Developing and Reasoning about Probabilistic Programs in *pGCL*

Annabelle McIver[1] and Carroll Morgan[2]

[1] Department of Computer Science
Macquarie University
NSW, Australia

[2] Department of Computer Science and Engineering
University of New South Wales
NSW, Australia

As explained in Chapter 1, Dijkstra's guarded-command language, which we call *GCL*, was introduced as an intellectual framework for rigorous reasoning about imperative sequential programs; one of its novelties was that it contained explicit "demonic" nondeterminism, representing abstraction from (or ignorance of) which of two program fragments will be executed. By introducing *probabilistic* nondeterminism into *GCL*, we provide a means with which also probabilistic programs can be rigorously developed and reasoned about.

The programming logic of "weakest preconditions" for *GCL* becomes a logic of "greatest pre-expectations" for what we call *pGCL*. An *expectation* is a generalized predicate suitable for expressing quantitative properties such as "the probability of achieving a postcondition".

pGCL is suitable for describing random algorithms, at least over discrete distributions. In our presentation of it and its logic we give a number of small examples, and two case studies. The first illustrates probabilistic "almost-certain" termination; the second case study illustrates approximated probabilities, abstraction and refinement.

After a brief historical account of work on probabilistic semantics in Section 1, Section 2 gives a brief and shallow overview of *pGCL*, somewhat informal and concentrating on simple examples. Section 3 sets out the definitions and properties of *pGCL* systematically, and Section 4 treats an example of reasoning about probabilistic loops, showing how to use probabilistic *invariants.* Section 5 illustrates termination arguments via probabilistic *variants* with a thorough treatment of Rabin's choice-coordination algorithm [219]; Section 6 illustrates abstraction and refinement, as well as "approximated probabilities", by giving a two-level treatment of an almost-uniform selection algorithm. An impression of *pGCL* can be gained by reading Sections 2 and 4, with finally a glance over Sections 3.1 and 3.2; more thoroughly one would read Sections 2, 3.1 and 3.2, then 2 (again) and finally 4. The more theoretical Section 3.3 can be skipped on first reading. Appendix A describes basic concepts of probability theory needed in this chapter.

1 Introduction

Probabilistic programs and systems are increasingly relevant: often random algorithms are computationally feasible where their deterministic counterparts are not; some concurrent applications are impossible without the symmetry breaking that randomisation provides; and in hybrid systems the low-level hardware might be represented by probabilistic program text that models quantitative unreliability. Because of that relevance, there has been a renewed interest in techniques for establishing the correctness of such programs—for the more widespread they become, the more we will depend on understanding their behaviour, and their limits, exactly.

In this tutorial chapter we address that last concern, of understanding: we survey a method for rigorous reasoning about probabilistic programs and systems. We give an impression of how they work, an *operational* view, and we suggest how we should reason about them, a *logical* view—and we show how the two views are designed to fit together.

We use Dijkstra's Guarded Command Language *GCL* [81] as a simple and "pared-down" syntax for presenting our ideas: it is a weakest-precondition based method of describing computations and their meaning; here we extend it to probabilistic programs, and we give examples of its use.

Most sequential programming languages contain a construct for "deterministic" choice, where the program makes a selection in a predictable way: for example, in

$$\mathbf{if}\ \mathit{test}\ \mathbf{then}\ \mathsf{This}\ \mathbf{else}\ \mathsf{That}\ \mathbf{fi} \qquad (1)$$

the two-way choice between This and That is determined by *test* and the current state.

In contrast, Dijkstra's language of guarded commands brings to prominence nondeterministic or "demonic" choice, in which the program's behaviour is *not* predictable, is not determined by the current state. At first [81], demonic choice was presented as a consequence of "overlapping guards", as almost an accident, but as its importance became more widely recognized it developed a life of its own. Nowadays it merits an explicit operator: the construct

$$\mathsf{This} \sqcap \mathsf{That}$$

chooses between the alternatives unpredictably and, as a specification, indicates abstraction from the issue of which will be executed. The customer will be happy with either This or That; and the implementor may choose between them according to his own concerns. An alternative but equivalent view is that the choice between the alternatives is made at runtime by an adversarial "demon" whose aim is to make the program as unlikely as possible to achieve its goal.

Early research on probabilistic semantics took a different route: demonic choice was not regarded as fundamental. Rather it was abandoned altogether, being replaced by probabilistic choice [140, 90, 89, 133, 132], written for example

$$\mathsf{This}\ {}_{p}\!\oplus\ \mathsf{That}$$

to indicate a program that behaved like This with probability p, but otherwise like That. Without demonic choice, however, probabilistic semantics was divorced from the contemporaneous work on specification and refinement: there was no longer any means of abstraction.

More recently it has been discovered [131, 197, 247] how to bring the two topics back together, taking the more natural approach of *adding* probabilistic choice, while retaining demonic choice. In fact deterministic choice is a special case of probabilistic choice, which in turn is a refinement of demonic choice.

We give the resulting probabilistic extension of *GCL* the name "*pGCL*".

2 An impression of *pGCL*

Let square brackets $[\cdot]$ be used to embed Boolean-valued predicates within arithmetic formulae which, for reasons explained below, we call *expectations*; we allow them to range over the unit interval $[0, 1]$. Stipulating that [**false**] is 0 and [**true**] is 1 makes $[P]$ in a trivial sense the probability that a given predicate P holds: if false, P holds with probability 0; if true, it holds with probability 1.

For (our first) example, consider the simple program

$$x{:=}\ -y \ \ {}_{\frac{1}{3}}\!\oplus \ \ x{:=}\ +y \tag{2}$$

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

Recall [81] that for any predicate P over *final* states, and a standard command S, the "weakest precondition" predicate $wp.S.P$ acts over *initial* states: it holds just in those initial states from which S is guaranteed to reach P. (Throughout this chapter, we use *standard* to mean "non-probabilistic".) We also write $f.x$ instead of $f(x)$ for function f applied to argument x, with left association. Now suppose S is probabilistic, as Program (2) is; what can we say about the *probability* that $wp.S.P$ holds in some initial state?

It turns out that the answer is just $wp.S.[P]$, once we generalize $wp.S$ to expectations instead of predicates. For that, we begin with the two definitions

$$\begin{aligned} wp.(x{:=}\ E).R \ &\widehat{=}\ \text{"}R \text{ with } x \text{ replaced everywhere by } E\text{"}\,^{3} && (3)\\ wp.(S\ {}_p\!\oplus \mathsf{T}).R \ &\widehat{=}\ \qquad p * wp.S.R && (4)\\ &\quad\ + (1{-}p) * wp.\mathsf{T}.R && \end{aligned}$$

in which R 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 (2)?

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

To find out, we calculate $wp.S.[P]$ in this case; that is

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

Thus our answer is the last arithmetic formula above, which we could 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

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

Those results indeed correspond with our operational intuition about the effect of ${}_{\frac{1}{3}}\!\oplus$. Later we explain the use of "$\equiv$" rather than "$=$".

The above remarkable generalisation of sequential program correctness is due to Kozen [140], but in its original form was restricted to programs that did not contain demonic choice $\sqcap$. When He *et al.* [131] and Morgan *et al.* [197] successfully added demonic choice, it became possible to begin the long-overdue integration of probabilistic programming and formal program development: in the latter, demonic choice—as *abstraction*—plays a crucial role in specifications. The extension was based on a general approach to probabilistic power-domains due to Jones and Plotkin [132, 133], which recently has been further developed by Tix *et al.* [247].

To illustrate the use of abstraction, in our second example we abstract from probabilities: a demonic version of Program (2) is much more realistic in that we set its probabilistic parameters only within some tolerance. We say informally (but still with precision) that

$$\left.\begin{array}{l}
-\ x{:=}\ -y \text{ is to be executed with probability } \textit{at least } 1/3, \\
-\ x{:=}\ +y \text{ is to be executed with probability } \textit{at least } 1/4 \text{ and} \\
-\ \text{it is certain that one or the other will be executed.}
\end{array}\right\} \qquad (5)$$

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 (5) as

$$\begin{array}{rl}
& x{:=}\ -y \ {}_{\frac{1}{3}}\!\oplus\ x{:=}\ +y \\
\sqcap & x{:=}\ -y \ {}_{\frac{3}{4}}\!\oplus\ x{:=}\ +y
\end{array} \qquad (6)$$

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. A convenient notation for (6) would be based on the abbreviation

$$\mathsf{S}\ {}_p\oplus_q\ \mathsf{T}\ \ \widehat{=}\ \ (\mathsf{S}\ {}_p\oplus\ \mathsf{T})\ \sqcap\ (\mathsf{T}\ {}_q\oplus\ \mathsf{S})\quad \text{for } p+q \leq 1$$

we would then write $x{:=}\ -y\ {}_{\frac{1}{3}}\oplus_{\frac{1}{4}}\ x{:=}\ +y$.

To treat Program (6), we define the command

$$wp.(\mathsf{S} \sqcap \mathsf{T}).R\ \ \widehat{=}\ \ wp.\mathsf{S}.R\ \mathit{min}\ wp.\mathsf{T}.R \tag{7}$$

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

$$\begin{array}{lll} & wp.(\text{ Program (6) }).[x \geq 0] & \\ \equiv & \quad [y \leq 0]/3 + 2\,[y \geq 0]/3 & \text{using (3), (3), (7)} \\ & \mathit{min}\ \ 3\,[y \leq 0]/4 + [y \geq 0]/4 & \\ \equiv & [y < 0]/3\ +\ [y = 0]\ +\ [y > 0]/4 & \text{using arithmetic} \end{array}$$

Our interpretation is now

- When y is initially negative, the 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, the demon cannot avoid $x \geq 0$—either way the probability of $x \geq 0$ finally is 1.
- When y is initially positive, the 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. Suppose Program (6) represents some mass-produced physical device and, by examining the production method, we have determined the tolerance (5) on the devices produced. If we were to buy one arbitrarily, all we could conclude about its probability of establishing $x \geq 0$ is just as calculated above.

Refinement is the converse of abstraction: for two commands S, T we define

$$\mathsf{S} \sqsubseteq \mathsf{T}\ \ \widehat{=}\ \ wp.\mathsf{S}.R \Rrightarrow wp.\mathsf{T}.R\quad \text{for all } R \tag{8}$$

where we write $\Rrightarrow$ for "everywhere no more than" (which ensures [**false**] $\Rrightarrow$ [**true**] as the notation suggests). From (8) we see that in the special case when R is an embedded predicate $[P]$, the meaning of $\Rrightarrow$ ensures that a refinement T of S is at least as likely to establish P as S is. That accords with the usual definition of refinement for standard programs—for then we know $wp.\mathsf{S}.[P]$ is either 0 or 1, and whenever S is certain to establish P (whenever $wp.\mathsf{S}.[P] \equiv 1$) we know that T also is certain to do so (because then $1 \Rrightarrow wp.\mathsf{T}.[P]$).

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

$$x{:=}\ -y\ \ {}_{\frac{1}{2}}\oplus\ \ x{:=}\ +y \tag{9}$$

which clearly satisfies Specification (5); thus it should refine Program (6). With

Definition (8), we find for any R that

$$\begin{array}{lll}
& wp.(\,\text{Program (9)}\,).R & \\
\equiv & wp.(x{:=}\ {-y}).R/2 \ \ + \ \ wp.(x{:=}\ {+y}).R/2 & \\
\equiv & R^-/2 + R^+/2 & \text{introduce abbreviations} \\
\equiv & \quad (3/5)(R^-/3 + 2R^+/3) & \text{arithmetic} \\
+ & \quad (2/5)(3R^-/4 + R^+/4) & \\
\Leftarrow & \quad R^-/3 + 2R^+/3 & \text{any linear combination exceeds } \mathsf{min} \\
\mathsf{min} & \quad 3R^-/4 + R^+/4 & \\
\equiv & wp.(\,\text{Program (6)}\,).R &
\end{array}$$

The refinement relation (8) 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 at the program level might read

$$\begin{array}{lll}
& \text{Program (9)} & \\
= & x{:=}\ {-y} \ {}_{\frac{1}{2}}\!\oplus \ x{:=}\ {+y} & \\
= & \quad (x{:=}\ {-y} \ {}_{\frac{1}{3}}\!\oplus \ x{:=}\ {+y}) & \text{arithmetic} \\
{}_{\frac{3}{5}}\!\oplus & \quad (x{:=}\ {-y} \ {}_{\frac{3}{4}}\!\oplus \ x{:=}\ {+y}) & \\
\sqsupseteq & \quad x{:=}\ {-y} \ {}_{\frac{1}{3}}\!\oplus \ x{:=}\ {+y} & (\sqcap) \sqsubseteq ({}_p\!\oplus) \text{ for any } p \\
\sqcap & \quad x{:=}\ {-y} \ {}_{\frac{3}{4}}\!\oplus \ x{:=}\ {+y} & \\
\equiv & \text{Program (6)} &
\end{array}$$

3 Presentation of probabilistic *GCL*

In this section we give a concise presentation of probabilistic *GCL*—*pGCL*: its definitions, how they are to be interpreted and their (healthiness) properties.

3.1 Definitions of *pGCL* commands

In *pGCL*, commands act between "expectations" rather than predicates, where an *expectation* is an expression over (program or state) variables that takes its value in the unit interval $[0, 1]$. (A more general treatment is possible in which expectations are arbitrarily non-negative but bounded [197, 181].) To retain the use of predicates, we allow expectations of the form $[P]$ when P is Boolean-valued, defining $[\mathbf{false}]$ to be 0 and $[\mathbf{true}]$ to be 1.

Implication-like relations between expectations are

$$\begin{array}{rcl}
R \Rrightarrow R' & \mathrel{\widehat{=}} & R \text{ is everywhere no more than } R' \\
R \equiv R' & \mathrel{\widehat{=}} & R \text{ is everywhere equal to } R' \\
R \Lleftarrow R' & \mathrel{\widehat{=}} & R \text{ is everywhere no less than } R'
\end{array}$$

Note that $\models P \Rightarrow P'$ exactly when $[P] \Rrightarrow [P']$, and so on; that is the motivation

The probabilistic guarded command language *pGCL* acts over "expectations" rather than predicates: *expectations* take values in $[0, 1]$.

$wp.(x := E).R$	The expectation obtained after replacing all free occurrences of x in R by E, renaming bound variables in R if necessary to avoid capture of free variables in E.
$wp.\mathbf{skip}.R$	R
$wp.(\mathsf{S};\ \mathsf{T}).R$	$wp.\mathsf{S}.(wp.\mathsf{T}.R)$
$wp.(\mathsf{S} \sqcap \mathsf{T}).R$	$wp.\mathsf{S}.R\ \mathit{min}\ wp.\mathsf{T}.R$
$wp.(\mathsf{S}\ {}_p\!\oplus \mathsf{T}).R$	$p * wp.\mathsf{S}.R + (1-p) * wp.\mathsf{T}.R$
$\mathsf{S} \sqsubseteq \mathsf{T}$	$wp.\mathsf{S}.R \Rrightarrow wp.\mathsf{T}.R$ for all R

- R is an expectation (possibly but not necessarily $[P]$ for a predicate P);
- P is a predicate (not an expectation);
- $*$ is multiplication;
- S, T are probabilistic guarded commands (inductively);
- p is an expression over the program variables (possibly but not necessarily a constant), taking a value in $[0, 1]$; and
- x is a variable (or a vector of variables).

Deterministic choice **if** B **then** S **else** T **fi** is a special case of probabilistic choice: it is just $\mathsf{S}\ {}_{[B]}\!\oplus\ \mathsf{T}$. Recursions are handled by least fixed points in the usual way; in practice however, the special case of loops is more easily treated using (probabilistic) invariants and variants.

Fig. 1. *pGCL*—the probabilistic Guarded Command Language

for the symbols chosen.

The definitions of the commands in *pGCL* are given in Fig. 1.

3.2 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 $[P]$ assigns to each state a worth of 0 or of 1: states satisfying P are worth 1, and states not satisfying P are worth 0. 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 $[P]$, the expected worth of the initial state turns out to be just the probability that the program will reach P. That is because

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

where, of course, matters are greatly simplified by the fact that all states satisfying P have the same worth. We must, however, moderate this to "the greatest guaranteed probability" when there is demonic choice: this is why the general judgement is the inequality $p \Rrightarrow \mathit{wp}.\mathsf{S}.[P]$ rather than the special case of equality given at (10).

Typical analyses of programs S in practice lead to conclusions of the form

$$p \ \equiv \ \mathit{wp}.\mathsf{S}.[P] \tag{10}$$

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

1. the expected worth $[P]$ of the final state is at least the value of p in the initial state; or
2. the probability that S will establish P is at least p.

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

$$\begin{array}{lr}
\mathit{wp}.\begin{pmatrix} c{:=}\ H \ {}_{\frac{1}{2}}\!\oplus c{:=}\ T\,; \\ d{:=}\ H \ {}_{\frac{1}{2}}\!\oplus d{:=}\ T \end{pmatrix}.\,[c = d] & \\
\equiv & {}_{\frac{1}{2}}\!\oplus,\ {:=} \text{ and sequential composition} \\
\quad \mathit{wp}.(c{:=}\ H \ {}_{\frac{1}{2}}\!\oplus c{:=}\ T).([c = H]/\mathit{2} + [c = T]/\mathit{2}) & \\
\equiv \quad (\mathit{1/2})([H = H]/\mathit{2} + [H = T]/\mathit{2}) & {}_{\frac{1}{2}}\!\oplus \text{ and } {:=} \\
\quad + (\mathit{1/2})([T = H]/\mathit{2} + [T = T]/\mathit{2}) & \\
\equiv (\mathit{1/2})(\mathit{1/2} + \mathit{0/2}) + (\mathit{1/2})(\mathit{0/2} + \mathit{1/2}) & \text{definition } [\cdot]
\end{array}$$

$\equiv 1/2$ arithmetic

We can then use the second interpretation above to conclude that the faces are the same with probability (at least) $1/2$. Knowing there is no demonic choice in the program, we can in fact say it is exact.

But part of the above calculation involves the more general expression

$$wp.(c{:=}\ H\ {}_{\frac{1}{2}}\!\oplus\ c{:=}\ T).([c = H]/2 + [c = T]/2)$$

and what does that mean on its own? It must be given the first interpretation, since its post-expectation is not of the form $[P]$, and it means

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

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.

Exercise 1. We consider again the two coin-like variables c and d which are flipped in various ways. We use the notation $c{:=}\ H\ {}_p\!\oplus\ T$ to represent the assignment of H to c with probability p, and of T with probability $1-p$; similarly, we write $d{:=}\ H\ {}_p\!\oplus\ T$.

1. What if one of the two coins is not fair? Calculate

$$\begin{array}{rl} & wp.(c{:=}\ H\ {}_p\!\oplus\ T;\ \ d{:=}\ H\ {}_{1/2}\!\oplus\ T).\,[c = d] \\ \text{and} & wp.(c{:=}\ H\ {}_{1/2}\!\oplus\ T;\ \ d{:=}\ H\ {}_q\!\oplus\ T).\,[c = d] \end{array}$$

2. What if one of the two coins is not even flipped, but rather is placed face-up or -down at will? (At *whose* will?) Calculate

$$\begin{array}{rl} & wp.(c{:=}\ H \sqcap T;\ \ d{:=}\ H\ {}_{1/2}\!\oplus\ T).\,[c = d] \\ \text{and} & wp.(c{:=}\ H\ {}_{1/2}\!\oplus\ T;\ \ d{:=}\ H \sqcap T).\,[c = d]\ . \end{array}$$

3. Of the five answers to the questions above, (including the two-fair-coins example in the text) one is conspicuous. Which one? How do you explain that answer?

3.3 Properties of *pGCL*

Recall that all *GCL* constructs satisfy the property of conjunctivity—that is, for any *GCL* command S and post-conditions P, P' we have

$$wp.S.(P \wedge P') \quad = \quad wp.S.P \wedge wp.S.P'$$

That "healthiness property" [81] is used to prove general properties of programs.

In *pGCL* the healthiness condition becomes "sublinearity" [197], a generalisation of conjunctivity:

Definition 1 (Sub-linearity). *Let a, b and c be non-negative finite reals, and R and R' expectations; then all pGCL constructs S satisfy*

$$wp.\mathsf{S}.(aR + bR' \ominus c) \quad \Lleftarrow \quad a(wp.\mathsf{S}.R) + b(wp.\mathsf{S}.R') \ominus c \qquad (11)$$

This property of S is called sublinearity. *We have written aR for $a * R$, and so on. Truncated subtraction $\ominus$ is defined*

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

with syntactic precedence lower than $+$.

Sublinearity characterizes probabilistic *and demonic* commands. In Kozen's original probability-only formulation [140] the commands are not demonic, and there they satisfy the much stronger property of "linearity" [179].

Although it has a strange appearance, from sublinearity we can extract a number of very useful consequences, as we now show [197]. We begin with monotonicity, feasibility and scaling.

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

$$\begin{array}{lr}
wp.\mathsf{S}.R' & \\
\equiv\ wp.\mathsf{S}.(R + (R' - R)) & \\
\Lleftarrow\ wp.\mathsf{S}.R + wp.\mathsf{S}.(R' - R) & \text{sublinearity with } a, b, c := 1, 1, 0 \\
\Lleftarrow\ wp.\mathsf{S}.R & R' - R \text{ well defined, hence } 0 \Rrightarrow wp.\mathsf{S}.(R' - R)
\end{array}$$

feasibility: pre-expectations cannot be "too large". First note that $wp.\mathsf{S}.0$ must be 0, as we show below.

$$\begin{array}{lr}
wp.\mathsf{S}.0 & \\
\equiv\ wp.\mathsf{S}.(2 * 0) & \\
\Lleftarrow\ 2 * wp.\mathsf{S}.0 & \text{sublinearity with } a, b, c := 2, 0, 0
\end{array}$$

Now write $\mathsf{max}\, R$ for the maximum of R over all its variables' values; then

$$\begin{array}{lr}
0 & \\
\equiv\ wp.\mathsf{S}.0 & \text{feasibility above} \\
\equiv\ wp.\mathsf{S}.(R \ominus \mathsf{max}\, R) & R \ominus \mathsf{max}\, R \equiv 0 \\
\Lleftarrow\ wp.\mathsf{S}.R \ominus \mathsf{max}\, R & \text{sublinearity with } a, b, c := 1, 0, \mathsf{max}\, R
\end{array}$$

But from $0 \Lleftarrow wp.\mathsf{S}.R \ominus (\mathsf{max}\, R)$ we have trivially that

$$wp.\mathsf{S}.R \quad \Rrightarrow \quad \mathsf{max}\, R \qquad (12)$$

which we identify as the *feasibility* condition for *pGCL*. Conveniently, the general (12) implies the earlier special case $wp.\mathsf{S}.0 \equiv 0$.

scaling: multiplication by a non-negative constant distributes through commands. Note first that $wp.\mathsf{S}.(aR) \Lleftarrow a(wp.\mathsf{S}.R)$ directly from sublinearity. For $\Rrightarrow$ we have two cases: when a is 0, trivially from feasibility

$$wp.\mathsf{S}.(0 * R) \;\equiv\; wp.\mathsf{S}.0 \;\equiv\; 0 \;\equiv\; 0 * wp.\mathsf{S}.R$$

and for the other case $a \neq 0$ we reason as follows, establishing the identity $wp.\mathsf{S}.(aR) \equiv a(wp.\mathsf{S}.R)$ generally.

$$\begin{array}{ll} & wp.\mathsf{S}.(aR) \\ \equiv & a(1/a)wp.\mathsf{S}.(aR) \qquad\qquad\qquad\qquad\qquad a \neq 0 \\ \Rrightarrow & a(wp.\mathsf{S}.((1/a)aR)) \qquad\qquad\qquad \text{sublinearity using } 1/a \\ \equiv & a(wp.\mathsf{S}.R) \end{array}$$

That completes monotonicity, feasibility and scaling.

The remaining property we examine is probabilistic conjunction. Since standard conjunction $\wedge$ is not defined over numbers, we have many choices for a probabilistic analogue & of it, requiring only, for consistency with embedded Booleans, that

$$\begin{array}{c} 0 \;\&\; 0 = 0 \\ 0 \;\&\; 1 = 0 \\ 1 \;\&\; 0 = 0 \\ 1 \;\&\; 1 = 1 \end{array} \tag{13}$$

Obvious possibilities for & are multiplication $*$ and minimum min, and each of those has its uses; but neither satisfies anything like a generalisation of conjunctivity. Instead we define

$$R \,\&\, R' \;\;\widehat{=}\;\; R + R' \ominus 1 \tag{14}$$

whose right-hand side is inspired by sublinearity when $a, b, c := 1, 1, 1$. We now state a (sub-)distribution property for it, a direct consequence of sublinearity. This same operator (and its other propositional companions) was introduced by Łukasiewicz in the 1920's [103]; here we have synthesized it by quite different means.

sub-conjunctivity: the operator & subdistributes through commands. From sublinearity with $a, b, c := 1, 1, 1$ we have

$$wp.\mathsf{S}.(R \,\&\, R') \;\;\Lleftarrow\;\; wp.\mathsf{S}.R \;\&\; wp.\mathsf{S}.R'$$

for all S.

Unfortunately there does not seem to be a full (rather than sub-)conjunctivity property.

Beyond sub-conjunctivity, we say that & generalizes conjunction for several other reasons. The first is of course that it satisfies the standard properties (13).

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

$$wp.\mathsf{S}.([P] \mathbin{\&} [P']) \quad \equiv \quad wp.\mathsf{S}.[P] \;\&\; wp.\mathsf{S}.[P'] \tag{15}$$

First note that "$\Lleftarrow$" comes directly from sub-conjunctivity above, taking R, R' to be $[P], [P']$.

For "$\Rrightarrow$" we appeal to monotonicity, because $[P] \mathbin{\&} [P'] \Rrightarrow [P]$ whence we have $wp.\mathsf{S}.([P] \mathbin{\&} [P']) \Rrightarrow wp.\mathsf{S}.[P]$, and similarly for P'. Putting those together gives

$$wp.\mathsf{S}.([P] \mathbin{\&} [P']) \quad \Rrightarrow \quad wp.\mathsf{S}.[P] \;\mathsf{min}\; wp.\mathsf{S}.[P']$$

by elementary arithmetic properties of $\Rrightarrow$. But on standard expectations—which $wp.\mathsf{S}.[P]$ and $wp.\mathsf{S}.[P']$ are, because S is standard—the operators min and & agree.

A last attribute linking & to $\wedge$ comes straight from elementary probability theory. Let A and B be two events, unrelated by $\subseteq$ and not necessarily independent: then we can show that

> if the probabilities of A and B are at least p and q respectively, then the most that can be said about the joint event $A \cap B$ is that it has probability at least $p \mathbin{\&} q$ [235].

The & operator also plays a crucial role in the proof [193, 181] (not given here) of the probabilistic loop rule presented and used in the next section.

Exercise 2. Say that a probabilistic program is *standard* if it takes *0/1*-valued post-expectations to *0/1*-valued pre-expectations; typical examples are programs written in *pGCL* that nevertheless do not use ${}_p\oplus$. Show that such programs distribute *minimum* for all post-expectations. For hints, consult the reference text on *pGCL* [181].

4 Probabilistic invariants for loops

To show *pGCL* in action, we state a proof rule for probabilistic loops and apply it to a simple example.

Just as for standard loops, we can deal with invariants and termination separately: common sense suggests that the probabilistic reasoning should be an extension of standard reasoning, and indeed that is the case. One proves a predicate invariant under execution of a loop's body; and one finds a variant that ensures the loop's eventual termination: the conclusion is that if the invariant holds initially then the invariant and the negation of the loop guard together hold finally. Probability does lead to differences, however—and here are some of them:

- The invariant *may* be probabilistic, in which case its operational meaning is more general than just "the computation remains within a certain set of states".

- The variant might *have* to be probabilistically interpreted, since the usual "must strictly decrease and is bounded below" technique is no longer adequate, even for simple cases. (It remains sound.)
- When both the invariant and the termination condition are probabilistic, one cannot use Boolean conjunction to combine "correct if terminates" and "it does terminate".

4.1 Probabilistic invariants

In a standard loop, the invariant holds at every iteration of the loop. It describes a set of states from which continuing to execute the loop body is guaranteed to establish the postcondition, if the guard ever becomes false—that is, if termination occurs.

For a probabilistic loop we have a post-expectation rather than a postcondition, but otherwise the situation is much the same. Moreover, if that post-expectation is some $[P]$ say, then—as an aid to the intuition—we can look for an invariant that gives a lower bound on the probability that we will establish P by (continuing to) execute the loop body. Often that invariant will have the form

$$p \ * \ [I] \tag{16}$$

with p a probability and I a predicate, both expressions over the state. From the definition of $[\cdot]$ we know that the interpretation of (16) is

probability p if I holds, and probability 0 otherwise.

We see an example of such invariants in Section 4.3.

4.2 Termination

The probability that a program will terminate generalizes the usual definition: recalling that $[\mathbf{true}] \equiv \mathit{1}$ we see that a program's probability of termination is given by

$$wp.S.\mathit{1} \tag{17}$$

As a simple example of that, suppose S is the recursive program

$$S \ \mathrel{\widehat{=}} \ S \ {}_p\!\oplus \mathbf{skip} \tag{18}$$

in which we assume that p is some constant strictly less than 1: on each recursive call, P has probability $\mathit{1}{-}p$ of termination, continuing otherwise with further recursion. Elementary probability theory shows that S terminates with probability 1 (after an expected $p/(\mathit{1}{-}p)$ recursive calls). By calculation based on (17) we see that

$$\begin{aligned} & wp.S.\mathit{1} \\ \equiv\ & p * (wp.S.\mathit{1}) + (\mathit{1}{-}p) * (wp.\mathbf{skip}.\mathit{1}) \end{aligned}$$

$$\equiv p * (wp.S.1) + (1-p)$$

so that $(1-p) * (wp.S.1) \equiv 1-p$. Since p is not 1, we can divide by $1-p$ to see that indeed $wp.S.1 \equiv 1$: the recursion will terminate with probability 1 (for if p is not 1, the chance of recursing N times is p^N, which for $p < 1$ approaches 0 as N increases without bound).

We return to probabilistic termination in Section 5.

4.3 Probabilistic correctness of loops

As in the standard case, it is easy to show that if $[P] * I \Rrightarrow wp.S.I$ then

$$I \quad \Rrightarrow \quad wp.(\mathbf{do}\, P \rightarrow S\, \mathbf{od}).([\neg P] * I)$$

provided the loop terminates. Thus the notion of invariant carries over smoothly from the standard to the probabilistic case. This is an immediate consequence of the definition of loops as least fixed points: indeed, for the proof one simply carries out the standard reasoning almost without noticing that expectations rather than predicates are being manipulated. The precise treatment of "provided" uses weakest *liberal* pre-expectations [193, 180].

When termination is taken into account as well, we get the rule below [193].

Definition 2 (Proof rule for probabilistic loops). *For convenience, we write T for the termination probability of the loop, so that*

$$T \quad \widehat{=} \quad wp.(\boldsymbol{do}\, P \rightarrow S\, \boldsymbol{od}).1$$

Then partial loop correctness—preservation of a loop invariant I—implies total loop correctness if that invariant I nowhere exceeds T: that is,

$$\begin{array}{lrcl} \textit{if} & [P] * I & \Rrightarrow & wp.S.I \\ \textit{and} & I & \Rrightarrow & T \\ \textit{then} & I & \Rrightarrow & wp.(\boldsymbol{do}\, P \rightarrow S\, \boldsymbol{od}).([\neg P] * I) \end{array}$$

Note that it is not the same to say "implies total correctness from those initial states where I does not exceed T": in fact I must not exceed T in *any* state. The weaker alternative is not sound.

We illustrate the loop rule with a simple example. Suppose we have a machine that is supposed to sum the elements of a sequence ss of N elements indexed from 0 to $N-1$, except that the mechanism for moving along the sequence occasionally moves the wrong way. A program for the machine is given in Figure 2, where the unreliable component

$$k{:=}\ k+1 \quad {}_c\oplus \quad k{:=}\ k-1$$

misbehaves with probability $1-c$. With what probability does the machine accurately sum the sequence, establishing

$$r \quad = \quad \sum ss \tag{19}$$

on termination?

```
var $k\colon \mathbb{Z}$ •
    $r, k := 0, 0$;
    do $k < N \rightarrow$
        $r := r + ss.k$;
        $k := k + 1\ {}_c\!\oplus\ k := k - 1$        $\leftarrow$ failure possible here
    od
```

Fig. 2. An unreliable sequence-summer

We first find the invariant. Relying on our informal discussion above, we ask the following question:

> during the loop's execution, with what probability are we in a state from which completion of the loop would establish (19)?

The answer is in the form (16)—take p to be c^{N-k}, and let I be the standard invariant

$$0 \le k \le N \quad \wedge \quad r = \sum ss[0..k)$$

Then our probabilistic invariant—call it J—is just $p * [I]$, which is to say that

> if the standard invariant holds then it is c^{N-k}, the probability of going on to successful termination; if it does not hold, then it is 0.

Having chosen a possible invariant, to check it we calculate

$$\begin{array}{ll}
wp.\begin{pmatrix} r := r + ss.k; \\ k := k + 1\ {}_c\!\oplus\ k := k - 1 \end{pmatrix}.J & \\
\equiv\ wp.(r := ss.k).(& ;\ \text{and}\ {}_c\!\oplus \\
\qquad c * wp.(k := k + 1).J & \\
\qquad +(1-c) * wp.(k := k - 1).J) & \\
\Leftarrow\ wp.(r := r + ss.k). & \text{drop second term, and}\ wp.(:=) \\
\qquad c^{N-k} \ * \ \begin{bmatrix} 0 \le k + 1 \le N \\ r = \sum ss[0..k) \end{bmatrix} & \\
\equiv\ c^{N-k} \ * \ \begin{bmatrix} 0 \le k + 1 \le N \\ r + ss.k = \sum ss[0..k) \end{bmatrix} & wp.(:=) \\
\Leftarrow\ [k < N] * J & \text{arithmetic}
\end{array}$$

where in the last step the guard $k < N$, and $k \ge 0$ from the invariant, allow the removal of $+ss.k$ from both sides of the lower equality.

A more concise rendering of the above can be given using the following convention. When reasoning "backwards", as above, the compact notation

$$\begin{array}{ll}
\quad PostE & \\
\cdot\ \Leftarrow\ PreE & \text{applying}\ wp.\mathsf{Prog}
\end{array}$$

allows the linear "step-by-step" layout of the proof to be more easily continued.

The "·" at left warns that we are asserting $PostE \Lleftarrow wp.\mathsf{Prog}.PreE$ (rather than $PostE \Lleftarrow PreE$ itself). Using this convention we would have written instead

$$
\begin{array}{lll}
 & J & \\
\cdot \equiv & \quad c * wp.(k{:=}\ k+1).J & \text{applying } wp.(k{:=}\ k+1 \ {}_c\!\oplus k{:=}\ k-1) \\
 & + (1-c) * wp.(k{:=}\ k-1).J & \\
 & \Lleftarrow c^{N-k} * \begin{bmatrix} 0 \le k+1 \le N \\ r = \sum ss[0..k) \end{bmatrix} & \text{drop second term; } wp.(:=) \\
\cdot \equiv & c^{N-k} * \begin{bmatrix} 0 \le k+1 \le N \\ r + ss.k = \sum ss[0..k) \end{bmatrix} & \text{applying } wp.(r{:=}\ r+ss.k) \\
 & \Lleftarrow [k < N] * J &
\end{array}
$$

Now we turn to termination: we note (informally) that the loop terminates with probability at least

$$c^{N-k} \;\; * \;\; [0 \le k \le N]$$

which is just the probability of $N-k$ correct executions of $k{:=}\ k+1$, given that k is in the proper range to start with; hence trivially $J \Rrightarrow T$ as required by the loop rule.

That concludes reasoning about the loop itself, leaving only initialisation and the post-expectation of the whole program. For the latter we see that on termination of the loop we have $[k \ge N] * J$, which indeed "implies" (is in the relation $\Rrightarrow$ to) the post-expectation $[r = \sum ss]$ as required.

Turning finally to the initialisation we finish off with

$$
\begin{array}{l}
wp.(r, k{:=}\ 0, 0).J \\
\equiv c^N \;\; * \;\; \begin{bmatrix} 0 \le 0 \le N \\ 0 = \sum ss[0..0) \end{bmatrix} \\
\equiv c^N \;\; * \;\; [\mathbf{true}] \\
\equiv c^N
\end{array}
$$

and our overall conclusion is therefore

$$c^N \;\; \Rrightarrow \;\; wp.(\mathit{sequence\text{-}summer}).\,[r = \textstyle\sum ss]$$

just as we had hoped: the probability that the sequence is correctly summed is at least c^N.

Note the importance of the inequality $\Rrightarrow$ in our conclusion just above. It is not true that the probability of correct operation is *equal* to c^N in general, for it is certainly possible that r is correctly calculated in spite of the occasional malfunction of $k{:=}\ k+1$. The exact probability, should we try to calculate it, might depend intricately on the contents of ss. (It could be very involved if ss contained some mixture of positive and negative values.) If we were forced to calculate exact results (as in earlier work [238]), rather than just lower bounds as we did above, this method would not be at all practical.

Further examples of loops are given elsewhere [193].

5 First case study: probabilistic termination

In this case study, we treat an algorithm whose termination argument is fairly involved, showing how it is dealt with using probabilistic-variant arguments. This example has also been given an automated proof using the *pB* probabilistic extension of the *B* development method [182, 3]. For another example of "easy correctness but difficult termination", see the Probabilistic Dining Philosophers [149], [181, Section 3.2].

5.1 Introduction

Rabin's choice-coordination algorithm (explained in Sections 5.2 and 5.3 below) is an example of the use of probability for *symmetry-breaking*: identical processes with identical initial conditions must reach collectively an asymmetric state, all choosing one alternative or all choosing the other. The simplest example is a coin flipped between two people—each has equal right to win, the coin is fair, the initial conditions are thus symmetric; yet, at the end, one person has won and not the other. In this example, however, the situation is made more complex by insisting that the processes be *distributed*: they cannot share a central "coin".

Rabin's article [219] explains the algorithm he invented and relates it to a similar algorithm in nature, carried out by mites who must decide whether they should all infest the left or all the right ear of a moth, but he does not give a formal proof of its correctness. We do that here.

Section 5.3 writes the algorithm as a loop, containing probabilistic choice, and we show the loop terminates "with probability 1" in a desired state: we use invariants, to show that if it terminates it is in that state; and we use probabilistic variants to show that indeed it does terminate. "Termination with probability 1"' is the kind of termination exhibited for example by the algorithm "flip a fair coin repeatedly until you get heads, then stop". For our purposes that is as good as "normal" guarantees of termination.

In this example, the partial correctness argument is entirely standard and so does not illustrate the new probabilistic techniques. (It is somewhat involved, however, and thus interesting as an exercise in any case.) In such cases one treats probabilistic choice as nondeterministic choice and proceeds with standard reasoning, since the theory shows that any *wp*-style property proved of the "projected" nondeterministic program is valid for the original probabilistic program as well. More precisely, replacing probabilistic choice by nondeterministic choice is an anti-refinement.

The termination argument is novel however, since probabilistic variant techniques [107, 193] must be used.

5.2 Informal description of Rabin's algorithm

This informal description is based on Rabin's presentation [219].

A group of tourists are to decide between two meeting places: inside a (certain) church, or inside a museum. They may not communicate all at once as a group.

Each tourist carries a notepad on which he will write various numbers; outside each of the two potential meeting places is a noticeboard on which various messages will be written. Initially the number 0 appears on all the notepads and on the two noticeboards.

Each tourist decides independently (demonically) which meeting place to visit first, after which he strictly alternates his visits between them. At each visit he looks at the noticeboard there, and if it displays "here" goes inside. If it does not display "here" it will display a number instead, in which case the tourist compares that number K with the one on his notepad k and takes one of the following three actions:

if $k > K$ —The tourist writes "here" on the noticeboard (erasing K), and goes inside.

if $k = K$ —The tourist chooses K', the next even number larger than K, and then flips a coin: if it comes up heads, he increases K' by a further 1. He then writes K' on the noticeboard and on his notepad (erasing k and K), and goes to the other place. For example if K is 8 or 9, first K' becomes 10 and then possibly 11.

if $k < K$ —The tourist writes K on his notepad (erasing k), and goes to the other place.

Rabin's algorithm terminates with probability 1; and on termination all tourists will be inside, at the same meeting place.

5.3 The program

Here we make the description more precise by giving a *pGCL* program for it (see Figure 3). Each tourist is represented by an instance of the number on his pad.

The program informally Call the two places "left" and "right".

Bag *lout* (*rout*) is the bag of numbers held by tourists waiting to look at the left (right) noticeboard; bag *lin* (*rin*) is the bag of numbers held by tourists who have already decided on the left (right) alternative; number L (R) is the number on the left (right) noticeboard.

Initially there are M (N) tourists on the left (right), all holding the number 0; no tourist has yet made a decision. Both noticeboards show 0.

Execution is as follows. If some tourists are still undecided (so that *lout* (*rout*) is not yet empty), select one: the number he holds is l (r). If some tourist has (already) decided on this alternative (so that *lin* (*rin*) is not empty), this tourist does the same; otherwise there are three further possibilities:

If this tourist's number l (r) is greater than the noticeboard value L (R), then he decides on this alternative (joining *lin* (*rin*)).

```
$lout, rout := [\![0]\!]^M, [\![0]\!]^N$;
$lin, rin := \Box, \Box$;
$L, R := 0, 0$;

do  $lout \neq \Box \rightarrow$
      take $l$ from $lout$;
      if $lin \neq \Box$ then add $l$ to $lin$ else
           $l > L$  $\rightarrow$  add $l$ to $lin$
      []   $l = L$  $\rightarrow$  $L := L + 2 \;{}_{\frac{1}{2}}\!\oplus\; \overline{(L + 2)}$;    add $L$ to $rout$
      []   $l < L$  $\rightarrow$  add $L$ to $rout$
      fi

[]    $rout \neq \Box \rightarrow$
      take $r$ from $rout$;
      if $rin \neq \Box$ then add $r$ to $rin$ else
           $r > R$  $\rightarrow$  add $r$ to $rin$
      []   $r = R$  $\rightarrow$  $R := R + 2 \;{}_{\frac{1}{2}}\!\oplus\; \overline{(R + 2)}$;    add $R$ to $lout$
      []   $r < R$  $\rightarrow$  add $R$ to $lout$
      fi
od
```

Fig. 3. Rabin's choice-coordination algorithm

If this tourist's number equals the noticeboard value, he increases the noticeboard value, copies that value and goes to the other alternative (*rout* (*lout*)).
If this tourist's number is less than the noticeboard value, he copies that value and goes to the other alternative.

Notation We use the following notations in the program and in the subsequent analysis.

- $[\![\cdots]\!]$ — Bag (multiset) brackets.
- $\Box$ — The empty bag.
- $[\![n]\!]^N$ — A bag containing N copies of value n.
- $b0 + b1$ — The bag formed by putting all elements of $b0$ and $b1$ together into one bag.
- **take** n **from** b — A program command: choose an element nondeterministically from non-empty bag b, assign it to n and remove it from b.
- **add** n **to** b — Add element n to bag b.
- **if** B **then** Prog **else** $\cdots$ **fi** — Execute Prog if B holds, otherwise treat $\cdots$ as a collection of guarded alternatives in the normal way.
- $\overline{n}$ — The "conjugate" value $n + 1$ if n is even, and $n - 1$ if n is odd.
- $\widetilde{n}$ — The minimum n *min* $\overline{n}$ of n and $\overline{n}$.
- $\#b$ — The number of elements in bag b.
- $x := m\,{}_p\!\oplus n$ — Assign m to x with probability p, and n to x with probability $1-p$.

Correctness criteria We must show that the program is guaranteed with probability 1 to terminate, and that on termination it establishes

$$\#lin = M{+}N \wedge rin = \Box \;\;\vee\;\; lin = \Box \wedge \#rin = M{+}N$$

That is, on termination the tourists are either all inside on the left or all inside on the right.

5.4 Partial correctness

The arguments for partial correctness involve no probabilistic reasoning; but there are several invariants.

Three invariants The first invariant states that tourists are neither created nor destroyed:

$$\#lout + \#lin + \#rout + \#rin \;\;=\;\; M + N \tag{20}$$

It holds initially, and is trivially maintained.

The second invariant is

$$\begin{aligned} lin, lout \;&\leq\; R \\ rin, rout \;&\leq\; L \end{aligned} \tag{21}$$

and expresses that a tourist's number never exceeds the number posted at the *other* place. By $b \leq K$ we mean that no element in the bag b exceeds the integer K. To show invariance we reason as follows:

- It holds initially.
- Since L, R never decrease, it can be falsified only by adding elements to the bags.
- Adding elements to lin, rin cannot falsify it, since those elements come from $lout, rout$.
- The only commands adding elements to $lout, rout$ are

  add L **to** $rout$ and **add** R **to** $lout$

  and they maintain it trivially.

Our final invariant for partial correctness is

$$\begin{array}{ll} \mathsf{max}\, lin > L & \text{if } lin \neq \Box \\ \mathsf{max}\, rin > R & \text{if } rin \neq \Box \end{array} \tag{22}$$

expressing that if any tourist has gone inside, then at least one of the tourists inside must have a number exceeding the number posted outside.

By symmetry we need only consider the left (lin) case. The invariant holds on initialisation (when $lin = \Box$); and inspection of the program shows that it is trivially established when the first value is added to *lin* since the command concerned

$$l > L \;\rightarrow\; \textbf{add}\, l\, \textbf{to}\, \mathit{lin}$$

is executed when $\mathit{lin} = \square$ to establish $\mathit{lin} = \llbracket l \rrbracket$ for some $l > L$.

Since elements never leave *lin*, it remains non-empty and $\mathsf{max}\,\mathit{lin}$ can only increase; finally L cannot change when *lin* is non-empty.

On termination. . . With these invariants we can show that on termination (if it occurs) we have $\mathit{lout} = \mathit{rout} = \square$—in fact with invariant (20) we need only

$$\mathit{lin} = \square \;\;\vee\;\; \mathit{rin} = \square$$

Assuming for a contradiction that both *lin* and *rin* are non-empty, we then have from invariants (21) and (22) the inequalities

$$L \;\geq\; \mathsf{max}\,\mathit{rin} \;>\; R \;\geq\; \mathsf{max}\,\mathit{lin} \;>\; L$$

which give us the required impossibility.

5.5 Showing termination: the variant

For termination we need probabilistic arguments, since it is easy to see that no standard variant will do: suppose that the first $M + N$ iterations of the loop take us to the state below, differing from the initial state only in the use of 4's rather than 0's.

$$\begin{aligned} \mathit{lout}, \mathit{rout} &= \llbracket 4 \rrbracket^{M}, \llbracket 4 \rrbracket^{N} \\ \mathit{lin}, \mathit{rin} &= \square, \square \\ L, R &= 4, 4 \end{aligned}$$

All coin flips came up heads, and each tourist had exactly two turns. Since the program contains no absolute comparisons, we are effectively back where we started: the program checks only whether various numbers are greater than others, not what the numbers actually are. Because of that, there can be no standard variant that decreased on every step we took.

So is not possible to prove termination using a standard variant whose strict decrease is guaranteed. Instead we appeal to the following rule [107,193,181]:

Definition 3 (Probabilistic variant rule). *If an integer-valued function of the program state—a* probabilistic variant*—can be found that*

- *is bounded above,*
- *is bounded below and*
- *with probability at least p is decreased by the loop body, for some fixed non-zero p,*

then with probability 1 the loop will terminate. (Note that the invariant and guard *of the loop may be used in establishing the three properties.)*

The rule differs from the standard one in two respects: the variant must be bounded above (as well as below); and it is not guaranteed to decrease, but rather does so only with some probability bounded away from 0. Note that the probability of decrease may differ from state to state, but the point of "bounded away from zero"—distinguished from simply "not equal to zero"—is that over an infinite state space the various probabilities cannot be arbitrarily small. Over a finite state space there is no distinction.

To find our variant, we note that the algorithm exhibits two kinds of behaviour: the shuttling back-and-forth of the tourists, between the two meeting places (small scale); and the pattern of the two noticeboard numbers L, R as they increase (large scale). Our variant therefore will be "lexicographic", one within another: the small-scale *inner* variant will deal with the shuttling, and the large-scale *outer* variant will deal with L and R.

Inner variant: tourists' movements The aim of the inner variant is to show that the tourists cannot shuttle forever between the sites without eventually changing one of the noticeboards. Intuition suggests that indeed they cannot, since every such movement increases the number on some tourist's notepad, and from invariant (21) those numbers are bounded above by $L \mathsf{max} R$.

The inner variant is based on that idea. For neatness we make it increasing rather than decreasing, which is of no consequence since we have taken care to ensure that it is bounded above and below by fixed values, independent of L and R—we could always subtract it from the upper bound to convert it back to decreasing. The independence from L, R is important, given our variant rule, because L and R can themselves increase without bound. We define $V0$ to be

$$\begin{array}{ll} & \#[\![x\colon lout{+}rout \mid x \geq L]\!] \\ + & \#[\![x\colon lout{+}rout \mid x \geq R]\!] \\ + & 3 \times \#(lin{+}rin) \end{array} \qquad (23)$$

This is bounded above by $3(M{+}N)$, because

$$(23) \leq 2\#(lout{+}rout) + 3\#(lin{+}rin) \leq 3\#(lout{+}rout{+}lin{+}rin) = 3(M{+}N)$$

where the last equality is supplied by the invariant (20). Since the outer variant will deal with changes to L and R, in checking the increase of $V0$ we can restrict our attention to those parts of the loop body that leave L, R fixed—and we show in that case that the variant must increase on every step:

– If $lin \neq \square$ then an element is removed from $lout$ ($V0$ decreases by at most 2) and added to lin (but then $V0$ increases by 3); the same reasoning applies when $l > L$.
– If $l = L$ then L will change; so we need not consider that. (It will be dealt with by the outer variant.)
– If $l < L$ then $V0$ increases by at least 1, since l is replaced by L in $lout{+}rout$—and (before) $l \not\geq L$ but (after) $L \geq L$.

The reasoning for *rout*, on the right, is symmetric.

Outer variant: changes to L and R For the outer variant we need further invariants; the first is

$$\widetilde{L} - \widetilde{R} \;\in\; \{-2, 0, 2\} \tag{24}$$

stating that the notice-board values can never be "too far apart". It holds initially; and, from invariant (21), the command

$$L{:=}\;\; L+2 \;{}_{\frac{1}{2}}\!\oplus\; \overline{(L+2)}$$

is executed only when $L \leq R$, thus only when $\widetilde{L} \leq \widetilde{R}$, and has the effect

$$\widetilde{L}{:=}\;\; \widetilde{L}+2$$

Thus we can classify L, R into three sets of states:

- $\widetilde{L} = \widetilde{R} - 2 \,\vee\, \widetilde{L} = \widetilde{R} + 2$—write $L \ncong R$ for those states.
- $\overline{L} = R$ (equivalently $L = \overline{R}$)—write $L \cong R$.
- $L = R$.

Then we note that the underlying iteration of the loop induces state transitions as follows. (We write $\langle L = R\rangle$ for the set of states satisfying $L = R$, and so on; nondeterministic choice is indicated by $\sqcap$; the transitions are indicated by $\rightarrow$.)

$$\begin{array}{lcl}
\langle L \ncong R\rangle & \rightarrow & \langle L \ncong R\rangle \;\sqcap\; \langle L = R\rangle \;{}_{\frac{1}{2}}\!\oplus\; \langle L \cong R\rangle \\
\langle L = R\rangle & \rightarrow & \langle L = R\rangle \;\sqcap\; \langle L \ncong R\rangle \\
\langle L \cong R\rangle & \rightarrow & \langle L \cong R\rangle
\end{array}$$

To explain the absence of a transition leaving states $\langle L \cong R\rangle$ we need yet another invariant

$$\overline{L} \notin \mathit{rout} \;\;\wedge\;\; \overline{R} \notin \mathit{lout} \tag{25}$$

It holds initially, and cannot be falsified by the command **add** L **to** *rout*, because $\overline{L} \neq L$. That leaves the command $L{:=}\;\; L+2 \;{}_{\frac{1}{2}}\!\oplus\; \overline{(L+2)}$; but in that case, from (21), we have

$$\mathit{rout} \;\;\leq\;\; L \;\;<\;\; L+2, \overline{(L+2)} \;\;=\;\; \overline{\overline{(L+2)}}, \overline{(L+2)}$$

so that in neither case does the command set L to the conjugate of a value already in *rout*.

Thus with (25) we see that execution of the only alternatives that change L, R cannot occur if $L \cong R$, since, for example, selection of the guard $l = L$ implies $L \in \mathit{lout}$, impossible if $L \cong R$ and $\overline{R} \notin \mathit{lout}$.

For the outer variant we therefore define $V1$ to be

$$\begin{array}{rl}
2, & \text{if } L = R \\
1, & \text{if } L \ncong R \\
0, & \text{if } L \cong R
\end{array} \tag{26}$$

and note that whenever L or R changes, the quantity $V1$ decreases with probability at least $1/2$.

The two variants together If we put the two variants together lexicographically, with the outer variant $V1$ being the more significant, then the composite satisfies all the conditions required by the probabilistic variant rule. In particular it has probability at least $1/2$ of strict decrease on *every* iteration of the loop. Remember that the inner variant increases rather than decreases—we subtract it from $3(M+N)$ to make it decrease.

Thus the algorithm terminates with probability 1—and we are done.

Exercise 3. Argue informally that the loop

```
$c := H \ _p\oplus \ T$;
do $c \neq H \rightarrow$
    $c := H \ _p\oplus \ T$;
od
```

terminates with probability one provided $p > 0$. Then prove it formally by finding a variant function and using the PROBABILISTIC VARIANT RULE.

Exercise 4. Show that the loop

```
$c, d := H, H$;
do $c = d \rightarrow$
    $c := H \ _p\oplus \ T$;
    $d := H \ _p\oplus \ T$
od
```

establishes $c = H$ on termination with probability $1/2$ for *any* p, provided $0 < p < 1$. (Note that the two coins have the same bias, although it is almost arbitrary: think of it as the same coin flipped repeatedly, where in the loop guard we are comparing the last two results.) *Hint:* Consider the invariant (a real-valued function) defined by the matrix

$$\begin{pmatrix} \frac{1}{2} & 1 \\ 0 & \frac{1}{2} \end{pmatrix}$$

where c selects the row and d selects the column. Do not forget the variant.

Exercise 5. Let $\overline{H}$ be T and $\overline{T}$ be H, so that $d := \overline{d}$ simply turns d over. Show that the loop

```
$c, d := H, H$;
do $c = H \rightarrow$
    $c := H \ _{1/2}\oplus \ T$;
    $d := \overline{d}$
od
```

establishes $d = H$ on termination with probability exactly $1/3$. (This is a good way of dealing with the "one ice-cream, three sons" problem.) *Hint:* Consider the invariant

$$\begin{pmatrix} \frac{1}{3} & \frac{2}{3} \\ 1 & 0 \end{pmatrix}$$

6 Second case study: approximated probabilities, abstraction and refinement

In this case study, we give a small example of a probabilistic program developed in two stages, linked by abstraction and refinement, and in which the issue of "approximate" probabilities is highlighted. This section is based on an example in Hurd's thesis, where, however, the probabilities are exact [125]; we treat the exact case elsewhere [196].

For practical purposes we suppose a source of randomness is available as a stream of unbiased random bits; however many applications' correctness relies on more elaborate distributions. Those distributions can be generated by using various sampling methods; here (Figure 4) we consider a small program which uses (nearly) unbiased bits to generate a (nearly) uniform choice over a positive number N of alternatives. That is, we imagine we have access to a stream of bits, each equally likely to be 0 or 1, but we need to choose uniformly between N alternatives (rather than just 2, which the bits could do directly). We want to write a program to carry this out.

```
{ $[0 \leq K < N]/N_\varepsilon$ }
var $k: \mathbb{N}$ •
  $k := N$;
  do $k \geq N$ →
    var $n: \mathbb{N}$;
    $k, n := 0, N-1$;
    do $n \neq 0$ →
      $k := 2k \;_{\frac{1}{2}-\varepsilon}\!\oplus_{\frac{1}{2}-\varepsilon}\; k := 2k+1$;
      $n := n \,\mathsf{div}\, 2$
    od
  od
{ $[k=K]$ }
```

The inner loop selects k almost uniformly such that $0 \leq k < \$N$, where $\$N$ is the least power-of-two no less than N. The outer loop accepts that choice only if $k < N$; otherwise the inner loop is repeated. The effect overall is to select k almost uniformly so that $0 \leq k < N$.
The pre- and post-expectation annotations express that for any K the probability of achieving $k = K$ on termination is at least $1/N_\varepsilon$ if $0 \leq K < N$ (and at least zero otherwise), where $N_\varepsilon \geq N$. The "excess" $N_\varepsilon - N$ quantifies the inaccuracy, and should tend to zero as ε does.

Fig. 4. Almost-uniform selection algorithm

6.1 Approximation via nondeterminism

Suppose we have access to a stream of bits b each of which is independently unbiased but only to within some tolerance ε, by which we mean that the probability of a 1 (or 0) is only within ε of $1/2$ on each occasion. In *pGCL* we would express this by using the statement

$$b{:=}\ 0\ {}_{\frac{1}{2}-\varepsilon}{\oplus}_{\frac{1}{2}-\varepsilon}\ b{:=}\ 1$$

That is, we are using this statement to model what probably is a piece of hardware, and the ε in the probabilistic-choice operator represents how accurate we have observed this hardware to be: if it were completely accurate ($\varepsilon = 0$) then the statement would be just a "coin flip" of b.

In fact because we will always be "shifting left" our random bits into a bit-string represented by k, we will use the statement

$$k{:=}\ 2k\ {}_{\frac{1}{2}-\varepsilon}{\oplus}_{\frac{1}{2}-\varepsilon}\ k{:=}\ 2k+1 \tag{27}$$

at the point where we access the random bit-stream. We recall that, for $p+q \le 1$ in general, by $\mathsf{This}\ {}_p{\oplus}_q\ \mathsf{That}$ we mean the nondeterministic combination

$$\mathsf{This}\ {}_p{\oplus}\ \mathsf{That}\ \ \sqcap\ \ \mathsf{That}\ {}_q{\oplus}\ \mathsf{This} \tag{28}$$

of the two programs that (on the left) executes This with probability p (and That with probability $1-p$), and (on the right) executes That with probability q (and This with probability $1-q$).

The operational semantics of *pGCL* identifies Program (28) with one that chooses This (rather than That) with any probability r satisfying $p \le r \le 1-q$, because the space of possible program behaviours is "convex-closed" [131, 197, 181], reflecting that nondeterministic choices can be resolved to arbitrary probabilistic ones. Thus the program fragment (27) "flips the coin" with any probability r satisfying $1/2 - \varepsilon \le r \le 1/2 + \varepsilon$, which captures our intended meaning above of "unbiased only to within some tolerance ε". The value of r can vary between separate executions of the fragment, and we recall that it is adversarial in the sense that its choice is treated as worst-case by our program logic, in effect determined by a "demon" whose aim is to make our program as unlikely as possible to produce a uniform distribution.

What is the aim of the program? It is to set k to some value K in the range $[0, N)$, and the effect of the introduced nondeterminism will be to make it less likely to do that than the $1/N$ we would expect of an exactly uniform distribution.

6.2 Overall analysis strategy

We will give a conservative analysis of the program, which is safe but slightly pessimistic, on the grounds that it is simpler than an exact analysis would be, and that for small biases the assurance it gives us is good enough. Informally our reasoning will be as follows.

The inner loop chooses a number k in the range $[0, \$N)$, where $\$N$ is the smallest power-of-two no less than N; it does that by assembling a $\flat N$-bit number via a series of calls to the random bit generator, where $\flat N$ is the minimum number of bits sufficient to represent any number in the given range.

Because the bit generator is biased, however, the minimum guaranteed probability of producing any particular K with $0 \leq K < \$N$ is only $1/\delta^{\flat N}$ (instead of $1/2^{\flat N}$, that is $1/\$N$), where for convenience we set $1/\delta := 1/2 - \varepsilon$. Thus—informally—the maximum guaranteed probability x of producing K in the given range satisfies

$$x \;\; \geq \;\; 1/\delta^{\flat N} + (\$N - N)x/\delta^{\flat N} \tag{29}$$

where the second term is a lower bound on the probability that the inner loop chooses a k that is "too big", that is with $k \geq N$, thus forcing a subsequent iteration. It is only a lower bound because the actual probability of achieving $N \leq k < \$N$ is usually higher, given the way in which k is constructed.

For example, note that although the minimum guaranteed probability of setting k to $\$N{-}1$ is $1/\delta^{\flat N}$, and similarly to $\$N{-}2$, the probability of achieving either, that is $\$N{-}2 \leq k < \$N$, is in fact $1/\delta^{\flat N - 1}$ because in that case only the first $\flat N{-}1$ bits of k are constrained. That is more than the sum $1/\delta^{\flat N} + 1/\delta^{\flat N}$ given by considering the two values separately (unless there is no bias, that is unless $\delta = 2$ exactly).

This is the essence of our abstraction, that we ignore the bit-by-bit structure of k in order to get a good-enough result by simpler means. Solving (29) gives

$$x \;\; \geq \;\; 1/(N + (\delta^{\flat N} - \$N))$$

which identifies the quantity $\delta^{\flat N} - \$N$ as a sort of "excess" E which lowers the probability from the uniform $1/N$ to some $1/N_\varepsilon$ where $N_\varepsilon := N + E$.

6.3 Proofs for inner loop

We analyse the program in two levels, first the inner loop and then the outer loop.

As well as the definitions above, we let $\overline{\flat}n$ be the number of bits used in the binary representation of n, and let $\overline{\$}n$ be the smallest power-of-two strictly exceeding n, so that $2^{\overline{\flat}n} = \overline{\$}n$ (which makes it clear that $\overline{\flat}0 = 0$). These "strict" definitions have slightly better algebraic properties than the "non-strict" ones above, and simplify the calculation. In fact $\overline{\$}N = \$(N{+}1)$ of course, so we are just avoiding a mess of brackets and $+1$'s.

For the inner loop, where the selection range is $\$N$, a nice power of two, it's a reasonable guess that the effect of the bias introduced by the nondeterminism will be to reduce any particular K's chances from $1/\$N$, that is $1/2^{\flat N}$, down to $1/\delta^{\flat N}$—and we note (reassuringly) that when $\varepsilon = 0$ those two probabilities are equal. That suggests the overall precondition for our approximating inner loop, and similar considerations suggest an invariant for it: the resulting annotated loop is shown in Figure 5. In the following, we justify the annotations.

On initialisation This is straightforward; we reason

$$\left[k(\overline{\$}n) \leq K < (k{+}1)(\overline{\$}n)\right] / \delta^{\overline{\flat}n} \qquad \text{invariant}$$

$$\cdot \equiv \qquad \text{applying } wp.(k,n{:=}\ 0, N{-}1)$$

$$\left[0(\overline{\$}(N{-}1)) \leq K < (0{+}1)(\overline{\$}(N{-}1))\right] / \delta^{\overline{\flat}(N-1)}$$

$$\equiv [0 \leq K < \$N] / \delta^{\flat N} \qquad \text{arithmetic gives pre-expectation}$$

While iterating We reason backwards from the end of the loop body towards its beginning. The novelty here is the demonic nondeterminism in ${}_{\frac{1}{\delta}}\oplus_{\frac{1}{\delta}}$ which we interpret as at (28), leading to the use of **min** as indicated by the semantics given at in (7).

$$\left[k(\overline{\$}n) \leq K < (k{+}1)(\overline{\$}n)\right] / \delta^{\overline{\flat}n} \qquad \text{invariant}$$

$$\cdot \equiv \qquad \text{applying } wp.(n{:=}\ n\mathsf{div}2)$$

$$\left[k(\overline{\$}(n\mathsf{div}2)) \leq K < (k{+}1)(\overline{\$}(n\mathsf{div}2))\right] / \delta^{\overline{\flat}(n\mathsf{div}2)}$$

$$\cdot \equiv \qquad \text{applying } wp.(k{:=}\ 2k\ {}_{\frac{1}{\delta}}\oplus_{\frac{1}{\delta}}\ 2k+1)$$

$$\begin{array}{rl}
1/\delta * & \left[\begin{array}{l} \quad (2k)(\overline{\$}(n\mathsf{div}2)) \\ \leq K \\ < ((2k){+}1)(\overline{\$}(n\mathsf{div}2)) \end{array}\right] / \delta^{\overline{\flat}(n\mathsf{div}2)} \\
+ \quad (1{-}1/\delta) * & \left[\begin{array}{l} \quad (2k{+}1)(\overline{\$}(n\mathsf{div}2)) \\ \leq K \\ < ((2k{+}1){+}1)(\overline{\$}(n\mathsf{div}2)) \end{array}\right] / \delta^{\overline{\flat}(n\mathsf{div}2)} \\
\textbf{min} & \\
(1{-}1/\delta) * & \left[\begin{array}{l} \quad (2k)(\overline{\$}(n\mathsf{div}2)) \\ \leq K \\ < ((2k){+}1)(\overline{\$}(n\mathsf{div}2)) \end{array}\right] / \delta^{\overline{\flat}(n\mathsf{div}2)} \\
+ \qquad 1/\delta * & \left[\begin{array}{l} \quad (2k{+}1)(\overline{\$}(n\mathsf{div}2)) \\ \leq K \\ < ((2k{+}1){+}1)(\overline{\$}(n\mathsf{div}2)) \end{array}\right] / \delta^{\overline{\flat}(n\mathsf{div}2)}
\end{array}$$

$$\Lleftarrow \qquad \text{arithmetic; } 1/\delta \leq 1{-}1/\delta$$

$$\begin{array}{rl}
& \left[\begin{array}{l} \quad (2k)(\overline{\$}(n\mathsf{div}2)) \\ \leq K \\ < (2k{+}1)(\overline{\$}(n\mathsf{div}2)) \end{array}\right] / \delta^{\overline{\flat}(n\mathsf{div}2)+1} \\
+ & \left[\begin{array}{l} \quad (2k{+}1)(\overline{\$}(n\mathsf{div}2)) \\ \leq K \\ < (2(k{+}1))(\overline{\$}(n\mathsf{div}2)) \end{array}\right] / \delta^{\overline{\flat}(n\mathsf{div}2)+1}
\end{array}$$

$$\Lleftarrow [n \neq 0] * \qquad 2\overline{\$}(n\mathsf{div}2) = \overline{\$}n;\ \overline{\flat}(n\mathsf{div}2){+}1 = \overline{\flat}n$$

$$\begin{array}{rl}
(& \left[k(\overline{\$}n) \leq K < (2k{+}1)(\overline{\$}(n\mathsf{div}2))\right] / \delta^{\overline{\flat}n} \\
+ & \left[(2k{+}1)(\overline{\$}(n\mathsf{div}2)) \leq K < (k{+}1)(\overline{\$}n)\right] / \delta^{\overline{\flat}n}\ )
\end{array}$$

```
$\{\, [0 \le K < \$N] / \delta^{\flat N} \,\}$
$k, n{:=}\ 0, N{-}1;$
$\{\, \left[k(\overline{\$}n) \le K < (k{+}1)(\overline{\$}n)\right] / \delta^{\overline{\flat}n} \,\}$
do $n \neq 0 \ \rightarrow$
    $\{\, \left[k(\overline{\$}n) \le K < (k{+}1)(\overline{\$}n)\right] / \delta^{\overline{\flat}n} \,\}$
    $k{:=}\ 2k \ {}_{\frac{1}{\delta}}\!\oplus_{\frac{1}{\delta}} \ 2k + 1;$
    $\{\, \left[k\overline{\$}(n\mathsf{div}2) \le K < (k{+}1)\overline{\$}(n\mathsf{div}2)\right] / \delta^{\overline{\flat}(n\mathsf{div}2)} \,\}$
    $n{:=}\ n\ \mathsf{div}\ 2$
    $\{\, \left[k(\overline{\$}n) \le K < (k{+}1)(\overline{\$}n)\right] / \delta^{\overline{\flat}n} \,\}$
od
$\{\, [k{=}K] \,\}$
```

Fig. 5. Approximating inner loop with annotations

$\Leftarrow$ merging inequalities gives guard and invariant

$$[n \neq 0] * \left[k(\overline{\$}n) \le K < (k{+}1)(\overline{\$}n)\right] / \delta^{\overline{\flat}n}$$

On termination This is immediate; we have

$$\begin{array}{ll}
[n = 0] * \left[k(\overline{\$}n) \le K < (k{+}1)(\overline{\$}n)\right] / \delta^{\overline{\flat}n} & \text{negated guard and invariant} \\
\Rrightarrow \left[k(\overline{\$}0) \le K < (k{+}1)(\overline{\$}0)\right] / \delta^{\overline{\flat}0} & \text{arithmetic} \\
\Rrightarrow [k \le K < (k{+}1)] & \overline{\$}0 = 1 \\
\Rrightarrow [k = K] & k, K \in \mathbb{N} \text{ gives post-expectation}
\end{array}$$

6.4 The algebra of abstractions

We now use what we have proved about the inner loop to deduce a property for use in the outer loop. As mentioned in Section 6.2, we are taking a conservative view (though sound) to simplify the calculations. We write $PostE_k^K$ for $PostE$ with variable k replaced by constant K, and use *sub-linearity* from (11) to reason

$$\left.\begin{array}{ll}
wp.\mathsf{Inner}.PostE & \\
\Lleftarrow & \text{arithmetic, } wp.\mathsf{Inner} \text{ monotonic} \\
\quad wp.\mathsf{Inner}.(\sum_{0 \le K < \$N} \ PostE_k^K * [k = K]) & \\
\Lleftarrow & \textit{sublinearity } (11) \text{ used } \$N{-}1 times \\
\quad (\sum_{0 \le K < \$N} \ PostE_k^K * wp.\mathsf{Inner}.\,[k = K]\,) & \\
\Lleftarrow (\sum_{0 \le K < \$N} \ PostE_k^K * [0 \le K < \$N] / \delta^{\flat N}\,) & \text{Section 6.3} \\
\Lleftarrow (\sum_{0 \le K < \$N} \ PostE_k^K) / \delta^{\flat N} & \text{within summation } [0 \le K < \$N] = 1 \\
\Lleftarrow (\sum_{0 \le k < \$N} \ PostE) / \delta^{\flat N} & \text{change bound variable } K \text{ to } k
\end{array}\right\} \quad (30)$$

If we now took (30) as the *wp-definition* of Inner, rather than merely a property of it, we would effectively have an abstraction (*i.e.* an anti-refinement) of the actual inner loop. As usual for refinement, any conclusion we draw about Inner (such as its contribution to the correctness of the outer loop, argued below) are valid for the actual inner loop as well.

6.5 Proofs for outer loop

The complete annotations for our outer loop are given in Figure 6. Since it has nonzero probability $N/\delta^{\flat N}$ of termination on each iteration, its overall termination probability is one. We leave N_ε undetermined for now: at the appropriate moment in the proof we will discover what it must be.

```
{ $[0 \le K < N]/N_\varepsilon$ }
$k := N$;
{ $[k = K] \lhd k < N \rhd [0 \le K < N]/N_\varepsilon$ }
do $k \ge N$ →
    { $[0 \le K < N]/N_\varepsilon$ }
    Inner
    { $[k = K] \lhd k < N \rhd [0 \le K < N]/N_\varepsilon$ }
od
{ $[k = K]$ }
```

Fig. 6. Outer loop, with inner loop abstracted

The on-initialisation and on-termination arguments are trivial. The while-iterating argument is as follows:

$$\begin{array}{ll}
\quad [k = K] \lhd k < N \rhd [0 \le K < N]/N_\varepsilon & \text{invariant} \\
\equiv (\ \sum_{0 \le k < N}\ [k = K] & \text{applying } \textit{wp}.\mathsf{Inner} \text{ from (30), and } \$N \ge N \\
\quad +\sum_{N \le k < \$N}\ [0 \le K < N]/N_\varepsilon & \\
\quad)/\delta^{\flat N} & \\
\equiv (\ [0 \le K < N] & \text{arithmetic} \\
\quad +(\$N - N)\,[0 \le K < N]/N_\varepsilon & \\
\quad)/\delta^{\flat N} & \\
\Lleftarrow [0 \le K < N]/N_\varepsilon & \text{see below} \\
\Lleftarrow [k = K] \lhd k < N \rhd [0 \le K < N]/N_\varepsilon & \text{assuming guard } k \ge N
\end{array}$$

The deferred justification in the second-last step is the information we need to determine N_ε: it is sufficient to have

$$(1 + (\$N - N)/N_\varepsilon)/\delta^{\flat N} \quad \ge \quad 1/N_\varepsilon$$

that is $N_\varepsilon \ge N + (\delta^{\flat N} - \$N) = N + E$, say.

6.6 Discussion

The "excess" E can be regarded as a price we must pay for the bias in our random-bit source: because $\delta \geq 2$ and so $\delta^{\flat N} \geq \$N$, it is never negative; and, as expected, if the bias ε is zero then δ is 2 exactly, making the excess E zero as well.

Another special case is when N is an exact power of two, whence $N = \$N$ and so $N_\varepsilon \geq \delta^{\flat N}$, again as one would expect.

As an example of the general case, we suppose our bit-source is up to 1% biased either way, and we are using it to make uniform selections from 10 alternatives; then we would have

$$\begin{array}{rlcl} & \varepsilon & = & .01 \\ \text{and} & N & = & 10\,, \\ \\ \text{hence} & \delta & = & 1/0.49 \quad \approx \quad 2.04\,, \\ & E & \approx & 2.04^4 - 16 \quad \approx \quad 1.35 \\ \text{and} & N_\varepsilon & \geq & \sim 11.35 \end{array}$$

so that our conservative estimate gives each of our ten choices a guaranteed probability of just under one-in-eleven of being chosen. A more exact but informal analysis in our earlier style would look at the actual bit patterns as follows. The probability of setting $k{:=}\ K$ within the inner loop is at least $1/\delta^4$; otherwise there is a guaranteed probability that k will be set "high" so that the inner loop will be tried again, as in this table:

$$\text{inner-loop outcomes where } k \text{ is "high"} \quad \left\{ \begin{array}{ll} \left.\begin{array}{ll} 10- & 1010 \\ 11- & 1011 \end{array}\right\} & \text{probability } 1/\delta^3 \\ \left.\begin{array}{ll} 12- & 1011 \\ 13- & 1100 \\ 14- & 1101 \\ 15- & 1111 \end{array}\right\} & \text{probability } 1/\delta^2 \end{array} \right.$$

This leads to the inequality

$$x \quad \geq \quad 1/\delta^4 + (1/\delta^2 + 1/\delta^3)x$$

giving $N_\varepsilon \geq \sim 11.14$—which is not much improvement for the extra trouble. In general, exact calculations for the high-outcome probabilities would be unpleasant.

7 Conclusion

It seems that a little generalisation can go a long way: Kozen's use of expectations and the definition of ${}_p\oplus$ as a weighted average [140] is all that is needed for a simple probabilistic semantics, albeit one lacking abstraction. Then He's *sets* of distributions [131] and our *min* for demonic choice together with the fundamental property of sublinearity [197] take us the rest of the way, allowing abstraction and refinement to resume their central role—this time in a probabilistic context. And as Sections 4 and 5 illustrate, many of the standard reasoning principles carry over almost unchanged.

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 randomized algorithms that have been developed over the decades [201]. Our contribution—at this stage—is to make it possible in principle to locate and determine reliably what are the probabilistic/mathematical facts the construction of a randomized algorithm needs to exploit... which is of course just what standard predicate transformers do for conventional algorithms.

In practice however, one is interested not only in certain and correct termination of random algorithms, but in how long they take to do so. Such algorithms' performance cannot be put within bounds in the normal way: instead, one speaks of the *expected* time to termination, how long "on average" should one expect the algorithm to take. When the algorithm is also nondeterministic (as in Rabin's, where no assumptions are made about the order or frequency of the tourists' travels), the estimate would have to be "worst-case" expected.

And there is the larger issue of probabilistic modules, and the associated concern of probabilistic data refinement. That is a challenging problem, with lots of surprises: using our new tools we have already seen that probabilistic modules sometimes do not mean what they seem [183], and that equivalence or refinement between them depends subtly on the power of demonic choice and its interaction with probability.

Other areas in which probabilistic semantics is relevant include concurrent- and relational models. For the former there is an extremely large literature on probabilistic labelled transition systems in the *CCS* style [148, 248, for example], with (as usual) an emphasis on bisimulation; the denotational approach favoured by *CSP* is represented by a smaller but no less elegant body of research [234, 172, 198, 190]. A connection between the latter and our sequential approach can be made via action systems [195].

Probabilistic semantics has attractions for relational programming as well, where programs are represented directly as relations between initial and final states (or as predicates over them) as in the UTP (see Chapter 6). An attractive generalisation is to replace the Booleans by real values, so that the "extended relation" produces directly the probability of making a transition from a given initial to a given final state; a challenge is to do this without losing the ability to describe demonic nondeterminism as well.

Exercise 6. Let n be a natural number. The loop

```
$c, n := H, 0;$
do $c = H \rightarrow$
    $c := H \; {}_{1/2}\oplus \; T;$
    $n := n{+}1$
od
```

terminates with probability one, and can produce any positive integer as the final value of n. Thus it is not *image-finite*, a condition normally considered to be a "well-behavedness" criterion for sequential programs, and guaranteeing their continuity.

But what do we mean by continuity in this context? Is the above program continuous after all? If it is, can you give an example of a *pGCL* program that is not?

Exercise 7. A more immediate approach to probabilistic semantics might be to retain Boolean logic while extending the *wp* modality to include an explicit lower-bound probability: thus

$$wp_p.S.P \tag{31}$$

would describe those initial states from which termination of S in a final state satisfying P was guaranteed with probability at least p. (Free variables in p, if present, would be resolved in the initial state.) Thus we could write for example $wp_{\frac{1}{2}}.(c := H \; {}_{1/2}\oplus \, T).(c = H)$ to describe those states from which $c := H \; {}_{1/2}\oplus \, T$ is guaranteed to establish $c = H$ with probability at least $1/2$ (which is in fact all states).

1. Write the precondition $wp_p.S.P$ in our logic of expectations, thus showing that the latter is at least as expressive.
2. By considering the two programs

$$\begin{array}{ll} & x := A \sqcap (x := B \; {}_{1/2}\oplus \; x := C) \\ \text{and} & (x := A \sqcap x := C) \; {}_{1/2}\oplus \; (x := B \sqcap x := C)\,, \end{array}$$

   show that in fact (31) is not expressive enough.