Annabelle McIver
Carroll Morgan

Abstraction, Refinement and Proof for Probabilistic Systems

With 62 Figures

Springer

Annabelle McIver
Department of Computing
Macquarie University
Sydney 2109 Australia
anabel@ics.mq.edu.au

Carroll Morgan
Department of Computer Science
and Engineering
The University of New South Wales
Sydney 2052 Australia
carrollm@cse.unsw.edu.au

Preface

Probabilistic techniques in computer programs and systems are becoming more and more widely used, for increased efficiency (as in random algorithms), for symmetry breaking (distributed systems) or as an unavoidable artefact of applications (modelling fault-tolerance). Because interest in them has been growing so strongly, stimulated by their many potential uses, there has been a corresponding increase in the study of their correctness — for the more widespread they become, the more we will depend on understanding their behaviour, and their limits, exactly.

In this volume we address that last concern, of understanding: we present a method for *rigorous reasoning about probabilistic programs and systems.* It provides an operational model — "how they work" — and an associated program logic — "how we should reason about them" — that are designed to fit together. The technique is simple in principle, and we hope that with it we will be able to increase dramatically the effectiveness of our analysis and use of probabilistic techniques in practice.

Our contribution is a probabilistic calculus that operates at the level of the program text, and it is *light-weight* in the sense that the amount of reasoning is similar in size and style to what standard assertional techniques require. In the fragment at right, for example, each potential loop entry occurs with probability 1/2; the resulting iteration establishes $x \geq 1/2$ with probability exactly p for any $0 \leq p \leq 1$. It is thus an implementation of the general operation *choose with probability p*, but it uses only simple tests of unbiased random bits (to implement the loop guard). It should take only a little quantitative logic to confirm that claim, and indeed we will show that just four lines of reasoning suffice.

$x{:=}\ p\,;$
while $1/2$ **do**
  $x{:=}\ 2x;$
  if $x \geq 1$
    then $x{:=}\ x-1$
  fi
od

Economy and precision of reasoning are what we have come to expect for standard programs; there is no reason we should accept less when they are probabilistic.

The cover illustration comes from page 59.
The program fragment is adapted from Fig. 7.7.10 on page 210.

Scope and applicability

Methods for the analysis of probabilistic systems include automata, labelled transition systems, model checking and logic (*e.g.* dynamic or temporal). Our work falls into the last category: we overlay the Hoare-logic paradigm with probabilistic features imported from Markov processes, taking from each the essential characteristics required for a sound mathematical theory of refinement and proof. The aim is to accommodate modelling and analysis of both sequential and distributed probabilistic systems, and to allow — even encourage — movement between different levels of abstraction.

Our decision to focus on *logic* — and a proof system for it — was motivated by our experience with logical techniques more generally: they impose a discipline and order which promotes clarity in specifications and design; the resulting proofs can often be carried out, and checked, with astonishing conciseness and accuracy; and the calculation rules of the logic lead to an algebra that captures useful equalities and inequalities at the level of the programs themselves.

Although we rely ultimately on an operational model, we use it principally to validate the logic (and that, in turn, justifies the algebra) — direct reliance on the model's details for individual programs is avoided if possible. (However we do not hesitate to use such details to support our intuition.) We feel that operational reasoning is more suited to the algorithmic methods of verification used by model checkers and simulation tools which can, for specific programs, answer questions that are impractical for the general approach that a logic provides.

Thus the impact of our approach is most compelling when applied to programs which are intricate either in their implementation or their design, or have generic features such as undetermined size or other parameters. They might appear as probabilistic source-level portions of large sequential programs, or as abstractions from the probabilistic modules of a comprehensive system-level design; we provide specific examples of both situations. In the latter case the ability to abstract modules' properties has a significant effect on the overall verification enterprise.

Technical features

Because we generalise the well-established assertional techniques of specifications, pre- and postconditions, there is a natural continuity of reasoning style evident in the simultaneous use of the new and the familiar approaches: the probabilistic analysis can be deployed more, or less, as the situation warrants.

A major feature is that we place probabilistic choice and abstraction together, in the same framework, without having to factor either of them out for separate treatment unless we wish to (as in fact we do in Chap. 11). This justifies the *abstraction and refinement* of our title, and is what gives us access to the stepwise-development paradigm of standard programming where systems are "refined" from high levels of abstraction towards the low levels that include implementation detail.

As a side-effect of including abstraction, we retain its operational counterpart *demonic choice* as an explicit operator $\sqcap$ in the cut-down probabilistic programming language *pGCL* which we use to describe our algorithms — that is, the new probabilistic choice operator ${}_p\oplus$ refines demonic choice rather than replacing it. In Chap. 8 we consider angelic choice $\sqcup$ as well, which is thus a further refinement.

Probabilistic and demonic choice together allow an elementary treatment of the hybrid that selects "with probability *at least* p" (or similarly "*at most* p"), an abstraction which accurately models our unavoidable ignorance of exact probabilities in real applications. Thus in our mathematical model we are able to side-step the issue of "approximate refinement."

That is, rather than saying "this coin refines a fair coin with probability 95%," we would say "this coin refines one which is within 5% of being fair." This continues the simple view that either an implementation refines a specification or it does not, which simplicity is possible because we have retained the original treatment in terms of sets of behaviours: abstraction is inclusion; refinement is reverse inclusion; and demonic choice is union. In that way we maintain the important relationship between the three concepts. (Section 6.5 on pp. 169ff illustrates this geometrically.)

Organisation and intended readership

The material is divided into three major parts of increasing specialisation, each of which can to a large extent be studied on its own; a fourth part contains appendices. We include a comprehensive index and extensive cross-referencing.

Definitions of notation and explanations of standard mathematical techniques are carefully given, rather than simply assumed; they appear as footnotes at their first point of use and are made visually conspicuous by using SMALL CAPITALS for the defined terms (where grammar allows). Thus in many cases a glance should be sufficient to determine whether any footnote contains a definition. In any case all definitions, whether or not in footnotes, may be retrieved by name through the index; and those with numbers are listed in order at page xvii.

Because much of the background material is separated from the main text, the need for more advanced readers to break out of the narrative should be reduced. We suggest that on first reading it is better to consult the footnotes only when there is a term that appears to require definition — otherwise the many cross-references they contain may prove distracting, as they are designed for "non-linear" browsing once the main ideas have already been assimilated.

Part I, *Probabilistic guarded commands*, gives enough introduction to the probabilistic logic to prove properties of small programs such as the one earlier, for example at the level of an undergraduate course for Formal-Methods-inclined students that explains "what to do" but not necessarily "why it is correct to do that." These would be people who need to understand how to reason about programs (and why), but would see the techniques as intellectual tools rather than as objects of study in their own right.

We have included many small examples to serve as models for the approach (they are indexed under *Programs*), and there are several larger case studies (for example in Chap. 3).

Part II, *Semantic structures*, develops in detail the mathematics on which the probabilistic logic is built and with which is it justified. That is, whereas the earlier sections present and illustrate the new reasoning techniques, this part shows where they have come from, why they have the form they do and — crucially — why they are correct.

That last point is especially important for students intending to do research in logic and semantics, as it provides a detailed and extended worked example of the fundamental issue of proving reasoning techniques *themselves* to be correct (more accurately, "valid"), a higher-order concept than the more familiar theme of the previous part in which we presented the techniques *ex cathedra* and used them to verify particular programs.

This part would thus be suitable for an advanced final-year undergraduate or first-year graduate course, and would fit in well with other material on programming semantics. It defines and illustrates the use of many of the standard tools of the subject: lattices, approximation orders, fixed points, semantic injections and retractions *etc.*

Part III, *Advanced topics*, concentrates on more exotic methods of specification and design, in this case probabilistic temporal/modal logics. Its final chapter, for example, contains material only recently discovered and leads directly into an up-to-date research area. It would be suitable for graduate students as an introduction to this specialised research community.

Part IV includes appendices collecting material that either leads away from the main exposition — *e.g.* alternative approaches and why we have not taken them — or supports the text at a deeper level, such as some of the more detailed proofs.

It also contains a short list of algebraic laws that demonic/probabilistic program fragments satisfy, generated mainly by our needs in the examples and proofs of earlier sections. An interesting research topic would be a more systematic elaboration of that list with a view to incorporating it into probabilistic Kleene- or omega algebras for distributed computations.

Overall, readers seeking an introduction to probabilistic formal methods could follow the material in order from the beginning. Those with more experience might instead sample the first chapter from each part, which would give an indication of the scope and flavour of the approach generally.

Original sources

Much of the material is based on published research, done with our colleagues, in conference proceedings and journal articles; but here it has been substantially updated and rationalised — and we have done our best to bring the almost ten years' worth of developing notation into a uniform state.

For self-contained presentations of the separate topics, and extra background, readers could consult our earlier publications as shown overleaf.

At the end of each chapter we survey the way in which our ideas have been influenced by — and in some cases adopted from — the work of other researchers, and we indicate some up-to-date developments.

Acknowledgements

Our work on probabilistic models and logic was carried out initially at the University of Oxford, together with Jeff Sanders and Karen Seidel and with the support of the UK's *Engineering and Physical Sciences Research Council* (the EPSRC) during two projects led by Sanders and Morgan over the years 1994–2001.

Morgan spent sabbatical semesters in 1995–6 at the University of Utrecht, as the guest of S. Doaitse Swierstra, and at the University of Queensland and the Software Verification and Research Centre (SVRC), as the guest of David Carrington and Ian Hayes. The foundational work the EPSRC projects produced during that period — sometimes across great distances — benefited from the financial support of those institutions but especially from the academic environment provided by the hosts and by the other researchers who were receptive to our initial ideas [MMS96].

Ralph Back at Åbo Akademi hosted our group's visit to Turku for a week in 1996 during which we were able to explore our common interests in refinement and abstraction as it applied to the new domain; that led later to a three-month visit by Elena Troubitsyna from the *Turku Center for Computer Science* (TUCS), to our group in Oxford in 1997, and contributed to what has become Chap. 4 [MMT98].

David Harel was our host for a two-week visit to Israel in 1996, during which we presented our ideas and benefited from the interaction with researchers there.

Chapters' dependence on original sources

Chapter 1	*see*	[MM99b, SMM, MMS00]
Chapter 2	*see*	[Mor96, MMS00]
Chapter 3	*see*	[MM99b]
Chapter 4	*see*	[MMT98]
Chapter 5	*see*	[MMS96]
Chapter 6		*is new material*
Chapter 7	*see*	[Mor96, MM01b]
Chapter 8	*see*	[MM01a]
Chapter 9	*see*	[MM97]
Chapter 10	*see*	[MM99a]
Chapter 11	*see*	[MM02]

The sources listed opposite are in chronological order of writing, thus giving roughly the logical evolution of the ideas.

Subsequently we have continued to work with Sanders and with Ken Robinson, Thai Son Hoang and Zhendong Jin, supported by the *Australian Research Council* (ARC) over the (coming) years 2001–8 in their *Large Grant* and *Discovery* programmes, at the Universities of Macquarie and of New South Wales.

Joe Hurd from the Computer Laboratory at Cambridge University visited us in 2002, with financial assistance from Macquarie University; and Orieta Celiku was supported by TUCS when she visited in 2003. Both worked under McIver's direction on the formalisation of *pGCL*, and its logic, in the mechanised logic *HOL*.

Hoang, Jin and especially Eric Martin have helped us considerably with their detailed comments on the typescript; also Ralph Back, Ian Hayes, Michael Huth, Quentin Miller and Wayne Wheeler have given us good advice. Section B.1 on the algebraic laws satisfied by probabilistic programs has been stimulated by the work (and the critical eyes) of Steve Schneider and his colleagues at Royal Holloway College in the U.K.

We thank the members of IFIP Working Groups 2.1 and 2.3 for their many comments and suggestions.

Annabelle McIver — *LRI* Paris,
Carroll Morgan — May 2004

In memoriam AJMvG

List of sources in order of writing

[MMS96] C.C. Morgan, A.K. McIver, and K. Seidel. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems*, 18(3):325–53, May 1996.

[Mor96] C.C. Morgan. Proof rules for probabilistic loops. In He Jifeng, John Cooke, and Peter Wallis, editors, *Proceedings of the BCS-FACS 7th Refinement Workshop*, Workshops in Computing. Springer Verlag, July 1996.

[MM01b] A.K. McIver and C.C. Morgan. Partial correctness for probabilistic programs. *Theoretical Computer Science*, 266(1–2):513–41, 2001.

[SMM] K. Seidel, C.C. Morgan, and A.K. McIver. Probabilistic imperative programming: a rigorous approach. Extended abstract appears in Groves and Reeves [GR97], pages 1–2.

[MMT98] A.K. McIver, C.C. Morgan, and E. Troubitsyna. The probabilistic steam boiler: a case study in probabilistic data refinement. In J. Grundy, M. Schwenke, and T. Vickers, editors, *Proc. International Refinement Workshop, ANU, Canberra*, Discrete Mathematics and Computer Science, pages 250–65. Springer Verlag, 1998.

[MM01a] A.K. McIver and C.C. Morgan. Demonic, angelic and unbounded probabilistic choices in sequential programs. *Acta Informatica*, 37:329–54, 2001.

[MM99b] C.C. Morgan and A.K. McIver. *pGCL*: Formal reasoning for random algorithms. *South African Computer Journal*, 22, March 1999.

[MM97] C.C. Morgan and A.K. McIver. A probabilistic temporal calculus based on expectations. In Groves and Reeves [GR97], pages 4–22.

[MM99a] C.C. Morgan and A.K. McIver. An expectation-based model for probabilistic temporal logic. *Logic Journal of the IGPL*, 7(6):779–804, 1999.

[MMS00] A.K. McIver, C.C. Morgan, and J.W. Sanders. Probably Hoare? Hoare probably! In J.W. Davies, A.W. Roscoe, and J.C.P. Woodcock, editors, *Millennial Perspectives in Computer Science*, Cornerstones of Computing, pages 271–82. Palgrave, 2000.

[MM02] A.K McIver and C.C. Morgan. Games, probability and the quantitative μ-calculus qMu. In *Proc. LPAR*, volume 2514 of *LNAI*, pages 292–310. Springer Verlag, 2002.

Contents

List of definitions *etc.*

Definitions and notations for standard mathematical concepts are given in footnotes, rather than in the main text, so that they do not interrupt the flow. They can be found directly, via the index, where they are indicated by bold-face page references. Thus for example at "fixed point, least" we find that the *least fixed-point* and its associated "μ notation" are defined in Footnote 33 on p. 21, and a second form of the μ notation is defined in Footnote 32 on p. 102.

Part I

Probabilistic guarded commands and their refinement logic

1

Introduction to *pGCL*: Its logic and its model

1.1 Sequential program logic

Since the mid-1970's, any serious student of rigorous program development will have encountered "assertions about programs" — they are predicates which, when inserted into program code, are supposed to be "true at that point of the program." Formalised — *i.e.* made into a logic — they look like either

$$\left.\begin{array}{llr} & \{pre\}\ prog\ \{post\} & \text{Hoare-style} \\ \text{or} & pre \Rrightarrow wp.prog.post\ , & \text{Dijkstra-style} \end{array}\right\} \quad (1.1)$$

in each case meaning "from any state satisfying precondition *pre*, the sequential program *prog* is guaranteed to terminate in a state satisfying postcondition *post*." [1] Formulae *pre* and *post* are written in first-order predicate logic over the program variables, and *prog* is written in a sequential programming language. Often Dijkstra's *Guarded Command Language* [Dij76], called *GCL*, is used in simple expositions like this one, since it contains just the essential features, and no clutter.

A conspicuous feature of Dijkstra's original presentation of guarded commands was the novel "demonic" choice. He explained that it arose naturally if one developed programs hand-in-hand with their proofs of correctness: if a single specification admitted say two implementations, then a third possibility was program code that seemed to choose unpredictably between the two. Yet in its pure form, where for example

$$prog \quad \sqcap \quad prog' \qquad (1.2)$$

is a program that can unpredictably behave either as *prog* or as $prog'$, this "demonic" nondeterminism seemed at first — to some — to be an unnecessary and in fact gratuitously confusing complication. Why would anyone ever want to introduce unpredictability *deliberately*? Programs are unpredictable enough already.

If one really wanted programs to behave in some kind of "random" way, then more useful surely would be a construction like the

$$prog \quad {}_{\frac{1}{2}}\oplus \quad prog' \qquad (1.3)$$

that behaves as *prog* on half of its runs, and as $prog'$ on the other half. Of course on any *particular* run the behaviour is unpredictable, and even over many runs the proportions will not necessarily be exactly "50/50" — but over a long enough period one will find approximately equal evidence of each behaviour.

A logic and a model for programs like (1.3) was in fact provided in the early 1980's [Koz81, Koz85], where in the "Kozen style" the pre- and post-formulae became real- rather than Boolean functions of the state, and $\sqcap$ was replaced by ${}_p\oplus$ in the programming language. Those logical statements

[1] We will use the Dijkstra-style.

(1.1) now took on a more general meaning, that "if program *prog* is run many times from the same initial state, the average value of *post* in the resulting final states is at least the actual value that *pre* had in the initial state." Naturally we are relying on the expressions' *pre* and *post* having real- rather than Boolean type when we speak of their average, or *expected* value.

The original — *standard*, we call it — Boolean logic was still available of course via the embedding $\mathsf{false}, \mathsf{true} \mapsto 0, 1$.

Dijkstra's demonic $\sqcap$ was not so easily discarded, however. Far from being "an unnecessary and confusing complication," it is the very basis of what is now known as *refinement* and *abstraction* of programs. (The terms are complementary: an implementation refines its specification; a specification abstracts from its implementation.) To specify "set r to a square-root of s" one could write *directly in the programming language GCL*

$$r{:=}\ -\sqrt{s} \quad \sqcap \quad r{:=}\ \sqrt{s}\ , \quad ^{2} \tag{1.4}$$

something that had never been possible before. This explicit, if accidental, "programming feature" caught the tide that had begun to flow in that decade and the following: the idea that specifications and code were merely different ways of describing the same thing (as advocated by Abrial, Hoare and others; making an early appearance in Back's work [Bac78] on what became the Refinement Calculus [Mor88b, Bac88, Mor87, Mor94b, BvW98]; and as found at the heart of specification and development methods such as *Z* [Spi88] and *VDM* [Jon86]).

Unfortunately, *probabilistic* formalisms were left behind, and did not embrace the new idea: *replacing* $\sqcap$ by ${}_p\oplus$, they lost demonic choice; without demonic choice, they lost abstraction and refinement; and without those, they had no nontrivial path from specification to implementation, and no development calculus or method.

---

[2] Admittedly this is a rather clumsy notation when compared with those designed especially for specification, *e.g.*

$r{:}\,[r^2 = s]$	a specification statement (Back, Morgan, Morris)
$(r')^2 = s$	(the body of) a *Z* schema (Abrial, *Oxford*)
$\overrightarrow{r}^2 = \overleftarrow{s}$	*VDM* (Bjørner, Jones)
any r' **with** $(r')^2 = s$ **then** $r{:=}\ r'$ **end**	a generalised substitution (Abrial)

But the point is that the specification could be written in a "programming language" at all: it was beginning to be realised that there was no reason to distinguish the meanings of specifications and of programs (a point finally crystallised in the subtitle *Assigning Programs to Meanings* of Abrial's book [Abr96a], itself a reference 30 years further back to Floyd's paper [Flo67] where it all began).

To have a probabilistic *development method*, we need both $\sqcap$ and ${}_p\oplus$ — we cannot abandon one for the other. Using them together, we can for example describe "flip a nearly fair coin" as

$$c{:=}\ \mathsf{heads}\ {}_{0.49}\oplus\ \mathsf{tails} \quad \sqcap \quad c{:=}\ \mathsf{heads}\ {}_{0.51}\oplus\ \mathsf{tails}\ .$$

What we are doing here is specifying a coin which is within 1% of being fair — just as well, since perfect ${}_{0.5}\oplus$ coins do not exist in nature, and so we could never implement a specification that required one.[3] This program abstracts, slightly, from the precise probability of heads or tails.

In this introduction we will see how the seminal ideas of Floyd, Hoare, Dijkstra, Abrial and others can be brought together and replayed in the probabilistic context suggested by Kozen, and how the milestones of sequential program development and refinement — the concepts of

- program assertions;
- loop invariants;
- loop variants;
- program algebra (*e.g.* monotonicity and conjunctivity)

— can be generalised to include probability. Our simple programming language will be Dijkstra's, but with ${}_p\oplus$ added and — crucially — demonic choice $\sqcap$ retained: we call it *pGCL*.

Section 1.2 gives a brief overview of *pGCL* and its use of so-called *expectations* rather than predicates in its accompanying logic; Section 1.3 then supplies operational intuition by relating *pGCL* operationally to a form of gambling game. (The rigorous operational semantics is given in Chap. 5, and a deeper connection with games is given in Chap. 11.) Section 1.4 completes the background by reviewing elementary probability theory.

Section 1.5 gives the precise syntax and expectation-transformer semantics of *pGCL*, using the infamous "Monty Hall" game as an example. Finally, in Sec. 1.6 we make our first acquaintance with the algebraic properties of *pGCL* programs.

Throughout we write $f.x$ instead of $f(x)$ for *function application* of f to argument x, with left association so that $f.g.x$ is $(f(g))(x)$; and we use "$:=$" for *is defined to be*. For *syntactic substitution* we write $expr\ \langle var \mapsto term \rangle$

[3] That means that probabilistic formalisms without abstraction in their specifications must introduce probability into their refinement operator if they are to be of any practical use: writing for example $prog \sqsubseteq_{0.99} prog'$ can be given a sensible meaning even if the probability in *prog* is exact [DGJP02, vBMOW03, Yin03]. But we do not follow that path here.

to indicate replacing *var* by *term* in *expr*. We use "overbar" to indicate *complement* both for Booleans and probabilities: thus $\overline{\mathsf{true}}$ is false, and $\overline{p}$ is $1-p$.

1.2 The programming language *pGCL*

We'll use *square brackets* $[\cdot]$ to convert Boolean-valued predicates to arithmetic formulae which, for reasons explained below, we call *expectations*. Stipulating that $[\mathsf{false}]$ is zero and $[\mathsf{true}]$ is one makes $[P]$ in a trivial sense the probability that a given predicate P holds: if false, it holds with probability zero; if true, it holds with probability one.[4]

For our first example, consider the simple program

$$x{:=}\ -y \quad {}_{\frac{1}{3}}{\oplus} \quad x{:=}\ +y \tag{1.5}$$

over integer variables $x, y{:}\,\mathbb{Z}$, using the new construct ${}_{\frac{1}{3}}{\oplus}$ which we interpret as "choose the left branch $x{:=}\ -y$ with probability $1/3$, and choose the right branch with probability $1-1/3$."

Recall [Dij76] that for any predicate *post* over *final* states, and a standard command *prog*,[5] the "weakest precondition" predicate *wp.prog.post* acts over *initial* states: it holds just in those initial states from which *prog* is guaranteed to reach *post*. Now suppose *prog* is probabilistic, as Program (1.5) is; what can we say about the *probability* that *wp.prog.post* holds in some initial state?

It turns out that the answer is just *wp.prog.*$[post]$, once we generalise *wp.prog* to expectations instead of predicates. For that, we begin with the two definitions [6]

$$\begin{array}{rcl} wp.(x{:=}\ E).postE & := & \text{“}postE \text{ with } x \text{ replaced} \\ & & \text{everywhere by } E\text{”}\ ^{7} \end{array} \tag{1.6}$$

$$\begin{array}{rcl} wp.(prog\ {}_{p}{\oplus}\ prog').postE & := & p * wp.prog.postE \\ & & +\ \overline{p} * wp.prog'.postE\ , \end{array} \tag{1.7}$$

in which *postE* is an expectation, and for our example program we ask *what is the probability that the predicate "the final state will satisfy $x \geq 0$" holds in some given initial state of the program (1.5)?*

To find out, we calculate *wp.prog.*$[post]$ using the definitions above; that is

---

[4] Note that this nicely complements our "overbar" convention, because for any predicate P the two expressions $\overline{[P]}$ and $[\overline{P}]$ are therefore the same.

[5] Throughout we use STANDARD to mean "non-probabilistic."

[6] Here we are defining the language as we go along; but all the definitions are collected together in Fig. 1.5.3 (p. 26).

[7] In the usual way, we take account of free and bound variables, and if necessary rename to avoid variable capture.

$$
\begin{array}{lll}
 & wp.(x{:=}\ -y\ {}_{\frac{1}{3}}{\oplus}\ x{:=}\ +y).[x \geq 0] & \\
\equiv^{8} & \quad (1/3) * wp.(x{:=}\ -y).[x \geq 0] & \text{using (1.7)} \\
 & + \quad (2/3) * wp.(x{:=}\ +y).[x \geq 0] & \\
\equiv & (1/3)\,[-y \geq 0] + (2/3)\,[+y \geq 0] & \text{using (1.6)} \\
\equiv & [y < 0]/3 \ + \ [y = 0] \ + \ 2\,[y > 0]/3\ . & \text{using arithmetic}
\end{array}
$$

Thus our answer is the last arithmetic formula above, which we call a "pre-expectation" — and the probability we seek is found by reading off the formula's value for various initial values of y, getting

$$
\begin{array}{lrcl}
\text{when } y < 0, & 1/3 + 0 + 2(0)/3 & = & 1/3 \\
\text{when } y = 0, & 0/3 + 1 + 2(0)/3 & = & 1 \\
\text{when } y > 0, & 0/3 + 0 + 2(1)/3 & = & 2/3 \quad .
\end{array}
$$

Those results indeed correspond with our operational intuition about the effect of ${}_{\frac{1}{3}}{\oplus}$.

For our second example we illustrate abstraction from probabilities: a demonic version of Program (1.5) is much more realistic in that we set its probabilistic parameters only within some tolerance. We say informally (but still precisely) that

- $x{:=}\ -y$ is to be executed with probability *at least* 1/3,
- $x{:=}\ +y$ is to be executed with probability *at least* 1/4 and
- it is certain that one or the other will be executed.

(1.8)

Equivalently we could say that alternative $x{:=}\ -y$ is executed with probability between 1/3 and 3/4, and that otherwise $x{:=}\ +y$ is executed (therefore with probability between 1/4 and 2/3).

With demonic choice we can write Specification (1.8) as

$$
x{:=}\ -y\ {}_{\frac{1}{3}}{\oplus}\ x{:=}\ +y \quad \sqcap \quad x{:=}\ -y\ {}_{\frac{3}{4}}{\oplus}\ x{:=}\ +y\ , \tag{1.9}
$$

because we do not know or care whether the left or right alternative of $\sqcap$ is taken — and it may even vary from run to run of the program, resulting in an "effective" ${}_p{\oplus}$ with p somewhere between the two extremes.[9]

[8]Later we explain the use of "$\equiv$" rather than "$=$".

[9]We will see later that a convenient notation for (1.9) uses the abbreviation

$$
prog\ {}_p{\oplus}_q\ prog' \quad := \quad prog\ {}_p{\oplus}\ prog' \ \sqcap\ prog'\ {}_q{\oplus}\ prog\ ;
$$

we would then write it $x{:=}\ -y\ {}_{\frac{1}{3}}{\oplus}_{\frac{1}{4}}\ x{:=}\ +y$, or even $x{:=}\ -y\ {}_{\frac{1}{3}}{\oplus}_{\frac{1}{4}}\ +y$.

To treat Program (1.9) we need a third definition,

$$wp.(prog \sqcap prog').postE \quad := \quad wp.prog.postE \;\mathsf{min}\; wp.prog'.postE \;, \tag{1.10}$$

using min because we regard demonic behaviour as attempting to make the achieving of *post* as <u>im</u>probable as it can. Repeating our earlier calculation (but more briefly) gives this time

$$
\begin{array}{lll}
 & wp.(\text{ Program (1.9) }).[x \geq 0] & \\
\equiv & \quad [y \leq 0]/3 + 2\,[y \geq 0]/3 & \text{using (1.6), (1.7), (1.10)} \\
 & \mathsf{min} \quad 3\,[y \leq 0]/4 + [y \geq 0]/4 & \\
\equiv & [y < 0]/3 \;+\; [y = 0] \;+\; [y > 0]/4 \;. & \text{using arithmetic}
\end{array}
$$

Our interpretation has become

- When y is initially negative, a demon chooses the left branch of $\sqcap$ because that branch is more likely (2/3 *vs.* 1/4) to execute $x := +y$ — the best we can say then is that $x \geq 0$ will hold with probability at least 1/3.
- When y is initially zero, a demon cannot avoid $x \geq 0$ — either way the probability of $x \geq 0$ finally is one.
- When y is initially positive, a demon chooses the right branch because that branch is more likely to execute $x := -y$ — the best we can say then is that $x \geq 0$ finally with probability at least 1/4.

The same interpretation holds if we regard $\sqcap$ as abstraction instead of as run-time demonic choice. Suppose Program (1.9) represents some mass-produced physical device and, by examining the production method, we have determined the tolerance (1.8) we can expect from a particular factory. If we were to buy one from the warehouse, all we could conclude about its probability of establishing $x \geq 0$ is just as calculated above.

Refinement is the converse of abstraction: we have

Definition 1.2.1 PROBABILISTIC REFINEMENT For two programs *prog*, *prog′* we say that *prog′* is a refinement of *prog*, written $prog \sqsubseteq prog'$, whenever for all post-expectations *postE* we have

$$wp.prog.postE \quad \Rrightarrow \quad wp.prog'.postE \tag{1.11}$$

We use the symbol $\Rrightarrow$ for $\leq$ (extended pointwise) between expectations, which emphasises the similarity between probabilistic- and standard refinement.[10] □

[10] We are aware that "$\Rrightarrow$" looks more like "$\geq$" than it does "$\leq$"; but for us its resemblance to "$\Rightarrow$" is the important thing. . . .

From (1.11) we see that in the special case when expectation *postE* is an embedded predicate $[post]$, the meaning of $\Rrightarrow$ ensures that a refinement $prog'$ of *prog* is at least as likely to establish *post* as *prog* is.[11] That accords with the usual definition of refinement for standard programs — for then we know $wp.prog.[post]$ is either zero or one, and whenever *prog* is certain to establish *post* (whenever $wp.prog.[post] \equiv 1$) we know that $prog'$ also is certain to do so (because then $1 \Rrightarrow wp.prog'.[post]$).

For our third example we prove a refinement: consider the program

$$x\!:= -y \quad {}_{\frac{1}{2}}\!\oplus \quad x\!:= +y \ , \tag{1.12}$$

which clearly satisfies Specification (1.8); thus it should refine Program (1.9), which is just that specification written in *pGCL*. With Definition (1.11), we find for any *postE* that

$$\begin{array}{lll}
 & wp.(\text{ Program (1.12) }).postE & \\
\equiv & \quad wp.(x\!:= -y).postE/2 & \text{definition } {}_p\!\oplus\text{, at (1.7)} \\
 & + \quad wp.(x\!:= +y).postE/2 & \\
\equiv & postE^-/2 \quad + \quad postE^+/2 & \text{introduce abbreviations} \\
\equiv & \quad (3/5)(postE^-/3 + 2postE^+/3) & \text{arithmetic} \\
 & + \quad (2/5)(3postE^-/4 + postE^+/4) & \\
\Leftarrow & \qquad postE^-/3 + 2postE^+/3 & \text{any linear combination exceeds } \mathsf{min} \\
 & \mathsf{min} \quad 3postE^-/4 + postE^+/4 & \\
\equiv & wp.(\text{ Program (1.9) }).postE \ . &
\end{array}$$

The refinement relation (1.11) is indeed established for the two programs.

The introduction of 3/5 and 2/5 in the third step can be understood by noting that demonic choice $\sqcap$ can be implemented by any probabilistic choice whatever: in this case we used ${}_{\frac{3}{5}}\!\oplus$. Thus a proof of refinement using program algebra might read

$$\begin{array}{ll}
 & \text{Program (1.12)} \\
= & x\!:= -y \quad {}_{\frac{1}{2}}\!\oplus \quad x\!:= +y
\end{array}$$

···[10]Similar conflicts of interest arise when logicians use "$\supset$" for *implies* although, interpreted set-theoretically, *implies* is in fact "$\subseteq$". And then there is "$\sqsubseteq$" for refinement, which corresponds to "$\supseteq$" of behaviours.

[11]We see later in this chapter, however, and in Sec. A.1, that it is not sound to consider only post-expectations *postE* of the form $[post]$ in Def. 1.2.1: it is necessary for refinement, *but not sufficient*, that $prog'$ be at least as likely to establish any postcondition *post* as *prog* is.

$$
\begin{array}{llll}
= & \quad (x{:=}\ -y \ \ {}_{\frac{1}{3}}\!\oplus\ \ x{:=}\ +y) & & \text{Sec. B.1 Law 4} \\
 & {}_{\frac{3}{5}}\!\oplus \quad (x{:=}\ -y \ \ {}_{\frac{3}{4}}\!\oplus\ \ x{:=}\ +y) & & \\
 & & & \\
\sqsupseteq & \quad x{:=}\ -y \ \ {}_{\frac{1}{3}}\!\oplus\ \ x{:=}\ +y & & (\sqcap) \sqsubseteq ({}_p\!\oplus) \text{ for any } p\ ^{12} \\
 & \sqcap \quad x{:=}\ -y \ \ {}_{\frac{3}{4}}\!\oplus\ \ x{:=}\ +y & & \\
 & & & \\
= & \text{Program (1.9)}\ . & &
\end{array}
$$

1.3 An informal computational model: *pGCL* describes gambling

We now use a simple card-and-dice game as an informal introduction to the computational model for *pGCL*, to support the intuition for probabilistic choice, demonic choice and their interaction. To start with, we consider the simplest case: non-looping programs without $\sqcap$ or ${}_p\oplus$.

1.3.1 The standard game

Imagine we have a board of numbered squares, and a selection of numbered cards laid on it with at most one card per square; winning squares are indicated by coloured markers. The squares are the program states; the program is the pattern of numbered cards; the coloured markers indicate the postcondition.

To play the game

> An initial square is chosen (according to certain rules which do not concern us); subsequently
>
> - if the square contains a card the card is removed, and play continues from the square whose number appeared on the card, and
> - if the square does not contain a card, the game is over.
>
> When the game is over the player has won if his final square contains a marker — otherwise he has lost.

This simple game is deterministic: any initial state always leads to the same final state. And because the cards are removed after use it is also guaranteed to terminate, if the board is finite. It is easily generalised however to include other features of standard programs:

---

[12] By $(\sqcap) \sqsubseteq ({}_p\oplus)$ we mean that for all $prog, prog'$ we have

$$prog \sqcap prog' \quad \sqsubseteq \quad prog\ {}_p\!\oplus\ prog' ,$$

which is an instance of our Law 7 given on p. 323, in Sec. B.1 on program algebra.

looping If the cards are *not* removed after use, the game can "loop." A looping-forever player loses.

aborting If a card reads *go to jail*, the program is said to "abort" and the player can be sent to any square whatever, including a special supplementary "jail" square from which there is no escape. A jailed player loses.

demonic nondeterminism If each square can contain several cards, face-down, and the rules are modified so that the next state is determined by choosing just one of them "blind," then play is nondeterministic. Taking the demonic (pessimistic) view, the player should expect to lose unless he is guaranteed to reach a winning position no matter which blind choices he makes.

In the standard game, for each (initial) square one can examine the cards before playing to determine whether a win is guaranteed from there. But once the game has started, the cards are turned face-down.

The set of squares from which a win is guaranteed is the weakest precondition.[13]

1.3.2 The probabilistic game

Suppose now that each card contains not just one but, rather, a list of successor squares, and the choice from the list is made by rolling a die. In this deterministic game,[14] play becomes a succession of die rolls, taking the player from square to square; termination (no card) and winning (marker) are defined as before.

When squares can contain several cards face down, each with a separate list of successors to be resolved by die roll, we are dealing with probability and demonic nondeterminism together: first the card is chosen "blind" (*i.e.* demonically); the card is turned over and a die roll (probability) determines which of its listed alternatives to take.

In the probabilistic game one can ask for the *greatest guaranteed probability* of winning; as in the standard case, the prediction will vary depending on the initial square. (It's because of demonic nondeterminism, as illustrated below, that the probability might be only a lower bound.)

---

[13] A glance at Fig. 6.7.1 (p. 173) will show where we are headed in the visualisation of probabilistic preconditions!

[14] Note that we still call this game "deterministic," in spite of the probabilistic choices, and there are good mathematical reasons for doing so. (In Chap. 5, for example, we see that such programs are maximal in the refinement order.) An informal justification is that deterministic programs are those with repeatable behaviours and, even for probabilistic programs, the output *distribution* is repeatable (to within statistical confidence measures) provided the program contains no demonic choice; see *e.g.* p. 135.

In Fig. 1.3.1 is an example game illustrating some of the above points. The greatest guaranteed probability of winning from initial state 0 is only 1/2, in spite of the fact that the player can win every time if he is lucky enough to choose the first card in the pile; but he might be unlucky enough never to choose the first card, and we must assume the worst.

1.3.3 Expected winnings in the probabilistic game

For standard programs, the computational model of execution supports a complementary, "logical" view — given a set of final states (the postcondition) we can examine the program to determine the largest set of initial states (the weakest precondition) from which execution of the program is guaranteed to reach the designated final states. The sets of states are *predicates*, and the program is being regarded as a predicate *transformer*.

Regarding sets of states as characteristic functions (from the state space into $\{0, 1\}$), we generalise to "probabilistic predicates" by extending the range of those functions to all of $\mathbb{R}_{\geq}$, the non-negative reals.[15]

Probabilistic programs become functions from probabilistic postconditions to probabilistic weakest preconditions — we call them *post-expectations* and *greatest pre-expectations*. The corresponding generalisation in the game is as follows.

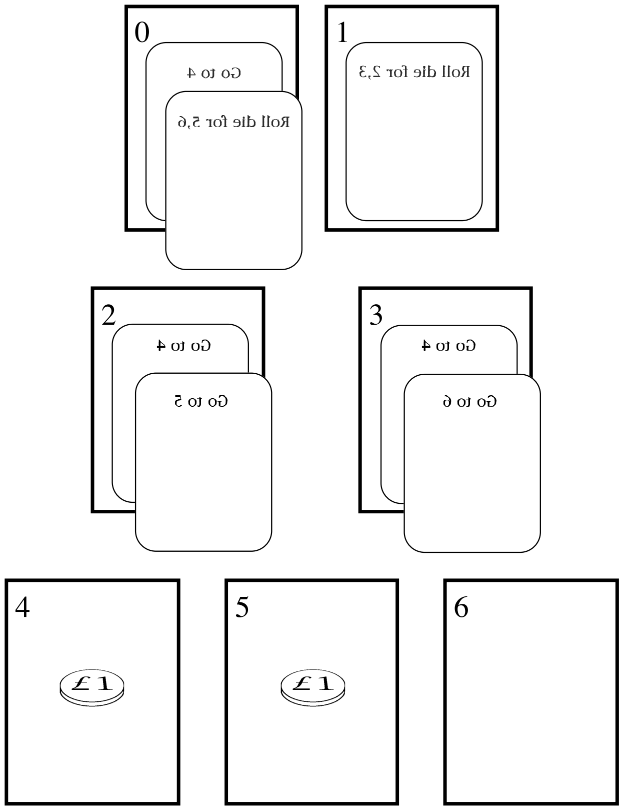
Rather than placing winning markers on the board, we place *money* — rather than strictly winning or losing, the player simply keeps whatever money he finds in his final square. In Fig. 1.3.2 we show the effect of translating our original game. In fact, not much changes: the *probability* of winning (in Fig. 1.3.1) translates into the equivalent *expected payoff* (Fig. 1.3.2) as the corresponding fraction of £1, illustrating this important fact:

> The expected value of a characteristic function over a distribution is the same as the probability assigned to the set that function describes.

Thus using expectations is at least as general as using probabilities explicitly, since we can always restrict ourselves to $\{0, 1\}$-valued functions from which probabilities are then recovered.

For probabilistic programs, the operational interpretation of execution thus supports a "logical" view also — given a function from final states to $\mathbb{R}_{\geq}$ (the post-expectation) one can examine the program beforehand to determine for each initial state the minimum expected (or "average") win when the game is played repeatedly from there (the greatest pre-expectation) — also therefore a function from states to $\mathbb{R}_{\geq}$.

[15] In later chapters we will be more precise about the range of expectations, requiring them in particular to be *bounded above*.



To play from a square, you first pick one of the face-down cards. (In the diagram, we are seeing what's on the cards with our *x-ray* vision.) Then you roll a die to choose one of the alternatives on the card. (In this case the die is two-sided, *i.e.* it is a coin.)
As special cases, a *standard* step (non-probabilistic) has only one alternative per card, but possibly many cards; and a *deterministic* step has only one card, but possibly many alternatives on it. A standard and deterministic step has one card, and only one alternative.

The winning final positions — the postcondition — are the states $\{4, 5\}$, marked with a £1 coin. From initial state 2 a win is guaranteed; from state 0 or 1 the minimum guaranteed probability of winning is $1/2$; from state 3 the minimum probability is zero, since the second card might be chosen every time.

The probabilities are summarised in Fig. 1.3.2.

Figure 1.3.1. Card-and-dice game operational semantics for *pGCL*

The *post-expectation*:

Final state	0	1	2	3	4	5	6
Payoff awarded if this state reached	0	0	0	0	£1	£1	0

The probability of winning (ending on a £1) (from Fig. 1.3.1):

Initial state	0	1	2	3	4	5	6
Greatest guaranteed probability of winning	1/2	1/2	1	0	1	1	0

The *greatest pre-expectation*:

Initial state	0	1	2	3	4	5	6
Greatest guaranteed expected payoff	50p	50p	£1	0	£1	£1	0

Figure 1.3.2. A PROBABILISTIC AND NONDETERMINISTIC GAMBLING GAME

Since the functions are *expectations*, the program is being regarded as an *expectation transformer*.[16]

We are not limited to £1 coins for indicating postconditions — that is only an artefact of embedding standard postconditions into the probabilistic world. In general any amount of money can be placed in a square, and that is the key to allowing a smooth sequential composition of programs at the logical level — for if the program *game* of Fig. 1.3.2 were executed after some other program *prog*, the precondition of the two together with respect to the postcondition $\{4,5\}$ would be calculated by applying *wp.prog* to the *greatest pre-expectation* table for *game*. That is because sequential composition of programs becomes, as usual, functional composition of the corresponding transformers: we have

$$wp.(prog; game).\{4,5\} \quad := \quad wp.prog.(\overbrace{wp.game.\{4,5\}}^{\text{expected win table}})\ ,$$

and that table contains non-integer values (for example 50p).

Another reason for allowing arbitrary values in $\mathbb{R}_{\geq}$ is that using only standard postconditions ($\{0,1\}$-valued) — equivalently, using explicit probabilities (recall the important fact above) — is not discriminating enough when nondeterminism is present: certain programs are identified that should be distinguished, and the semantics becomes *non-compositional*. (See Sec. A.1 for why this happens.)

[16] For deterministic (yet probabilistic) programs, the card-game model and the associated transformers are essentially Kozen's original construction [Koz81, Koz85]. We have added demonic (and later angelic) nondeterminism.

1.4 Behind the scenes: elementary probability theory

In probability theory, an *event* is a subset of some given *sample space* S, so that the event is said to have occurred if the sampled value is in that set; a *probability distribution* Pr over the sample space is a function from its events into the closed interval $[0,1]$, giving for each event the probability of its occurrence. In the general case, for technical reasons, not necessarily all subsets of the sample space are events.[17]

In our case we consider countable sample spaces, and take every (sub-)set of S to be an event — and so we can regard a probability distribution more simply as a function from S directly to probabilities (rather than from its subsets). Thus $\mathrm{Pr}\colon S \to [0,1]$, and the probability of a more general event is now just the sum of the probabilities of its elements: we are using *discrete* distributions.[18]

A *random variable* X is a function from the sample space to the non-negative reals;[19] and the *expected value* $\mathrm{Exp}.X$ of that random variable is defined in terms of the (discrete) probability distribution Pr; we have the summation

$$\mathrm{Exp}.X \quad := \quad (\sum_{s \in S} \mathrm{Pr}.s * X.s) \;. \quad ^{20} \tag{1.13}$$

It represents the "average" value of $X.s$ over many repeated samplings of s according to the distribution Pr.[21]

In fact expected values can also be characterised without referring directly to an underlying probability distribution:

> If a function Exp is of type $(S \to \mathbb{R}_{\geq}) \to \mathbb{R}_{\geq}$, and it is
>
> **non-negative** so that $\mathrm{Exp}.X \geq 0$ for all $X\colon S \to \mathbb{R}_{\geq}$,
> **linear** so that for $X, Y\colon S \to \mathbb{R}_{\geq}$ and $c, d\colon \mathbb{R}_{\geq}$ we have
>
> $$\mathrm{Exp}.(c * X + d * Y) \quad = \quad c * \mathrm{Exp}.X + d * \mathrm{Exp}.Y$$

---

[17] This may occur if the sample space is uncountable, for example; the general technique for such cases involves σ-algebras [GS92]. See Footnote 7 on p. 297 for an example.

[18] The price paid for using discrete distributions is that there are some "everyday" situations we cannot describe, such as the uniform "continuous" distribution over the real interval $[0,1]$ that might be the result of the program "choose a real number x randomly so that $0 \leq x \leq 1$." We get away with it because no such program can be written in *pGCL* — at least, not at this stage.

[19] Footnote 12 on p. 134 gives a more generous definition.

[20] Although the parentheses may look odd around $\sum$ — we write $(\sum \cdots)$ rather than $\sum(\cdots)$ — we always indicate the scope of bound variables (like s) with explicit delimiters, since it helps to avoid errors when doing calculations.

[21] Our "important fact" (p. 13) is now stated "if X is the characteristic function of some event P, then $\mathrm{Exp}.X$ is the probability that event P will occur."

and **normalised** so that it satisfies $\text{Exp}.\underline{1} = 1$, where $\underline{1}$ is the constant function returning 1 for all arguments in S,

then it is an expectation over some probability distribution: it can be shown that it is expressible uniquely in the form (1.13) for some Pr.[22]

The relevance of the above is that our real-valued expressions over the state — what we are calling "expectations" — are random variables, and that the expression

$$wp.prog.postE\ , \tag{1.14}$$

as a function of *initial* values for the state variables, is a random variable as well. As a function of state variables, it is the expected value of the random variable *postE* (also a function of state variables, but those taken *after* execution) over the distribution of final states produced by executions of *prog*, and so

$$preE \quad \Rrightarrow \quad wp.prog.postE \tag{1.15}$$

says that *preE* gives in any initial state a lower bound for the expected value of *postE* in the final distribution reached via execution of *prog* begun in that initial state.

In general, we call random variables *post-expectations* when they are to be evaluated in a final state, and we call them *pre-expectations* when they are calculated as at (1.14). And, like pre- and postconditions in standard programs, if placed "between" two programs a single random variable is a post-expectation for the first and a pre-expectation for the second.

But how do *prog* and an initial state determine a distribution? In fact the underlying distributions are found on the cards of the game from Sec. 1.3 — the sample space is the set of squares, and each card gives an explicit distribution over that space. If we consider the deterministic game, and regard "make *one* move in the game" as a program in its own right, then we have a function from initial state to final distribution — the function taking a square to the card that square contains.[23] For any postcondition *postE* written, say, as an expression over names N of squares, and initial square N_0, the expression $wp.move.postE\ \langle N \mapsto N_0 \rangle$ is the expectation of *postE* over the distribution of square names given on the card found at N_0.

[22] It is a special case of the *Riesz Representation Theorem* which states, loosely speaking, that knowledge of the expectation (assumed to be given directly) of every random variable uniquely determines an underlying probability distribution. See for instance Feller [Fel71, p. 135].

[23] For nondeterministic programs we are thus considering a function from state to *sets* of distributions, from a square to the set of cards there; again we see the general computational model underlying the expectation-transformer semantics.

For example, in Figs. 1.3.1 and 1.3.2 we see the above features: program *move* is given by the layout of the cards (Fig. 1.3.1); and the resulting pre- and post-expectations are tabulated in Fig. 1.3.2. All three tables there are random variables over the state space $\{0, \cdots, 6\}$.

When we move to more general programs, we must relax the conditions that characterise expectations. If *prog* is possibly nonterminating — if it is recursive or contains **abort** — then *wp.prog.postE* may violate the normalisation condition $\text{Exp}.\underline{1} = 1$. However as a function which satisfies the first two conditions it can still be regarded as an expectation in a weak sense. That was shown by Kozen [Koz81] and later Jones [Jon90], who defined expectations with regard to "probability distributions" which may sum to less than one. Those are in fact a special case of Jones's *evaluations*,[24] and she gave conditions similar to the above for their existence [Jon90, p. 117].

Finally, if program *prog* is not deterministic then we move further away from elementary theory, because *wp.prog.postE* is no longer an expectation even in the weak sense: it not linear. It is still however the minimum of a set of expectations: if *prog* and $prog'$ are deterministic programs then $wp.(prog \sqcap prog').postE$ is the pointwise minimum of the two expectations *wp.prog.postE* and $wp.prog'.postE$. This definition is one of the main features of this approach.

Thus although linearity is lost, it is not gone altogether: we retain so-called *sub*-linearity,[25] which implies that for any $c_1, c_2 \colon \mathbb{R}_{\geq}$ and any program *prog* we still have

$$\begin{array}{ll} & wp.prog.(c_1 * postE_1 + c_2 * postE_2) \\ \Leftarrow & c_1 * wp.prog.postE_1 \quad + \quad c_2 * wp.prog.postE_2 \ . \end{array}$$

And clearly non-negativity continues to hold.

The characterisations of expectations given above for the simpler cases might suggest that non-negative and sublinear functionals uniquely determine a *set* of probability distributions — and, in Chap. 5, that is indeed shown to be the case: sublinearity is the key "healthiness condition" for expectation transformers.[26]

1.5 Basic syntax and semantics of *pGCL*

1.5.1 Syntax

Let *prog* range over programs and p over real number expressions taking values between zero and one inclusive; assume that x stands for a list of distinct variables, and *expr* for a list of expressions (of the same length as x

[24] She was working in a much more general context.
[25] The actual property is slightly more general than we give here; see Sec. 1.6.
[26] Halpern and Pucella [HP02] have recently studied similar properties.

where appropriate); and let the program *scheme* $\mathcal{C}$ be a program in which program *names* like *xxx* can appear. The syntax of *pGCL* is as follows:

$$\begin{array}{lll} \textit{prog} & := & \textbf{abort} \mid \textbf{skip} \mid x := E \mid \textit{prog}; \textit{prog} \mid \\ & & \textit{prog}\ {}_p\!\oplus\ \textit{prog} \mid \textit{prog} \sqcap \textit{prog} \mid \\ & & (\textbf{mu}\ \textit{xxx} \bullet \mathcal{C}) \end{array} \tag{1.16}$$

The first four constructs, namely **abort**, **skip**, assignment and sequential composition, are just the conventional ones [Dij76].

The remaining constructs are for probabilistic choice, nondeterministic choice and recursion: given p in the closed interval $[0,1]$ we write $\textit{prog}\ {}_p\!\oplus\ \textit{prog}'$ for the probabilistic choice between programs *prog* and *prog*′; they have probability p and $1{-}p$ respectively of being selected. In many cases p will be a constant, but in general it can be an expression over the state variables.

1.5.2 Shortcuts and "syntactic sugar"

For convenience we extend our logic and language with the following notations.

Boolean embedding — For predicate *pred* we write [*pred*] for the expectation "1 **if** *pred* **else** 0".[27]

Conditional — The conditional

$$\begin{array}{ll} & \textit{prog}\ \textbf{if}\ \textit{pred}\ \textbf{else}\ \textit{prog}' \\ \text{or} & \textbf{if}\ \textit{pred}\ \textbf{then}\ \textit{prog}\ \textbf{else}\ \textit{prog}'\ \textbf{fi}\ , \end{array}$$

chooses program *prog* (resp. *prog*′) if Boolean *pred* is true (resp. false). It is defined $\textit{prog}\ {}_{[\textit{pred}]}\!\oplus\ \textit{prog}'$.

If **else** is omitted then **else skip** is assumed. (See also the "hybrid" conditional of Sec. 3.1.2.)

Implication-like relations — For expectations *exp*, *exp*′ we write

$$\begin{array}{llll} \textit{exp} \Rrightarrow \textit{exp}' & \text{for} & \textit{exp}\ \text{is everywhere less than or equal to}\ \textit{exp}' \\ \textit{exp} \equiv \textit{exp}' & \text{for} & \textit{exp}\ \text{and}\ \textit{exp}'\ \text{are everywhere equal} \\ \textit{exp} \Lleftarrow \textit{exp}' & \text{for} & \text{exp is everywhere greater than or equal to exp}' \end{array}$$

We distinguish $\textit{exp} \Rrightarrow \textit{exp}'$ from $\textit{exp} \leq \textit{exp}'$ — the former is a statement *about* *exp* and *exp*′, thus true or false as a whole; the latter is itself a Boolean-valued expression over the state, possibly true in some states and false in others.[28] Similarly we regard $\textit{exp} = \textit{exp}'$ as

[27] We will not distinguish predicates from Boolean-valued expressions.

[28] Note that $\textit{exp} \Rightarrow \textit{exp}'$ is different again, in fact badly typed if *exp* and *exp*′ are expectations: one real-valued function cannot "imply" another.

true in just those states where *exp* and *exp*$'$ are equal, and false in the rest.
The closest standard equivalent of $\Rrightarrow$ is the entailment relation $\models$ between predicates[29] — and in fact $post \models post'$ exactly when $[post] \Rrightarrow [post']$, meaning that the "embedding" of $\models$ is $\Rrightarrow$.

Multi-way probabilistic choices — A probabilistic choice over N alternatives can be written horizontally

$$(prog_1 \; @ \; p_1 \mid \cdots \mid prog_N \; @ \; p_N)$$

or vertically

$$\left|\begin{array}{ll} prog_1 & @ \; p_1 \\ prog_2 & @ \; p_2 \\ \quad\vdots & \\ prog_N & @ \; p_N \end{array}\right.$$

in which the probabilities are enumerated and sum to no more than one.[30] We can also write a "probabilistic comprehension" $(\| \; i{:}\, I \bullet prog_i \; @ \; p_i)$ over some countable index set I. In general, we have

$$\begin{array}{ll} & wp.(prog_1 \; @ \; p_1 \mid \cdots \mid prog_N \; @ \; p_N).postE \\ := & p_1 * wp.prog_1.postE + \cdots + p_N * wp.prog_N.postE \; . \end{array}$$

It means "execute $prog_1$ with probability at least p_1, and $prog_2$ with probability at least $p_2 \ldots$"[31]
If the probabilities sum to 1 exactly, then it is a simple N-way probabilistic branch; if there is a deficit $1 - \Sigma_i p_i$, it gives the probability of aborting.
When all the programs $prog_i$ are assignments with the same left-hand side, say $x{:=}\; expr_i$, we write even more briefly

$$x{:=}\; (expr_1 \; @ \; p_1 \mid \cdots \mid \; expr_N \; @ \; p_N) \; .$$

Variations on $_p\oplus$ — By $prog \oplus_p prog'$ we mean $prog' \;_p\oplus\; prog$, and in general we write $prog \;_p\oplus_{p'}\; prog'$ for

$$\left|\begin{array}{lll} prog & @ & p \\ prog' & @ & p' \\ prog \sqcap prog' & @ & 1 - (p{+}p') \; , \end{array}\right.$$

the program that executes *prog* with probability at least p and $prog'$

[29] One predicate ENTAILS another, written $\models$, just when it implies the other in all states.
[30] See Sec. 4.3 for an example of the vertical notation.
[31] It is "at least p_i" because if the probabilities sum to less than one there will be an "aborting" component, which might behave like $prog_i$.

with probability at least p'; we assume $p + p' \leq 1$.
By ${}_{\geq p}\oplus$ we mean ${}_{p}\oplus{}_{0}$, and so on. (See also (B.3) on p. 328.)

Demonic choice — We write demonic choice between assignments to the same variable x as

$$\begin{array}{rl} & x{:}\in\ \{expr_1, expr_2, \cdots\}\ , \\ \text{or} & x{:}=\ expr_1 \sqcap expr_2 \sqcap \cdots\ , \end{array} \qquad (1.17)$$

in each case abbreviating $x{:}=\ expr_1 \sqcap x{:}=\ expr_2 \sqcap \cdots$. More generally we can write $x{:}\in expr$ or $x{:}\notin expr$ if $expr$ is set-valued, provided the implied choice is finite.[32]

Iteration — The construct (**mu** $xxx \bullet \mathcal{C}$) behaves as prescribed by the program context $\mathcal{C}$ except that it invokes itself recursively whenever it reaches a point where the program name xxx appears in $\mathcal{C}$. Then, in the usual way, iteration is a special case of recursion:

$$\begin{array}{ll} & \textbf{do}\ pred \rightarrow body\ \textbf{od} \\ := & (\textbf{mu}\ xxx \bullet\ (body\,;xxx)\ \textbf{if}\ pred\ \textbf{else skip})\ . \end{array} \quad ^{33} \qquad (1.18)$$

1.5.3 *Example of syntax: the "Monty Hall" game*

We illustrate the syntax of our language with the example program of Fig. 1.5.1. There are three curtains, labelled A, B and C, and a prize is hidden nondeterministically behind one of them, say *pc*. A contestant hopes to win the prize by guessing where it is hidden: he chooses randomly to

---

[32] None of our examples requires a choice from the empty set. We see later that the finiteness requirement is so that our programs will be continuous (Footnote 60 on p. 71); and in some cases — for example, the third and fourth statements of the program shown in Fig. 1.5.1 — we rely on type information for that finiteness.

[33] An equivalent but simpler formulation is given by the least fixed-point definition

$$wp.(\textbf{do}\ pred \rightarrow body\ \textbf{od}).R \quad := \quad (\mu Q \bullet\ wp.body.Q\ \textbf{if}\ pred\ \textbf{else}\ R)\ , \qquad (1.19)$$

which matches Dijkstra's original formulation more closely [Dij76]. But there is some technical work required to get between the two, as we explain later at (7.12). The expression on the right can be read

the *least* pre-expectation Q such that

$$Q \quad \equiv \quad wp.body.Q\ \textbf{if}\ pred\ \textbf{else}\ R\ ,$$

and is called a FIXED POINT because placing Q in the expression does not alter its value — this is the mathematical equivalent of "and the same again" when the loop returns to its starting point for potentially more iterations.

The "least," for us, means the *lowest* expectation — that reflects the view, appropriate for elementary sequential programming, that unending iteration should have little worth (in fact, zero). For standard programming, the order is $\mathsf{false} \leq \mathsf{true}$ so that taking the least fixed-point means adopting the view that an infinite loop does not establish any postcondition (*i.e.*, has precondition false).

A more discriminating treatment of unending computations is given in Part III.

$pc{:}\in\ \{A, B, C\}$;	Prize hidden behind curtain.
$cc{:}=\ (A\ @\ \frac{1}{3} \mid B\ @\ \frac{1}{3} \mid C\ @\ \frac{1}{3})$;	Contestant chooses randomly.
$ac{:}\notin\ \{pc, cc\}$;	Another curtain opened; it's empty.
$(cc{:}\notin \{cc, ac\})$ **if** *clever* **else skip**	Changes his mind — or not?

The three "curtain" variables ac, cc, pc are of type $\{A, B, C\}$.
Written in full, the first three statements would be

$$
\begin{array}{l}
pc{:}=\ A \sqcap pc{:}=\ B \sqcap pc{:}=\ C;\\
cc{:}=\ A\ {}_{\frac{1}{3}}\!\oplus (cc{:}=\ B\ {}_{\frac{1}{2}}\!\oplus\ cc{:}=\ C);\\
ac{:}\in\ \{A, B, C\} - \{pc, cc\}\ .
\end{array}
$$

The fourth statement is written using $\notin$ just for convenience — in fact it executes deterministically, since cc and ac are guaranteed to be different at that point.

Figure 1.5.1. The "Monty Hall" program

point to curtain cc. The host then tries to get the contestant to change his choice, showing that the prize is *not* behind some other curtain ac — which means that either the contestant has chosen it already or it is behind the other closed curtain.

Should the contestant change his mind?

1.5.4 Intuitive interpretation of pGCL expectations

In its full generality, an expectation is a function describing how much each program state is "worth."

The special case of an embedded predicate [*pred*] assigns to each state a worth of zero or of one: states satisfying *pred* are worth one, and states not satisfying *pred* are worth zero. The more general expectations arise when one estimates, in the *initial* state of a probabilistic program, what the worth of its *final* state will be. That estimate, the "expected worth" of the final state, is obtained by summing over all final states

> the worth of the final state multiplied by the probability the program "will go there" from the initial state.

Naturally the "will go there" probabilities depend on "from where," and so that expected worth is a function of the initial state.

When the worth of final states is given by [*post*], the expected worth of the initial state turns out to be just the probability that the program will reach *post*. That is because

$$
\begin{array}{ll}
& \text{expected worth of initial state} \\
\equiv & \quad (\text{probability } \mathit{prog} \text{ reaches } \mathit{post}) \\
& * \ (\text{worth of states satisfying } \mathit{post}) \\
+ & \quad (\text{probability } \mathit{prog} \text{ does not reach } \mathit{post}) \\
& * \ (\text{worth of states not satisfying } \mathit{post}) \\
\equiv & \quad (\text{probability } \mathit{prog} \text{ reaches } \mathit{post}) * 1 \\
+ & \quad (\text{probability } \mathit{prog} \text{ does not reach } \mathit{post}) * 0 \\
\equiv & \text{probability } \mathit{prog} \text{ reaches } \mathit{post} \ ;
\end{array}
$$

note we have relied on the fact that all states satisfying *post* have worth one.

More general analyses of programs *prog* in practice lead to conclusions of the form

$$p \quad \equiv \quad wp.prog.[post]$$

for some p and *post* which, given the above, we can interpret in two equivalent ways:

- the expected worth $[post]$ of the final state is at least the value of p in the initial state; or
- the probability that *prog* will establish *post* is at least p.[34]

Each interpretation is useful, and in the following example we can see them acting together: we ask for the probability that two fair coins when flipped will show the same face, and calculate

$$
\begin{array}{lll}
& wp.\left(\begin{array}{l} x{:=}\ H \ {}_{\frac{1}{2}}{\oplus}\ x{:=}\ T \quad ; \\ y{:=}\ H \ {}_{\frac{1}{2}}{\oplus}\ y{:=}\ T \end{array}\right).[x = y] & \\
\equiv & & {}_{\frac{1}{2}}{\oplus},\ :=\ \text{and sequential composition}^{35} \\
& wp.(x{:=}\ H \ {}_{\frac{1}{2}}{\oplus}\ x{:=}\ T).([x = H]/2 + [x = T]/2) & \\
\equiv & \quad (1/2)([H = H]/2 + [H = T]/2) & {}_{\frac{1}{2}}{\oplus} \text{ and } := \\
+ & \quad (1/2)([T = H]/2 + [T = T]/2) &
\end{array}
$$

[34] We must say "at least" in general, because possible demonic choice in *prog* means that the pre-expectation is only a lower bound for the actual expected value the program could deliver; and some analyses give only the weaker $p \Rrightarrow wp.prog.[post]$ in any case. See also Footnote 14 on p. 89.

[35] See Fig. 1.5.3 for this definition.

$$\equiv \quad (1/2)(1/2+0/2)+(1/2)(0/2+1/2) \qquad \text{definition } [\cdot]$$
$$\equiv \quad 1/2\ . \qquad \text{arithmetic}$$

We can then use the second interpretation above to conclude that the faces are the same with probability 1/2.[36]

But part of the above calculation involves the more general expression

$$wp.(x{:=}\ H\ {}_{\frac{1}{2}}{\oplus}\ x{:=}\ T).([x=H]/2+[x=T]/2)\ , \tag{1.20}$$

and what does that mean on its own? It must be given the first interpretation, that is as an expected worth, since "will *establish* $[x=H]/2+[x=T]/2$" makes no sense. Thus it means

> the expected value of the expression $[x=H]/2+[x=T]/2$
> after executing the program $x{:=}\ H\ {}_{\frac{1}{2}}{\oplus}\ x{:=}\ T$,

which the calculation goes on to show is in fact 1/2. But for our overall conclusions we do not need to think about the intermediate expressions — they are only the "glue" that holds the overall reasoning together.[37]

1.5.5 Semantics

The probabilistic semantics is derived from generalising the standard semantics in the way suggested in Sec. 1.3. Let the state space be S.

Definition 1.5.2 Expectation space The space of expectations over S is defined

$$\mathbb{E}S \quad := \quad (S\to\mathbb{R}_{\geq}, \Rrightarrow)\ ,$$

where the entailment relation $\Rrightarrow$, as we have seen, is inherited pointwise from the normal $\leq$ ordering in $\mathbb{R}_{\geq}$. The expectation-transformer model for programs is

$$\mathbb{T}S \quad := \quad (\mathbb{E}S \leftarrow \mathbb{E}S, \sqsubseteq)\ ,$$

where we write the functional arrow backward just to emphasise that such transformers map final post-expectations to initial pre-expectations, and where the *refinement* order $\sqsubseteq$ is derived pointwise from entailment $\Rrightarrow$ on $\mathbb{E}S$. □

---

[36](Recall Footnote 34.) If we do know, by other means say, that the program is deterministic (though still probabilistic), then we can say the pre-expectation is exact.

[37]See p. 271 for an example of this same analogy, but in the context of temporal logic.

Although both $\mathbb{E}S$ and $\mathbb{T}S$ are lattices, neither is a complete partial order,[38] because $\mathbb{R}_{\geq}$ itself is not. (It lacks an adjoined ∞ element.) In addition, when S is infinite (see *e.g.* Sec. 8.2 of Part II) we must impose the condition on elements of $\mathbb{E}S$ that each of them be *bounded above* by some non-negative real.[39]

In Fig. 1.5.3 we give a probabilistic semantics to the constructs of our language. It has the important feature that the standard programming constructs behave as usual, and are described just as concisely.

Note that our semantics states how *wp.prog* in each case transforms an *expression* in the program variables: that is, we give a procedure for calculating the greatest pre-expectation by purely syntactic manipulation. An alternative view is to see the post-expectations as mathematical *functions* of type $\mathbb{E}S$, and the expressions *wp.prog* are then of type $\mathbb{T}S$.

The expression-based view is more convenient in an introduction, and for the treatment of specific programs; the function-based view is more convenient (and, for recursion, necessary) for general properties of expectation transformers. In this chapter and the rest of Part I we retain the

---

[38] A PARTIAL ORDER differs from the familiar "total" orders like "$\leq$" in that two elements can be "incomparable"; the most common example is subset $\subseteq$ between sets, which satisfies REFLEXIVITY (a set is a subset of itself), ANTI-SYMMETRY (two sets cannot be subsets of each other without being the same set) and TRANSITIVITY (one set within a second within a third is a subset of the third directly as well). But it is not true that for any two sets one is necessarily a subset of the other.

A LATTICE is a non-empty partially ordered set where for all x, y in the set there is a GREATEST LOWER BOUND $x \sqcap y$ and and a LEAST UPPER BOUND $x \sqcup y$. This holds *e.g.* for the lattice of sets, as above; but the collection of *non-empty* sets is not a lattice, because $x \cap y$ (which is how $x \sqcap y$ is written for sets) is not necessarily non-empty even if x and y are.

A partial order $\sqsubseteq$ is CHAIN- or DIRECTED COMPLETE — then called a CPO — when it contains all limits of chains or directed sets respectively, where a CHAIN is a set totally ordered by $\sqsubseteq$ and a set is $\sqsubseteq$-DIRECTED if for any x, y in the set there is a z also in the set such that $x, y \sqsubseteq z$. (Since a chain is directed, directed completeness implies chain completeness; in fact with the Axiom of Choice, chain- and directed completeness are equivalent.)

All of these details can be found in standard texts [DP90].

[39] There is a difference between requiring that there be an upper bound for all expectations (we do not) and requiring that each expectation separately have an upper bound (we do).

In the first case, we would be saying that there is some M such that every expectation α in $\mathbb{E}S$ satisfied $\alpha \Rrightarrow \underline{M}$. That would be convenient because it would make both $\mathbb{E}S$ and $\mathbb{T}S$ complete partial orders, trivially; and that would *e.g.* allow us to use a standard treatment of fixed points.

But we adopt the second case where, for each expectation α separately, there is some M_α such that $\alpha \Rrightarrow \underline{M_\alpha}$; and, as α varies, these M_α's can increase without bound. That is why $\mathbb{E}S$ is not complete and is, therefore, why we will need a slightly special argument when dealing with fixed points.

$$
\begin{array}{rcl}
wp.\mathbf{abort}.postE & := & 0 \\
wp.\mathbf{skip}.postE & := & postE \\
wp.(x{:=}\ expr).postE & := & postE\ \langle x \mapsto expr\rangle \\
wp.(prog\,;prog').postE & := & wp.prog.(wp.prog'.postE) \\
wp.(prog \sqcap prog').postE & := & wp.prog.postE \ \ \mathsf{min}\ \ wp.prog'.postE \\
wp.(prog\ {}_p\!\oplus prog').postE & := & p * wp.prog.postE \ \ + \ \ \overline{p} * wp.prog'.postE
\end{array}
$$

Recall that $\overline{p}$ is the complement of p.

The expression on the right gives the *greatest pre-expectation* of *postE* with respect to each *pGCL* construct, where *postE* is an expression of type $\mathbb{E}S$ over the variables in state space S. (For historical reasons we continue to write *wp* instead of *gp*.)

In the case of recursion, however, we cannot give a purely syntactic definition. Instead we say that

$$
\begin{array}{rcl}
(\mathbf{mu}\ xxx \cdot \mathcal{C}) & := & \text{least fixed-point of the function } cntx{:}\,\mathbb{T}S \to \mathbb{T}S \\
 & & \text{defined so that } cntx.(wp.xxx) = wp.\mathcal{C}.\ [40]
\end{array}
$$

Figure 1.5.3. PROBABILISTIC *wp*-SEMANTICS OF *pGCL*

expression-based view as far as possible; but in Part II we use the more mathematical notation. (See for example Sec. 5.3.)

The worst program **abort** cannot be guaranteed to terminate in any proper state and therefore maps every post-expectation to 0. The immediately terminating program **skip** does not change anything, therefore the expected value of post-expectation *postE* after execution of **skip** is just its actual value before. The pre-expectation of the assignment $x{:=}\ expr$ is the postcondition with the expression *expr* substituted for x. Sequential composition is functional composition. The semantics of demonic choice $\sqcap$ reflects the dual metaphors for it: as abstraction, we must take the minimum because we are giving a guarantee over all possible implementations; as a demon's behaviour, we assume he acts to make our expected winnings as small as possible.

The pre-expectation of probabilistic choice is the weighted average of the pre-expectations of its branches. Since any such average is no less than the minimum it follows immediately that probabilistic choice refines demonic

[40] Because $\mathbb{T}S$ is not complete, to ensure existence of the fixed point we insist that the transformer-to-transformer function *cntx* be "feasibility-preserving," *i.e.* that if applied to a feasible transformer it returns a feasible transformer again. "Feasibility" of transformers is one of the "healthiness conditions" we will encounter in Sec. 1.6. For convenience, we usually assume that *cntx* is continuous as well.

See Lem. 5.6.8 on p. 148.

choice, which corresponds to our intuition. In fact we consider probabilistic choice to be a *deterministic* programming construct; that is we say that a program is deterministic if it is free of demonic nondeterminism unless it aborts.[41]

Finally, recursive programs have least-fixed-point semantics as usual.

1.5.6 Example of semantics: Monty Hall again

We illustrate the semantics by returning to the program of Fig. 1.5.1. Consider the post-expectation $[pc = cc]$, which takes value one just in those final states in which the candidate has correctly chosen the prize. Working backwards through the program's four statements, we have first (by standard *wp* calculations) that

$$\begin{array}{ll} & wp.\ \ ((cc{:}\notin \{cc, ac\})\ \mathbf{if}\ \ clever\ \mathbf{else\ skip})\ \ .[pc = cc] \\ \equiv & [clever] * [\{ac, cc, pc\} = \{A, B, C\}]\ \ +\ \ [\neg clever] * [pc = cc]\ \ , \end{array}$$

because (in case *clever*) the nondeterministic choice is guaranteed to pick *pc* only when it cannot avoid doing so.[42]

Standard reasoning suffices for our next step also:

$$\begin{array}{ll} & wp.\ (ac{:}\notin\ \{pc, cc\}). \\ & \quad ([clever] * [\{ac, cc, pc\} = \{A, B, C\}] + [\neg clever] * [pc = cc]) \\ \equiv & [clever] * [pc \neq cc] + [\neg clever] * [pc = cc]\ \ . \end{array}$$

For the *clever* case note that $\{ac, cc, pc\} = \{A, B, C\}$ holds (in the post-expectation) iff all three elements differ, and that the statement itself establishes only two of the required three inequalities — that $ac \neq pc$ and $ac \neq cc$. The weakest precondition supplies the third.

For the $\neg clever$ case note that neither *pc* nor *cc* is assigned to by $ac{:}\notin\ \{pc, cc\}$, so that $pc = cc$ holds afterwards iff it held before.

The next statement is probabilistic, and so produces a probabilistic pre-expectation involving the factors 1/3 given explicitly in the program; we have

$$\begin{array}{ll} & wp.\ (cc{:}=\ (A\ @\ \tfrac{1}{3}\ \vert\ B\ @\ \tfrac{1}{3}\ \vert\ C\ @\ \tfrac{1}{3})). \\ & \quad ([clever] * [pc \neq cc] + [\neg clever] * [pc = cc]) \\ \equiv & \qquad [clever]/3\ \ *\ \ ([pc \neq A] + [pc \neq B] + [pc \neq C]) \\ & +\ \ [\neg clever]/3\ \ *\ \ ([pc = A] + [pc = B] + [pc = C]) \end{array}$$

[41] Some writers call that PRE-DETERMINISM: "deterministic if terminating."

[42] In Fig. 1.5.1 we said that this fourth statement "executes deterministically"; yet here we have called it nondeterministic.

On its own, it is nondeterministic; but in the context of the program its nondeterminism is limited to making a choice from a singleton set, as our subsequent calculations will show.

$$
\begin{array}{lll}
\equiv & ([clever]/3)*2 \ + \ ([\neg clever]/3)*1 & \text{type of } pc \text{ is } \{A,B,C\}\ ^{43} \\
\equiv & 2\,[clever]/3 + [\neg clever]/3\ . &
\end{array}
$$

Then for the first statement $pc{:}\in\{A,B,C\}$ we only note that pc does not appear in the final condition above, thus leaving it unchanged under *wp*: with simplification it becomes

$$(1+[clever])/3\ ,$$

which is thus the pre-expectation for the whole program.

Since the post-expectation $[pc = cc]$ is standard (it is the characteristic function of the set of states in which $pc = cc$), we are able to interpret the pre-expectation directly as the probability that $pc = cc$ will be satisfied on termination: we conclude that the contestant has $2/3$ probability of finding the prize if he is clever, and only $1/3$ if he is not.

1.6 Healthiness and algebra for *pGCL*

Recall that all standard *GCL* constructs satisfy the important property of conjunctivity[44] — that is, for any *GCL* command *prog* and post-conditions $post, post'$ we have

$$wp.prog.(post \wedge post') \quad = \quad wp.prog.post \wedge wp.prog.post'\ .$$

That "healthiness condition" [Dij76] is used to prove many general properties of programs.

In *pGCL* the healthiness condition becomes "sublinearity," a generalisation of conjunctivity: [45]

Definition 1.6.1 Sublinearity of *pGCL* Let c_0, c_1, c_2 be non-negative reals, and $postE_1, postE_2$ expectations; then all *pGCL* constructs *prog* satisfy

$$
\begin{array}{ll}
 & wp.prog.(c_1 * postE_1 + c_2 * postE_2 \ \ominus\ c_0) \\
\Lleftarrow & c_1 * wp.prog.postE_1 \ + \ c_2 * wp.prog.postE_2 \ \ \ominus\ c_0\ ,
\end{array}
$$

which property of *prog* is called *sublinearity*. Truncated subtraction $\ominus$ is defined

$$x \ominus y \quad := \quad (x-y)\ \mathsf{max}\ 0\ ,$$

[43] Footnote 50 on p. 33 explains how typing might be propagated this way.

[44] They satisfy monotonicity too, which is implied by conjunctivity.

[45] Having discovered a probabilistic analogue of conjunctivity, we naturally ask for an analogue of disjunctivity. That turns out to be "super-linearity" — which when combined with sublinearity gives (just) linearity, and is characteristic of *deterministic* probabilistic programs, just as disjunctivity (with conjunctivity) characterises deterministic standard programs. See Sec. 8.3.

the maximum of the normal difference and zero. It has syntactic precedence lower than $+$. □

Although it has a strange appearance, from sublinearity we can extract a number of very useful consequences, as we now show. We begin with monotonicity, feasibility and scaling.[46]

Definition 1.6.2 Healthiness conditions

- *monotonicity:* increasing a post-expectation can only increase the pre-expectation. Suppose $postE \Rrightarrow postE'$ for two expectations $postE, postE'$; then

$$
\begin{array}{lll}
 & wp.prog.postE' & \\
\equiv & wp.prog.(postE + (postE' - postE)) & \\
\Lleftarrow & & postE'-postE \Lleftarrow 0\text{, hence well defined;} \\
 & & \text{sublinearity with } c_0, c_1, c_2 := 0, 1, 1 \\
 & wp.prog.postE + wp.prog.(postE'-postE) & \\
\Lleftarrow & wp.prog.postE\ . & 0 \Rrightarrow wp.prog.(postE'-postE)
\end{array}
$$

- *feasibility:* pre-expectations cannot be "too large." First note that

$$
\begin{array}{lll}
 & wp.prog.0 & \\
\equiv & wp.prog.(2 * 0) & \\
\Lleftarrow & 2 * wp.prog.0\ , & \text{sublinearity with } c_0, c_1, c_2 := 0, 2, 0
\end{array}
$$

so that $wp.prog.0$ must be zero.
Now write $\mathsf{max}\ postE$ for the maximum of $postE$ over all its variables' values; then

$$
\begin{array}{lll}
 & 0 & \\
\equiv & wp.prog.0 & \text{feasibility above} \\
\equiv & wp.prog.(postE \ominus \mathsf{max}\ postE) & postE \ominus \mathsf{max}\ postE \equiv 0 \\
\Lleftarrow & wp.prog.postE \ominus \mathsf{max}\ postE\ . & c_0, c_1, c_2 := \mathsf{max}\ postE, 1, 0
\end{array}
$$

But from $0 \Lleftarrow wp.prog.postE \ominus \mathsf{max}\ postE$ we have trivially that

$$wp.prog.postE \quad \Rrightarrow \quad \mathsf{max}\ postE\ , \tag{1.21}$$

which we identify as the *feasibility* condition for *pGCL*.[47]

- *scaling:* multiplication by a non-negative constant distributes through commands. Note first that $wp.prog.(c * postE) \Lleftarrow c * wp.prog.postE$ directly from sublinearity.

[46] These properties are collected together in Sec. 5.6, and restated in Part II as Defs. 5.6.3–5.6.5.

[47] Note how the general (1.21) implies the strictness condition $wp.prog.0 \equiv 0$, a direct numeric embedding of Dijkstra's *Law of the Excluded Miracle.*

For $\Rrightarrow$ we have two cases: when c is zero, trivially from feasibility

$$wp.prog.(0 * postE) \quad \equiv \quad wp.prog.0 \quad \equiv \quad 0 \quad \equiv \quad 0 * wp.prog.postE \ ;$$

and for the other case $c \neq 0$ we reason

$$\begin{array}{lll} & wp.prog.(c * postE) & \\ \equiv & c(1/c) * wp.prog.(c * postE) & c \neq 0 \\ \Rrightarrow & c * wp.prog.((1/c)c * postE)) & \text{sublinearity using } 1/c \\ \equiv & c * wp.prog.postE \ , & \end{array}$$

thus establishing $wp.prog.(c * postE) \equiv c * wp.prog.postE$ generally. (See p. 53 for an example of scaling's use.)

□

The remaining property we examine is so-called "probabilistic conjunctivity." Since standard conjunction "$\wedge$" is not defined over numbers, we have many choices for a probabilistic analogue "&" of it, requiring only that

$$\begin{array}{rcl} 0 \,\&\, 0 & = & 0 \\ 0 \,\&\, 1 & = & 0 \\ 1 \,\&\, 0 & = & 0 \\ 1 \,\&\, 1 & = & 1 \end{array} \tag{1.22}$$

for consistency with embedded Booleans.

Obvious possibilities for & are multiplication $*$ and minimum min, and each of those has its uses; but neither satisfies anything like a generalisation of conjunctivity. Return for example to the program of Fig. 1.5.1, and consider its second statement

$$cc := \ (A \ @ \ \tfrac{1}{3} \ \mid \ B \ @ \ \tfrac{1}{3} \ \mid \ C \ @ \ \tfrac{1}{3}) \ .$$

Writing *prog* for the above, with postcondition $[cc \neq C] \ \mathsf{min} \ [cc \neq A]$ we find

$$\begin{array}{ll} & wp.prog.(\,[cc \neq C]\ \mathsf{min}\ [cc \neq A]\,) \\ \equiv & wp.prog.[cc \neq C \wedge cc \neq A] \\ \equiv & wp.prog.[cc = B] \\ \equiv & 1/3 \\ \not\equiv & 2/3 \ \mathsf{min} \ 2/3 \\ \equiv & wp.prog.[cc \neq C]\ \mathsf{min}\ wp.prog.[cc \neq A] \ . \end{array}$$

Thus probabilistic programs do *not* distribute min in general, and we must find something else. Instead we define

$$exp \,\&\, exp' \quad := \quad exp + exp' \ominus 1 \ , \tag{1.23}$$

whose right-hand side is inspired by sublinearity when $c_0, c_1, c_2 := 1, 1, 1$. The operator is commutative; and if we restrict expectations to $[0,1]$ it is associative as well. Note however that it is not idempotent.[48]

We now state a (sub-)distribution property for &, a direct consequence of sublinearity.

sub-conjunctivity: the operator & sub-distributes through expectation transformers. From sublinearity with $c_0, c_1, c_2 := 1, 1, 1$ we have

$$wp.prog.(postE \,\&\, postE') \quad \Leftarrow \quad wp.prog.postE \;\;\&\;\; wp.prog.postE'$$

for all *prog*.

(Unfortunately there does not seem to be a full ($\equiv$) conjunctivity property for expectation transformers.)

Beyond sub-conjunctivity, we say that & generalises conjunction for several other reasons as well. The first is of course that it satisfies the standard properties (1.22).

The second reason is that sub-conjunctivity (a consequence of sublinearity) implies "full" conjunctivity for standard programs. Standard programs, containing no probabilistic choices, take standard $[post]$-style post-expectations to standard pre-expectations: they are the embedding of *GCL* in *pGCL*, and for standard *prog* we now show that

$$\begin{array}{ll} & wp.prog.([post] \,\&\, [post']) \\ \equiv & wp.prog.[post] \;\;\&\;\; wp.prog.[post'] \;. \end{array} \tag{1.24}$$

First note that "$\Leftarrow$" comes directly from sub-conjunctivity above, taking $postE, postE'$ to be $[post]\,, [post']$.

For "$\Rrightarrow$" we appeal to monotonicity, because $[post] \,\&\, [post'] \Rrightarrow [post]$ whence $wp.prog.([post] \,\&\, [post']) \Rrightarrow wp.prog.[post]$, and similarly for $post'$. Putting those together gives

$$wp.prog.([post] \,\&\, [post']) \quad \Rrightarrow \quad wp.prog.[post] \;\; \mathsf{min} \;\; wp.prog.[post'] \;,$$

by elementary arithmetic properties of $\Rrightarrow$. But on standard expectations — which $wp.prog.[post]$ and $wp.prog.[post']$ are, because *prog* is standard — the operators min and & agree.

A last attribute linking & to $\wedge$ comes straight from elementary probability theory. Let X and Y be two events, not necessarily independent: then

> if the probability of X is at least p, and the probability of Y is at least q, the most that can be said in general about the joint event $X \cap Y$ is that it has probability at least $p \,\&\, q$.

[48] A binary operator $\odot$ is IDEMPOTENT just when $x \odot x = x$ for all x.

To see this, we begin by recalling that for any events X, Y and any probability distribution Pr we have[49]

$$
\begin{array}{ll}
 & \Pr.(X \cap Y) \\
= & \Pr.X + \Pr.Y - \Pr.(X \cup Y) \\
\geq & \qquad\qquad \text{because } \Pr.(X \cup Y) \leq 1 \text{ and } \Pr.(X \cap Y) \geq 0 \\
 & (\Pr.X + \Pr.Y - 1) \sqcup 0 \ .
\end{array}
$$

We are not dealing with exact probabilities however: when demonic nondeterminism is present we have only lower bounds. Thus we address the question

> Given only $\Pr.X \geq p$ and $\Pr.Y \geq q$, what is the most precise lower bound for $\Pr.(X \cap Y)$ in terms of p and q?

From the reasoning above we obtain

$$(p + q - 1) \sqcup 0 \tag{1.25}$$

immediately as a lower bound. But to see that it is the *greatest* lower bound we must show that for any X, Y, p, q there is a probability distribution Pr such that the bound is attained; and that is illustrated in Fig. 1.6.3, where an explicit distribution is given in which $\Pr.X = p$, $\Pr.Y = q$ and $\Pr.(X \cap Y)$ is as low as possible, reaching $(p + q - 1) \sqcup 0$ exactly.

Returning to our example, but using &, we now have equality:

$$
\begin{array}{ll}
 & wp.prog.([cc \neq C] \,\&\, [cc \neq A]) \\
\equiv & wp.prog.[cc = B] \\
\equiv & 1/3 \\
\equiv & 2/3 \,\&\, 2/3 \\
\equiv & wp.prog.[cc \neq C] \;\;\&\;\; wp.prog.[cc \neq A] \ .
\end{array}
$$

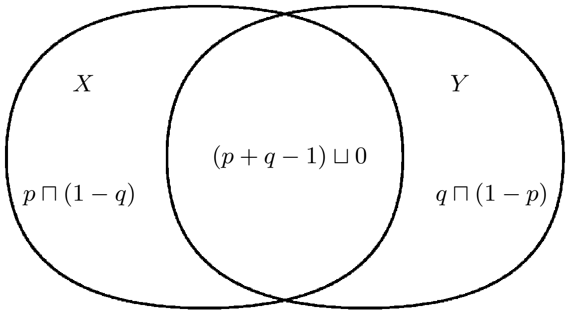
The & operator also plays a crucial role in the proof (Chap. 7) of our probabilistic loop rule, presented in Chap. 2 and used in the examples to come.

1.7 Healthiness example: modular reasoning

As an example of the use of healthiness conditions, we formulate and prove a simple but very powerful property of *pGCL* programs, important for "modular" reasoning about them.

By *modular* reasoning in this case we mean determining, first, that a program *prog* of interest has some standard property; then for subsequent (possibly probabilistic) reasoning we assume that property. This makes

[49] The first step is the *modularity law* for probabilities.



$$\begin{array}{rclcl}
\Pr.X & = & p \sqcap (1-q) + (p+q-1) \sqcup 0 & = & p \\
\Pr.Y & = & q \sqcap (1-p) + (p+q-1) \sqcup 0 & = & q \\
\Pr.(X \cap Y) & = & (p+q-1) \sqcup 0 & = & p \,\&\, q
\end{array}$$

The lower bound $p \,\&\, q$ is the best possible.

Figure 1.6.3. PROBABILISTIC CONJUNCTION & DEPICTED

---

the reasoning modular in the sense that we do not have to prove all the properties at once.[50]

We formulate the principle as a lemma.

Lemma 1.7.1 MODULAR REASONING Suppose for some program *prog* and predicates *pre* and *post* we have

$$[pre] \quad \Rrightarrow \quad wp.prog.[post]\ , \qquad (1.26)$$

which is just the embedded form of a standard Hoare-triple specification. Then in any state satisfying *pre* we have for any bounded post-expectations $postE, postE'$ that

$$wp.prog.postE \quad = \quad wp.prog.postE'\ , \quad ^{51}$$

provided *post* implies that *postE* and $postE'$ are equal.

That is, with (1.26) we can assume the truth of *post* when reasoning about the post-expectation, provided *pre* holds in the initial state.

[50] A typical use of this appeals to standard reasoning, in a “first pass,” to establish that some (Boolean) property — such as a variable’s typing — is invariant in a program; then, in the “second pass” during which probabilistic reasoning might be carried out, we can assume that invariant everywhere without comment. Recall Footnote 43 on p. 28; see also the treatment of Fig. 7.7.11 on p. 211 to come.

[51] We write “=” rather than “≡” because the equality holds only in some states (those satisfying *pre*), as indicated in the text above. Thus writing “$\equiv, \Rrightarrow, \Lleftarrow$” as we do elsewhere is just an alternative for the text “in all states”.

Proof: We use the healthiness conditions of the previous section, and we assume that the post-expectations $postE, postE'$ are bounded above by some nonzero M. Given that the current state satisfies *pre*, we then have

$$
\begin{array}{lll}
 & wp.prog.([post] * postE) & \\
= & M * wp.prog.([post] * postE/M) & \text{scaling} \\
 & & \\
= & M * wp.prog.([post] \ \&\ (postE/M)) & [post] \text{ is standard;} \\
 & & postE/M \Rrightarrow 1 \\
 & & \\
\geq & M * (wp.prog.[post] \ \&\ wp.prog.(postE/M)) & \text{sub-conjunctivity} \\
\geq & M * ([pre] \ \&\ wp.prog.(postE/M)) & \text{Assumption (1.26)} \\
= & M * (1 \ \&\ wp.prog.(postE/M)) & pre \text{ holds in current state} \\
= & M * wp.prog.(postE/M) & \text{arithmetic} \\
= & wp.prog.postE\ . & \text{scaling}
\end{array}
$$

The opposite inequality is immediate (in all states) from the monotonicity healthiness property, since $[post] * postE \Rrightarrow postE$. Thus, still assuming *pre* in the current state, we conclude with

$$
\begin{array}{lll}
 & wp.prog.postE & \\
= & wp.prog.([post] * postE) & \text{above} \\
= & wp.prog.([post] * postE') & \text{assumption about } postE, postE' \\
= & wp.prog.postE'\ . & \text{as above, but for } postE'
\end{array}
$$

□

This kind of reasoning is nothing new for standard programs, and indeed is usually taken for granted (although its formal justification appeals to conjunctivity). It is important that it is available in *pGCL* as well.[52]

1.8 Interaction between probabilistic- and demonic choice

We conclude with some illustrations of the interaction of demonic and probabilistic choice. Consider two variables x, y, one chosen demonically and the other probabilistically. Suppose first that x is chosen demonically and y probabilistically, and take post-expectation $[x = y]$. Then

[52] Lem. 1.7.1 holds even when $postE, postE'$ are unbounded, provided of course that *wp.prog* is defined for them; the proof of that can be given by direct reference to the definition of *wp* over the model, as set out in Chap. 5.

We will need that extension for our occasional excursions beyond the "safe" bounded world we have formally dealt with in the logic (*e.g.* Sections 2.11 and 3.3).

$$
\begin{array}{ll}
 & wp.(\,(x\!:=\,1 \sqcap x\!:=\,2);\;(y\!:=\,1\;{}_{\frac{1}{2}}\!\oplus y\!:=\,2)\,).[x=y] \\
\equiv & wp.(x\!:=\,1 \sqcap x\!:=\,2).([x=1]\,/2+[x=2]\,/2) \\
\equiv & ([1=1]\,/2+[1=2]\,/2)\;\;\mathsf{min}\;\;([2=1]\,/2+[2=2]\,/2) \\
\equiv & (1/2+0/2)\;\;\mathsf{min}\;\;(0/2+1/2) \\
\equiv & 1/2\;,
\end{array}
$$

from which we see that program establishes $x = y$ with probability at least $1/2$: no matter which value is assigned to x, with probability $1/2$ the second command will assign the same to y.

Now suppose instead that it is the second choice that is demonic. Then we have

$$
\begin{array}{ll}
 & wp.(\,(x\!:=\,1\;{}_{\frac{1}{2}}\!\oplus x\!:=\,2);\;(y\!:=\,1 \sqcap y\!:=\,2)\,).[x=y] \\
\equiv & wp.(x\!:=\,1\;{}_{\frac{1}{2}}\!\oplus x\!:=\,2).([x=1]\;\mathsf{min}\;[x=2]) \\
\equiv & ([1=1]\;\mathsf{min}\;[1=2])/2\;\;+\;\;([2=1]\;\mathsf{min}\;[2=2])/2 \\
\equiv & (1\;\mathsf{min}\;0)/2\;\;+\;\;(0\;\mathsf{min}\;1)/2 \\
\equiv & 0\;,
\end{array}
$$

reflecting that no matter what value is assigned probabilistically to x, the demon could choose subsequently to assign a different value to y.

Thus it is clear that the execution order of occurrence of the two choices plays a critical role in their interaction, and in particular that the demon in the first case cannot make the assignment "clairvoyantly" to x in order to avoid the value that later will be assigned to y.

1.9 Summary

Being able to reason formally about probabilistic programs does not of course remove *per se* the complexity of the mathematics on which they rely: we do not now expect to find astonishingly simple correctness proofs for all the large collection of randomised algorithms that have been developed over the decades [MR95]. However it should be possible in principle to locate and determine reliably what are the probabilistic/mathematical facts the construction of a randomised algorithm needs to exploit... which is of course just what standard predicate transformers do for conventional algorithms.

In the remainder of Part I we concentrate on proof rules that can be derived for *pGCL* — principally for loops — and on examples.

The theory of expectation transformers with nondeterminism is given in Part II, where in particular the role of sublinearity is identified and proved: it characterises a subspace of the predicate transformers that has an equivalent operational semantics of relations between initial and final probabilistic distributions over the state space — a formalisation of the gambling game of Sec. 1.3. All the programming constructs of the probabilistic language of guarded commands belong to that subspace, which means that the programmer who uses the language can elect to reason about it either axiomatically or operationally.

Chapter notes

In the mid-1970's, Rabin demonstrated how randomisation could be used to solve a variety of programming problems [Rab76]; since then, the range of applications has increased considerably [MR95], and indeed we analyse several of them as case studies in later chapters. In the meantime — fuelled by randomisation's impressive applicability — the search for an effective logic of probabilistic programs became an important research topic around the beginning of the 1980's, and remained so until the mid-1990's. Ironically, the major technical difficulty was due, in the main, to one of standard programming's major successes: *demonic nondeterminism*, the basis for abstraction. It was a challenging problem to decide what to do about it, and how it should interact with the new *probabilistic nondeterminism*.

The first probabilistic logics did not treat demonic nondeterminism at all — Feldman and Harel [FH84] for instance proved soundness and completeness for a probabilistic *PDL* which was (in our terms) purely deterministic. The logical language allowed statements about programs to be made at the level of probability *distributions* and, as we discuss in Sec. A.2, that proves to be an impediment to the natural introduction of a demon. A Hoare-style logic based on similar principles has also been explored by den Hartog and de Vink [dHdV02].

The crucial step of a *quantitative* logic of expectations was taken by Kozen [Koz85]. Subsequently Jones [Jon90], with Plotkin and using the *evaluations* from earlier work of Saheb-Djahromi [SD80] that were based directly on topologies rather than on σ- or Borel algebras, worked on more general probabilistic powerdomains; as an example of her technique she specialised it to the Kozen-style logic for deterministic programs, resulting in the *sub-probability measures* that provide a neat way to quantify nontermination.[53]

In 1997 He *et al.* [HSM97] finally proposed the *operational model* containing all the ingredients for a full treatment of abstraction and program refinement in the context of probability — and that model paved the way for the "demonic/probabilistic" program logic based on expectation transformers. Subsequently Ying [Yin03] has worked towards a probabilistic *refinement calculus* in the style of Back [BvW98].

[53] The notion of sub-probability measures to characterise termination was present much earlier, for example in the work of Feldman and Harel [FH84].