniteness problem is defined by*

$$\overline{\mathcal{COF}} = \mathsf{GCL} \setminus \mathcal{COF} .$$

$\overline{\mathcal{COF}}$ is Π_3^0–complete.

In Figure 10.5, we provide an overview of the lower levels of the arithmetical hierarchy and depict precisely for which level in the arithmetical hierarchy the problems we have presented in this chapter are complete.

Finally, we need to note a rather peculiar fact about the Δ_n^0–sets: many–one completeness of a Δ_n^0 problem is *not a well–behaved notion*, as the following hard–to–find[6] theorem demonstrates:

THEOREM 10.10 ([Rog67, Exercise 14–14, p. 332]):
For $n \geq 2$, there exists no Δ_n^0–complete set.[7]

The above theorem will be relevant to classifying problems about approximating covariances (see Chapter 13).[8]

4 In this context, a set is cofinite iff its *relative complement*, i.e. its complement with respect to some appropriate universe, is finite.
5 i.e. iff $\Sigma_C \setminus \{\sigma \in \Sigma_C \mid (C, \sigma) \in \mathcal{H}\}$ is finite.
6 Many thanks to Wolfgang Thomas for pointing us to this Theorem.
7 In the sense of many–one completeness.
8 More forthrightly, it will be our excuse for not providing a completeness result.

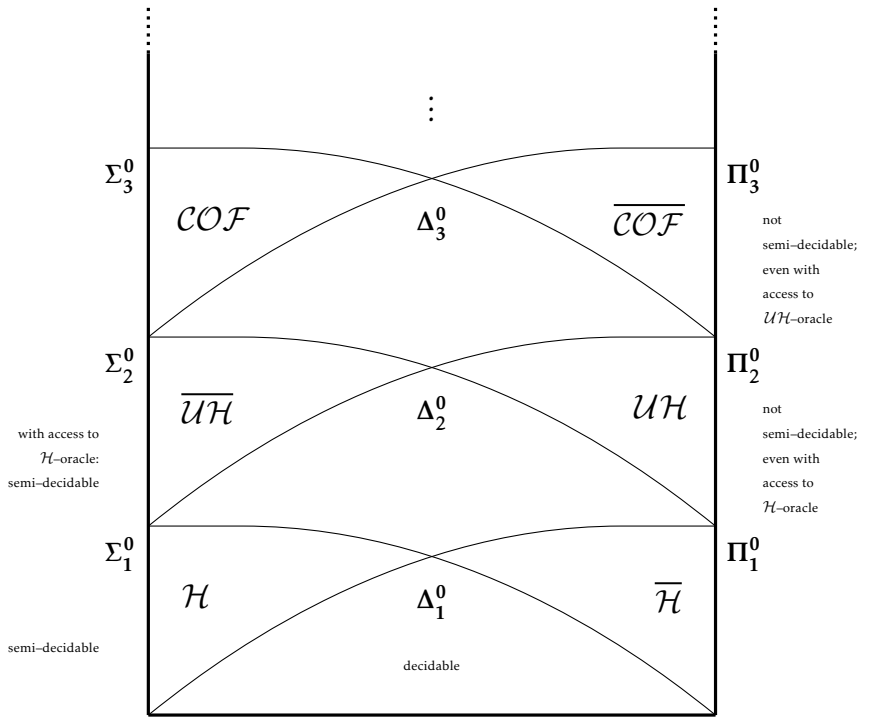


Figure 10.5: The complexity landscape of analysis problems for deterministic programs. All problems are complete for the respective level at which they lie in the arithmetical hierarchy.

APPROXIMATING PREEXPECTATIONS 11

Systematic reasoning about properties of probabilistic programs using dedicated calculi was the subject matter of Parts I and II of this thesis. Arguably the most difficult task in reasoning about programs is dealing with loops. For deterministic programs, loops are what brings about undecidability in reasoning about the programs.

In Chapter 5, we have presented rules that allowed us to tackle reasoning about loops in probabilistic programs. Our reasoning techniques were based on the concept of weakest preexpectations. Recall from Chapter 4 that *the weakest preexpectation of program C with respect to postexpectation f evaluated in σ* is the expected value of f evaluated in the final states reached after executing C on input σ.

A particular verification task that we came across was to obtain upper and/or lower bounds on weakest preexpectations, see Section 5.2. We argued that reasoning about lower bounds seems to be rather difficult, whereas reasoning about upper bounds seems to be easier. In this chapter, we will classify precisely how hard it really is to approximate weakest preexpectation from above and from below. To our surprise, we will find that the exact opposite is true from a computational hardness perspective: Though both algorithmically intractable, obtaining upper bounds is *strictly harder* than obtaining lower bounds.

In order to keep our analyses simple, we will restrict in this entire chapter (and the chapters to come) to *tame probabilistic programs* (see Definition 3.1 e.) whose only source of randomness are *binary probabilistic choices*, i.e. we do neither consider random assignments, nor nondeterminism. Thus, whenever we speak of pGCL programs in this entire part of the thesis, we mean programs of the just–described form.

Omitting random assignments from our programming lanugage does not really restrict our formalism. In principle, we could even restrict to *fair* probabilistic choices, i.e. choices with a probability of $1/2$, because a biased choice can be simulated by a program having only access to fair choices in expected constant time [AB09, Lemma 7.14]. One can even show that every *enumerable semi–measure*, i.e. every discrete probability distribution for which lower bounds on probabilities are computably enumerable, corresponds exactly to a probabilistic Turing machine having access only to fair probabilisitic choices [Ica17]. Thus anything we could hope to *compute* exploiting some source of randomness is also computable from just fair coin flips.

Towards analyzing the complexity of approximating weakest preexpectations, recall Definition 3.8, in which we defined the distribution $[\![C]\!]_\sigma$ over final states yielded by executing a probabilistic program C on input σ as

$$[\![C]\!]_\sigma(\tau) = \sum_{\gamma \in K} q\,, \qquad \text{where}$$
$$K = \left\{ \gamma \;\middle|\; \langle C, \sigma, 0, \varepsilon, \varepsilon, 1\rangle \vdash^* \langle \downarrow, \tau, n, \theta, \eta, q\rangle = \gamma \right\}.$$

Since we are interested in approximating weakest preexpectations, we will now construct *computable* approximations of the distribution $[\![C]\!]_\sigma$ as well as computable approximations of weakest preexpectations, namely by restricting the number of allowed computation steps, i.e. by restricting n above.

Definition 11.1:

A. *Let C be a tame* pGCL *program and $\sigma \in \Sigma$ be an initial program state. Then the distribution over final states established by executing C on input σ for exactly k steps, denoted $[\![C]\!]^{=k}_\sigma$, is a (sub)probability distribution over program states[1] given by*

$$[\![C]\!]^{=k}_\sigma(\tau) = \sum_{\gamma \in K} q\,, \qquad \text{where}$$
$$K = \left\{ \gamma \;\middle|\; \langle C, \sigma, 0, \varepsilon, \varepsilon, 1\rangle \vdash^* \langle \downarrow, \tau, n, \theta, \eta, q\rangle = \gamma, \quad n = k \right\}.$$

B. *Based on $[\![C]\!]^{=k}_\sigma$, we define an approximation of the weakest preexpectation* wp $[\![C]\!]\,(f)$ *as*

$$\mathsf{wp}^{=k}\,[\![C]\!]\,(f)(\sigma) = \sum_{\tau \in \Sigma_C} [\![C]\!]^{=k}_\sigma(\tau) \cdot f(\tau)\,.$$

As a consequence of the Kleene Normal Form Theorem [Kle43], *$[\![C]\!]^{=k}_\sigma$ is computable and hence* $\mathsf{wp}^{=k}\,[\![C]\!]\,(f)(\sigma)$ *is also computable, provided that f is computable. Moreover, note that*

$$\mathsf{wp}\,[\![C]\!]\,(f)(\sigma) = \sum_{k=0}^{\omega} \mathsf{wp}^{=k}\,[\![C]\!]\,(f)(\sigma)\,.$$

Even though weakest preexpectations will turn out not to be computably approximable, restricting to computable postexpectations is perfectly sensible because there is no hope of determining the value of a non–computable postexpectation in a final state.

In order to investigate the complexity of approximating weakest preexpectations, we define three sets: $\mathcal{LEXP}$, which relates to the set of rational lower bounds on wp $[\![C]\!]\,(f)(\sigma)$, $\mathcal{REXP}$, which relates to the set of rational upper bounds, and $\mathcal{EXP}$ which relates to the exact value of wp $[\![C]\!]\,(f)(\sigma)$:

1 I.e. $[\![C]\!]^{=k}_\sigma : \Sigma \to [0,1]$, such that $\sum_{\tau\in\Sigma} [\![C]\!]^{=k}_\sigma(\tau) \leq 1$.

DEFINITION 11.2 (Approximation Problems for wp [KK15b; KKM18]): *Analogously to* Definition 10.5 A., *in order to pair probabilistic programs with valid inputs and a* countable *space of* computable *postexpectations, we define the set*[2]

$$\mathsf{pGCL} \otimes \Sigma \otimes \mathbb{E} = \Big\{ (C, \sigma, f) \;\Big|\; C \in \mathsf{pGCL}, \quad \sigma \in \Sigma_C, \quad f : \Sigma_C \to \mathbb{Q}_{\geq 0}, \quad f \text{ computable} \Big\}.$$

The sets $\mathcal{LEXP}, \mathcal{REXP}, \mathcal{EXP} \subset \mathsf{pGCL} \otimes \Sigma \otimes \mathbb{E} \times \mathbb{Q}_{\geq 0}$ *are defined as*

$$\begin{aligned} (C, \sigma, f, q) \in \mathcal{LEXP} \quad &\text{iff} \quad q < \mathsf{wp}\,[\![C]\!]\,(f)(\sigma)\,, \\ (C, \sigma, f, q) \in \mathcal{REXP} \quad &\text{iff} \quad q > \mathsf{wp}\,[\![C]\!]\,(f)(\sigma)\,, \quad \text{and} \\ (C, \sigma, f, q) \in \mathcal{EXP} \quad &\text{iff} \quad q = \mathsf{wp}\,[\![C]\!]\,(f)(\sigma)\,. \end{aligned}$$

The computational hardness of approximating weakest preexpectations coincides with the hardness of deciding these problem sets.

11.1 LOWER BOUNDS

The first hardness result we establish is the Σ^0_1–completeness of $\mathcal{LEXP}$. For that, we show that $\mathcal{LEXP}$ is a Σ^0_1–problem and then show by a reduction from the (non–universal) halting problem for deterministic programs that $\mathcal{LEXP}$ is Σ^0_1–hard.

THEOREM 11.3 (Hardness of Lower Bounds on wp [KK15b; KKM18]): *$\mathcal{LEXP}$ is Σ^0_1–complete.*

Proof. For showing that $\mathcal{LEXP}$ is a member of Σ^0_1, consider the following:

$$\begin{aligned} & (C, \sigma, f, q) \in \mathcal{LEXP} \\ \text{iff} \quad & q < \mathsf{wp}\,[\![C]\!]\,(f)(\sigma) && \text{(by Definition 11.2)} \\ \text{iff} \quad & q < \sum_{k=0}^{\omega} \mathsf{wp}^{=k}\,[\![C]\!]\,(f)(\sigma) && \text{(by Definition 11.1)} \\ \text{iff} \quad & \exists\, y\colon \quad q < \sum_{k=0}^{y} \mathsf{wp}^{=k}\,[\![C]\!]\,(f)(\sigma) && \text{(all summands positive)} \\ \text{implies} \quad & \mathcal{LEXP} \in \Sigma^0_1 && \text{(the above is a } \Sigma^0_1\text{–formula)} \end{aligned}$$

Figure 11.1 gives an intuition on the resulting Σ^0_1–formula: With increasing maximum computation length y, more and more mass of the expected value can be accumulated until eventually an expected value mass strictly larger than the threshold q has been accumulated.

2 We let computable postexpectations map to $\mathbb{Q}_{\geq 0}$ instead of, say, computable reals with an infinity element for simplicity of the presentation.

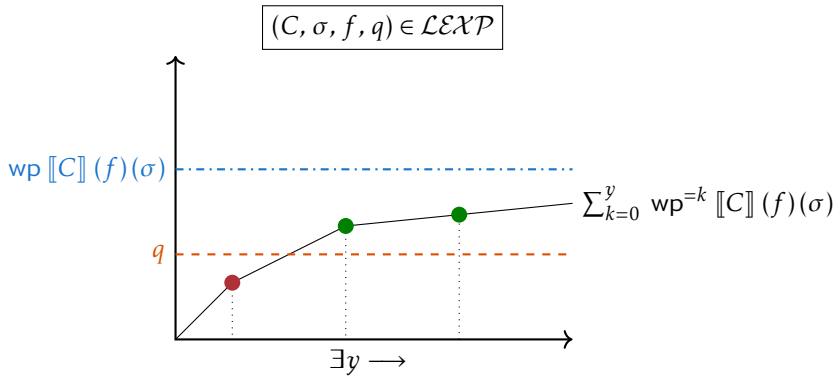


Figure 11.1: Schematic depiction of the formulae defining $\mathcal{LEXP}$. The solid line represents the monotonically increasing graph of $\sum_{k=0}^{y}$ wp$^{=k}$ $[\![C]\!]\,(f)(\sigma)$ plotted over increasing y.

It remains to show that $\mathcal{LEXP}$ is Σ_1^0–hard. For that, we construct a reduction function $r\colon \mathcal{H} \leq_{\mathrm{m}} \mathcal{LEXP}$ that reduces the Σ_1^0–complete non–universal halting problem (see Theorem 10.6 A.) to $\mathcal{LEXP}$. This function r takes a deterministic program $Q \in \mathsf{GCL}$ and a state σ as its input and computes

$$r(Q, \sigma) \;=\; (C, \sigma, 1, 1/2)\,,$$

where C is the probabilistic program

$$\{\texttt{skip}\}\,[1/2]\,\{Q\}\,.$$

Correctness of the reduction. There are two cases:

A. Q terminates on input σ. Then C terminates on σ with probability 1 and the expected value of 1 after executing the program C on input σ is thus 1. As $1/2 < 1$, we have that $(C, \sigma, 1, 1/2) \in \mathcal{LEXP}$.

B. Q does not terminate on input σ. Then C terminates with probability $1/2$ and the expected value of 1 after executing the program C on input σ is thus $1/2 \cdot 1 = 1/2$ since the right branch contributes 0 to the expected value of 1. As $1/2 \not< 1/2$, we have that $(C, \sigma, 1, 1/2) \notin \mathcal{LEXP}$. Q.E.D.

11.2 UPPER BOUNDS

As an immediate consequence of Theorem 11.3, $\mathcal{LEXP}$ is computably enumerable, see Definition 10.1 F. This means that all lower bounds for

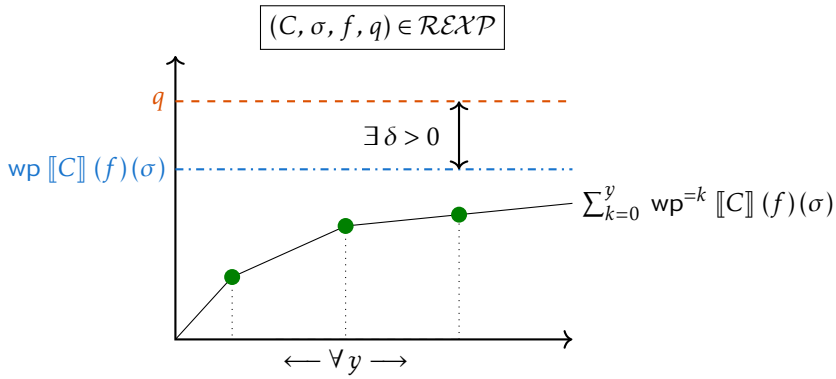


Figure 11.2: Schematic depiction of the formulae defining $\mathcal{REXP}$. The solid line represents the monotonically increasing graph of $\sum_{k=0}^{y} \mathsf{wp}^{=k}\,[\![C]\!]\,(f)(\sigma)$ plotted over increasing y.

preexpectations (with respect to a fixed initial state) can be effectively enumerated by some algorithm. Now, if upper bounds were computably enumerable as well, then preexpectations would be *computable reals*. However, we show that the contrary holds, because $\mathcal{REXP}$ is Σ_2^0–complete. Thus, by Theorem 10.4, we have $\mathcal{REXP} \notin \Sigma_1^0$, which means that upper bounds on preexpectations are not computably enumerable.

Theorem 11.4 (Hardness of Upper Bounds on wp [KK15b; KKM18]): *$\mathcal{REXP}$ is Σ_2^0–complete.*

Proof. For showing that $\mathcal{REXP}$ is a member of Σ_2^0, consider the following:

$$
\begin{aligned}
& (C, \sigma, f, q) \in \mathcal{REXP} \\
\text{iff}\quad & q > \mathsf{wp}\,[\![C]\!]\,(f)(\sigma) && \text{(by Definition 11.2)} \\
\text{iff}\quad & q > \sum_{k=0}^{\omega} \mathsf{wp}^{=k}\,[\![C]\!]\,(f)(\sigma) && \text{(by Definition 11.1)} \\
\text{iff}\quad & \exists\, \delta > 0\ \forall\, y\colon \quad q - \delta > \sum_{k=0}^{y} \mathsf{wp}^{=k}\,[\![C]\!]\,(f)(\sigma) \\
\text{implies}\quad & \mathcal{REXP} \in \Sigma_2^0 && \text{(the above is a } \Sigma_2^0\text{–formula)}
\end{aligned}
$$

Figure 11.2 gives an intuition on the resulting Σ_2^0–formula: No matter what maximum computation length y we allow and thereby no matter how much probability mass of the actual expected value we accumulate, this probability mass is strictly smaller than q (ensured by the safety margin δ).

It remains to show that $\mathcal{REXP}$ is Σ_2^0–hard: We do this by constructing a reduction function $r\colon \overline{\mathcal{UH}} \leq_{\mathrm{m}} \mathcal{REXP}$ that reduces the Σ_2^0–complete complement of the universal halting problem (see Theorem 10.8 B.) to $\mathcal{REXP}$: This function r takes a deterministic program $Q \in \mathsf{GCL}$ as its input and computes the tuple

$$r(Q) \;=\; (C, \sigma, v, 1)\,,$$

where σ is an arbitrary but fixed valid input for C, and $C \in \mathsf{pGCL}$ is the following probabilistic program:

$$
\begin{array}{l}
i := 0 \,;\\
\{c := 0\}\,[1/2]\,\{c := 1\}\,;\\
\\
\texttt{while}(c = 1)\{\\
\quad i := i + 1\,;\\
\quad \{c := 0\}\,[1/2]\,\{c := 1\}\\
\}\,;\\
\\
\\
k := 0\,;\\
\{c := 0\}\,[1/2]\,\{c := 1\}\,;\\
\\
\texttt{while}(c = 1)\{\\
\quad k := k + 1\,;\\
\quad \{c := 0\}\,[1/2]\,\{c := 1\}\\
\}\,;\\
\\
\\
v := \mathsf{wp}^{=k}\,[\![Q]\!]\,(1)\big(g_Q(i)\big)\cdot 2^{k+1}\,,
\end{array}
$$

where $g_Q\colon \mathbb{N} \to \Sigma_Q$ is some computable enumeration of valid inputs for Q. The last assignment of this program is a shortcut for the program that computes the right–hand–side of the assignment, which is computable (see Definition 11.1 B.), and stores the result of the computation in variable v. Recalling that $\mathsf{wp}\,[\![Q]\!]\,(1)(\sigma)$ is the „probability" of Q terminating on input σ, variable v takes value 2^{k+1} if and only if Q terminates on input $g_Q(i)$ after exactly k steps (otherwise it returns 0).

Correctness of the reduction. The two while loops generate independent geometric distributions with parameter $1/2$ on i and k, respectively. Thus, the probability of generating exactly the numbers i and k is

$$\frac{1}{2^{i+1}} \cdot \frac{1}{2^{k+1}} \;=\; \frac{1}{2^{i+k+2}}\,.$$

The expected valuation of v after executing the program C is hence indepen-

dent of the input σ and given by

$$\sum_{i=0}^{\infty}\sum_{k=0}^{\infty}\frac{\mathsf{wp}^{=k}\,[\![Q]\!]\,(1)\big(g_Q(i)\big)\cdot 2^{k+1}}{2^{i+k+2}} \;=\; \sum_{i=0}^{\infty}\sum_{k=0}^{\infty}\frac{\mathsf{wp}^{=k}\,[\![Q]\!]\,(1)\big(g_Q(i)\big)}{2^{i+1}}\,.$$

For each input, the number of steps until termination of Q is either unique or does not exist. Therefore $\mathsf{wp}^{=k}\,[\![Q]\!]\,(1)\big(g_Q(i)\big)$ is either 1 for exactly one k and 0 for all other k's, or 0 for all k's. Hence, the formula for the expected outcome reduces to

$$\sum_{i=0}^{\infty}\frac{1}{2^{i+1}} \;=\; 1$$

if and only if Q terminates on *every* input (valid for Q) after some finite number of steps. Thus, if there exists an input on which Q *does not* eventually terminate, then $(C,\sigma,v,1)\in\mathcal{REXP}$ as then the preexpectation of v is strictly less than 1. If, on the other hand, Q *does* terminate on every input, then this preexpectation is exactly 1 and hence $(C,\sigma,v,1)\notin\mathcal{REXP}$. Q.E.D.

11.3 EXACT VALUES

As mentioned before, a consequence of Theorem 11.3 and Theorem 11.4 for approximating preexpectations is that upper bounds are not computable at all whereas lower bounds are at least computably enumerable. Upper bounds would be computably enumerable if we had access to an oracle for the (non–universal) halting problem $\mathcal{H}$ for deterministic programs. Given a rational q it would then be semi–decidable whether q is an upper bound when provided access to an oracle for $\mathcal{H}$. Next, we establish that this is not the case for the problem of deciding whether q *equals* the value of the preexpectation. Formally, we establish the following hardness result:

Theorem 11.5 (Hardness of Exact Values for wp [KK15b; KKM18]): *$\mathcal{EXP}$ is Π^0_2–complete.*

Proof. For showing that $\mathcal{EXP}$ is a member of Π^0_2, consider the following: By Theorem 11.4 there exists a decidable relation $\mathcal{R}$, such that

$$(C,\sigma,v,q)\;\in\;\mathcal{REXP}\qquad\text{iff}\qquad\exists\, r_1\,\forall\, r_2\colon\quad(r_1,r_2,C,\sigma,v,q)\in\mathcal{R}\,. \tag{†}$$

Furthermore, by Theorem 11.3 there exists a decidable relation $\mathcal{L}$, such that

$$(C,\sigma,v,q)\;\in\;\mathcal{LEXP}\qquad\text{iff}\qquad\exists\,\ell\colon\quad(\ell,C,\sigma,v,q)\in\mathcal{L}\,. \tag{‡}$$

Let $\neg\mathcal{R}$ and $\neg\mathcal{L}$ be the (decidable) negations of $\mathcal{R}$ and $\mathcal{L}$, respectively. Then:

$$
\begin{aligned}
& (C, \sigma, v, q) \in \mathcal{EXP} \\
\text{iff}\quad & q = \mathsf{wp}[\![C]\!](v)(\sigma) && \text{(by Definition 11.2)} \\
\text{iff}\quad & q \leq \mathsf{wp}[\![C]\!](v)(\sigma) \quad \text{and} \quad q \geq \mathsf{wp}[\![C]\!](v)(\sigma) \\
\text{iff}\quad & \neg\big(q > \mathsf{wp}[\![C]\!](v)(\sigma)\big) \quad \text{and} \quad \neg\big(q < \mathsf{wp}[\![C]\!](v)(\sigma)\big) \\
\text{iff}\quad & \neg\big(\exists\, r_1\ \forall\, r_2 \colon \quad (r_1, r_2, C, \sigma, v, q) \in \mathcal{R}\big) && \text{(by † and ‡ above)} \\
& \qquad \text{and} \quad \neg\big(\exists\, \ell \colon \quad (\ell, C, \sigma, v, q) \in \mathcal{L}\big) \\
\text{iff}\quad & \forall\, r_1\ \exists\, r_2 \colon \quad (r_1, r_2, C, \sigma, v, q) \in \neg\mathcal{R} \\
& \qquad \text{and} \quad \forall\, \ell \colon \quad (\ell, C, \sigma, v, q) \in \neg\mathcal{L} \\
\text{iff}\quad & \forall\, r_1\ \forall\, \ell\ \exists\, r_2 \colon \quad (r_1, r_2, C, \sigma, v, q) \in \neg\mathcal{R} \\
& \qquad\qquad \text{and} \quad (\ell, C, \sigma, v, q) \in \neg\mathcal{L} \\
\text{implies}\quad & \mathcal{EXP} \in \Pi^0_2 && \text{(the above is a } \Pi^0_2\text{–formula)}
\end{aligned}
$$

It remains to show that $\mathcal{EXP}$ is Π^0_2–hard. We do this by reducing the Π^0_2–complete universal halting problem (see Theorem 10.8 A.) to $\mathcal{EXP}$. Reconsider for that the reduction function r from the proof of Theorem 11.4: Given a deterministic program Q, r computes the tuple $(C, \sigma, v, 1)$, where C is a probabilistic program with $\int_\Sigma 1 \, d\,[\![C]\!]_\sigma = 1$ if and only if Q terminates on all inputs. Thus $Q \in \mathcal{UH}$ iff $(C, \sigma, v, 1) \in \mathcal{EXP}$ and hence $r \colon \mathcal{UH} \leq_{\mathrm{m}} \mathcal{EXP}$. Q.E.D.

11.4 UPPER BOUNDS VS. LOWER BOUNDS

In Section 5.2, we argued that obtaining lower bounds on weakest preexpectations seems to be more difficult than obtaining lower bounds. In this chapter, we saw that the exact opposite is true: approximating weakest preexpectations (with respect to a given initial state) from above is strictly harder than from below. There is a slight discrepancy inherent in this comparison: In Section 5.2, we argued about non–strict *expectation* bounds which *map each initial state to some value*, whereas in this chapter we argued only about strict bounds with respect to *specified initial states*.

This mismatch could be rectified by considering for lower bounds instead of $\mathcal{LEXP}$ the problem $\mathcal{LEXP}_{\mathbb{E}} \subset \mathsf{pGCL} \times \Sigma \times \mathbb{E} \times \mathbb{E}$, defined by

$$
\begin{aligned}
& (C, f, g) \in \mathcal{LEXP}_{\mathbb{E}} \qquad \text{iff} \\
& \qquad \forall \sigma \in \Sigma_C \colon \quad \big(C, \sigma, f, g(\sigma)\big) \in \mathcal{LEXP} \ \text{ or } \ \big(C, \sigma, f, g(\sigma)\big) \in \mathcal{EXP}\,,
\end{aligned}
$$

which is a universal quantification over the disjunction of a Σ_1–formula and a Π^0_2–formula, which gives a Π^0_2–formula. Intuitively,

$$
(C, f, g) \in \mathcal{LEXP}_{\mathbb{E}} \qquad \text{iff} \qquad g \preceq \mathsf{wp}[\![C]\!](f)\,,
$$

i.e. if $g(\sigma)$ is a non–strict lower bound on the expected value of f measured in the final states reached after executing C on input σ, for all inputs σ.

An analogous lifting of $\mathcal{REXP}$ to $\mathcal{REXP}_{\mathbb{E}}$ would yield a Π^0_3–formula. It would be an interesting direction for future work to obtain lower bounds on the hardness of $\mathcal{LEXP}_{\mathbb{E}}$ and $\mathcal{REXP}_{\mathbb{E}}$ as well. However, *we strongly conjecture that $\mathcal{LEXP}_{\mathbb{E}}$ and $\mathcal{REXP}_{\mathbb{E}}$ are Π^0_2– and Π^0_3–complete, respectively*, although we did not investigate $\mathcal{LEXP}_{\mathbb{E}}$ and $\mathcal{REXP}_{\mathbb{E}}$ as part of this thesis.

Under the working hypothesis that $\mathcal{LEXP}_{\mathbb{E}}$ and $\mathcal{REXP}_{\mathbb{E}}$ are Π^0_2–complete and Π^0_3–complete, respectively, we would still be left with the situation that obtaining an upper bound on preexpectations is strictly harder than obtaining a lower bound, even though from a reasoning perspective the opposite should be expected, as we have extensively discussed in Section 5.2. We are not aware of any solution or explanation to this complexity mismatch, and this issue is thus an interesting, though intricate, direction for future work. We expect that resolving this complexity discrepancy might give new insights either into the computational complexity of probabilistic program analysis or into new techniques for reasoning about probabilistic programs (or both).

11.5 FINITENESS

Apart from approximating preexpectations, we also consider the question whether preexpectations are finite. This problem is closely related to the definedness of covariances (see Chapter 13) and also to the finiteness of expected runtimes (see Chapter 12). The finiteness problem is formalized by the problem set $\mathcal{FEXP}$:

Definition 11.6 (Finiteness Problem for wp [KKM18]):
The problem set $\mathcal{FEXP} \subset \mathsf{pGCL} \otimes \Sigma \otimes \mathbb{E}$ is defined as

$$(C, \sigma, f) \in \mathcal{FEXP} \quad \text{iff} \quad \mathsf{wp}\,[\![C]\!]\,(f)(\sigma) < \infty.$$

Since deciding whether a given rational number is an upper bound of a preexpectation is Σ^0_2–complete (cf. Theorem 11.4), it is not surprising that deciding finiteness of preexpectations is also in Σ^0_2, since we just have to existentially quantify an upper bound. In fact, it is Σ^0_2–complete as well.

Theorem 11.7 (Hardness of Finiteness of wp [KKM18]):
$\mathcal{FEXP}$ is Σ^0_2–complete.

Proof. For proving the membership $\mathcal{FEXP} \in \Sigma^0_2$, consider the following:

$$\begin{aligned} & (C, \sigma, f) \in \mathcal{FEXP} \\ \text{iff} \quad & \mathsf{wp}\,[\![C]\!]\,(f)(\sigma) < \infty \end{aligned}$$

$$
\begin{aligned}
&\text{iff} && \exists\, q\colon \quad \mathsf{wp}\,[\![C]\!]\,(f)(\sigma) \;<\; q \\
&\text{iff} && \exists\, q\colon \quad (C,\sigma,f,q) \in \mathcal{REXP} && \text{(Definition 11.2)} \\
&\text{implies} && \mathcal{FEXP} \;\in\; \Sigma^0_2 && \text{(Theorem 11.4, the above is a } \Sigma^0_2\text{–formula)}
\end{aligned}
$$

The proof that $\mathcal{FEXP}$ is Σ^0_2–hard is deferred to Lemma 12.7, because we use a reduction from the positive almost–sure termination problem (Definition 12.5), which is studied in detail in the next section.

By showing $\mathcal{FEXP} \in \Sigma^0_2$ and by (for now just) believing that $\mathcal{FEXP}$ is Σ^0_2–hard, we get that $\mathcal{FEXP}$ is Σ^0_2–complete. Q.E.D.

11.6 CONCLUSION AND FUTURE WORK

Our findings on the computational hardness of approximating preexpectations are summarized in Figure 11.3. Besides resolving the discrepancy in the difficulty of handling lower and upper bounds from a reasoning perspective on the one hand and from the computational hardness perspective on the other hand (see Section 11.4), we would like to mention three more directions for future work.

One direction would be to study the effect of reintroducing nondeterminism, i.e. considering a potentially non–tame program C. One would then study the hardness of approximating $\mathsf{wp}\,[\![C]\!]\,(f)$ and contrast it with the hardness of approximating $\mathsf{awp}\,[\![C]\!]\,(f)$ (cf. Definition 4.5). We conjecture that all of our results are preserved under both interpretations of nondeterminism, although there is also evidence that the complexity under angelic nondeterminism is different from the complexity under demonic nondeterminism, at least for questions of probabilistic termination [Cha+16].

Another direction for future work would be to study the impact of conditioning (see Chapter 8) on approximating preexpectations. We have currently no strong conjecture as to whether approximating conditional expected values is computationally harder or not. However, Ackerman, Freer, and Roy have shown that the operation of conditioning itself already introduces undecidability: Conditioning a computable random variable X with *computable* distribution $\mathsf{P}(X)$ on another random variable Y with *computable* distribution $\mathsf{P}(Y)$, yields a new random variable with *non–computable* distribution $\mathsf{P}(X \mid Y)$ [AFR11]. We view this as weak evidence that approximating conditional expected values might in fact be computationally harder than approximating non–conditional ones.

A final direction for future work would be to study the hardness of approximating weakest *liberal* preexpectations. Thereby, we would understand whether reasoning about partial correctness is really easier than reasoning about total correctness. In the next chapter, we study computational hardness aspects of analyzing the termination behavior of probabilistic programs.

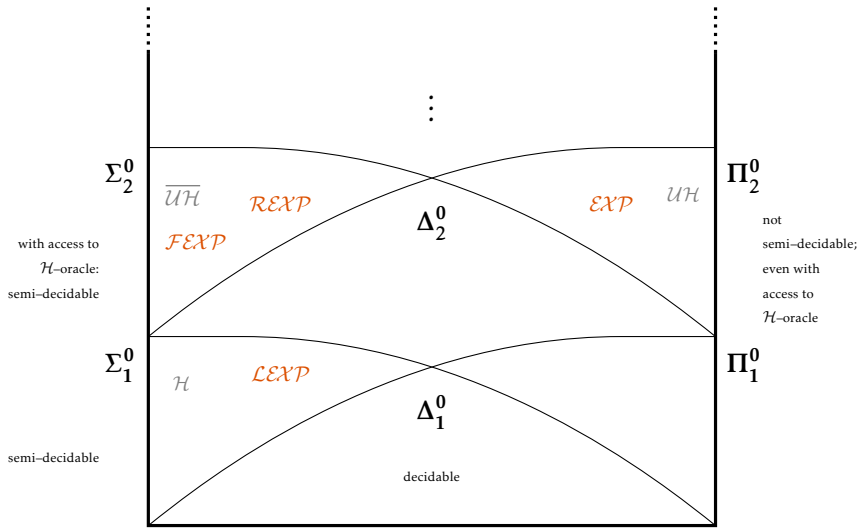


Figure 11.3: The complexity landscape of approximating weakest preexpectations. All problems are complete for the respective level at which they lie in the arithmetical hierarchy.

DECIDING PROBABILISTIC TERMINATION 12

Termination is one of the most elementary properties one can analyze an algorithm for. Almost a century ago, as early as 1920, Emil Leon Post had already anticipated the impossibility of finding effective procedures for perfoming such an analysis for nonprobabilistic algorithms [Pos04]. Soon after the advent of Alonzo Church's and Alan Mathison Turing's seminal undecidability results on the *Entscheidungsproblem* [Chu36; Tur37] — a decision problem posed by David Hilbert in 1928, which Turing then reduced to the termination problem for Turing machines —, scholars started to explore *degrees of unsolvability* for different decision problems related to the termination of algorithms. In this chapter, we explore the degree of unsolvability of *probabilistic termination*.

In Chapter 6, we presented methods for *reasoning* about two notions of probabilistic termination: almost–sure termination and positive almost–sure termination. The former describes that a probabilistic program terminates with probability 1, the latter that it terminates within finite expected time. In particular, we have presented in Chapter 7 the ert calculus for reasoning about expected runtimes. Naturally, this calculus can be used to reason about positive almost–sure termination. We also saw that reasoning about positive almost–sure termination using the induction rule of the ert calculus is conceptually rather simple (see Section 7.6.2), whereas reasoning about almost–sure termination appears much more involved (see Section 6.2).

In this chapter, we will classify precisely and rigorously how hard it really is to decide almost–sure termination and positive almost–sure termination of probabilistic programs. Similarly to the situation for approximating pre-expectations (see Chapter 11), we will find to our surprise that — at least for the universal versions of the probabilistic termination problems — the opposite of what we described above is true from a computational hardness perspective: Though both algorithmically intractable, deciding universal positive almost–sure termination is *strictly harder* than deciding universal almost–sure termination. Furthermore, we will argue based on our findings why perhaps *positive* almost–sure termination and not almost–sure termination should be considered *the* proper notion of probabilistic termination.

Recall from Chapter 11 that — in order to keep our analyses simple — we restrict pGCL in this entire part of the thesis to *tame probabilistic programs* (see Definition 3.1 e.) whose only source of randomness are *binary probabilistic choices*, i.e. we neither consider random assignments, nor do we consider nondeterminism.

12.1 ALMOST–SURE TERMINATION

In order to investigate the complexity of deciding almost–sure termination, we define two sets: $\mathcal{AST}$, which relates to almost–sure termination of a probabilistic program on a *specific* input, and $\mathcal{UAST}$, which relates to almost–sure termination on *all* valid inputs:

Definition 12.1 (A.–s. Termination Problem Sets [KK15b; KKM18]): *The sets* $\mathcal{AST} \subset \mathsf{pGCL} \otimes \Sigma$ *and* $\mathcal{UAST} \subset \mathsf{pGCL}$ *are defined as*

$$(C, \sigma) \in \mathcal{AST} \quad \text{iff} \quad \mathsf{wp}\,[\![C]\!]\,(1)(\sigma) = 1\,, \qquad \text{and}$$
$$C \in \mathcal{UAST} \quad \text{iff} \quad \forall\, \sigma \in \Sigma_C\colon \quad (C, \sigma) \in \mathcal{AST}\,.$$

As a first hardness result on probabilistic termination, we establish that deciding almost–sure termination on a specific input is Π^0_2–complete:

Theorem 12.2 (Hardness of A.–s. Termination [KK15b; KKM18]): *$\mathcal{AST}$ is Π^0_2–complete.*

Proof. For proving $\mathcal{AST} \in \Pi^0_2$, we show $\mathcal{AST} \leq_{\mathrm{m}} \mathcal{EXP}$. For that, consider the following reduction function $r\colon \mathcal{AST} \leq_{\mathrm{m}} \mathcal{EXP}$ which takes a probabilistic program C and a state σ as its input and computes $r(C, \sigma) = (C, \sigma, 1, 1)$.

Correctness of the reduction. Recall that $\mathsf{wp}\,[\![C]\!]\,(1)(\sigma)$ is precisely the probability of C terminating on input σ. Thus

$$(C, \sigma, 1, 1) \in \mathcal{EXP} \quad \text{iff} \quad (Q, \sigma) \in \mathcal{AST}$$

and therefore $r\colon \mathcal{AST} \leq_{\mathrm{m}} \mathcal{EXP}$. Since $\mathcal{EXP}$ is Π^0_2–complete by Theorem 11.5, it follows that $\mathcal{AST} \in \Pi^0_2$.

It remains to show that $\mathcal{AST}$ is Π^0_2–hard. For that, we reduce the Π^0_2–complete universal halting problem (see Theorem 10.8 A.) to $\mathcal{AST}$ by means of the reduction function $r'\colon \mathcal{UH} \leq_{\mathrm{m}} \mathcal{AST}$. r' takes a deterministic program Q as its input and computes the pair (C', σ), where σ is some fixed arbitrary valid input for C' and C' is the probabilistic program

$$\begin{aligned}
&i := 0\,;\\
&\{c := 0\}\,[1/2]\,\{c := 1\}\,;\\
&\texttt{while}\,(c = 1)\,\{\\
&\qquad i := i + 1\,;\\
&\qquad \{c := 0\}\,[1/2]\,\{c := 1\}\\
&\}\,;\\
&SQ\big(g_Q(i)\big)\,,
\end{aligned}$$

where $SQ\big(g_Q(i)\big)$ is a deterministic program that simulates the deterministic program Q on input $g_Q(i)$, and $g_Q\colon \mathbb{N} \to \Sigma_Q$ is some computable enumeration of valid inputs for Q.

Correctness of the reduction. The loop in C' establishes a geometric distribution with parameter ½ on i and hence a geometric distribution on all valid inputs for Q. After the while loop, the program Q is simulated on the probabilistically sampled input $g_Q(i)$. The entire program C' then terminates almost–surely on any arbitrary input σ, if and only if the simulation of Q terminates on every input. Thus $Q \in \mathcal{UH}$ if and only if $(C', \sigma) \in \mathcal{AST}$. Q.E.D.

While for deterministic programs there is a complexity gap between the halting problem and the universal halting problem (Σ_1^0–complete vs. Π_2^0–complete, see Figure 10.4), we establish that *there is no such gap for almost–sure termination*, i.e. $\mathcal{UAST}$ is exactly as hard to decide as $\mathcal{AST}$:

THEOREM 12.3 (Hardness of Universal A.–s. Term. [KK15b; KKM18]**):** *$\mathcal{UAST}$ is Π_2^0–complete.*

Proof. For proving the membership $\mathcal{UAST} \in \Pi_2^0$, consider that, by Theorem 12.2, there exists a decidable relation $\mathcal{R}$, such that

$$(C, \sigma) \in \mathcal{AST} \qquad \text{iff} \qquad \forall\, y_1\, \exists\, y_2\colon \quad (y_1, y_2, C, \sigma) \in \mathcal{R}\,.$$

By that we have that

$$C \in \mathcal{UAST} \qquad \text{iff} \qquad \forall\, \sigma \in \Sigma_C\, \forall\, y_1\, \exists\, y_2\colon \quad (y_1, y_2, C, \sigma) \in \mathcal{R}$$

which is a Π_2^0–formula and therefore $\mathcal{UAST} \in \Pi_2^0$.

It remains to show that $\mathcal{UAST}$ is Π_2^0–hard. We do this by reducing the Π_2^0–complete almost–sure termination problem $\mathcal{AST}$ (see Theorem 12.2) to $\mathcal{UAST}$ as follows: On input (C, σ) the reduction function $r\colon \mathcal{AST} \leq_{\mathrm{m}} \mathcal{UAST}$ computes a probabilistic program C' that first initializes all variables according to σ and then executes C. This reduction is clearly correct. Q.E.D.

As mentioned above, Theorem 12.3 stands in some contrast to the corresponding results for deterministic programs. With deterministic programs, a Σ_1^0–formula expressing their termination on a specified input is prepended with a universal quantifier over all valid inputs, yielding a Π_2^0–formula. The reason for the missing complexity gap between non–universal and universal *almost–sure* termination is that non–universal almost–sure termination is already a Π_2^0–property, basically due to the inherent universal quantification over all resolutions of probabilistic choices. Prepending this Π_2^0–formula with another universal quantifier over all valid inputs does not increase the complexity, as two universal quantifiers can computably be contracted to a single one, yielding again a Π_2^0–formula.

12.2 POSITIVE ALMOST–SURE TERMINATION

Towards analyzing the complexity of deciding almost–sure termination, we get from combining Definition 6.1 and Theorem 7.5, and from restricting to tame programs that

$$\operatorname{ert}[\![C]\!](0)(\sigma) = \sum_{i=1}^{\omega}\left(1 - \sum_{\langle\downarrow,\tau,n,\theta,\eta,q\rangle\in K^{<i}} q\right), \qquad \text{where}$$
$$K^{<i} = \left\{\langle\downarrow,\tau,n,\theta,\eta,q\rangle \,\middle|\, \langle C,\sigma,0,\varepsilon,\varepsilon,1\rangle \vdash^{*} \langle\downarrow,\tau,n,\theta,\eta,q\rangle,\ n<i\right\}.$$

Since we will need to approximate expected runtimes, we will now construct *computable* approximations of expected runtimes, namely by restricting the maximum number of allowed computation steps.

Definition 12.4:
Let $C \in \mathsf{pGCL}$ and $\sigma \in \Sigma_C$. Then the expected runtime of executing C on input σ for at most k steps, denoted $\operatorname{ert}^{\leq k}[\![C]\!](\sigma)$, is defined as

$$\operatorname{ert}^{\leq k}[\![C]\!](0)(\sigma) = \sum_{i=1}^{k}\left(1 - \sum_{\langle\downarrow,\tau,n,\theta,\eta,q\rangle\in K^{<i}} q\right),$$

where $K^{<i}$ as above. Due to the Kleene Normal Form Theorem [Kle43], $\operatorname{ert}^{\leq k}[\![C]\!](0)(\sigma)$ is computable. Moreover, note that

$$\operatorname{ert}[\![C]\!](f)(\sigma) = \sup_{k\in\mathbb{N}} \operatorname{ert}^{\leq k}[\![C]\!](0)(\sigma).$$

In order to investigate the complexity of deciding positive almost–sure termination, we define two sets: $\mathcal{PAST}$, which relates to positive almost–sure termination of a probabilistic program on a *specific* input, and $\mathcal{UPAST}$, which relates to positive almost–sure termination on *all* valid inputs:

Definition 12.5 (Positive A.–s. Term. Problem Sets [KK15b; KKM18]):
The sets $\mathcal{PAST} \subset \mathsf{pGCL} \otimes \Sigma$ and $\mathcal{UPAST} \subset \mathsf{pGCL}$ are defined as

$$(C,\sigma) \in \mathcal{PAST} \quad \text{iff} \quad \operatorname{ert}[\![C]\!](0)(\sigma) < \infty, \quad \text{and}$$
$$C \in \mathcal{UPAST} \quad \text{iff} \quad \forall\,\sigma \in \Sigma_C\colon \ (C,\sigma) \in \mathcal{PAST}.$$

Notice that both $\mathcal{PAST} \subset \mathcal{AST}$ and $\mathcal{UPAST} \subset \mathcal{UAST}$ hold.

We now investigate the computational hardness of deciding positive almost–sure termination: It turns out that deciding $\mathcal{PAST}$ is Σ^0_2–complete. Thus, $\mathcal{PAST}$ becomes semi–decidable when given access to an $\mathcal{H}$–oracle whereas

$\mathcal{AST}$ does not. We establish Σ^0_2–hardness by a reduction from $\overline{\mathcal{UH}}$. The implications of this reduction are rather counterintuitive: the reduction function *effectively* transforms each deterministic program that *does not terminate* on all inputs into a probabilistic program that *does terminate* within an expected finite number of steps.

Theorem 12.6 (Hardness of Positive A.–s. Term. [KK15b; KKM18]): *$\mathcal{PAST}$ is Σ^0_2–complete.*

Proof. For proving the membership $\mathcal{PAST} \in \Sigma^0_2$, consider the following:

$$
\begin{aligned}
& (C, \sigma) \in \mathcal{PAST} \\
\text{iff}\quad & \operatorname{ert}[\![C]\!](0)(\sigma) < \infty && \text{(by Definition 12.5)} \\
\text{iff}\quad & \exists\, c\colon \quad \operatorname{ert}[\![C]\!](0)(\sigma) < c \\
\text{iff}\quad & \exists\, c\colon \quad \sup_{k\in\mathbb{N}} \operatorname{ert}^{\leq k}[\![C]\!](0)(\sigma) < c && \text{(by Definition 12.4)} \\
\text{iff}\quad & \exists\, c\, \forall\, \ell\colon \quad \operatorname{ert}^{\leq \ell}[\![C]\!](0)(\sigma) < c \\
\text{implies}\quad & \mathcal{PAST} \in \Sigma^0_2 && \text{(the above is a } \Sigma^0_2\text{–formula)}
\end{aligned}
$$

It remains to show that $\mathcal{PAST}$ is Σ^0_2–hard. For that, we reduce the Σ^0_2–complete complement of the universal halting problem (see Theorem 10.8 b.) $\overline{\mathcal{UH}}$ to $\mathcal{PAST}$ by the reduction function $r(Q) = (C, \sigma)$, where σ is an arbitrary valid input for C and C is the probabilistic program

$c := 1\,;\ i := 0\,;\ x := 0\,;\ term := 0\,;$
$initQ\big(g_Q(i)\big)\,;$

`while`$(c = 1)\{$
  $stepQ\,;$
  `if` $(term = 1)\{$
    $Cheer(x)$
    $i := i + 1\,;\ term := 0\,;$
    $initQ\big(g_Q(i)\big)\,;$
  $\}$`else{skip}`
  $x := x + 1\,;$
  $\{c := 0\}\,[1/2]\,\{c := 1\}$
$\}$

where $InitQ\big(g_Q(i)\big)$ is a deterministic program that initializes a simulation of the deterministic program Q on input $g_Q(i)$ (recall the enumeration g_Q from Theorem 11.4), $StepQ$ is a deterministic program that does one single

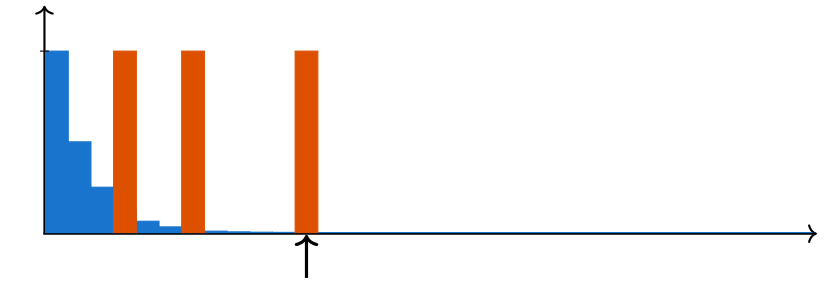


Figure 12.1: The cheering behavior of the program C from the proof of Theorem 12.6 in case that program Q does *not* terminate on every input. Orange bars correspond to loop iterations with cheering. The expected runtime of C corresponds to the integral over the bars. $g_Q(i)$ is the input with minimal i, such that Q does *not* terminate on $g_Q(i)$. Consequently, $g_Q(i-1)$ is the input with maximal i, such that Q terminates on all inputs $g_Q(0), \ldots, g_Q(i-1)$. Cheering occurs only finitely often, so the integral over the bars converges to a finite value.

(further) step of that simulation and sets *term* to 1 if that step has led to termination of Q, and $Cheer(x)$ is a deterministic program that executes 2^x many effectless computation steps. We refer to this as „cheering“.[1]

Correctness of the reduction. Intuitively, the program C starts by simulating Q on input $g_Q(0)$. During the simulation, it — figuratively speaking — gradually looses interest in further simulating Q by tossing a coin after each simulation step to decide whether to continue the simulation or not. In variable x, the program C counts the number of coin tosses it has made.

If eventually C finds that Q has terminated on input $g_Q(0)$, it „cheers“ for a number of steps exponential in the number of coin tosses that were made so far, namely for 2^x steps. C then continues with the same procedure for the next input $g_Q(1)$, and so on.

The variable x keeps track of the number coin tosses, and thus in effect also of the number of loop iterations (starting from 0). The x–th loop iteration takes place with probability $1/2^x$.

Notice that the simulation of a single step of the deterministic program Q, i.e. the program *StepQ*, requires (in our runtime model) at most a number linear in the number of instructions in program Q. Since the size of program Q is fixed in the construction of program C, we consider the time required to execute *StepQ* to be constant.

1 The program C cheers as it was able to prove the termination of Q on input $g_Q(i)$.

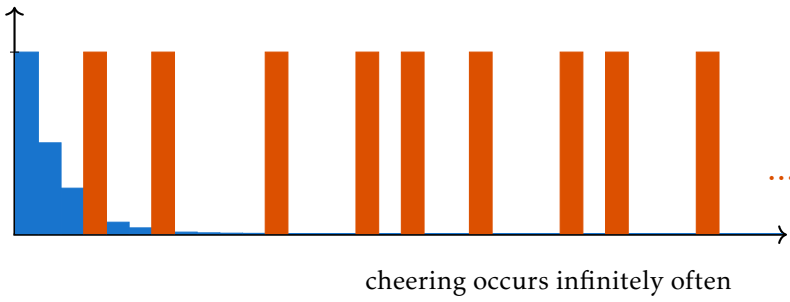


Figure 12.2: The cheering behavior of the program C from the proof of Theorem 12.6 in case that program Q *does* terminate on every input. Orange bars correspond to loop iterations with cheering. The expected runtime of C corresponds to the integral over the bars. Cheering occurs infinitely often, so the integral diverges to ∞

One loop iteration then consists of a constant number c_1 of steps in case Q did not terminate on input $g_Q(i)$ in the current simulation step. Such an iteration therefore contributes $c_1/2^x$ to the expected runtime of C. In case Q did terminate, a loop iteration takes a constant number c_2 of steps plus 2^x additional „cheering“ steps. Such an iteration therefore contributes

$$\frac{c_2 + 2^x}{2^x} = \frac{c_2}{2^x} + 1 > 1$$

to the expected runtime of C. Overall, the expected runtime ert $[\![C]\!](0)(\sigma)$ roughly resembles a geometric series with exponentially decreasing summands. However, for each time the program Q terminates on an input, a summand of the form $c_2/2^x + 1$ appears in this series. There are now two cases:

A. $Q \in \overline{\mathcal{UH}}$, so there exists some valid input σ with minimal i such that $g_Q(i) = \sigma$ on which Q does not terminate. This situation is depicted in Figure 12.1. Since $Q \in \overline{\mathcal{UH}}$, summands of the form $c_2/2^x + 1$ appear only $i - 1$ times in the series and therefore, the series converges—the expected runtime is finite, so $(C, \sigma) \in \mathcal{PAST}$.

B. $Q \notin \overline{\mathcal{UH}}$, so Q terminates on every input. This situation is depicted in Figure 12.2. Since $Q \notin \overline{\mathcal{UH}}$, summands of the form $c_2/2^x + 1$ appear infinitely often in the series and therefore, the series diverges—the expected runtime is infinite, so $(C, \sigma) \notin \mathcal{PAST}$. Q.E.D.

We now have the prerequisites to present the missing part of the proof of Theorem 11.7: We show that deciding $\mathcal{FEXP}$, i.e. the question whether a pre-

expectation computed by a probabilistic program for a given input is finite, is Σ_2^0–hard by reduction from the positive almost–sure termination problem.

LEMMA 12.7 (Hardness of Finiteness of wp [KKM18]): *$\mathcal{FEXP}$ is Σ_2^0–hard.*

Proof. We reduce the Σ_2^0–complete positive almost–sure termination problem $\mathcal{PAST}$ (see Theorem 12.6) to $\mathcal{FEXP}$ by the reduction function

$$r(C, \sigma) \;=\; (C', \sigma', v)\,,$$

where σ' is an arbitrary valid input for C' and C' is the probabilistic program

$c := 1 \,;\, k := 0 \,;$

`while`$(c = 1)\{$

$\quad k := k + 1$

$\quad \{c := 0\}\,[1/2]\,\{c := 1\}$

$\}\,;$

$v := \mathsf{wp}^{=k}\,[\![C]\!]\,(1)(\sigma) \cdot k \cdot 2^k\,.$

Recall that $\mathsf{wp}^{=k}\,[\![C]\!]\,(1)(\sigma)$ is the probability that C terminates on input σ after exactly k computation steps. For any concrete k, this is clearly computable (see also Definition 11.1 B.).

Correctness of the reduction. Regardless of its input σ', the while loop of program C' establishes a geometric distribution on variable k such that the probability that $k = i$ is given by $1/2^i$. This while loop terminates almost–surely. Thereafter, the program computes the assignment to v. Due to the geometric distribution on k, the program in effect stores for any $i \in \mathbb{N}$ the value $\mathsf{wp}^{=i}\,[\![C]\!]\,(1)(\sigma) \cdot i \cdot 2^i$ in variable v with probability $1/2^i$. The expected value of variable v is thus given by

$$\begin{aligned}
&\sum_{i=0}^{\omega} \frac{\mathsf{wp}^{=i}\,[\![C]\!]\,(1)(\sigma) \cdot i \cdot 2^i}{2^i} \\
&= \sum_{i=0}^{\omega} \mathsf{wp}^{=i}\,[\![C]\!]\,(1)(\sigma) \cdot i \\
&= \mathsf{ert}\,[\![C]\!]\,(0)(\sigma)\,.
\end{aligned}$$

We see that the expected value of v after executing C' on an arbitrary input equals exactly the expected runtime of C on input σ and this expected value is infinite if and only if the expected runtime is infinite. Thus, we have

$$(C, \sigma) \in \mathcal{PAST} \quad \text{iff} \quad r(C, \sigma) = (C', \sigma', v) \in \mathcal{FEXP}$$

and hence $r\colon \mathcal{PAST} \leq_{\mathrm{m}} \mathcal{FEXP}$. Q.E.D.

It is noteworthy that — as discussed in Section 7.2 — we cannot just annotate the given program C with a runtime counter and determine the expected value of that runtime counter. Therefore, we need the more involved construction presented in the proof of Lemma 12.7.

Coming back to termination problems, the last problem we study is *universal* positive almost–sure termination. In contrast to the non–positive version, we *do* have a complexity gap between non–universal and universal positive almost–sure termination. We will establish that $\mathcal{UPAST}$ is Π^0_3–complete and thus *strictly harder* to decide than $\mathcal{UAST}$. We do this by a reduction from $\overline{\mathcal{COF}}$, the complement of the cofiniteness problem (see Theorem 10.9):

Theorem 12.8 (Hardness of Univ. Pos. A.–s. Term. [KK15b; KKM18]**):** *$\mathcal{UPAST}$ is Π^0_3–complete.*

Proof. By Theorem 12.6, there exists a decidable relation $\mathcal{R}$, such that

$$(C,\sigma) \in \mathcal{PAST} \quad \text{iff} \quad \exists\, y_1\, \forall\, y_2 \colon \quad (y_1, y_2, C, \sigma) \in \mathcal{R}\,.$$

Therefore $\mathcal{UPAST}$ is definable by

$$C \in \mathcal{UPAST} \quad \text{iff} \quad \forall\, \sigma \in \Sigma_C\, \exists\, y_1\, \forall\, y_2 \colon \quad (y_1, y_2, C, \sigma) \in \mathcal{R}\,,$$

which is a Π^0_3–formula and therefore $\mathcal{UPAST} \in \Pi^0_3$.

It remains to show that $\mathcal{UPAST}$ is Π^0_3–hard. For that we reduce the Π^0_3–complete complement of the cofiniteness problem (see Theorem 10.9 b.) to $\mathcal{UPAST}$ using the following reduction function r: r takes a deterministic program Q as its input and computes the probabilistic program C given by

$c := 1\,;\ x := 0\,;\ term := 0\,;$
$initQ\big(g_Q(i)\big)\,;$

`while`$(c = 1)\{$
  $stepQ\,;$

  `if` $(term = 1)\{$
    $Cheer(x)$
    $i := i + 1\,;\ term := 0\,;$
    $initQ\big(g_Q(i)\big)\,;$
  $\}$`else{skip}`

  $x := x + 1\,;$
  $\{c := 0\}\,[1/2]\,\{c := 1\}$
$\}\,,$

where i assumed to range over the natural numbers, $InitQ(i)$ is a deterministic program that initializes a simulation of the program Q on input $g_Q(i)$

(recall the enumeration g_Q from Theorem 11.4), *StepQ* is a deterministic program that does one single (further) step of that simulation and sets *term* to 1 if that step has led to termination of Q, and *Cheer*(x) is a deterministic program that executes 2^x many effectless computation steps.

Note that program C is the same program as in the proof of Theorem 12.6 with one exception: The variable i is not initialized with 0, but instead left *uninitialized*. Thus, for every input σ, the program C skips all inputs for Q (in the order given by g_Q) up to the $\sigma(i)$-th input. After that, program C simulates Q on all remaining inputs starting from input $g_Q\big(\sigma(i)\big)$.

This ability to skip any number of inputs for some input state σ is crucial for the correctness of the reduction. Intuitively, program C terminates in finite expected time on all inputs, i.e. $C \in \mathcal{UPAST}$, if it is impossible to find an input state σ (and thus a value $\ell \in \mathbb{N}$ determined by $\sigma(i)$) such that executing C on σ skips all inputs on which Q does not terminate. Otherwise, C (when executed on such an input σ) keeps simulating terminating runs of Q and thus „cheers“ infinitely often. In this case, the expected runtime of C on input σ becomes infinite, i.e. $C \notin \mathcal{UPAST}$.

Correctness of the reduction. $\overline{\mathcal{COF}}$ can alternatively be defined as

$$Q \in \overline{\mathcal{COF}} \quad \text{iff} \quad \left\{ \sigma \in \Sigma_Q \;\middle|\; (Q, \sigma) \in \overline{\mathcal{H}} \right\} \text{ is infinite .}$$

There are now two cases:

1. $Q \notin \overline{\mathcal{COF}}$. Then there are only *finitely* many inputs on which Q does not terminate. Say $\ell \in \mathbb{N}$ is a minimal value such that Q does not terminate on input $g_Q(\ell)$, i.e. the program Q terminates on all inputs $g_Q(j)$ with $j > \ell$. Now, consider the execution of program C on some input σ with $\sigma(i) > \ell$. Then the „cheering“ steps in the `if`–branch of the while loop of C are executed infinitely often. Consequently, the runtime of C on that input σ is infinite (analogously to the proof of Theorem 12.6). Hence, $C \notin \mathcal{UPAST}$.

2. $Q \in \overline{\mathcal{COF}}$. Then there are *infinitely* many inputs on which Q does not terminate. For every input σ of C (and thus regardless of the number of skipped inputs for Q, i.e. the value ℓ that is initially assigned to variable i), the variable i will eventually be incremented to some value $j > \ell$ such that Q does not terminate on input $g_Q(j)$. From this point on, the „cheering“ steps in the `if`–branch of the while loop of C are *not* executed anymore. Consequently, for every input σ, the expected runtime of C on σ is finite. Hence, $C \in \mathcal{UPAST}$. Q.E.D.

Our hardness results are summarized in Figure 12.3. In the next section, we discuss possible implications and interpretations of our results.

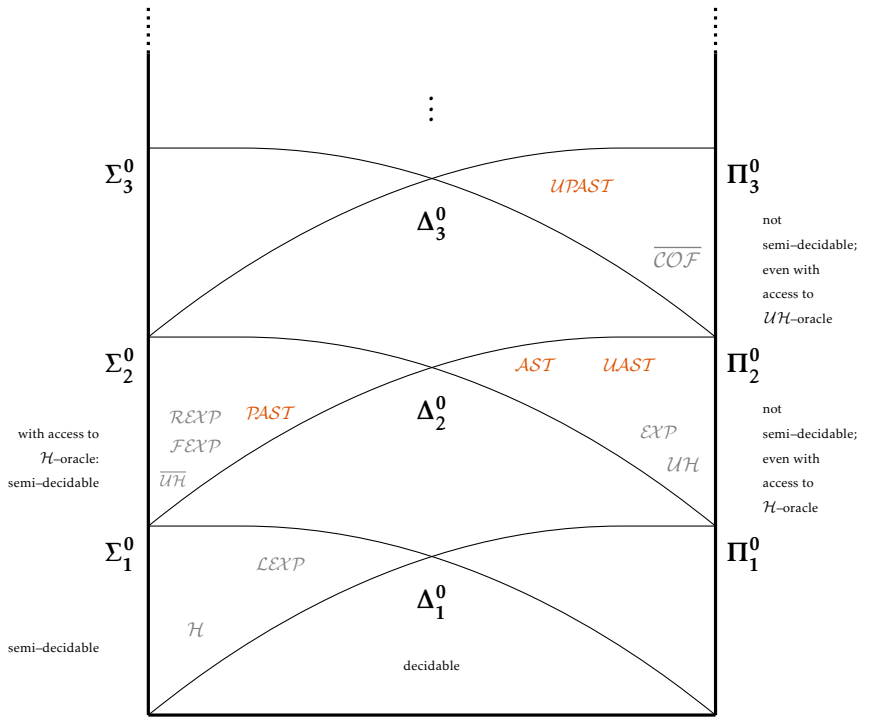


Figure 12.3: The complexity landscape of deciding probabilistic termination. All problems are complete for the respective level at which they lie in the arithmetical hierarchy.

12.3 ON THE PROPER NOTION OF TERMINATION

What should be the probabilistic analogon to termination of a program? Termination with probability 1? Or termination within finite expected time? In this section, we will argue why perhaps the latter should be considered *the* proper notion of probabilistic termination, even though our arguments are more of a philosophical and informal nature.

We have seen in Chapter 10, in particular Figure 10.4, that there is a jump in complexity when moving from the non–universal halting problem for deterministic programs to its universal variant. More specifically, we go from Σ_1^0–complete, which is not decidable but semi–decidable, to Π_2^0–complete, which is not even semi–decidable.

With probabilistic termination, we have two different situations: For almost–sure termination, which is often considered the probabilistic counterpart to deterministic termination, there is *no complexity jump* when comparing the non–universal to the universal variant. Both variants are Π_2^0–complete, which seems somewhat counterintuitive.

For *positive* almost–sure termination, we have a different situation: when moving from the non–universal to the universal variant, there *is* a complexity jump, namely from Σ_2^0–complete, which is not decidable but semi–decidable when having access to an oracle for the halting problem, to Π_3^0–complete, which is not semi–decidable, even if we did have access to an oracle for the halting problem.

In Figure 12.4, we have summarized the different complexity jumps. We can see that the complexity jump that presents itself for the positive almost–sure termination problem very closely resembles the complexity jump of the halting problem for deterministic programs — everything is just shifted up one level in the arithmetical hierarchy. The missing jump for the almost–sure termination problem, on the other hand, appears not to fit in as naturally into the diagram. We thus believe that the problem of (universal) positive almost–sure termination is maybe a more natural probabilistic analog to the (universal) halting problem for deterministic programs.

Another argument in favor of preferring the notion of positive almost–sure termination is concerned with the quantifier ordering and the type of objects that are quantified: For deterministic programs, we can define the halting problem by existentially quantifying over a witness computation length and then running the (unique) computation of at most that length on a given input. For the universal halting problem, we additionally universally quantify over all valid inputs.

Somewhat analogously, for probabilistic programs, we can define the problem of positive almost–sure termination by existentially quantifying over a witness expected computation length, then running all computations on a given input (captured by a $\forall$–quantifier), and seeing whether their accumulated expected computation length stays below the witness expected com-

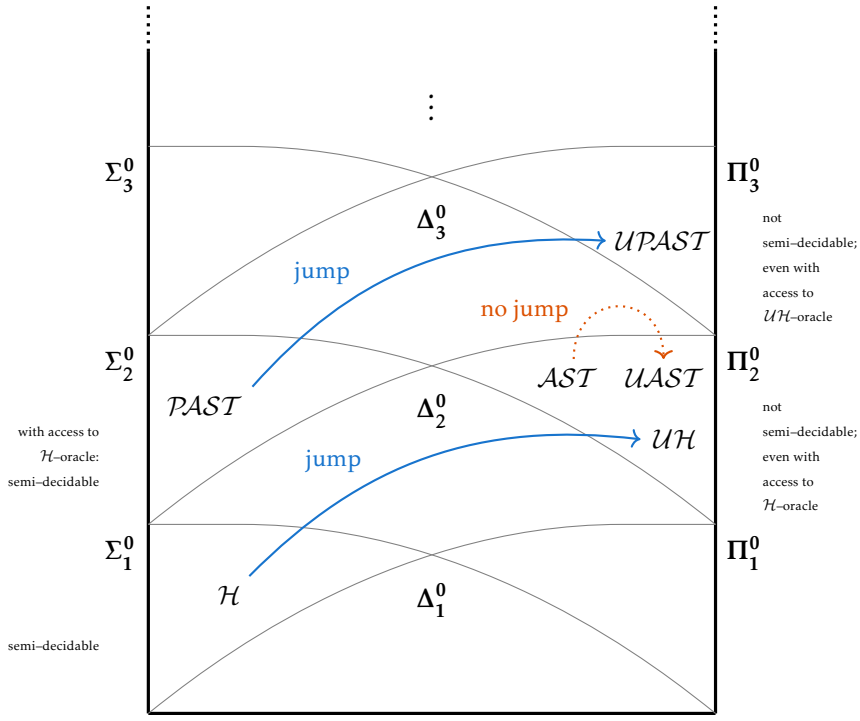


Figure 12.4: Complexity jumps that *do* or *do not* occur when moving from non–universal to universal termination problems.

putation length. For the universal version of the problem, we additionally universally quantify over all valid inputs.

A third argument pro positive almost–sure termination stems more from a user perspective: For a user, the expected runtime of an algorithm might be more relevant than its termination probability. After all, an algorithm whose runtime can at least be estimated to some finite value is likely more useful in practice than an algorithm for which one has to *expect to wait forever* until the algorithm finishes its computation (even if it does so with probability 1). Along this line of thought, the expected runtime of an algorithm is also a key notion in defining probabilistic complexity classes such as ZPP — the class of decision problems that can be decided by a probabilistic program that always gives the correct answer *within expected polynomial time* [Gil77].

12.4 FUTURE AND RELATED WORK

There are very few results on computational hardness in connection with probabilistic programs in general, like non–semi–decidability results for probabilistic rewriting logic by Bournez & Garnier [BG05] and decidability results for restricted probabilistic programming languages by Murawski & Ouaknine [MO05]. Precise classifications of the computational hardness of almost–sure termination, however, have received little attention. As a notable exception, Tiomkin established that deciding almost–sure termination of certain concurrent probabilistic programs is in Π^0_2 [Tio89]; a result which is in accordance with what we have established in the absence of concurrency.

As for the applicability of the work presented in this chapter, our results on the hardness of deciding almost–sure termination have been utilized by Breuvart & dal Lago [BDL18]. They define a type system for a probabilistic λ–calculus in which type derivations correspond to termination probabilities of λ–terms. They exploit our Π^0_2–completeness result to show that it is impossible to define a type system in which the exact termination probability of a λ–term is provable by a finite derivation.

A direction for future work would be to study the effect of reintroducing nondeterminism. For instance, will positive almost–sure termination still Σ^0_2–complete under angelic and/or demonic nondeterminism? We conjecture that all of our results are preserved under both interpretations of nondeterminism, although there is also evidence that the complexity of deciding probabilistic termination under angelic nondeterminism is different from the complexity under demonic nondeterminism [Cha+16].

The most interesting[2] source for future insights, however, is the current mismatch between the *conceptual complexity of reasoning* about and the *arithmetical computational complexity of deciding* probabilistic termination: While universal almost–sure termination is strictly easier to decide than universal

2 According to the author, that is.

positive almost–sure termination, reasoning about the former seems much more involved than reasoning about the latter (compare e.g. Theorem 6.8 with Theorem 7.16). In this thesis, we were neither able to resolve this enigma nor develop any intuition on why this complexity discrepancy exists. We do, however, strongly believe that resolving this mystery will yield either a more fine–grained view on the computational complexity of probabilistic program analysis or lead us to new techniques for reasoning about probabilistic programs (or both).

APPROXIMATING COVARIANCES 13

The covariance of two random variables f and g is one of the most basic measures of their *correlation*. If the covariance of f and g is positive, then f and g tend to be monotonically correlated, i.e. high values of f tend to correspond with high values of g. If the covariance is negative, on the other hand, then f and g tend to be antitonically correlated. Many other measures are ultimately built up on the notion of covariance, e.g. the *correlation coefficient*, the *variance*, or the *standard deviation*. Applications of the covariance range from biology over finance to meterology [Wikd].

In this chapter, we study the hardness of approximating the covariance of two postexpectations f and g (read: random variables) under the distribution $[\![C]\!]_\sigma$ obtained by executing a probabilistic program C on input σ (cf. Definition 3.8). Like we did for approximating preexpectations (read: expected values; cf. Chapter 4) in Chapter 11, we will give hardness results for approximating covariances from below, from above, and for deciding whether a given rational is equal to the sought–after covariance.

We show that obtaining bounds on covariances is computationally more difficult than for preexpectations. In particular, we prove that computing upper *and lower* bounds on covariances is both Σ^0_2–complete, thus *not computably enumerable*. In contrast to that, lower bounds on preexpectation *are* computable enumerable, thus preexpectations can be computably approximated from below, whereas covariances can not. We also show that determining the precise values of covariances is in Δ^0_3 and both Σ^0_2– and Π^0_2–hard. The covariance problem is thus a problem that lies „properly“ in Δ^0_3. Finally, we show how our findings carry over to approximating the variance of a single postexpectation.

The textbook definition of the covariance of two random variables f and g under distribution μ (see e.g. [ADD00, Definition 4.10.10, Lemma 4.10.6] or [Kle13, Definition 5.1]) is given by

$$\begin{aligned}\mathsf{Cov}(f,g) &= \int \left(f - \int f\,d\mu\right)\cdot\left(g - \int g\,d\mu\right) d\mu \\ &= \int f\cdot g\,d\mu - \int f\,d\mu \cdot \int g\,d\mu\,,\end{aligned}$$

provided that all integrals are finite. Note that $\int h\,d\mu$ is the expected value of random variable h under distribution μ. Recalling that the wp transformer yields precisely expected values (see Section 4.1), we obtain the following definition of covariance in the context of probabilistic programs:

Definition 13.1 (Covariance [KKM16; KKM18]):
Let $C \in \mathsf{pGCL}$, $\sigma \in \Sigma$, and $f, g \in \mathbb{E}$. Then the covariance of f and g after executing C on σ is given by

$$\mathsf{Cov}_{[\![C]\!]_\sigma}(f,g) \;=\; \mathsf{wp}\,[\![C]\!]\,(f\cdot g)(\sigma) \\ -\mathsf{wp}\,[\![C]\!]\,(f)(\sigma)\cdot\mathsf{wp}\,[\![C]\!]\,(g)(\sigma)\,,$$

if $\mathsf{wp}\,[\![C]\!]\,(f\cdot g)(\sigma)$, $\mathsf{wp}\,[\![C]\!]\,(f)(\sigma)$, and $\mathsf{wp}\,[\![C]\!]\,(g)(\sigma)$ are finite; otherwise, the covariance is undefined.

Since obtaining bounds on undefined covariances is not meaningful, we first need to deal with the definedness problem.

13.1 DEFINEDNESS

Since definedness of the covariance can generally not just be assumed bona fide, we first address the hardness of deciding whether a covariance is defined. According to Definition 13.1, the covariance $\mathsf{Cov}_{[\![C]\!]_\sigma}(f,g)$ is defined if and only if all preexpectations occurring on the right–hand–side of this definition are finite. Hence, we are concerned with the following problem set:

Definition 13.2 (Definedness of Covariance [KKM18]):
Analogously to Definition 11.2, in order to pair probabilistic programs with valid inputs and a countable space of pairs of computable postexpectations, we define the set

$$\mathsf{pGCL}\otimes\Sigma\otimes\mathbb{E}\otimes\mathbb{E} \;=\; \Big\{(C,\sigma,f,g)\;\Big|\; C\in\mathsf{pGCL},\quad \sigma\in\Sigma_C, \\ f,g\colon\Sigma_C\to\mathbb{Q}_{\geq 0},\quad f,g \text{ computable}\Big\}\,.$$

The problem set $\mathcal{DCOVAR}\subset\mathsf{pGCL}\otimes\Sigma\otimes\mathbb{E}\otimes\mathbb{E}$ is defined as

$$(C,\sigma,f,g)\in\mathcal{DCOVAR} \qquad \text{iff} \\ \mathsf{wp}\,[\![C]\!]\,(f)(\sigma)<\infty \quad\text{and}\quad \mathsf{wp}\,[\![C]\!]\,(g)(\sigma)<\infty \\ \text{and}\quad \mathsf{wp}\,[\![C]\!]\,(f\cdot g)(\sigma)<\infty\,.$$

The definition of $\mathcal{DCOVAR}$ is a conjunction of assertions that an expected value of a random variable is finite. As a consequence of the hardness of deciding whether a preexpectation is finite (Theorem 11.7), we obtain the following result on the hardness of deciding definedness of covariances:

Theorem 13.3 (Hardness of Definedness of Covariances [KKM16]):
$\mathcal{DCOVAR}$ is Σ^0_2–complete.

Proof. By Theorem 11.7, there is a decidable relation $\mathcal{F}$ such that

$$\begin{aligned}
(C,\sigma,f)\in\mathcal{FEXP} \quad &\text{iff} \quad \mathsf{wp}\,[\![C]\!]\,(f)(\sigma) < \infty \qquad \text{(by Definition 11.6)}\\
&\text{iff} \quad \exists\, y_1\,\forall\, y_2 \colon (y_1,y_2,C,\sigma,f) \in \mathcal{F} \qquad (\dagger)
\end{aligned}$$

Now consider the following:

$$\begin{aligned}
&(C,\sigma,f,g)\in\mathcal{DCOVAR}\\
\text{iff}\quad &\mathsf{wp}\,[\![C]\!]\,(f)(\sigma) < \infty \qquad \text{(by Definition 13.2)}\\
&\quad\text{and}\quad \mathsf{wp}\,[\![C]\!]\,(g)(\sigma) < \infty\\
&\quad\text{and}\quad \mathsf{wp}\,[\![C]\!]\,(f\cdot g)(\sigma) < \infty\\
\text{iff}\quad &\exists\, y_1\,\forall\, y_2 \colon \quad (y_1,y_2,C,\sigma,f) \in \mathcal{F} \qquad \text{(by } \dagger \text{ above)}\\
&\quad\text{and}\quad \exists\, y_1'\,\forall\, y_2' \colon \quad (y_1',y_2',C,\sigma,g) \in \mathcal{F}\\
&\quad\text{and}\quad \exists\, y_1''\,\forall\, y_2'' \colon \quad (y_1'',y_2'',C,\sigma,f\cdot g) \in \mathcal{F}\\
\text{iff}\quad &\exists\, y_1\,\exists\, y_1'\,\exists\, y_1''\,\forall\, y_2\,\forall\, y_2'\,\forall\, y_2'' \colon \quad (y_1,y_2,C,\sigma,f) \in \mathcal{F}\\
&\quad\text{and}\quad (y_1',y_2',C,\sigma,g) \in \mathcal{F}\\
&\quad\text{and}\quad (y_1'',y_2'',C,\sigma,f\cdot g) \in \mathcal{F}\\
\text{implies}\quad &\mathcal{DCOVAR} \in \Sigma_2^0 \qquad \text{(the above is a } \Sigma_2^0\text{–formula)}
\end{aligned}$$

It remains to show that $\mathcal{DCOVAR}$ is Σ_2^0–hard. For showing this, we reduce the Σ_2^0–complete problem $\mathcal{FEXP}$ (see Theorem 11.7) to $\mathcal{DCOVAR}$ by means of the reduction function $r(C,\sigma,f) = (C,\sigma,f,0)$.

Correctness of the reduction. Consider the following:

$$\begin{aligned}
&(C,\sigma,f) \in \mathcal{FEXP}\\
\text{iff}\quad &\mathsf{wp}\,[\![C]\!]\,(f)(\sigma) < \infty \qquad \text{(by Definition 11.6)}\\
\text{iff}\quad &\mathsf{wp}\,[\![C]\!]\,(f)(\sigma) < \infty \quad \text{and} \quad 0 < \infty \quad \text{and} \quad 0 < \infty\\
\text{iff}\quad &\mathsf{wp}\,[\![C]\!]\,(f)(\sigma) < \infty \qquad \text{(by strictness, Theorem 4.14 A.)}\\
&\quad\text{and}\quad \mathsf{wp}\,[\![C]\!]\,(0)(\sigma) < \infty \quad \text{and} \quad \mathsf{wp}\,[\![C]\!]\,(0)(\sigma) < \infty\\
\text{iff}\quad &\mathsf{wp}\,[\![C]\!]\,(f)(\sigma) < \infty\\
&\quad\text{and}\quad \mathsf{wp}\,[\![C]\!]\,(0)(\sigma) < \infty \quad \text{and} \quad \mathsf{wp}\,[\![C]\!]\,(f\cdot 0)(\sigma) < \infty\\
\text{iff}\quad &(C,\sigma,f,0) \in \mathcal{DCOVAR} \qquad \text{(by Definition 13.2)}
\end{aligned}$$

Thus, we have that

$$(C,\sigma,f) \in \mathcal{FEXP} \quad \text{iff} \quad r(C,\sigma,f) \in \mathcal{DCOVAR}$$

and hence $r\colon \mathcal{FEXP} \leq_{\mathrm{m}} \mathcal{DCOVAR}$. Q.E.D.

13.2 BOUNDS ON COVARIANCES

Now that we know the computational complexity of deciding the definedness of a covariance, we can study the hardness of approximating covari-

ances. Analogously to our studies on the hardness of approximating preexpectations in Chapter 11, we define three problem sets: one for lower bounds, one for upper bounds, and one for the exact value of $\mathsf{Cov}_{[\![C]\!]_\sigma}(f,g)$.

Definition 13.4 (Approx. Prob. for Covariances [KKM16; KKM18]): *The three sets $\mathcal{LCOVAR}, \mathcal{RCOVAR}, \mathcal{COVAR} \subset \mathsf{pGCL} \otimes \Sigma \otimes \mathbb{E} \otimes \mathbb{E} \times \mathbb{Q}$ are defined as*

$$
\begin{aligned}
&(C,\sigma,f,g,q) \in \mathcal{LCOVAR} \qquad \text{iff} \\
&\quad (C,\sigma,f,g) \in \mathcal{DCOVAR} \quad \text{and} \quad q < \mathsf{Cov}_{[\![C]\!]_\sigma}(f,g)\,,
\end{aligned}
$$

$$
\begin{aligned}
&(C,\sigma,f,g,q) \in \mathcal{RCOVAR} \qquad \text{iff} \\
&\quad (C,\sigma,f,g) \in \mathcal{DCOVAR} \quad \text{and} \quad q > \mathsf{Cov}_{[\![C]\!]_\sigma}(f,g)\,,
\end{aligned}
$$

$$
\begin{aligned}
&(C,\sigma,f,g,q) \in \mathcal{COVAR} \qquad \text{iff} \\
&\quad (C,\sigma,f,g) \in \mathcal{DCOVAR} \quad \text{and} \quad q = \mathsf{Cov}_{[\![C]\!]_\sigma}(f,g)\,.
\end{aligned}
$$

Thus, for instance $(C,\sigma,f,g,q) \in \mathcal{LCOVAR}$ if the covariance of f and g after executing C on σ is defined and q is a strict lower bound on that covariance.

The first fact we establish on the computational hardness of approximating covariances is that approximating lower bounds is Σ^0_2–complete:

Theorem 13.5 (Hardn. of Lower Bounds on Cov. [KKM16; KKM18]): *$\mathcal{LCOVAR}$ is Σ^0_2–complete.*

Proof. The proof of the membership $\mathcal{LCOVAR} \in \Sigma^0_2$ is very similar to the membership proof in Theorem 11.4 and thus omitted here. For more details, refer to [KKM18].

For proving the Σ^0_2–hardness of $\mathcal{LCOVAR}$ we reduce the complement of the almost–sure termination problem $\overline{\mathcal{AST}}$ — which as a consequence of Theorem 12.2 and Corollary 10.3 is Σ^0_2–complete — to $\mathcal{LCOVAR}$. Consider for that the reduction function $r\colon \overline{\mathcal{AST}} \leq_{\mathrm{m}} \mathcal{LCOVAR}$ with $r(C,\sigma) = (C',\sigma,1,1,0)$, where C' is given by

$$\{\,\mathtt{skip}\,\}\,[1/2]\,\{\,C\,\}\,.$$

Correctness of the reduction. First, note that by the feasibility property (Theorem 4.15), we have that $\mathsf{wp}\,[\![C']\!]\,(1)(\sigma) = \mathsf{wp}\,[\![C]\!]\,\big(1^2\big)(\sigma)$ must be bounded from above by 1 and is hence finite. Thus, the covariance $\mathsf{Cov}_{[\![C']\!]_\sigma}(1,1)$ is defined and given by

$$
\begin{aligned}
\mathsf{Cov}_{[\![C']\!]_\sigma}(1,1) &= \mathsf{wp}\,[\![C']\!]\,\big(1^2\big)(\sigma) - \mathsf{wp}\,[\![C']\!]\,(1)(\sigma)^2 \quad \text{(by Definition 13.1)} \\
&= \mathsf{wp}\,[\![C']\!]\,(1)(\sigma) - \mathsf{wp}\,[\![C']\!]\,(1)(\sigma)^2\,.
\end{aligned}
$$

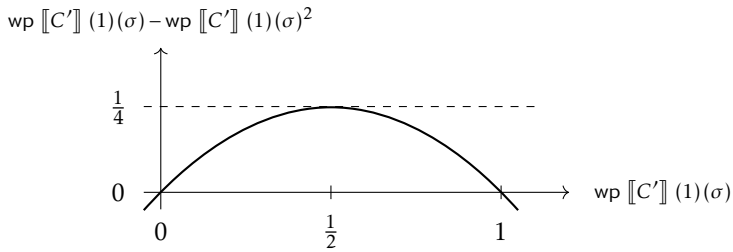


Figure 13.1: Plot of the termination probability of a program C' on input σ against the resulting variance. The curve is the one of the polynomial $x - x^2$.

Recall that $\mathsf{wp}\,[\![C']\!]\,(1)(\sigma)$ is precisely the probability of C' terminating on input σ. $\mathsf{Cov}_{[\![C']\!]_\sigma}(1,1) = \mathsf{Var}_{[\![C']\!]_\sigma}(1)$ is hence the variance of the termination probability of C'. The result of plotting this termination probability against the termination variance is shown in Figure 13.1. We observe that

$$\mathsf{Cov}_{[\![C']\!]_\sigma}(1,1) \;=\; \mathsf{wp}\,[\![C']\!]\,(1)(\sigma) - \mathsf{wp}\,[\![C']\!]\,(1)(\sigma)^2 \;>\; 0$$

holds iff C' terminates *neither* with probability 0 *nor* with probability 1. But since C' terminates by construction *at least* with probability $1/2$, we have here that $\mathsf{Cov}_{[\![C']\!]_\sigma}(1,1) > 0$ iff C' terminates with probability less than 1, which is the case iff C terminates with probability less than 1. Thus,

$$r(C,\sigma) \;=\; (C',\sigma,1,1,0) \;\in\; \mathcal{LCOVAR} \qquad \text{iff} \qquad (C,\sigma) \;\in\; \overline{\mathcal{AST}}$$

and therefore $r\colon \overline{\mathcal{AST}} \leq_{\mathrm{m}} \mathcal{LCOVAR}$. Q.E.D.

Next, we show that approximating upper bounds for covariances is exactly as hard as approximating lower bounds, namely Σ^0_2–complete:

THEOREM 13.6 (Hardn. of Upper Bounds on Cov. [KKM16; KKM18]): *$\mathcal{RCOVAR}$ is Σ^0_2–complete.*

Proof. Again, the proof of the membership $\mathcal{RCOVAR} \in \Sigma^0_2$ is very similar to the membership proof in Theorem 11.4 and thus omitted here. For more details, refer to [KKM18].

For proving the Σ^0_2–hardness of $\mathcal{RCOVAR}$, we reduce the Σ^0_2–complete $\overline{\mathcal{AST}}$ to $\mathcal{RCOVAR}$. Consider the reduction function $r(C,\sigma) = (C',\sigma,1,1,1/4)$, with C' being the program

$$\{\texttt{diverge}\}\,[1/2]\,\{C\}\,.$$

Analogously to the situation in the proof of Theorem 13.5, $\mathsf{Cov}_{[\![C']\!]_\sigma}(1,1)$ is defined and we have

$$\mathsf{Cov}_{[\![C']\!]_\sigma}(1,1) \;=\; \mathsf{wp}\,[\![C']\!]\,(1)(\sigma) - \mathsf{wp}\,[\![C']\!]\,(1)(\sigma)^2\,.$$

Recall that $\mathsf{wp}\,[\![C']\!]\,(1)(\sigma)$ is exactly the probability of C' terminating on input σ. By reconsidering Figure 13.1, we observe that

$$\mathsf{Cov}_{[\![C']\!]_\sigma}(1,1) \;=\; \mathsf{wp}\,[\![C']\!]\,(1)(\sigma) - \mathsf{wp}\,[\![C']\!]\,(1)(\sigma)^2 \;<\; \frac{1}{4}$$

holds iff C' does not terminate with probability $1/2$. Since by construction C' terminates with a probability of at most $1/2$, it follows that $\mathsf{Cov}_{[\![C']\!]_\sigma}(v,v) < 1/4$ holds iff C' terminates with probability less than $1/2$, which is the case iff C terminates with probability less than 1. Thus,

$$r(C,\sigma) \;=\; (C',\sigma,1,1,1/4) \;\in\; \mathcal{RCOVAR} \quad \text{iff} \quad (C,\sigma) \;\in\; \overline{\mathcal{AST}}$$

and therefore we have $r\colon \overline{\mathcal{AST}} \leq_{\mathrm{m}} \mathcal{RCOVAR}$. Q.E.D.

As a consequence of Theorems 13.5 and 13.6, computing upper bounds and computing lower bounds for covariances is equally difficult. This stands *in contrast to the case for preexpectations*: While computing upper bounds on preexpectations is Σ^0_2–complete, we have seen that computing lower bounds is „only“ Σ^0_1–complete, thus lower bounds are computably enumerable. We can even computably enumerate a sequence that *converges monotonically* to the sought–after preexpectation. By Theorems 13.5 and 13.6 this is *not possible* for a covariance as Σ^0_2–sets are in general not computably enumerable.

An approach to overcome this problem and reason about both lower and upper bounds on covariances bases on using a mixture of superinvariants and ω–subinvariants (see Section 5.1) and is outlined in [KKM16]. In a nutshell, the idea is to (A) *over*approximate $\mathsf{wp}\,[\![C]\!]\,(f \cdot g)$ using the induction rule (see Theorem 5.4), and (B) *under*approximate both $\mathsf{wp}\,[\![C]\!]\,(f)$ and $\mathsf{wp}\,[\![C]\!]\,(g)$ using ω–rules (see Theorem 5.9) in order to *over*approximate

$$\mathsf{Cov}_{[\![C]\!]_\sigma} \;=\; \mathsf{wp}\,[\![C]\!]\,(f \cdot g) - \mathsf{wp}\,[\![C]\!]\,(f) \cdot \mathsf{wp}\,[\![C]\!]\,(g)\,.$$

A dual approach works for underapproximating covariances. The approach outlined in [KKM16] also extends to reasoning about *runtime variances*, which can be useful to exclude timing side–channel attacks.

13.3 EXACT VALUES

Regarding the hardness of deciding whether a given rational is equal to the covariance, we establish that $\mathcal{COVAR} \in \Delta^0_3 = \Sigma^0_3 \cap \Pi^0_3$, $\mathcal{COVAR}$ is Π^0_2–hard, and $\mathcal{COVAR}$ is Σ^0_2–hard. $\mathcal{COVAR}$ is therefore at least as hard as deciding whether a deterministic program terminates on all inputs or deciding whether a probabilistic program terminates positively almost–surely. Let us first establish membership of $\mathcal{COVAR}$ in Δ^0_3.

LEMMA 13.7 ([KKM18]):
$\mathcal{COVAR}$ is in Δ_3^0.

Proof. To prove that $\mathcal{COVAR}$ is a member of $\Delta_3^0 = \Sigma_3^0 \cap \Pi_3^0$, consider that we have $(C, \sigma, f, g, q) \in \mathcal{COVAR}$ if and only if

$$(C, \sigma, f, g) \in \mathcal{DCOVAR} \quad \text{and}$$
$$(C, \sigma, f, g, q) \notin \mathcal{LCOVAR} \quad \text{and} \quad (C, \sigma, f, g, q) \notin \mathcal{RCOVAR}.$$

Now, by Theorem 13.3, Theorem 13.5, and Theorem 13.6 there exist *decidable* relations $\mathcal{D}$, $\mathcal{L}$, and $\mathcal{R}$ such that

$$\begin{aligned}
& (C, \sigma, f, g, q) \in \mathcal{COVAR} \\
\text{iff} \quad & \exists x_1\, \forall x_2 \colon \quad (x_1, x_2, C, \sigma, f, g) \in \mathcal{D} \\
& \quad \text{and} \quad \neg\exists y_1\, \forall y_2 \colon \quad (y_1, y_2, C, \sigma, f, g, q) \in \mathcal{L} \\
& \quad \text{and} \quad \neg\exists z_1\, \forall z_2 \colon \quad (z_1, z_2, C, \sigma, f, g, q) \in \mathcal{R} \\
\text{iff} \quad & \exists x_1\, \forall x_2 \colon \quad (x_1, x_2, C, \sigma, f, g) \in \mathcal{D} \\
& \quad \text{and} \quad \forall y_1\, \exists y_2 \colon \quad (y_1, y_2, C, \sigma, f, g, q) \notin \mathcal{L} \\
& \quad \text{and} \quad \forall z_1\, \exists z_2 \colon \quad (z_1, z_2, C, \sigma, f, g, q) \notin \mathcal{R} \\
\text{iff} \quad & \exists x_1\, \forall x_2\, \forall y_1\, \forall z_1\, \exists y_2\, \exists z_2 \colon \quad (x_1, x_2, C, \sigma, f, g) \in \mathcal{D} \qquad (\dagger) \\
& \quad \text{and} \quad (y_1, y_2, C, \sigma, f, g, q) \notin \mathcal{L} \\
& \quad \text{and} \quad (z_1, z_2, C, \sigma, f, g, q) \notin \mathcal{R} \\
\text{iff} \quad & \forall y_1\, \forall z_1\, \exists y_2\, \exists z_2\, \exists x_1\, \forall x_2 \colon \quad (x_1, x_2, C, \sigma, f, g) \in \mathcal{D} \qquad (\ddagger) \\
& \quad \text{and} \quad (y_1, y_2, C, \sigma, f, g, q) \notin \mathcal{L} \\
& \quad \text{and} \quad (z_1, z_2, C, \sigma, f, g, q) \notin \mathcal{R}
\end{aligned}$$

Note that † is a Σ_3^0–formula, whereas ‡ is a Π_3^0–formula. Hence, there exists a Σ_3^0–formula and a Π_3^0–formula, which each define $\mathcal{COVAR}$, and therefore $\mathcal{COVAR}$ is a member of $\Sigma_3^0 \cap \Pi_3^0 = \Delta_3^0$. Q.E.D.

With $\mathcal{COVAR} \in \Delta_3^0$, we have an *upper bound* on the computational hardness of $\mathcal{COVAR}$. In particular, we immediately get the following corollary:

COROLLARY 13.8:
$\mathcal{COVAR}$ is neither Σ_3^0–hard nor Π_3–hard.

As for a *lower bound* on the computational hardness of deciding $\mathcal{COVAR}$, we establish that $\mathcal{COVAR}$ is both Σ_2^0–hard and Π_2^0–hard.

LEMMA 13.9 ([KKM16; KKM18]):
$\mathcal{COVAR}$ is Π_2^0–hard.

Proof. We reduce the Π^0_2–complete almost–sure termination problem $\mathcal{AST}$ (see Theorem 12.2) to $\mathcal{COVAR}$. Let (C, σ) be an instance of $\mathcal{AST}$. Consider the reduction function $r(C, \sigma) = (C', \sigma, 1, 1, 1/4)$, with C' being the program

$$\{\texttt{diverge}\}\,[1/2]\,\{C\}\,.$$

Again, since $\mathsf{wp}\,[\![C']\!]\,(1)$ is bounded by 1, the covariance $\mathsf{Cov}_{[\![C']\!]_\sigma}(1,1)$ is defined and we have

$$\mathsf{Cov}_{[\![C']\!]_\sigma}(1,1) \;=\; \mathsf{wp}\,[\![C']\!]\,(1)(\sigma) - \mathsf{wp}\,[\![C']\!]\,(1)(\sigma)^2\,,$$

of which the plot is depicted in Figure 13.1. Recall that $\mathsf{wp}\,[\![C']\!]\,(1)(\sigma)$ is exactly the probability of C' terminating on input σ. We can see that

$$\mathsf{Cov}_{[\![C']\!]_\sigma}(1,1) \;=\; \mathsf{wp}\,[\![C']\!]\,(1)(\sigma) - \mathsf{wp}\,[\![C']\!]\,(1)(\sigma)^2 \;=\; \frac{1}{4}$$

iff C' terminates with probability $1/2$. Since C' terminates at most with probability $1/2$, we obtain that $\mathsf{Cov}_{[\![C']\!]_\sigma}(1,1) = 1/4$ iff C' terminates with probability $1/2$, which is the case iff C terminates almost–surely. Thus

$$r(C,\sigma) \;=\; \left(C', \sigma, 1, 1, \tfrac{1}{4}\right) \;\in\; \mathcal{COVAR} \qquad \text{iff} \qquad (C,\sigma) \;\in\; \mathcal{AST}\,,$$

and therefore $r\colon \mathcal{AST} \leq_{\mathrm{m}} \mathcal{COVAR}$. Q.E.D.

Lemma 13.10 ([KKM18]):
$\mathcal{COVAR}$ is Σ^0_2–hard.

Proof. We reduce the Σ^0_2–complete $\mathcal{FEXP}$ (see Theorem 11.7) to $\mathcal{COVAR}$. For that, consider the reduction function $r(C, \sigma, f) = (C, \sigma, f, 0, 0)$.

Correctness of the reduction. The covariance of f and 0 is defined if and only if the expected value of f is finite, the expected value of 0 is finite (which is trivially satisfied), and the expected value of $f \cdot 0 = 0$ is finite (which is again trivially satisfied). In case that this covariance is defined, it is equal to 0 by definition of the covariance of any random variable f and 0. Thus, the covariance is defined and its value is 0 if and only if the expected value of f after executing C on input σ is finite. Hence, $r\colon \mathcal{FEXP} \leq_{\mathrm{m}} \mathcal{COVAR}$. Q.E.D.

Our results on the computational hardness of deciding whether a given rational equals the covariance of two postexpectations after executing a probabilistic program on a given input are summarized in the following theorem:

Theorem 13.11 (Hardness of Exact Values for Covariances):

A. *$\mathcal{COVAR}$ is a member of Δ^0_3.*

B. *$\mathcal{COVAR}$ is Σ^0_2–hard.*

C. *$\mathcal{COVAR}$ is Π^0_2–hard.*

We have not classified the computational hardness of $\mathcal{COVAR}$ as precisely as we have classified the other decision problems we studied in this dissertation. By $\mathcal{COVAR} \in \Delta_3$ and $\mathcal{COVAR}$ being both Σ_2^0–hard and Π_2^0–hard, we know that $\mathcal{COVAR}$ must lie „properly“ in Δ_3^0, more precisely

$$\mathcal{COVAR} \in \Delta_3^0 \setminus \left(\Sigma_2^0 \cup \Pi_2^0\right).$$

For all other problems we studied, we established *completeness* for the respective Σ– or Π–level at which they lie in the arithmetical hierarchy. Intuitively, this means that those problems lie at the *top* of their respective Σ– or Π–level. We do not know, however, where precisely $\mathcal{COVAR}$ lies in Δ_3^0.

Part of the reason that we do not have a completeness result for $\mathcal{COVAR}$ is that by Theorem 10.10 there exist no Δ_n^0–complete sets, for $n \geq 2$, at least in the sense of many–one reducibility. There is, however, a plethora of other other reducibility notions which we did not investigate, e.g. *truth–table reducibility* or *Turing reducibility* [Rog67]. Indeed, there are e.g. Δ_n^0–complete sets in the sense of Turing reducibility, for instance the halting problem, which is Δ_n^2–complete by Post's Theorem [Pos48; Odi92], but we did not study those other notions of reducibility in this thesis.

Another approach towards locating $\mathcal{COVAR}$ within Δ_3^0 is refining the Δ_3^0–class itself. Indeed, there is for instance a hierarchy *within* the Δ_2^0–sets, called the *Ershov hierarchy* [SYY10], which can likely be relativized to the Δ_3^0–sets [Sch17]. It would be a good direction for future work to see where $\mathcal{COVAR}$ lies within such a relativized Ershov hierarchy.

13.4 VARIANCES

The variance of a random variable is a measure of how much the values that a random variable f assumes are spread out from the average value of f. It plays a central role in statistics, particle physics, and finance, to name only a few fields. Formally, the variance of a random variable f is defined as the covariance of f with itself. Recalling Definition 13.1, we obtain the following definition of a variance in the context of probabilistic programs:

Definition 13.12 (Variance [KKM16; KKM18]):
Let $C \in \mathsf{pGCL}$, $\sigma \in \Sigma_C$, and $f, g \in \mathbb{E}$. Then the variance of f after executing C on σ is given by

$$\mathsf{Var}_{[\![C]\!]_\sigma}(f) \;=\; \mathsf{wp}\,[\![C]\!]\left(f^2\right)(\sigma) - \left(\mathsf{wp}\,[\![C]\!]\,(f)(\sigma)\right)^2,$$

if $\mathsf{wp}\,[\![C]\!]\,(f)(\sigma)$ is finite; otherwise, the variance is undefined. In particular, if $\mathsf{wp}\,[\![C]\!]\left(f^2\right)(\sigma)$ is also finite, then

$$\mathsf{Var}_{[\![C]\!]_\sigma}(f) \;=\; \mathsf{Cov}_{[\![C]\!]_\sigma}(f, f).$$

As for the computational hardness of variance approximation, we can state that this problem is *not easier* than covariance approximation, i.e. the same hardness results as in Theorems 13.5 through 13.11 hold for analogous variance approximation problems. In fact, we have always reduced to approximating a variance — the variance of termination — for obtaining our hardness results on covariances. The only exception is the proof of Lemma 13.10, where two different expectations are used.

We note furthermore that variance approximation is *not harder* than covariance approximation. The proofs are analogous to the corresponding proofs for covariances presented in this section. The main difference is that we additionally have to consider the case that $\mathsf{wp}\,[\![C]\!]\,(f^2)(\sigma)$ is infinite (otherwise, we have $\mathsf{Var}_{[\![C]\!]_\sigma}(f) = \mathsf{Cov}_{[\![C]\!]_\sigma}(f, f)$).

Considering approximation of lower bounds of a variance, it suffices in this case to drop the finiteness check for $\mathsf{wp}\,[\![C]\!]\,(f^2)(\sigma)$ from $\mathcal{DCOVAR}$. This does not change the complexity, because we still have to check that the expected value of f is finite. With regard to approximating upper bounds, no change is required: If $\mathsf{wp}\,[\![C]\!]\,(f^2)(\sigma)$ is infinite, so is the variance and no constant q is an upper bound of the variance. As for exact variances, we argue as we did in Lemma 13.7.

13.5 FUTURE AND RELATED WORK

Our complexity results on the computational hardness of approximating covariances are summarized in Figure 13.2. Each of the examined problems — except for $\mathcal{COVAR}$ — is complete for their respective level of the arithmetical hierarchy. For $\mathcal{COVAR}$ we have established that it is both Σ^0_2– and Π^0_2–hard but in Δ^0_3.

An interesting issue that is raised by our results is the following: Obtaining upper bounds on covariances is computationally exactly as hard as obtaining upper bounds on preexpectations. Yet, while we do have an induction rule for proving upper bounds on preexpectations (see Theorem 5.4), we only know of a proof rule that additionally involves ω–invariants (see Theorem 5.9) for proving upper bounds on covariances [KKM16]. This adds yet another conundrum to our „upper vs. lower bounds“ theme: Computationally, it should be exactly as difficult to obtain upper bounds on preexpectations as it is on covariances. From a reasoning perspective, however, proving upper bounds on preexpectations seems conceptually rather simple, while proving upper bounds on covariances seems to be as involved as proving lower bounds on preexpectations. It would be a promising direction for future research to study the connection between the computational hardness of obtaining bounds and the conceptual intricacy of reasoning about bounds.

Another direction for future research would be to study the hardness of obtaining bounds on covariances for *mixed–sign* expectations as studied in

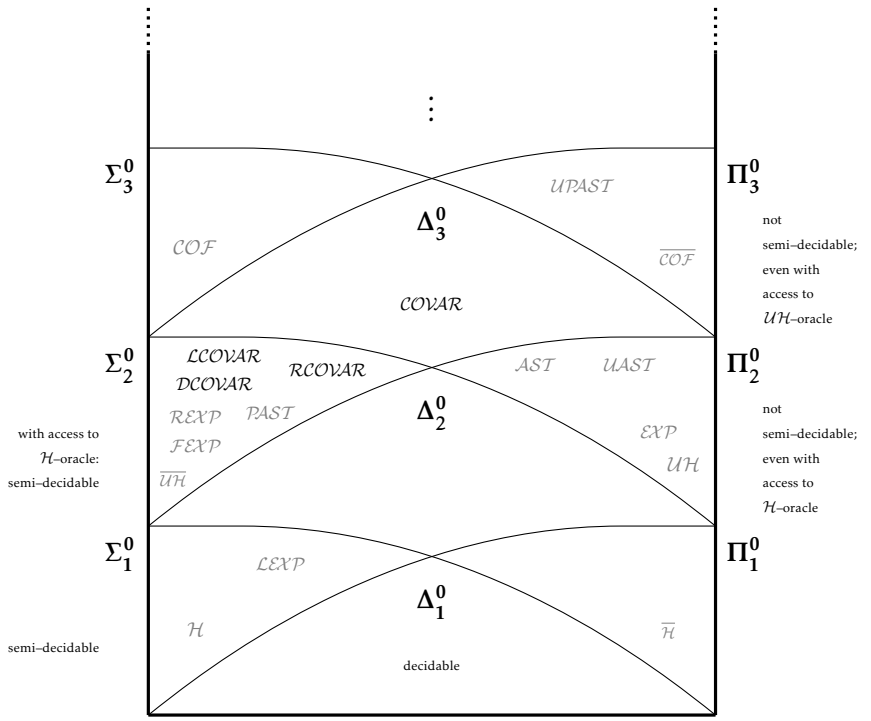


Figure 13.2: The complexity landscape of approximating covariances. All analysis problems — except for $\mathcal{COVAR}$ — are complete for the respective level at which they lie in the arithmetical hierarchy.

Chapter 9. Yet another direction for future research is to study the computational hardness of obtaining bounds on *runtime variances*. An approach for *reasoning* about runtime variances was presented in [KKM16].

CONCLUSION AND FUTURE WORK 14

For most chapters in this thesis, we have already provided an individual conclusion as well as directions for future work. We will not repeat those conclusions here. Instead, we aim to draw a bigger picture and extract the essence of some of the conclusions that share a common motif.

14.1 LOWER BOUNDS ARE HARD

In Part II of this thesis, we have presented advanced calculi tailored to specific tasks of compositional reasoning about quantitative properties of probabilistic programs on source code level. In particular, we have presented calculi for reasoning about

A. expected runtimes,

B. conditional expected values and conditional probabilities, and

C. expected values of mixed–sign random variables.

A recurring observation we made for these calculi is that reasoning about *upper bounds* on the desired quantities appears conceptually *easy* and can be carried out by simple *induction*. In more detail, the quantities of interest were expressed as *least fixed points* of some monotonic function Φ. The induction rule

$$\Phi(d) \sqsubseteq d \qquad \text{implies} \qquad \mathsf{lfp}\ \Phi \sqsubseteq d$$

provides a simple rule for proving that some d is in fact an upper bound on the least fixed point of Φ. By *simple* we mean that we have to apply Φ only *once* to d in order to assert that d is an upper bound. In particular, we *do not require infinitary means* such as, for example, finding the limit of some sequence or iterating Φ ad infinitum.

While the induction rule does not provide us with much insight on how to *find* inductive upper bounds, at least it provides us with simple means to *prove* them. Reasoning about *lower bounds*, on the other hand, is conceptually *more involved* and we do not know proof rules as simple as the induction rule above. A direction for future work is thus to discover rules for proving lower bounds which are conceptually as simple (or nearly as simple) as induction.

There is evidence that such simple proof rules for lower bounds might indeed exist: For proving lower bounds on runtimes of *deterministic* programs, Frohn *et al.* have provided a proof rule as simple as the induction rule above [Fro+16b]. For probabilistic programs, their rule fails to be sound.

For proving almost–sure termination of probabilistic programs, i.e. proving that 1 is a (non–strict) lower bound on the termination probability of a program, a reasonably simple rule is presented in Section 6.2.3. Unfortunately, it is not known whether this rule is complete. Induction for upper bounds on termination probabilitities, on the other hand, is complete.

For reasoning about lower bounds of general expected values, there exist results in probability theory on proving lower bounds on limit processes, most notably the *optional stopping theorem*. This theorem can be used to find a reasonably simple, but again incomplete, proof rule for lower bounds on probabilities, expected values and expected runtimes [Gie+19]. An unpleasant drawback, however, is that it requires almost–sure termination of the analyzed program and it can hence not be used to reason about lower bounds on termination probabilities.

14.2 LOWER BOUNDS SHOULD BE EASIER

While in Part II we investigated *techniques for actual reasoning*, we investigated in Part III of this thesis the *computational hardness of that reasoning*. Our results are summarized in Figure 14.1. Surprisingly, we can see that from a computational hardness perspective, the exact *opposite* of what we found in Part II should be expected: Lower bounds should be easier to obtain than upper bounds. For instance, approximating expected values from below ($\mathcal{LEXP}$) is computationally *strictly easier* than approximating them from above ($\mathcal{REXP}$). This constitutes a paradoxical *mismatch* between

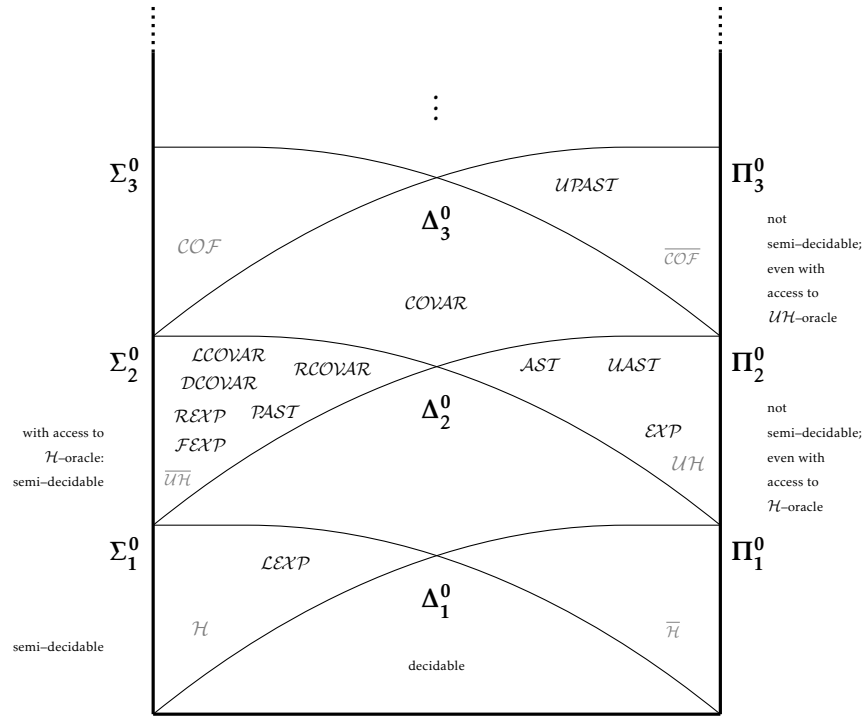
1. the *conceptual complexity of reasoning* and
2. the *arithmetical computational complexity of that reasoning*.

To the best of our knowledge, no *good* explanation for this discrepancy is known (an *unsatisfactory* explanation is provided in Section 5.2.6), but we believe that studying this discrepancy is a promising source for future insights: Ideally, we would either obtain a more fine–grained view on the computational complexity of probabilistic program analysis or we would discover new techniques for reasoning about probabilistic programs (or both).

Another conclusion we draw from our hardness considerations is that positive almost–sure termination, i.e. termination of a probabilistic program within *finite expected time* might be a more natural notion of probabilistic termination than almost–sure termination, i.e. termination with probability 1. See Section 12.3 for a more detailed discussion.

14.3 FUTURE WORK

Regarding the development of further calculi for reasoning about probabilistic programs, a direction we believe is quite promising is to endow



Problem Descriptions:

$\mathcal{LEXP}$	Lower bounds on expected values
$\mathcal{REXP}$	Upper bounds on expected values
$\mathcal{EXP}$	Exact expected values
$\mathcal{FEXP}$	Finiteness of expected values
$\mathcal{AST}$	Almost–sure termination
$\mathcal{UAST}$	Universal almost–sure termination
$\mathcal{PAST}$	Positive almost–sure termination
$\mathcal{UPAST}$	Universal positive almost–sure termination
$\mathcal{LCOVAR}$	Lower bounds on covariances
$\mathcal{RCOVAR}$	Upper bounds on covariances
$\mathcal{COVAR}$	Exact covariances
$\mathcal{DCOVAR}$	Definedness of covariances

Figure 14.1: The complexity landscape of analyzing probabilistic programs within the arithmetical hierarchy. All analysis problems — except for $\mathcal{COVAR}$ — are many–one complete for the respective level at which they lie.

the expectation–based weakest preexpectation calculus with the *local reasoning* capabilities of separation logic in order to perform reasoning about probabilistic programs with pointers. A first attempt at this is *quantitative separation logic* [Bat+19], but there remain many unanswered questions with this approach. For one, quantitative separation logic's *frame rule* allows for local reasoning on *lower bounds on partial correctness properties*. Finding a corresponding frame rule for *upper bounds* on expected values, i.e. (quantitative) total correctness properties, poses a problem. Another issue is to devise a calculus for *local reasoning about expected runtimes*, i.e. marrying quantitative separation logic with the ert calculus. In this context, an „upper bound frame rule" would be very useful since it would allow for local reasoning about upper bounds on expected runtimes.

Another direction for future work is *relational reasoning*: Given two programs and a relation R on initial states, one would like to prove relational properties about the final states. Relational reasoning about probabilistic programs using *couplings* has been studied [Bar+12; Bar+15; Hsu17; Agu+18]. This style of reasoning can be used to reason about the *probability* that some relation over final states holds. It would be interesting, however, to study how *relational weakest preexpectations* could enable reasoning over expected differences of the outputs of two programs or the covariance of two programs.

Further directions for future work emerge from combining the calculi we have presented in this thesis. For instance, could we combine the ert calculus for expected runtimes with the cwp calculus for conditional expected values to obtain a calculus for *conditional expected runtimes*? Or could we combine the ert calculus with the iwp calculus for mixed–sign expectations to reason about *amortized expected runtimes?*

Part IV

APPENDICES

DOMAIN THEORY

A

Domain theory is a field created in 1969 by Dana Steward Scott [Sco69]. In this thesis, we make heavy use of the basic concepts underlying this theory, although we will really just make use of the part that Samson Abramsky and Achim Jung call a „first step towards Domain Theory". For a comprehensive treatment, we refer to their introduction in the *Handbook of Logic in Computer Science* [AJ94]. We will here only briefly recall here some concepts, so that this thesis is somewhat self–contained.

The first concept we recall is that of a complete lattice: a set equipped with an order in which ascending and descending sequences in some sense converge. Indeed, Abramsky and Jung speak of „convergence spaces".[1]

Definition A.1 (Complete Lattices [AJ94]):

a. *Let D be some universe. Then $(D, \sqsubseteq)$, where $\sqsubseteq$ is a binary relation $\sqsubseteq \subset D \times D$, is a partial order, iff $\sqsubseteq$ is*

a) reflexive, i.e. for all $a \in D$

$$a \sqsubseteq a\,,$$

b) transitive, i.e. for all $a, b, c \in D$

$$a \sqsubseteq b \quad \text{and} \quad b \sqsubseteq c \qquad \text{implies} \qquad a \sqsubseteq c\,,$$

c) and antisymmetric, i.e. for all $a, b \in D$

$$a \sqsubseteq b \quad \text{and} \quad b \sqsubseteq a \qquad \text{implies} \qquad a = b\,.$$

Whenever the universe D is evident from the context, we may omit the D from $(D, \sqsubseteq)$ and simply speak of the partial order $\sqsubseteq$.

b. *A partial order $(D, \sqsubseteq)$ is called a complete lattice, if every subset $S \subseteq D$ has a supremum $\sup S \in D$.*

Note that every complete lattice $(D, \sqsubseteq)$ has a least element $\bot$ and dually a greatest element $\top$ which satisfy

$$\forall a \in D: \qquad \bot \sqsubseteq a \sqsubseteq \top\,.$$

1 Strictly speaking, Abramsky and Jung speak of *directed–complete partial orders* as convergence spaces, which make up a slightly more general notion. We, however, do not require this level of generality for our theory. In any case, any complete lattice *is* a directed complete partial order.

Note furthermore that every ascending or descending sequence forms a subset and we can thus ensure that ascending and descending sequences converge within a complete lattice in the sense that they will have a least upper or greatest lower bound, respectively.

The sets on which the transformers we have presented in this thesis act are equipped with some partial order and (mostly) form complete lattices. This is important for us, because it ultimately ensures that certain functions admit least fixed points. The class of those functions is the class of so called *Scott–continuous* or just *continuous* functions.

Definition A.2 (Continuity [AJ94]):
Let $(D, \sqsubseteq)$ be a complete lattice and let $\Phi\colon D \to D$. Then Φ is called continuous, iff for every chain $S = \{s_0 \sqsubseteq s_1 \sqsubseteq s_2 \sqsubseteq \ldots\} \subseteq D$

$$\Phi(\sup S) \;=\; \sup\, \Phi(S)\,,$$

where $\Phi(S)$ is the standard shorthand for the set $\{\Phi(a) \mid a \in S\}$.

The notion of continuity we choose here is what Abramsky and Jung call ω–*continuity* because it requires the functions to preserve suprema of chains of length ω. The more general notion of continuity is based on *directed sets*. We, choose ω–continuity because we find it more demonstrative, but we could have build our theory on the more general notion of continuity as well.

A notion, that is uncontestedly equal in both flavors of the theory is the notion of monotonicity of functions:

Definition A.3 (Monotonicity [AJ94]):
Let $(D, \sqsubseteq)$ be a complete lattice and let $\Phi\colon D \to D$. Then Φ is called monotonic, iff for all $a, b \in D$

$$a \sqsubseteq b \quad \text{implies} \quad \Phi(a) \sqsubseteq \Phi(b)\,.$$

General continuity, i.e. continuity with respect to directed sets, requires that continuous functions are *by definition* monotonic. For the notion of continuity we use, i.e. continuity with respect to ω–chains, however, monotonicity follows from continuity.

Theorem A.4:
Every continuous function is monotonic.

Proof. Let $a \sqsubseteq b$. Then $\{a \sqsubseteq b\}$ is a chain with

$$\sup\{a, b\} \;=\; b\,, \tag{†}$$

because b is greater than or equal to a.

Now, let Φ be continuous. Then

$$\begin{aligned}
\Phi(a) &\sqsubseteq \sup\left\{\Phi(a), \Phi(b)\right\} \\
&= \Phi\Big(\sup\{a, b\}\Big) && \text{(by continuity of } \Phi\text{)} \\
&= \Phi(b) && \text{(by } \dagger\text{)}
\end{aligned}$$

and hence Φ is monotonic. Q.E.D.

We now have all notions readily available to formulate the two fixed point theorems of which we make heavy use in this thesis. The origins of the fixed point theorems date back at least to 1928, namely to Bronisław Knaster, who proved a closely related theorem in set theory [Kna28; LNS82]. The version we use here is often attributed to Kleene, though the origins are somewhat unclear. It it thus considered a folk theorem [LNS82].

Theorem A.5 (Kleene Fixed Point Theorem [AJ94; LNS82]):
Let $(D, \sqsubseteq)$ be a complete lattice with least element $\bot$ and greatest element $\top$. Moreover, let $\Phi\colon D \to D$ be continuous (thus monotonic).

Then Φ has a least fixed point $\mathsf{lfp}\,\Phi$ and a greatest fixed point $\mathsf{gfp}\,\Phi$, respectively given by

$$\mathsf{lfp}\,\Phi = \sup_{n\in\mathbb{N}} \Phi^n(\bot) \qquad \text{and} \qquad \mathsf{gfp}\,\Phi = \inf_{n\in\mathbb{N}} \Phi^n(\top)\,.$$

We will use the Kleene Fixed Point Theorem to establish existence of least fixed point, which we use heavily in our definitions of transformers for while loops. The theorem due to Knaster's [Kna28] is actually more similar to a principle that is today called *Park's Lemma*, *Park's Theorem*, or *Park induction*, attributed to David Park.

Lemma A.6 (Park's Lemma [Par69]):
Let $(D, \sqsubseteq)$ be a complete lattice, let $d \in D$, and let $\Phi\colon D \to D$ be a monotonic function. Then

$$\Phi(d) \sqsubseteq d \quad \text{implies} \quad \mathsf{lfp}\,\Phi \sqsubseteq d$$

and dually

$$d \sqsubseteq \Phi(d) \quad \text{implies} \quad d \sqsubseteq \mathsf{gfp}\,\Phi\,.$$

The correctness of most of the proof rules for loops presented in this thesis are a direct consequence of Park's Lemma. This concludes our overview of domain–theoretical notions needed for this thesis.

MARKOV DECISION PROCESSES

B

Giving semantics to probabilistic programs can be done by means of *Markov decision processes*, see Section 3.3.3 or [GKM12; GKM14] for more details. We recap here very briefly basics on Markov decision processes. For an in–depth treatment, see [Put05]; for a more gentle introduction with an emphasis on verification, see [BK08, Chapter 10].

Markov decision processes are extensions of Markov chains by *nondeterminism*. We go here the other way: we first define Markov decision processes and define Markov chains as special cases.

Definition B.1 (Markov Decision Processes):
A Markov decision process (MDP) $M = (S, \iota, A, P)$ comprises of

- *a countable set of states S,*
- *an initial state $\iota \in S$,*
- *a set of actions A, and*
- *a transition probability function $P \colon S \times A \to S \to [0,1]$ mapping each state–action pair to a probability distribution over successor states, thus obeying $\forall \alpha, s \colon \ \sum_{s'} P(s,\alpha)(s') = 1$.*

We are mostly interested in reachability probabilities. An MDP, however, does not necessarily give rise to a unique probability measure. In fact, in general it will not. The reason is the inherent nondeterminism in the MDP which gives rise to many probability measures. In order to resolve the nondeterminism, we introduce *schedulers*.

Definition B.2 (Schedulers):
Let $M = (S, \iota, A, P)$ be an MDP.

A. *A scheduler $\mathfrak{s}$ of M is a function*

$$\mathfrak{s} \colon S^* \to A$$

mapping a sequence of states of M to an action.

B. *A scheduler $\mathfrak{s}$ is called positional or history–independent, iff for all prefixes of state sequences $\rho \in S^*$ and all states $s \in S$,*

$$\mathfrak{s}(\rho s) \ = \ \mathfrak{s}(s)\,,$$

i.e. the action chosen by the scheduler depends solely on the current state and not on the history of visited states.

Once a scheduler resolves the nondeterminism in a Markov decision process, there are no nondeterministic decisions left to be made. All decisions that are left are due to randomness. Such Markov decision processes are called *Markov chains*. Any given scheduler thus induces from a Markov decision process a Markov chain.

DEFINITION B.3 (Markov Chains Induced by Schedulers):

A. *A Markov chain (MC) $M = (S, \iota, P)$ comprises of*
   - *a countable set of states S,*
   - *an initial state $\iota \in S$,*
   - *a transition probability matrix $P\colon S \to S \to [0,1]$ mapping each state to a probability distribution over successor states, thus obeying $\forall s\colon \sum_{s'} P(s)(s') = 1$.*

B. *Let $M = (S, \iota, A, P')$ be an MDP and let $\mathfrak{s}$ be a scheduler of M. Then the Markov chain of M induced by scheduler $\mathfrak{s}$, denoted $M_{\mathfrak{s}}$, is given by (S, ι, P), where for all $s, s' \in S$*

$$P(s)(s') \;=\; P'\Big(s, \mathfrak{s}(s)\Big)(s')\,.$$

In MC's, we can now sensibly define unique reachability probabilities.

DEFINITION B.4 (Reachability Probabilities):
Let $M = (S, \iota, P)$ be a Markov chain and let $B \subseteq S$ be a set of states of interest. Then the probability of eventually reaching B, denoted $\Pr_M(\Diamond B)$, is given by

$$\Pr_M(\Diamond B) \;=\; \sum_{\substack{s_0 \cdots s_n \in (S \setminus B)^* B, \\ s_0 = \iota}} \prod_{i=0}^{n-1} P(s_i)(s_{i+1})$$

Another concept we will use are *expected rewards*. Intuitively, each state in an MC is associated with a non–negative real–valued reward which is collected upon visiting the state. The expected reward is the expected accumulated reward that is collected along the paths through a Markov chain, weighted by the probabilities associated with the according path.

DEFINITION B.5 (Expected Rewards):
Let $M = (S, \iota, P)$ be a Markov chain.

A. *A reward function is a function*

$$rew\colon S \to \mathbb{R}^{\infty}_{\geq 0}$$

B. *Let $B \subseteq S$ be a set of states of interest. The expected reward of eventually reaching B, denoted $\mathsf{ExpRew}_M(\Diamond B)$ is given by*

$$\mathsf{ExpRew}_M(\Diamond B) = \sum_{\substack{s_0 \cdots s_n \in (S \setminus B)^* B, \\ s_0 = \iota}} \prod_{i=0}^{n-1} \left(P(s_i)(s_{i+1}) \cdot rew(s_i) \right)$$

C OMITTED CALCULATIONS

C.1 DETAILED FIXED POINT ITERATION FOR EXAMPLE 2.6

The first three iterations of the fixed point iteration for Φ in Example 2.6 are:

$$
\begin{aligned}
&\Phi(\mathsf{false}) \\
&= (x = 0) \lor (x > 0 \land \mathsf{wp}\, [\![z := z + 1 \fatsemi x := x - 1]\!]\, (\mathsf{false})) \\
&= (x = 0) \lor (x > 0 \land \mathsf{wp}\, [\![z := z + 1]\!]\, (\mathsf{wp}\, [\![x := x - 1]\!]\, (\mathsf{false}))) \\
&= (x = 0) \lor (x > 0 \land \mathsf{wp}\, [\![z := z + 1]\!]\, (\mathsf{false})) \\
&= (x = 0) \lor (x > 0 \land \mathsf{false}) \\
&= (x = 0) \lor \mathsf{false} \\
&= (x = 0)
\end{aligned}
$$

$$
\begin{aligned}
&\Phi^2(\mathsf{false}) \\
&= (x = 0) \lor (x > 0 \land \mathsf{wp}\, [\![z := z + 1 \fatsemi x := x - 1]\!]\, (x = 0)) \\
&= (x = 0) \lor (x > 0 \land \mathsf{wp}\, [\![z := z + 1]\!]\, (\mathsf{wp}\, [\![x := x - 1]\!]\, (x = 0))) \\
&= (x = 0) \lor (x > 0 \land \mathsf{wp}\, [\![z := z + 1]\!]\, (x - 1 = 0)) \\
&= (x = 0) \lor (x > 0 \land \mathsf{wp}\, [\![z := z + 1]\!]\, (x = 1)) \\
&= (x = 0) \lor (x > 0 \land x = 1) \\
&= (x = 0) \lor (x = 1)
\end{aligned}
$$

$$
\begin{aligned}
&\Phi^3(\mathsf{false}) \\
&= (x = 0) \lor (x > 0 \land \mathsf{wp}\, [\![z := z + 1 \fatsemi x := x - 1]\!]\, (x = 0 \lor x = 1)) \\
&= (x = 0) \\
&\qquad \lor (x > 0 \land \mathsf{wp}\, [\![z := z + 1]\!]\, (\mathsf{wp}\, [\![x := x - 1]\!]\, (x = 0 \lor x = 1))) \\
&= (x = 0) \lor (x > 0 \land \mathsf{wp}\, [\![z := z + 1]\!]\, (x - 1 = 0 \lor x - 1 = 1)) \\
&= (x = 0) \lor (x > 0 \land \mathsf{wp}\, [\![z := z + 1]\!]\, (x = 1 \lor x = 2)) \\
&= (x = 0) \lor (x > 0 \land (x = 1 \lor x = 2)) \\
&= (x = 0) \lor (x > 0 \land x = 1) \lor (x > 0 \land x = 2) \\
&= (x = 0) \lor (x = 1) \lor (x = 2)
\end{aligned}
$$

C.2 DETAILED FIXED POINT ITERATION FOR EXAMPLE 2.11 B

The first three iterations of the fixed point iteration for Φ in Example 2.11 B. are:

$$
\begin{aligned}
\Phi(0) &= [x \leq 0] \cdot z \;+\; [0 < x] \cdot \mathsf{wp}\, [\![z := z+1 \,;\, x := x-1]\!]\, (0) \\
&= [x \leq 0] \cdot z \;+\; [0 < x] \cdot \mathsf{wp}\, [\![z := z+1]\!]\, (\mathsf{wp}\, [\![x := x-1]\!]\, (0)) \\
&= [x \leq 0] \cdot z \;+\; [0 < x] \cdot \mathsf{wp}\, [\![z := z+1]\!]\, (0) \\
&= [x \leq 0] \cdot z \;+\; [0 < x] \cdot 0 \\
&= [x \leq 0] \cdot z \qquad (\dagger) \\
&= [x \leq 0] \cdot z \;+\; \underbrace{\underbrace{[0 < x \leq 0]}_{=0} \cdot \lceil x \rceil}_{=0}
\end{aligned}
$$

$$
\begin{aligned}
\Phi^2(0) &= [x \leq 0] \cdot z \;+\; [0 < x] \cdot \mathsf{wp}\, [\![z := z+1 \,;\, x := x-1]\!]\, (\Phi(0)) \\
&= [x \leq 0] \cdot z \;+\; [0 < x] \cdot \mathsf{wp}\, [\![z := z+1]\!]\, (\mathsf{wp}\, [\![x := x-1]\!]\, (\Phi(0))) \\
&= [x \leq 0] \cdot z \;+\; [0 < x] \cdot \mathsf{wp}\, [\![z := z+1]\!]\, ([x-1 \leq 0] \cdot z) \qquad \text{(see } \dagger\text{)} \\
&= [x \leq 0] \cdot z \;+\; [0 < x] \cdot \mathsf{wp}\, [\![z := z+1]\!]\, ([x \leq 1] \cdot z) \\
&= [x \leq 0] \cdot z \;+\; [0 < x] \cdot [x \leq 1] \cdot (z+1) \\
&= [x \leq 0] \cdot z \;+\; [0 < x \leq 1] \cdot (z+1) \\
&= [x \leq 1] \cdot z \;+\; [0 < x \leq 1] \cdot \lceil x \rceil
\end{aligned}
$$

$$
\begin{aligned}
\Phi^3(0) &= [x \leq 0] \cdot z \;+\; [0 < x] \cdot \mathsf{wp}\, [\![z := z+1 \,;\, x := x-1]\!]\, \left(\Phi^2(0)\right) \\
&= [x \leq 0] \cdot z \\
&\qquad +\; [0 < x] \cdot \mathsf{wp}\, [\![z := z+1]\!]\, \left(\mathsf{wp}\, [\![x := x-1]\!]\, \left(\Phi^2(0)\right)\right) \\
&= [x \leq 0] \cdot z \;+\; [0 < x] \\
&\qquad \cdot \mathsf{wp}\, [\![z := z+1]\!]\, ([x-1 \leq 1] \cdot z \;+\; [0 < x-1 \leq 1] \cdot \lceil x-1 \rceil) \\
&= [x \leq 0] \cdot z \;+\; [0 < x] \\
&\qquad \cdot \mathsf{wp}\, [\![z := z+1]\!]\, ([x \leq 2] \cdot z \;+\; [1 < x \leq 2] \cdot (\lceil x \rceil - 1)) \\
&= [x \leq 0] \cdot z \;+\; [0 < x] \cdot ([x \leq 2] \cdot (z+1) \;+\; [1 < x \leq 2] \cdot (\lceil x \rceil - 1)) \\
&= [x \leq 0] \cdot z \;+\; [0 < x \leq 2] \cdot (z+1) \;+\; [1 < x \leq 2] \cdot (\lceil x \rceil - 1) \\
&= [x \leq 0] \cdot z \;+\; [0 < x \leq 1] \cdot (z+1) \\
&\qquad +\; [1 < x \leq 2] \cdot (z+1+\lceil x \rceil - 1) \\
&= [x \leq 2] \cdot z \;+\; [0 < x \leq 1] \cdot \lceil x \rceil \;+\; [1 < x \leq 2] \cdot \lceil x \rceil \\
&= [x \leq 2] \cdot z \;+\; [0 < x \leq 2] \cdot \lceil x \rceil
\end{aligned}
$$

A MORE DETAILED NOTE ON CONTRIBUTIONS OF THE AUTHOR

D

Es gibt immer die juristische Seite
und die Seite des gesunden Menschenverstandes.
Wir sind nun in dem langwierigen und schwierigen Prozess,
den gesunden Menschenverstand durch juristische Regelungen zu ersetzen.

— A faculty member

This note is mandatory under doctoral regulations of the RWTH Aachen University Faculty of Mathematics, Computer Science and Natural Sciences. Below, I give in reverse chronological order a complete list of peer–reviewed publications I coauthored that emerged from the research done in the course of writing this thesis. I then (have to) discuss in detail my own contributions to these publications.

[Kam+18] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. „Weakest Precondition Reasoning for Expected Runtimes of Randomized Algorithms." In: *Journal of the ACM* 65.5 (2018), 30:1–30:68

[Bat+18a] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. „A Program Analysis Perspective on Expected Sampling Times." In: *Extended Abstracts of the International Conference on Probabilistic Programming (PROBPROG)*. 2018

[KKM18] Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. „On the Hardness of Analyzing Probabilistic Programs." In: *Acta Informatica* (2018)

[Bat+18b] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. „How long, O Bayesian network, will I sample thee? A program analysis perspective on expected sampling times." In: *Proc. of the European Symposium on Programming Languages and Systems (ESOP)*. vol. 10801. Lecture Notes in Computer Science. Springer, 2018, pp. 186–213

[McI+18] Annabelle McIver, Carroll Morgan, Benjamin Lucien Kaminski, and Joost-Pieter Katoen. „A New Proof Rule for Almost–sure Termination." In: *Proc. of the Symposium on Principles of Programming Languages (POPL)* 2.POPL (2018), 33:1–33:28

[Olm+18] Federico Olmedo, Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Annabelle McIver. „Conditioning in Probabilistic Programming." In: *Trans. on Programming Languages and Systems* 40.1 (2018), 4:1–4:50

[KK17b] Benjamin Lucien Kaminski and Joost-Pieter Katoen. „A Weakest Pre–expectation Semantics for Mixed–sign Expectations." In: *Proc. of the Annual Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, 2017, pp. 1–12

[KK17a] Benjamin Lucien Kaminski and Joost-Pieter Katoen. „A Weakest Pre–Expectation Semantics for Mixed–Sign Expectations." In: *Extended Abstracts of the 2nd Workshop on Probabilistic Programming Semantics (PPS)*. 2017

[KKM16] Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. „Inferring Covariances for Probabilistic Programs." In: *Proc. of the International Conference on Quantitative Evaluation of Systems (QEST)*. vol. 9826. Lecture Notes in Computer Science. Springer, 2016, pp. 191–206

[Kam+16] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. „Weakest Precondition Reasoning for Expected Run–Times of Probabilistic Programs." In: *Proc. of the European Symposium on Programming Languages and Systems (ESOP)*. vol. 9632. Lecture Notes in Computer Science. Springer, 2016, pp. 364–389

[Gre+16] Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Federico Olmedo. „On the Semantic Intricacies of Conditioning." In: *Extended Abstracts of the 1st Workshop on Probabilistic Programming Semantics (PPS)* (2016)

[Kat+15] Joost-Pieter Katoen, Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, and Federico Olmedo. „Understanding Probabilistic Programs." In: *Correct System Design — Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday*. Vol. 9360. Lecture Notes in Computer Science. Springer, 2015, pp. 15–32

[KK15b] Benjamin Lucien Kaminski and Joost-Pieter Katoen. „On the Hardness of Almost–Sure Termination." In: *Proc. of the International Symposium on Mathematical Foundations of Computer Science (MFCS)*. vol. 9234. Lecture Notes in Computer Science. Springer, 2015, pp. 307–318

[Jan+15a] Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Federico Olmedo, Friedrich Gretz, and Annabelle McIver. „Conditioning in Probabilistic Programming." In: *Proc. of*

the Conference on Mathematical Foundations of Programming Semantics (MFPS) 319 (2015), pp. 199–216

[KK15a] Benjamin Lucien Kaminski and Joost-Pieter Katoen. „Analyzing Expected Outcomes and (Positive) Almost–sure Termination of Probabilistic Programs is Hard.“ In: *Proc. of the Young Researchers' Conference „Frontiers of Formal Methods“ (FFM)*. vol. 9234. Aachener Informatik Berichte. 2015, pp. 179–184

The doctoral regulations of the RWTH Aachen University Faculty of Mathematics, Computer Science and Natural Sciences require that I state explicitly what contributions I made to the individual publications. I have had many discussions with colleagues and representatives of the faculty about the sense and nonsense of this regulation. At the end of the day, I strongly believe that this regulation does not harmonize at all with the everyday reality of research in theoretical computer science, at least not in my case: In general, lots of ideas emerge naturally from discussions and one would more often than not be wrong to want to attribute a specific idea to a specific person. I would also like to note that for some of these publications I am at this point in time supposed to state explicitly what my contributions were some *four years after the research was conducted*. I made the following statements to the best of my knowledge and belief and tried my best not to do injustice to anyone in stating my own contributions to the publications listed above. In order to give any additional value whatsoever to this note, I will also describe briefly how we (or I) came up with the ideas for conducting the research covered in the respective publications. I proceed in what is roughly chronological order.

The technical contributions of the hardness results for approximating expected values and deciding probabilistic termination covered in [KK15a] and [KK15b] are exclusively due to me, although I did occasionally discuss minor details with others; I also came up with the ideas and the initiative for conducting this particular research myself (inspired by a fantastic lecture on Recursion Theory given by Wolfgang Thomas) and the publications were for the most part authored by me. I would, however, like to take this opportunity to thank my supervisor Joost-Pieter Katoen once again for his extremely thorough proofreading efforts and his invaluable feedback (this goes not only for the just–mentioned publications but also for the ones I cover below). In addition, Joost-Pieter often authored major parts of the introductions of the papers or at least helped greatly to bring them into shape. He also helped me by suggesting appropriate venues for publication. Without his assistance, I would certainly have had a much harder time getting these papers published.

The initiative for conducting the research covered in [Jan+15a], [Kat+15], [Gre+16], and [Olm+18] (the conditional weakest preexpectation calculus) came from Joost-Pieter Katoen. The cwp and cwlp calculus itself (see Sec-

tion 8.2) was jointly developed in the course of numerous discussions by Nils Jansen, Friedrich Gretz, Federico Olmedo, and myself. The issues that occur when adding nondeterminism to the language (see Section 8.4) were mainly investigated by me. The invariant–based reasoning for cwp and cwlp, at least the non–ω–rules were also suggested by me. If I remember correctly, I also came up with the small examples given in [Kat+15] showing the intricacies that arise when bringing together conditioning and loops, but I am almost certain, that we had discussions about those as well, even though I honestly do not remember anymore. I contributed substantially to the writing of all four papers.

The idea for conducting the research covered in [Kam+16] and [Kam+18] (the expected runtime calculus) emerged from an idea by Nils Jansen and Joost-Pieter Katoen to investigate techniques for reasoning about approximate computing. Federico Olmedo and I soon came to the conclusion that for that we would need a calculus for reasoning about expected resource consumption and got Christoph Matheja on board. The expected runtime calculus itself was jointly developed in the course of numerous discussions by Christoph, Federico, and myself. It would seem to me that the simple idea of adding a $1 + \_\_$ to the weakest preexpectation calculus (compare Table 4.1 on p. 84 with Table 7.1 on p. 163) is due to me, but I specifically remember that this idea first came to me in the middle of a discussion we had. As such, I feel it is absolutely impossible to attribute the calculus to any single person. The connection of the calculus to an operational model was established by Christoph and me. The invariant–based proof rules and the bound refinement were developed by me. The Coupon Collector case study was suggested to us by Gilles Barthe and then conducted by Christoph, Federico, and me. The connection to the Nielsen logic was established by Christoph; the idea to investigate this connection is due to Christoph and Federico, inspired by a lecture of Thomas Noll. I contributed substantially to the writing of the two papers.

The initiative for conducting the research covered in [KKM16] (hardness of approximating covariances) came from me. I got Christoph Matheja on board and we conducted much of the research together. My original idea was to develop a weakest–preexpectation–style calculus for reasoning about variances, but that did not work out as neatly as hoped for and, as such, only the hardness aspects are covered in this thesis. The reductions covered in this thesis and the invariant–based reasoning techniques are due to me. We were nudged towards the definedness issues about the covariances by an anonymous referee of [KKM18] to whom I would once again like to express my deepest gratitude for their thorough and extremely constructive feedback. Christoph and I resolved the issues together. The part about runtime variances is due to Christoph. [KKM18] summarizes all hardness–related contributions in [KK15a], [KK15b], and [KKM16]. I contributed substantially to the writing of the papers.

The idea of extending the weakest preexpectation calculus to mixed–sign random variables covered in [KK17a] and [KK17b] is due to me and emerged from trying to extend the expected runtime calculus developed in [Kam+16] to a calculus for *amortized* expected runtimes. I authored the papers myself for the most part. I would like, however, to repeat what I said above about Joost-Pieter Katoen's invaluable help.

The main idea for the new proof rule for almost–sure termination covered in [McI+18] is solely due to Annabelle McIver & Carroll Morgan, see [MM16] for their early draft. I discussed their draft with Annabelle at a workshop organized by Prakash Panangaden at the Bellairs Research Institute and suggested to formalize their proof rule in terms of the weakest preexpectation calculus. My contribution was thus the formalization of the rule and the case studies as well as providing a formal soundness proof. I participated in intensive discussions and also contributed to writing the paper.

The goal of the research covered by [Bat+18b] and [Bat+18a] was to make substantial progress towards automating the Coupon Collector case study we had studied earlier in [Kam+16]. The initiative came from Christoph Matheja and me and we were lucky having Kevin Batz do his Bachelor's thesis on this topic. Kevin, Christoph, and I regularly had intensive and fruitful discussions, but the bulk of the work was done by Kevin. To study a connection with Bayesian networks was suggested by Christoph, but, once again, it was Kevin who did most of the work, both with regard to theory as well as to implementation. The papers that emerged were authored by Christoph and myself for the most part.

BIBLIOGRAPHY

[AJ94] Samson Abramsky and Achim Jung. „Domain Theory.“ In: *Handbook of Logic in Computer Science*. Ed. by Samson Abramsky, Dov M. Gabbay, and Thomas Stephen Edward Maibaum. Vol. 3. *Corrected and expanded version* available at http://www.cs.bham.ac.uk/~axj/pub/papers/handy1.pdf. Oxford University Press, 1994, pp. 1–168.

[AFR11] Nathanael Leedom Ackerman, Cameron E. Freer, and Daniel M. Roy. „Noncomputable Conditional Distributions.“ In: *Proc. of the Annual Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, 2011, pp. 107–116.

[AH92] Leonard Max Adleman and Ming-Deh Huang. „Primality Testing and Abelian Varieties over Finite Fields.“ In: *Lecture Notes in Mathematics* 1512 (1992).

[ACN18] Sheshansh Agrawal, Krishnendu Chatterjee, and Petr Novotný. „Lexicographic Ranking Supermartingales: An Efficient Approach to Termination of Probabilistic Programs.“ In: *Proc. of the Symposium on Principles of Programming Languages (POPL)* 2.POPL (2018), 34:1–34:32.

[Agu+18] Alejandro Aguirre, Gilles Barthe, Lars Birkedal, Ales Bizjak, Marco Gaboardi, and Deepak Garg. „Relational Reasoning for Markov Chains in a Probabilistic Guarded Lambda Calculus.“ In: *Proc. of the European Symposium on Programming Languages and Systems (ESOP)*. Vol. 10801. Lecture Notes in Computer Science. Springer, 2018, pp. 214–241.

[AK02] Susanne Albers and Marek Karpinski. „Randomized Splay Trees: Theoretical and Experimental Results.“ In: *Information Processing Letters* 81.4 (2002), pp. 213–221.

[AR08] Miguel E. Andrés and Peter van Rossum. „Conditional Probabilities over Probabilistic and Nondeterministic Systems.“ In: *Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 157–172.

[APZ03] Tamarah Arons, Amir Pnueli, and Lenore D. Zuck. „Parameterized Verification by Probabilistic Abstraction.“ In: *Proc. of the International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*. Vol. 2620. Lecture Notes in Computer Science. Springer, 2003, pp. 87–102.

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity — A Modern Approach*. Cambridge University Press, 2009.

[Art+09] Rob Arthan, Ursula Martin, Erik Arne Mathiesen, and Paulo Oliva. „A General Framework for Sound and Complete Floyd–Hoare Logics." In: *Trans. on Computational Logic* 11.1 (2009).

[ADD00] Robert B. Ash and Catherine Doleans-Dade. *Probability and Measure Theory*. Academic Press, 2000.

[APM09] Philippe Audebaud and Christine Paulin-Mohring. „Proofs of randomized algorithms in Coq." In: *Science of Computer Programming* 74.8 (2009), pp. 568–589.

[BW89] Ralph-Johan Back and Joakim von Wright. „Refinement Calculus, Part I: Sequential Nondeterministic Programs." In: *Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness, REX Workshop*. Vol. 430. Lecture Notes in Computer Science. Springer, 1989, pp. 42–66.

[BW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus — A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[Bai+14] Christel Baier, Joachim Klein, Sascha Klüppelholz, and Steffen Märcker. „Computing Conditional Probabilities in Markovian Models Efficiently." In: *Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Vol. 8413. Lecture Notes in Computer Science. Springer, 2014, pp. 515–530.

[Bar+12] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. „Probabilistic Relational Reasoning for Differential Privacy." In: *Proc. of the Symposium on Principles of Programming Languages (POPL)*. ACM, 2012, pp. 97–110.

[Bar+15] Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, Léo Stefanesco, and Pierre-Yves Strub. „Relational Reasoning via Probabilistic Coupling." In: *Proc. of the International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. Vol. 9450. Lecture Notes in Computer Science. Springer, 2015, pp. 387–401.

[Bat+18a] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. „A Program Analysis Perspective on Expected Sampling Times." In: *Extended Abstracts of the International Conference on Probabilistic Programming (PROBPROG)*. 2018.

[Bat+18b] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. „How long, O Bayesian network, will I sample thee? A program analysis perspective on expected sampling times." In: *Proc. of the European Symposium on Programming Languages and Systems (ESOP)*. Vol. 10801. Lecture Notes in Computer Science. Springer, 2018, pp. 186–213.

[Bat+19] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. „Quantitative Separation Logic — A Logic for Reasoning about Probabilistic Programs." In: *Proc. of the Symposium on Principles of Programming Languages (POPL)*. [to appear]. ACM, 2019.

[Ben83] Michael Ben-Or. „Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols (Extended Abstract)." In: *Proc. of the Annual Symposium on Principles of Distributed Computing (PODC)*. ACM, 1983, pp. 27–30.

[BM04] Rudolf Berghammer and Markus Müller-Olm. „Formal Development and Verification of Approximation Algorithms Using Auxiliary Variables." In: *Proc. of the International Symposium on Logic–Based Program Synthesis and Transformation (LOPSTR)*. Vol. 3018. Lecture Notes in Computer Science. Springer, 2004, pp. 59–74.

[BGV18] Benjamin Bichsel, Timon Gehr, and Martin T. Vechev. „Fine–grained Semantics for Probabilistic Programs." In: *Proc. of the European Symposium on Programming Languages and Systems (ESOP)*. Vol. 10801. Lecture Notes in Computer Science. Springer, 2018, pp. 145–185.

[Bor09] Félix Édouard Justin Émile Borel. „Les probabilités dénombrables et leurs applications arithmétiques." In: *Rendiconti del Circolo Matematico di Palermo* 27.2 (1909), pp. 247–271.

[Bor+11] Johannes Borgström, Andrew D. Gordon, Michael Greenberg, James Margetson, and Jurgen Van Gael. „Measure Transformer Semantics for Bayesian Machine Learning." In: *Proc. of the European Symposium on Programming Languages and Systems (ESOP)*. Vol. 6602. Lecture Notes in Computer Science. Springer, 2011, pp. 77–96.

[Bor+16] Johannes Borgström, Andrew D. Gordon, Long Ouyang, Claudio V. Russo, Adam Scibior, and Marcin Szymczak. „Fabular: regression formulas as probabilistic programming." In: *Proc. of the Symposium on Principles of Programming Languages (POPL)*. ACM, 2016, pp. 271–283.

[BG05] Olivier Bournez and Florent Garnier. „Proving Positive Almost–Sure Termination.“ In: *Proc. of the International Conference on Term Rewriting and Applications (RTA)*. Vol. 3467. Lecture Notes in Computer Science. Springer, 2005, pp. 323–337.

[BG06] Olivier Bournez and Florent Garnier. „Proving Positive Almost Sure Termination Under Strategies.“ In: *Proc. of the International Conference on Term Rewriting and Applications (RTA)*. Vol. 4098. Lecture Notes in Computer Science. Springer, 2006, pp. 357–371.

[Brá+15] Tomás Brázdil, Stefan Kiefer, Antonín Kucera, and Ivana Hutarová Vareková. „Runtime Analysis of Probabilistic Programs with Unbounded Recursion.“ In: *Journal of Computer and System Sciences* 81.1 (2015), pp. 288–310.

[BDL18] Flavien Breuvart and Ugo Dal Lago. „On Intersection Types and Probabilistic Lambda Calculi.“ In: *Proc. of the International Symposium on Principles and Practice of Declarative Programming (PPDP)*. ACM, 2018, 8:1–8:13.

[BM12] Mark Burgin and Gunter Meissner. „Negative Probabilities in Financial Modeling.“ In: *Wilmott* 2012.58 (2012), pp. 60–65.

[Can17] Francesco Paolo Cantelli. „Sulla probabilità come limite della frequenza.“ In: *Atti Della Accademia Nazionale Dei Lincei* 26.1 (1917), pp. 39–45.

[CMR13] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. „Verifying Quantitative Reliability for Programs that Execute on Unreliable Hardware.“ In: *Proc. of the International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. ACM, 2013, pp. 33–52.

[Car+17] Bob Carpenter, Andrew Gelman, Matthew Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. „Stan: A Probabilistic Programming Language.“ In: *Journal of Statistical Software* 76.1 (2017).

[CM05] Orieta Celiku and Annabelle McIver. „Compositional Specification and Analysis of Cost–Based Properties in Probabilistic Programs.“ In: *Proc. of the International Symposium on Formal Methods (FM)*. Vol. 3582. Lecture Notes in Computer Science. Springer, 2005, pp. 107–122.

[CS13] Aleksandar Chakarov and Sriram Sankaranarayanan. „Probabilistic Program Analysis with Martingales.“ In: *Proc. of the International Conference on Computer–Aided Verification (CAV)*. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 511–526.

[CS14] Aleksandar Chakarov and Sriram Sankaranarayanan. „Expectation Invariants for Probabilistic Program Loops as Fixed Points.“ In: *Proc. of the Static Analysis Symposium (SAS)*. Vol. 8723. Lecture Notes in Computer Science. Springer, 2014, pp. 85–100.

[CKS81] Ashok Kumar Chandra, Dexter Campbell Kozen, and Larry Joseph Stockmeyer. „Alternation.“ In: *Journal of the ACM* 28.1 (1981), pp. 114–133.

[Cha09] Robert N. Charette. „This Car Runs on Code.“ In: *IEEE Spectrum* 46.3 (2009), p. 3.

[CF17] Krishnendu Chatterjee and Hongfei Fu. „Termination of Nondeterministic Recursive Probabilistic Programs.“ In: *CoRR* abs/1701.02944 (2017). arXiv: 1701.02944. URL: http://arxiv.org/abs/1701.02944.

[CFG16] Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. „Termination Analysis of Probabilistic Programs Through Positivstellensatz's.“ In: *Proc. of the International Conference on Computer–Aided Verification (CAV)*. Vol. 9779. Lecture Notes in Computer Science. Springer, 2016, pp. 3–22.

[CNZ17] Krishnendu Chatterjee, Petr Novotný, and Dorde Zikelic. „Stochastic Invariants for Probabilistic Termination.“ In: *Proc. of the Symposium on Principles of Programming Languages (POPL)*. ACM, 2017, pp. 145–160.

[Cha+16] Krishnendu Chatterjee, Hongfei Fu, Petr Novotný, and Rouzbeh Hasheminezhad. „Algorithmic Analysis of Qualitative and Quantitative Termination Problems for Affine Probabilistic Programs.“ In: *Proc. of the Symposium on Principles of Programming Languages (POPL)*. ACM, 2016, pp. 327–342.

[CS09] Yifeng Chen and Jeff W. Sanders. „Unifying Probability with Nondeterminism.“ In: *Proc. of the International Symposium on Formal Methods (FM)*. Vol. 5850. Lecture Notes in Computer Science. Springer, 2009, pp. 467–482.

[CW08] Yixiang Chen and Hengyang Wu. „Domain Semantics of Possibility Computations.“ In: *Information Sciences* 178.12 (2008), pp. 2661–2679.

[CJ17] Kenta Cho and Bart Jacobs. „Disintegration and Bayesian Inversion, Both Abstractly and Concretely.“ In: *CoRR* abs/1709.00322 (2017). arXiv: 1709.00322. URL: http://arxiv.org/abs/1709.00322.

[Chu36] Alonzo Church. „A Note on the Entscheidungsproblem.“ In: *Journal of Symbolic Logic* 1.1 (1936), pp. 40–41.

[Coc14] David Cock. „pGCL for Isabelle.“ In: *Archive of Formal Proofs* 2014 (2014).

[DLG17] Ugo Dal Lago and Charles Grellois. „Probabilistic Termination by Monadic Affine Sized Typing.“ In: *Proc. of the European Symposium on Programming Languages and Systems (ESOP)*. Vol. 10201. Lecture Notes in Computer Science. Springer, 2017, pp. 393–419.

[Dav58] Martin David Davis. *Computability and Unsolvability*. McGraw–Hill Series in Information Processing and Computers. McGraw–Hill, 1958.

[Deh+17] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. „A Storm is Coming: A Modern Probabilistic Model Checker.“ In: *Proc. of the International Conference on Computer–Aided Verification (CAV)*. Springer, 2017, pp. 592–600.

[Dij68] Edsger Wybe Dijkstra. „Letters to the Editor: Go To Statement Considered Harmful.“ In: *Communications of the ACM* 11.3 (1968), pp. 147–148.

[Dij72] Edsger Wybe Dijkstra. „The Humble Programmer.“ In: *Communications of the ACM* 15.10 (1972), pp. 859–866.

[Dij75] Edsger Wybe Dijkstra. „Guarded Commands, Nondeterminacy and Formal Derivation of Programs.“ In: *Communications of the ACM* 18.8 (1975), pp. 453–457.

[Dij76] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice–Hall, 1976.

[Dir42] Paul Adrien Maurice Dirac. „Bakerian Lecture. The Physical Interpretation of Quantum Mechanics.“ In: *Proc. of the Royal Society of London. Series A, Mathematical and Physical Sciences* 180.980 (1942), pp. 1–40.

[Dur10] Rick Durrett. *Probability: Theory and Examples*. Cambridge University Press, 2010.

[EGK12] Javier Esparza, Andreas Gaiser, and Stefan Kiefer. „Proving Termination of Probabilistic Programs Using Patterns.“ In: *Proc. of the International Conference on Computer–Aided Verification (CAV)*. Vol. 7358. Lecture Notes in Computer Science. Springer, 2012, pp. 123–138.

[Fen+17] Yijun Feng, Lijun Zhang, David Nicolaas Jansen, Naijun Zhan, and Bican Xia. „Finding Polynomial Loop Invariants for Probabilistic Programs.“ In: *Proc. of the International Symposium on Automated Technology for Verification and Analysis (ATVA)*. Lecture Notes in Computer Science. 2017.

[FH15] Luis María Ferrer Fioriti and Holger Hermanns. „Probabilistic Termination: Soundness, Completeness, and Compositionality.“ In: *Proc. of the Symposium on Principles of Programming Languages (POPL)*. ACM, 2015, pp. 489–501.

[FLP83] Michael John Fischer, Nancy Ann Lynch, and Mike Paterson. „Impossibility of Distributed Consensus with One Faulty Process.“ In: *Proc. Symposium on Principles of Database Systems (PODS)*. ACM, 1983, pp. 1–7.

[Flo67a] Robert W Floyd. „Assigning Meanings to Programs.“ In: *Mathematical Aspects of Computer Science* 19.19-32 (1967), p. 1.

[Flo67b] Robert W Floyd. „Nondeterministic Algorithms.“ In: *Journal of the ACM* 14.4 (1967), pp. 636–644.

[Fra98] Gudmund Skovbjerg Frandsen. „Randomised Algorithms.“ Lecture notes of the Lecture „Randomised Algorithms“ held at University of Aarhus, Denmark. Accessed online March 22, 2018. 1998. URL: http://www.cs.au.dk/~gudmund/Documents/randompearlnotes.pdf.

[Fre79] Rūsiņš Mārtiņš Freivalds. „Fast Probabilistic Algorithms.“ In: *Proc. of the International Symposium on Mathematical Foundations of Computer Science (MFCS)*. Vol. 74. Lecture Notes in Computer Science. Springer, 1979, pp. 57–69.

[Fro+16a] Florian Frohn, Matthias Naaf, Jera Hensel, Marc Brockschmidt, and Jürgen Giesl. „Lower Runtime Bounds for Integer Programs.“ In: *Aachener Informatik-Berichte* AIB-2016-03 (2016). URL: http://aib.informatik.rwth-aachen.de/2016/2016-03.pdf.

[Fro+16b] Florian Frohn, Matthias Naaf, Jera Hensel, Marc Brockschmidt, and Jürgen Giesl. „Lower Runtime Bounds for Integer Programs.“ In: *Proc. of the International Joint Conference on Automated Reasoning (IJCAR)*. Vol. 9706. Lecture Notes in Computer Science. Springer, 2016, pp. 550–567.

[Gie+19] Jürgen Giesl, Marcel Hark, Benjamin Lucien Kaminski, and Joost-Pieter Katoen. „Aiming Low is Harder — Inductive Lower Bounds in Probabilistic Program Verification.“ [under review]. 2019.

[Gil77] John Gill. „Computational Complexity of Probabilistic Turing Machines.“ In: *SIAM Journal on Computing* 6.4 (1977), pp. 675–695.

[Goo+08] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. „Church: A Language for Generative Models." In: *Proc. of the Conference in Uncertainty in Artificial Intelligence (UAI)*. AUAI Press, 2008, pp. 220–229.

[Gor+14] Andrew D. Gordon, Thomas A. Henzinger, Aditya Vithal Nori, and Sriram K. Rajamani. „Probabilistic programming." In: *Proc. of Future of Software Engineering (FOSE)*. ACM, 2014, pp. 167–181.

[GKM12] Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. „Operational Versus Weakest Precondition Semantics for the Probabilistic Guarded Command Language." In: *Proc. of the International Conference on Quantitative Evaluation of Systems (QEST)*. IEEE Computer Society, 2012, pp. 168–177.

[GKM14] Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. „Operational versus Weakest Pre–Expectation Semantics for the Probabilistic Guarded Command Language." In: *Performance Evaluation* 73 (2014), pp. 110–132.

[Gre+16] Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Federico Olmedo. „On the Semantic Intricacies of Conditioning." In: *Extended Abstracts of the 1st Workshop on Probabilistic Programming Semantics (PPS)* (2016).

[Gri99] Edward Ronald Griffor. *Handbook of Computability Theory*. Vol. 140. Studies in Logic and the Foundations of Mathematics. Elsevier, 1999.

[Hai04] Thomas Haigh. „Biographies: Robert W Floyd, in Memoriam." In: *Annals of the History of Computing* 26.2 (2004), pp. 75–83.

[HJ94] Hans Hansson and Bengt Jonsson. „A Logic for Reasoning about Time and Reliability." In: *Formal Aspects of Computing* 6.5 (1994), pp. 512–535.

[HSP82] Sergiu Hart, Micha Sharir, and Amir Pnueli. „Termination of Probabilistic Concurrent Programs." In: *Proc. of the Symposium on Principles of Programming Languages (POPL)*. ACM Press, 1982, pp. 1–6.

[HSP83] Sergiu Hart, Micha Sharir, and Amir Pnueli. „Termination of Probabilistic Concurrent Program." In: *Trans. on Programming Languages and Systems* 5.3 (1983), pp. 356–380.

[Hau04] Espen Gaarder Haug. „Why so negative to negative probabilities?" In: *Wilmott Magazine* (2004), pp. 34–38.

[Heh98] Eric Charles Roy Hehner. „Formalization of Time and Space." In: *Formal Aspects of Computing* 10.3 (1998), pp. 290–306.

[Heh11] Eric Charles Roy Hehner. „A Probability Perspective." In: *Formal Aspects of Computing* 23.4 (2011), pp. 391–419.

[Hen13] Thomas A. Henzinger. „Quantitative Reactive Modeling and Verification." In: *Computer Science — R&D* 28.4 (2013), pp. 331–344.

[HS55] Edwin Hewitt and Leonard Jimmie Savage. „Symmetric Measures on Cartesian Products." In: *Transactions of the American Mathematical Society* 80.2 (1955), pp. 470–501.

[HC88] Timothy Hickey and Jacques Cohen. „Automating Program Analysis." In: *Journal of the ACM* 35.1 (1988), pp. 185–220.

[Hin+16] Wataru Hino, Hiroki Kobayashi, Ichiro Hasuo, and Bart Jacobs. „Healthiness from Duality." In: *Proc. of the Annual Symposium on Logic in Computer Science (LICS)*. ACM, 2016, pp. 682–691.

[Hir15] Jerry Hirsch. *Elon Musk: Model S not a car but a 'sophisticated computer on wheels'*. Accessed online April 18, 2015. 2015. URL: http://www.latimes.com/business/autos/la-fi-hy-musk-computer-on-wheels-20150319-story.html.

[Hoa62] Charles Antony Richard Hoare. „Quicksort." In: *The Computer Journal* 5.1 (1962), pp. 10–15.

[Hoa69] Charles Antony Richard Hoare. „An Axiomatic Basis for Computer Programming." In: *Communications of the ACM* 12.10 (1969), pp. 576–580.

[Höl16] Johannes Hölzl. „Formalising Semantics for Expected Running Time of Probabilistic Programs." In: *Proc. of the International Conference on Interactive Theorem Proving (ITP)*. Vol. 9807. LNCS. Springer, 2016, pp. 475–482.

[Hsu17] Justin Hsu. „Probabilistic Couplings for Probabilistic Reasoning." PhD thesis. University of Pennsylvania, USA, 2017.

[Hur+14] Chung-Kil Hur, Aditya Vithal Nori, Sriram K. Rajamani, and Selva Samuel. „Slicing Probabilistic Programs." In: *Proc. of the Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2014, pp. 133–144.

[Hur02] Joe Hurd. „A Formal Approach to Probabilistic Termination." In: *Theorem Proving in Higher Order Logics (TPHOL)*. Vol. 2410. LNCS. Springer Berlin Heidelberg, 2002, pp. 230–245.

[Hur03] Joe Hurd. „Formal Verification of Probabilistic Algorithms." PhD thesis. University of Cambridge, UK, 2003.

[Iee] *IEEE 802.3-2015 — IEEE Standard for Ethernet*. Accessed online September 15, 2018. 2016. URL: https://standards.ieee.org/standard/802_3-2015.html.

[Ica17] Thomas Icard. „Beyond Almost–sure Termination.“ In: *Proc. of the Annual Meeting of the Cognitive Science Society*. Cognitive Science Society, 2017.

[IO01] Samin S. Ishtiaq and Peter William O'Hearn. „BI as an Assertion Language for Mutable Data Structures.“ In: *Proc. of the Symposium on Principles of Programming Languages (POPL)*. ACM, 2001, pp. 14–26.

[JGP00] Jacek Jachymski, Leslaw Gajek, and Piotr Pokarowski. „The Tarski–Kantorovitch Prinicple and the Theory of Iterated Function Systems.“ In: *Bulletin of the Australian Mathematical Society* 61.2 (2000), pp. 247–261.

[Jan+15a] Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Federico Olmedo, Friedrich Gretz, and Annabelle McIver. „Conditioning in Probabilistic Programming.“ In: *Proc. of the Conference on Mathematical Foundations of Programming Semantics (MFPS)* 319 (2015), pp. 199–216.

[Jan+15b] Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. „Probabilistic Programs — A Natural Model for Approximate Computations.“ In: *Extended Abstracts of the Workshop on Approximate Computing (AC 15)*. 2015.

[Jan+16] Nils Jansen, Christian Dehnert, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Lukas Westhofen. „Bounded Model Checking for Probabilistic Programs.“ In: *Proc. of the International Symposium on Automated Technology for Verification and Analysis (ATVA)*. Vol. 9938. Lecture Notes in Computer Science. 2016, pp. 68–85.

[Jin18] Jimmy Jin. *Elchanan Mossel's Dice Problem*. Accessed online July 15, 2018. 2018. URL: http://www.yichijin.com/files/elchanan.pdf.

[Jon90] Claire Jones. „Probabilistic Non–Determinism.“ PhD thesis. University of Edinburgh, UK, 1990.

[KK15a] Benjamin Lucien Kaminski and Joost-Pieter Katoen. „Analyzing Expected Outcomes and (Positive) Almost–sure Termination of Probabilistic Programs is Hard.“ In: *Proc. of the Young Researchers' Conference „Frontiers of Formal Methods“ (FFM)*. Vol. 9234. Aachener Informatik Berichte. 2015, pp. 179–184.

[KK15b] Benjamin Lucien Kaminski and Joost-Pieter Katoen. „On the Hardness of Almost–Sure Termination." In: *Proc. of the International Symposium on Mathematical Foundations of Computer Science (MFCS)*. Vol. 9234. Lecture Notes in Computer Science. Springer, 2015, pp. 307–318.

[KK17a] Benjamin Lucien Kaminski and Joost-Pieter Katoen. „A Weakest Pre–Expectation Semantics for Mixed–Sign Expectations." In: *Extended Abstracts of the 2nd Workshop on Probabilistic Programming Semantics (PPS)*. 2017.

[KK17b] Benjamin Lucien Kaminski and Joost-Pieter Katoen. „A Weakest Pre–expectation Semantics for Mixed–sign Expectations." In: *Proc. of the Annual Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, 2017, pp. 1–12.

[KK17c] Benjamin Lucien Kaminski and Joost-Pieter Katoen. „A Weakest Pre–expectation Semantics for Mixed–sign Expectations." In: *CoRR* abs/1703.07682 (2017). arXiv: 1703.07682. URL: http://arxiv.org/abs/1703.07682.

[KKM16] Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. „Inferring Covariances for Probabilistic Programs." In: *Proc. of the International Conference on Quantitative Evaluation of Systems (QEST)*. Vol. 9826. Lecture Notes in Computer Science. Springer, 2016, pp. 191–206.

[KKM18] Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. „On the Hardness of Analyzing Probabilistic Programs." In: *Acta Informatica* (2018).

[KM17] Benjamin Lucien Kaminski and Carroll Morgan. [unpublished personal email communication]. 2017.

[Kam+16] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. „Weakest Precondition Reasoning for Expected Run–Times of Probabilistic Programs." In: *Proc. of the European Symposium on Programming Languages and Systems (ESOP)*. Vol. 9632. Lecture Notes in Computer Science. Springer, 2016, pp. 364–389.

[Kam+18] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. „Weakest Precondition Reasoning for Expected Runtimes of Randomized Algorithms." In: *Journal of the ACM* 65.5 (2018), 30:1–30:68.

[Kat16] Joost-Pieter Katoen. „The Probabilistic Model Checking Landscape." In: *Proc. of the Annual Symposium on Logic in Computer Science (LICS)*. ACM, 2016, pp. 31–45.

[Kat+10] Joost-Pieter Katoen, Annabelle McIver, Larissa Meinicke, and Carroll Morgan. „Linear–invariant Generation for Probabilistic Programs: Automated Support for Proof–based Methods." In: *Proc. of the Static Analysis Symposium (SAS)*. Vol. 6337. Lecture Notes in Computer Science. Springer, 2010, pp. 390–406.

[Kat+15] Joost-Pieter Katoen, Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, and Federico Olmedo. „Understanding Probabilistic Programs." In: *Correct System Design — Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday*. Vol. 9360. Lecture Notes in Computer Science. Springer, 2015, pp. 15–32.

[Kei15] Klaus Keimel. „Healthiness Conditions for Predicate Transformers." In: *Proc. of the Conference on Mathematical Foundations of Programming Semantics (MFPS)* 319 (2015), pp. 255–270.

[KP17] Klaus Keimel and Gordon David Plotkin. „Mixed Powerdomains for Probability and Nondeterminism." In: *Logical Methods in Computer Science* 13.1 (2017).

[Keu+18] Maurice van Keulen, Benjamin Lucien Kaminski, Christoph Matheja, and Joost-Pieter Katoen. „Rule–based Conditioning of Probabilistic Data Integration." In: *Proc. of the International Conference on Scalable Uncertainty Management (SUM)*. Lecture Notes in Artificial Intelligence. Springer, 2018.

[Kle43] Stephen Cole Kleene. „Recursive Predicates and Quantifiers." In: *Transactions of the AMS* 53.1 (1943), pp. 41–73.

[Kle13] Achim Klenke. *Probability Theory: A Comprehensive Course*. Springer, 2013.

[Kna28] Bronisław Knaster. „Un Théorème sur les Functions D'ensembles." In: *Annales de la Societe Polonaise de Mathematique* 6 (1928), pp. 133–134.

[Koc+18] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. „Spectre Attacks: Exploiting Speculative Execution." In: *CoRR* (2018). arXiv: 1801.01203.

[KF09] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models — Principles and Techniques*. MIT Press, 2009.

[Koz79] Dexter Kozen. „Semantics of Probabilistic Programs." In: *Proc. of the Annual Symposium on Foundations of Computer Science (FOCS)*. 1979, pp. 101–114.

[Koz81] Dexter Kozen. „Semantics of Probabilistic Programs." In: *Journal of Computer and System Sciences* 22.3 (1981), pp. 328–350.

[Koz83] Dexter Kozen. „A Probabilistic PDL.“ In: *Proc. of the Annual Symposium on Theory of Computing (STOC)*. 1983, pp. 291–297.

[Koz85] Dexter Kozen. „A Probabilistic PDL.“ In: *Journal of Computer and System Sciences* 30.2 (1985), pp. 162–178.

[Koz18] Dexter Kozen. *On Disintegration in Probabilistic Semantics*. Tech. rep. [withdrawn]. 2018.

[KUH19] Satoshi Kura, Natsuki Urabe, and Ichiro Hasuo. „Tail Probabilities for Randomized Program Runtimes via Martingales for Higher Moments.“ In: *Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Lecture Notes in Computer Science. [to appear]. Springer, 2019.

[LNS82] Jean-Louis Lassez, V. L. Nguyen, and Liz Sonenberg. „Fixed Point Theorems and Semantics: A Folk Tale.“ In: *Information Processing Letters* 14.3 (1982), pp. 112–116.

[LBW17] Tuan Anh Le, Atilim Gunes Baydin, and Frank D. Wood. „Inference Compilation and Universal Probabilistic Programming.“ In: *Proc. of the International Conference on Artificial Intelligence and Statistics (AISTATS)*. Vol. 54. Proceedings of Machine Learning Research. PMLR, 2017, pp. 1338–1348.

[Lip+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. „Meltdown.“ In: *CoRR* (2018). arXiv: 1801.01207.

[MP74] Zohar Manna and Amir Pnueli. „Axiomatic Approach to Total Correctness of Programs.“ In: *Acta Informatica* 3 (1974), pp. 243–263.

[MM01] Annabelle McIver and Carroll Morgan. „Partial Correctness for Probabilistic Demonic Programs.“ In: *Theoretical Computer Science* 266.1-2 (2001), pp. 513–541.

[MM05] Annabelle McIver and Carroll Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Monographs in Computer Science. Springer, 2005.

[MM16] Annabelle McIver and Carroll Morgan. „A New Rule for Almost–certain Termination of Probabilistic and Demonic Programs.“ In: *CoRR* abs/1612.01091 (2016). arXiv: 1612.01091. URL: http://arxiv.org/abs/1612.01091.

[McI+18] Annabelle McIver, Carroll Morgan, Benjamin Lucien Kaminski, and Joost-Pieter Katoen. „A New Proof Rule for Almost–sure Termination.“ In: *Proc. of the Symposium on Principles of Programming Languages (POPL)* 2.POPL (2018), 33:1–33:28.

[MOW04] Michael W. Mislove, Joël Ouaknine, and James Worrell. „Axioms for Probability and Nondeterminism.“ In: *Electronic Notes in Theoretical Computer Science* 96 (2004), pp. 7–28.

[Mis00] Michael William Mislove. „Nondeterminism and Probabilistic Choice: Obeying the Laws.“ In: *Proc. of the International Conference on Concurrency Theory (CONCUR)*. 2000, pp. 350–364.

[Mis06] Michael William Mislove. „On Combining Probability and Nondeterminism.“ In: *Electronic Notes in Theoretical Computer Science* 162 (2006), pp. 261–265.

[MU05] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[Mon01] David Monniaux. „An Abstract Analysis of the Probabilistic Termination of Programs.“ In: *Proc. of the Static Analysis Symposium (SAS)*. Vol. 2126. Lecture Notes in Computer Science. Springer, 2001, pp. 111–126.

[Mon05] David Monniaux. „Abstract Interpretation of Programs as Markov Decision Processes.“ In: *Science of Computer Programming* 58.1–2 (2005), pp. 179–205.

[MM99] Carroll Morgan and Annabelle McIver. „An Expectation–Transformer Model for Probabilistic Temporal Logic.“ In: *Logic Journal of the Interest Group in Pure and Applied Logics* 7.6 (1999), pp. 779–804.

[MMS96] Carroll Morgan, Annabelle McIver, and Karen Seidel. „Probabilistic Predicate Transformers.“ In: *Trans. on Programming Languages and Systems* 18.3 (1996), pp. 325–353.

[MJ84] Francis Lockwood Morris and Cliff B. Jones. „An Early Program Proof by Alan Turing.“ In: *IEEE Annals of the History of Computing* 6.2 (1984), pp. 139–143. DOI: 10.1109/MAHC.1984.10017. URL: https://doi.org/10.1109/MAHC.1984.10017.

[Mos47] Andrzej Mostowski. „On Definable Sets of Positive Integers.“ In: *Fundamenta Mathematicae* 34.1 (1947), pp. 81–112.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[MO05] Andrzej S. Murawski and Joël Ouaknine. „On Probabilistic Program Equivalence and Refinement.“ In: *Proc. of the International Conference on Concurrency Theory (CONCUR)*. Vol. 3653. Lecture Notes in Computer Science. Springer, 2005, pp. 156–170.

[NCH18] Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. „Bounded Expectations: Resource Analysis for Probabilistic Programs." In: *Proc. of the Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2018, pp. 496–512.

[Nie87] Hanne Riis Nielson. „A Hoare–like Proof System for Analysing the Computation Time of Programs." In: *Science of Computer Programming* 9.2 (1987), pp. 107–136.

[Nor+14] Aditya V. Nori, Chung-Kil Hur, Sriram K. Rajamani, and Selva Samuel. „R2: An Efficient MCMC Sampler for Probabilistic Programs." In: *Proc. of the AAAI Conference on Artificial Intelligence (AAAI)*. AAAI Press, 2014.

[Odi92] Piergiorgio Odifreddi. *Classical Recursion Theory: The Theory of Functions and Sets of Natural Numbers*. Elsevier, 1992.

[Odi99] Piergiorgio Odifreddi. *Classical Recursion Theory, Volume II*. Elsevier, 1999.

[Olm+16] Federico Olmedo, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. „Reasoning about Recursive Probabilistic Programs." In: *Proc. of the Annual Symposium on Logic in Computer Science (LICS)*. ACM, 2016, pp. 672–681.

[Olm+18] Federico Olmedo, Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Annabelle McIver. „Conditioning in Probabilistic Programming." In: *Trans. on Programming Languages and Systems* 40.1 (2018), 4:1–4:50.

[Pro] *PROBABILISTIC-PROGRAMMIN.org*. Accessed online August 20, 2018. URL: http://probabilistic-programming.org/wiki/Home.

[Pan01] Prakash Panangaden. „Does Combining Nondeterminism and Probability Make Sense?" In: *Bulletin of the EATCS* 75 (2001), pp. 182–189.

[PZ83] Christos Harilaos Papadimitriou and Stathis Zachos. „Two Remarks on the Power of Counting." In: *Proc. of the GI Symposium on Theoretical Computer Science*. Vol. 145. Lecture Notes in Computer Science. Springer, 1983, pp. 269–276.

[Par69] David Park. „Fixpoint Induction and Proofs of Program Properties." In: *Machine intelligence* 5 (1969).

[Pfe09] Avi Pfeffer. „Figaro: An Object–oriented Probabilistic Programming Language." In: *Charles River Analytics Technical Report* 137 (2009), p. 96.

[Plo04] Gordon David Plotkin. „The Origins of Structural Operational Semantics." In: *Journal of Logic and Algebraic Programming* 60–61 (2004), pp. 3–15.

[Pos44] Emil Leon Post. „Recursively Enumerable Sets of Positive Integers and their Decision Problems.“ In: *Bulletin of the American Mathematical Society* 50.5 (1944), pp. 284–316.

[Pos48] Emil Leon Post. „Degrees of Recursive Unsolvability.“ In: *Bulletin of the American Mathematical Society* 54.7 (1948), pp. 641–642.

[Pos04] Emil Leon Post. „Absolutely Unsolvable Problems and Relatively Undecidable Propositions. Account of an Anticipation.“ In: *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions*. Ed. by Martin David Davis. Courier Corporation, 2004.

[Put05] Martin Lee Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 2005.

[Rab63] Michael Oser Rabin. „Probabilistic Automata.“ In: *Information and Control* 6.3 (1963), pp. 230–245.

[Rab76] Michael Oser Rabin. „Probabilistic Algorithms.“ In: *Algorithms and Complexity: New Directions and Recent Results*. Ed. by Joseph Frederick Traub. Academic Press, 1976, pp. 21–39.

[RS59] Michael Oser Rabin and Dana Stewart Scott. „Finite Automata and Their Decision Problems.“ In: *IBM Journal of Research and Development* 3.2 (1959), pp. 114–125.

[RKT07] Luc de Raedt, Angelika Kimmig, and Hannu Toivonen. „ProbLog: A Probabilistic Prolog and Its Application in Link Discovery.“ In: *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*. 2007, pp. 2462–2467. URL: http://ijcai.org/Proceedings/07/Papers/396.pdf.

[Rey02] John Charles Reynolds. „Separation Logic: A Logic for Shared Mutable Data Structures.“ In: *Proc. of the Annual Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, 2002, pp. 55–74.

[Ric53] Henry Gordon Rice. „Classes of Recursively Enumerable Sets and Their Decision Problems.“ In: *Trans. of the American Mathematical Society* 74.2 (1953), pp. 358–366.

[Rie67] Bernhard Riemann. *Ueber die Darstellbarkeit einer Function durch eine trigonometrische Reihe*. Königliche Gesellschaft der Wissenschaften zu Göttingen, 1867.

[Rob15] Borut Robič. *The Foundations of Computability Theory*. Springer, 2015.

[Rog59] Hartley Rogers. „Computing Degrees of Unsolvability.“ In: *Mathematische Annalen* 138.2 (1959), pp. 125–140.

[Rog67] Hartley Rogers. *Theory of Recursive Functions and Effective Computability*. Vol. 5. McGraw–Hill New York, 1967.

[Roj97] Raúl Rojas. „Konrad Zuse's Legacy: The Architecture of the Z1 and Z3." In: *IEEE Annals of the History of Computing* 19.2 (1997), pp. 5–16.

[Rot16] Jurriaan Roth. „Coalgebra, Lecture 13: Induction; Coinduction in Lattices and Categories." Lecture notes of the Lecture „Coalgebra" held at Radboud University, The Netherlands. Accessed online December 12, 2017. 2016. URL: http://www.cs.ru.nl/~jrot/coalgebra/coalg-lect13.pdf.

[Sam+14] Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S. McKinley, Dan Grossman, and Luis Ceze. „Expressing and Verifying Probabilistic Assertions." In: *Proc. of the Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2014, pp. 112–122.

[Sas+11] Raimondas Sasnauskas, Oscar Soria Dustmann, Benjamin Lucien Kaminski, Klaus Wehrle, Carsten Weise, and Stefan Kowalewski. „Scalable Symbolic Execution of Distributed Systems." In: *Proc. of the International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society, 2011, pp. 333–342.

[Sch17] Noah Schweber. Δ^0_n*–complete sets in the arithmetical hierarchy*. Mathematics Stack Exchange. Accessed online August 1, 2018. 2017. URL: https://math.stackexchange.com/q/2174728.

[Sco69] Dana Stewart Scott. *A Type–theoretical Alternative to ISWIM, CUCH, OWHY*. Manuscript. 1969.

[Scu] Marco Scutari. *Bayesian Network Repository*. Accessed online April 23, 2018. URL: http://www.bnlearn.com.

[Sel17] Peter Selinger. „Quantum Programming." Slides of the Lecture „Quantum Programming" held at the *1st School on Foundations of Programming and Software Systems (FoPSS) — Probabilistic Programming*. Accessed online September 7, 2018. 2017. URL: http://alfa.di.uminho.pt/~nevrenato/probprogschool_slides/Peter.pdf.

[SR17] Chung-chieh Shan and Norman Ramsey. „Exact Bayesian Inference by Symbolic Disintegration." In: *Proc. of the Symposium on Principles of Programming Languages (POPL)*. ACM, 2017, pp. 130–144.

[SPH84] Micha Sharir, Amir Pnueli, and Sergiu Hart. „Verification of Probabilistic Programs." In: *SIAM Journal of Computing* 13.2 (1984), pp. 292–314.

[Sho09] Victor Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, 2009.

[Smi00] Michiel Smid. „Closest–Point Problems in Computational Geometry." In: *Handbook of Computational Geometry*. Ed. by Jörg-Rüdiger Sack and Jorge Urrutia. North–Holland, 2000, pp. 877–935.

[SS11] Jon Sneyers and Danny De Schreye. „Probabilistic Termination of CHRiSM Programs." In: *Proc. of the International Symposium on Logic–Based Program Synthesis and Transformation (LOPSTR)*. Vol. 7225. Lecture Notes in Computer Science. Springer, 2011, pp. 221–236.

[SS77] Robert Solovay and Volker Strassen. „A Fast Monte–Carlo Test for Primality." In: *SIAM Journal on Computing* 6.1 (1977), pp. 84–85.

[SYY10] Frank Stephan, Yue Yang, and Liang Yu. „Turing Degrees and the Ershov Hierarchy." In: *Proc. of the Asian Logic Conference*. World Scientific. 2010, pp. 300–321.

[Tio89] Michael L. Tiomkin. „Probabilistic Termination Versus Fair Termination." In: *Theoretical Computer Science* 66.3 (1989), pp. 333–340.

[TKP09] Regina Tix, Klaus Keimel, and Gordon David Plotkin. „Semantic Domains for Combining Probability and Non–Determinism." In: *Electronic Notes in Theoretical Computer Science* 222 (2009), pp. 3–99.

[Tur37] Alan Mathison Turing. „On computable numbers, with an application to the Entscheidungsproblem." In: *Proc. of the London Mathematical Society* 2.1 (1937), pp. 230–265.

[Tur49] Alan Mathison Turing. „Checking a Large Routine." In: *Report of a Conference on High Speed Automatic Calculating Machines*. Univ. Math. Lab., Cambridge, 1949, pp. 67–69.

[Var03] Daniele Varacca. „Probability, Nondeterminism and Concurrency. Two Denotational Models for Probabilistic Computation." PhD thesis. BRICS – Aarhus University, 2003. URL: www.brics.dk/DS/03/14/.

[VW06] Daniele Varacca and Glynn Winskel. „Distributing Probability over Non–Determinism." In: *Mathematical Structures in Computer Science* 16.1 (2006), pp. 87–113.

[Var85] Moshe Ya'akov Vardi. „Automatic Verification of Probabilistic Concurrent Finite–State Programs." In: *Proc. of the Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society, 1985, pp. 327–338.

[Wika] Wikipedia. *Ariane 5*. Accessed online August 22, 2018. URL: https://en.wikipedia.org/wiki/Ariane_5.

[Wikb] Wikipedia. *Borel–Cantelli Lemma*. Accessed online December 27, 2017. URL: https://en.wikipedia.org/wiki/Borel-Cantelli_lemma.

[Wikc] Wikipedia. *Bra–ket Notation*. Accessed online September 5, 2017. URL: https://en.wikipedia.org/wiki/BraâĂŞket_notation.

[Wikd] Wikipedia. *Covariance*. Accessed online August 1, 2018. URL: https://en.wikipedia.org/wiki/Covariance.

[Wike] Wikipedia. *Disintegration Theorem*. Accessed online July 15, 2018. URL: https://en.wikipedia.org/wiki/Disintegration_theorem.

[Wikf] Wikipedia. *Hewitt–Savage Zero–One Law*. Accessed online December 27, 2017. URL: https://en.wikipedia.org/wiki/Hewitt-Savage_zero-one_law.

[Wikg] Wikipedia. *Iverson Bracket*. Accessed online April 15, 2017. URL: https://en.wikipedia.org/wiki/Iverson_bracket.

[Wikh] Wikipedia. *Kolmogorov's Zero–One Law*. Accessed online December 27, 2017. URL: https://en.wikipedia.org/wiki/Kolmogorov's_zero-one_law.

[Wiki] Wikipedia. *Monotonic Function*. Accessed online December 28, 2017. URL: https://en.wikipedia.org/wiki/Monotonic_function.

[Wikj] Wikipedia. *Rosetta Stone*. Accessed online September 13, 2018. URL: https://en.wikipedia.org/wiki/Rosetta_Stone.

[Wikk] Wikipedia. *Sublinear Function*. Accessed online July 20, 2017. URL: https://en.wikipedia.org/wiki/Sublinear_function.

[Wikl] Wikipedia. *Therac–25*. Accessed online August 22, 2018. URL: https://en.wikipedia.org/wiki/Therac-25.

[Wikm] Wikipedia. *Turing Completeness*. Accessed online May 13, 2017. URL: https://en.wikipedia.org/wiki/Turing_completeness.

[Wil14] Virginia Vassilevska Williams. *Multiplying Matrices in $O(n^{2.373})$ Time*. Accessed online August 27, 2018. 2014. URL: http://theory.stanford.edu/~virgi/matrixmult-f.pdf.

[WC11] Hengyang Wu and Yixiang Chen. „The Semantics of *wlp* and *slp* of Fuzzy Imperative Programming Languages." In: *Proc. of the International Conference on Nonlinear Mathematics for Uncertainty and its Applications (NL–MUA)*. Vol. 100. Advances in Intelligent and Soft Computing. Springer, 2011, pp. 357–364.

[WC12] Hengyang Wu and Yixiang Chen. „Semantics of Non–Deterministic Possibility Computation.“ In: *Fuzzy Sets and Systems* 199 (2012), pp. 47–63.

[Zus90] Konrad Zuse. *Der Computer — Mein Lebenswerk (2. Auflage)*. Springer, 1990.

INDEX

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre eidesstattlich, dass ich die Dissertation selbstständig verfasst und alle in Anspruch genommenen Hilfen in der Dissertation angegeben habe.

Aachen, 15. Februar 2019

Benjamin Lucien Kaminski