pGCL: formal reasoning for random algorithms*

Carroll Morgan and Annabelle McIver

Programming Research Group, University of Oxford
http://www.comlab.ox.ac.uk/oucl/groups/probs
{carroll,anabel}@comlab.ox.ac.uk

Abstract

Dijkstra's guarded-command language GCL contains explicit 'demonic' nondeterminism, representing abstraction from (or ignorance of) which of two program fragments will be executed. We introduce probabilistic *nondeterminism to the language, calling the result pGCL. Important is that both forms of nondeterminism are present — both demonic and probabilistic: unlike earlier approaches, we do not deal only with one or the other.*
The programming logic of 'weakest preconditions' for GCL becomes a logic of 'greatest pre-expectations' for pGCL: we embed predicates (Boolean-valued expressions over state variables) into arithmetic by writing $[P]$, an expression that is 1 when P holds and 0 when it does not. Thus in a trivial sense $[P]$ is the probability that P is true, and such embedded predicates are the basis for the more elaborate arithmetic expressions that we call "expectations". pGCL is suitable for describing random algorithms, at least over discrete distributions. In our presentation of it and its logic we give two examples: an erratic 'sequence accumulator', that fails with some probability to move along the sequence; and Rabin's 'choice-coordination' algorithm. The first illustrates probabilistic invariants; the second illustrates probabilistic variants.
Keywords: *Program correctness, probability, demonic nondeterminism, random algorithm, predicate transformer, weakest precondition, guarded command, correctness proof, invariant, variant.*
Computing Review Categories: *D.2.4, D.3.1, F.1.2, F.3.1, G.1.6, G.3.*

1 Introduction

Dijkstra's Guarded Command Language *GCL* [2] is a weakest-precondition based method of describing computations and their meaning; here we extend it to probabilistic programs, those that implement random algorithms, and we give examples of its use.

Most sequential programming languages contain a construct for 'deterministic' choice, where the program selects one from a number of alternatives in some predictable way: for example, in

$$\textbf{if } \mathit{test} \textbf{ then } \mathit{this} \textbf{ else } \mathit{that} \textbf{ fi} \qquad (1)$$

the choice between *this* and *that* is determined by *test* and the current state.

In contrast, Dijkstra's language of guarded commands brings nondeterministic or 'demonic' choice to prominence, in which the program's behaviour is *not* predictable, not determined by the current state. At first [2], demonic choice was presented as a consequence of 'overlapping guards', almost an accident — but as its importance became more widely recognised it developed a life of its own. Nowadays it merits an explicit operator: the construct

$$\mathit{this} \sqcap \mathit{that}$$

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.

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 [9, 4, 3, 8, 7]. Thus probabilistic semantics was divorced from the contemporaneous work on specification and refinement, because without demonic choice there is no means of abstraction.

More recently however it has been discovered [6, 15] 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*'.

Section 2 gives a brief and shallow overview of

*Part of this report is a 'transliteration' of another report [16] from generalised substitutions [1] to guarded commands. The case study (Rabin's algorithm) has not appeared before.

pGCL, somewhat informal and concentrating on simple examples. Section 3 sets out the definitions and properties of *pGCL* systematically, and Sec. 4 treats an example of reasoning about probabilistic loops, showing how to use probabilistic invariants. Section 5 illustrates probabilistic variants with a thorough treatment of Rabin's choice-coordination algorithm [18].

An impression of *pGCL* can be gained by reading Secc. 2 and 4, with finally a glance over Secc. 3.1 and 3.2; more thoroughly one would read Secc. 2, 3.1 and 3.2, then 2 (again) and finally 4. The more theoretical Sec. 3.3 can be skipped on first reading; and Sec. 5 can be read independently.

Throughout we write $f.x$ instead of $f(x)$ for function f applied to argument x, with left association; and we use $:=$ for "is defined to be".

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 $[\mathsf{false}]$ is 0 and $[\mathsf{true}]$ 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

$$x:=\ -y \quad {}_{\frac{1}{3}}\oplus \quad x:=\ +y \qquad (2)$$

over variables $x, y\colon \mathbb{Z}$, using a 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 [2] that for any predicate P over *final* states, and a standard[1] 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. 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 generalise $wp.S$ to expectations instead of predicates. For that, we begin with the two definitions

$$\begin{array}{lcll} wp.(x:=\ E).R & := & \text{`}R \text{ with } x \text{ replaced everywhere by } E\text{'}^{2} & (3)\\ wp.(S\ {}_p\oplus\ T).R & := & p * wp.S.R & (4)\\ & & +\quad (1{-}p) * wp.T.R\ , & \end{array}$$

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)?

[1] Throughout we use *standard* to mean 'non-probabilistic'.

[2] 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

$$\begin{array}{lll} & wp.(x:=\ -y\ {}_{\frac{1}{3}}\oplus\ x:=\ +y).\,[x \geq 0] & \\ \equiv^{3} & (1/3) * wp.(x:=\ -y).\,[x \geq 0] & \text{using (4)}\\ & +\quad (2/3) * wp.(x:=\ +y).\,[x \geq 0] & \\ \equiv & (1/3)\,[-y \geq 0] + (2/3)\,[+y \geq 0] & \text{using (3)}\\ \equiv & & \text{using arithmetic}\\ & [y<0]\,/3\ +\ [y=0]\ +\ 2\,[y>0]\,/3\ . & \end{array}$$

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

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

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

The above remarkable generalisation of sequential program correctness is due to Kozen [9], but until recently was restricted to programs that did not contain demonic choice $\sqcap$. When He *et al.* [6] and Morgan *et al.* [15] 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.

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

- $x:=\ -y$ 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.

(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

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

because we do not know or care whether the left or right alternative of $\sqcap$ is taken — and it may even vary

[3] Later we explain the use of '$\equiv$' rather than '$=$'.

from run to run of the program, resulting in an 'effective' ${}_p\oplus$ with p somewhere between the two extremes.[4]

To treat Program (6), we define the command

$$wp.(S \sqcap T).R \quad := \quad wp.S.R \ \mathsf{min}\ wp.T.R \;, \qquad (7)$$

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

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

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 substitutions S, T we define

$$S \sqsubseteq T \quad := \quad wp.S.R \Rrightarrow wp.T.R \quad \text{for all } R, \qquad (8)$$

where we write $\Rrightarrow$ for 'everywhere no more than' (which ensures $[\mathsf{false}] \Rrightarrow [\mathsf{true}]$ 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 $wp.S.[P]$ is either 0 or 1, and whenever S is certain to establish P (whenever $wp.S.[P] \equiv 1$) we know that T also is certain to do so (because then $1 \Rrightarrow wp.T.[P]$).

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

$$x := -y \quad {}_{\frac{1}{2}}\oplus \quad x := +y \;, \qquad (9)$$

which clearly satisfies Specification (5); thus it should refine Program (6). With Definition (8), we find for any R that

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

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

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

3 Presentation of probabilistic *GCL*

In this section we give a concise presentation of probabilistic *GCL* — *pGCL* — as a whole: 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 $[0, 1]$. To retain the use of predicates, we allow expectations of the form $[P]$ when P

[4] A convenient notation for (6) would be based on the abbreviation

$$S \ {}_{[p,q]}\oplus\ T \quad := \quad S\ {}_p\oplus T \ \sqcap \ S\ {}_q\oplus T \;;$$

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

is Boolean-valued, defining [false] to be 0 and [true] to be 1.

Implication-like relations between expectations are

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

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

The definitions of the substitutions 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 — very nearly — to be just the probability that the program will reach P. That is because

expected worth of initial state

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

where matters are greatly simplified by the fact that all states satisfying P have the same worth.

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

$$p \quad \equiv \quad wp.S.[P]$$

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[5] 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

$$wp.\left(\begin{array}{ll} x{:=}\ H \ {}_{\frac{1}{2}}{\oplus}\ x{:=}\ T & ; \\ y{:=}\ H \ {}_{\frac{1}{2}}{\oplus}\ y{:=}\ T & \end{array}\right).[x = y]$$

$$\begin{array}{ll} \equiv & \hfill {}_{\frac{1}{2}}{\oplus},\ := \text{ and sequential composition} \\ & wp.(x{:=}\ H \ {}_{\frac{1}{2}}{\oplus}\ x{:=}\ T).([x = H]/2 + [x = T]/2) \\ & \\ \equiv & \hfill {}_{\frac{1}{2}}{\oplus} \text{ and } := \\ & \quad (1/2)([H = H]/2 + [H = T]/2) \\ & + \ (1/2)([T = H]/2 + [T = T]/2) \\ & \\ \equiv & \hfill \text{definition } [\cdot] \\ & (1/2)(1/2 + 0/2) + (1/2)(0/2 + 1/2) \\ \equiv & 1/2\ . \hfill \text{arithmetic} \end{array}$$

We can then use the second interpretation above to conclude that the faces are the same with probability (at least[6]) 1/2.

But part of the above calculation involves the more general expression

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

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 $[x = H]/2 + [x = T]/2$ after executing $x{:=}\ H \ {}_{\frac{1}{2}}{\oplus}\ x{:=}\ T$,

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.

3.3 Properties of *pGCL*

Recall that all *GCL* constructs satisfy the property of conjunctivity[7] — 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' [2] is used to prove general properties of programs.

[5] We must say 'at least' in general, because of possible demonic choice in S; and some analyses give only the weaker $p \Rrightarrow wp.S.[P]$ in any case.

[6] Knowing there is no demonic choice in the program, we can in fact say it is exact.

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

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
$wp.(S;T).R$	$wp.S.(wp.T.R)$
$wp.(S \sqcap T).R$	$wp.S.R \ \mathsf{min}\ wp.T.R$
$wp.(S\ _p\oplus\ T).R$	$p * wp.S.R \ + \ (1-p) * wp.T.R$
$S \sqsubseteq T$	$wp.S.R \Rrightarrow wp.T.R$ for all R

- R is an expectation (possibly but not necessarily $[P]$ for some 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 $S\ _{[B]}\oplus\ T$. 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.

Figure 1: *pGCL* — the probabilistic Guarded Command Language

In $pGCL$ the healthiness condition becomes 'sub-linearity' [15], a generalisation of conjunctivity:

> SUB-LINEARITY
>
> Let a, b, c be non-negative finite reals, and R, R' expectations; then all $pGCL$ constructs S satisfy
>
> $$\begin{array}{ll} & wp.S.(aR + bR' \ominus c) \\ \Leftarrow & a(wp.S.R) + b(wp.S.R') \ominus c \ , \end{array}$$
>
> which property of S is called *sublinearity.* We have written aR for $a * R$ *etc.*, and truncated subtraction $\ominus$ is defined
>
> $$x \ominus y \quad := \quad (x - y) \operatorname{\mathsf{max}} 0 \ ,$$
>
> with 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 [15]. We begin with monotonicity, feasibility and scaling.[8]

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

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

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

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

so that $wp.S.0$ must be 0.

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

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

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

$$wp.S.R \quad \Rrightarrow \quad \mathsf{max}\ R \ , \tag{10}$$

which we identify as the *feasibility* condition[9] for $pGCL$.

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

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

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

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

thus establishing $wp.S.(aR) \equiv a(wp.S.R)$ generally.

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 that

$$\begin{array}{rcl} 0 \,\&\, 0 & = & 0 \\ 0 \,\&\, 1 & = & 0 \\ 1 \,\&\, 0 & = & 0 \\ 1 \,\&\, 1 & = & 1 \end{array} \tag{11}$$

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. Instead we define

$$R \,\&\, R' \quad := \quad R + R' \ominus 1 \ , \tag{12}$$

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.

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

$$wp.S.(R \,\&\, R') \quad \Leftarrow \quad wp.S.R \ \& \ wp.S.R'$$

for all S.

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

Beyond sub-conjunctivity, we say that $\&$ generalises conjunction for several other reasons. The first is of course that it satisfies the standard properties (11).

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

[8] Sublinearity characterises probabilistic *and demonic* substitutions. In Kozen's original probability-only formulation [9] the substitutions are not demonic, and there they satisfy the much stronger property of 'linearity' [11].

[9] Conveniently, the the general (10) implies the earlier special case $wp.S.0 \equiv 0$.

pre-expectations: they are the embedding of GCL in $pGCL$, and for standard S we now show that

$$wp.S.([P] \mathbin{\&} [P']) \quad \equiv \quad wp.S.[P] \ \mathbin{\&} \ wp.S.[P'] \ . \quad (13)$$

First note that '$\Leftarrow$' 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 $wp.S.([P] \mathbin{\&} [P']) \Rrightarrow wp.S.[P]$, and similarly for P'. Putting those together gives

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

by elementary arithmetic properties of $\Rrightarrow$. But on standard expectations — which $wp.S.[P]$ and $wp.S.[P']$ 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:

> if the probability of A is at least p, and the probability of B is at least q,
>
> then the most that can be said about the joint event $A \cap B$ is that it has probability at least $p \mathbin{\&} q$ [19].

The & operator also plays a crucial role in the proof [14] (not given in this paper) of the probabilistic loop rule presented and used in the next section.

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 — 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 can't 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: and 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

$$p \quad * \quad [I] \qquad (14)$$

with p a probability and I a predicate, both expressions over the state. From the definition of $[\cdot]$ we know that the interpretation of (14) is

> probability p if I holds, and probability 0 otherwise.

We see an example of such invariants below.

4.2 Termination

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

$$wp.S.1 \ . \qquad (15)$$

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

$$S \quad := \quad S \ {}_p\!\oplus \ \mathbf{skip} \ , \qquad (16)$$

in which we assume that p is some constant strictly less than 1: on each recursive call, P has probability $1-p$ of termination, continuing otherwise with further recursion.[10] By calculation based on (15) we see that

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

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 Sec. 5.

[10] Elementary probability theory shows that S terminates with probability 1 (after an expected $p/(1-p)$ recursive calls).

4.3 Probabilistic correctness of loops

As in the standard case, it is easy[11] to show that if $[P] * I \Rrightarrow [S]I$ then

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

provided[12] the loop terminates. Thus the notion of invariant carries over smoothly from the standard to the probabilistic case.

When termination is taken into account as well, we get the following rule [14].

> PROOF RULE FOR PROBABILISTIC LOOPS
>
> For convenience write T for the termination probability of the loop, so that
>
> $$T \quad := \quad wp.(\mathbf{do}\ P \to S\ \mathbf{od}).1\ .$$
>
> Then partial loop correctness — preservation of a loop invariant I — implies total loop correctness if that invariant I nowhere[13] exceeds T: that is,
>
> $$\begin{array}{ll} \text{if} & [P] * I \Rrightarrow wp.S.I \\ \text{and} & I \Rrightarrow T \\ \text{then} & \end{array}$$
>
> $$I \Rrightarrow wp.(\mathbf{do}\ P \to S\ \mathbf{od}).([\neg P]*I)\ .$$

We illustrate the loop rule with a simple example. Suppose we have a machine that is supposed to sum the elements of a sequence, except that the mechanism for moving along the sequence occasionally moves the wrong way. A program for the machine is given in Fig. 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

$$r \quad = \quad \sum ss \tag{17}$$

on termination?

We first find the invariant: relying on our informal discussion above, we ask

> during the loop's execution, with what probability are we in a state from which completion of the loop would establish (17)?

The answer is in the form (14) — 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 it is

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

Having chosen a possible invariant, to check it we calculate

$$\begin{array}{lll}
& wp.\left(\begin{array}{l} r{:=}\ r+ss.k; \\ k{:=}\ k+1\ {}_c\oplus\ k{:=}\ k-1 \end{array}\right).J & \\[2ex]
\equiv & \begin{array}{l} wp.(r{:=}\ ss.k).(\\ \quad c * wp.(k{:=}\ k+1).J \\ + \quad (1{-}c) * wp.(k{:=}\ k-1).J) \end{array} & \text{; and } {}_c\oplus \\[2ex]
\Leftarrow & wp.(r{:=}\ r+ss.k). \quad c^{N-k} \quad * \quad \left[\begin{array}{c} 0 \le k+1 \le N \\ r = \sum ss[0..k) \end{array}\right] & \text{drop second term, and } := \\[2ex]
\equiv & c^{N-k} \quad * \quad \left[\begin{array}{c} 0 \le k+1 \le N \\ r+ss.k = \sum ss[0..k) \end{array}\right] & := \\[2ex]
\Leftarrow & [k < N] * J\ , &
\end{array}$$

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.

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

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

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

$$\begin{array}{ll}
& wp.(r,k{:=}\ 0,0).J \\
\equiv & c^N \quad * \quad \left[\begin{array}{c} 0 \le 0 \le N \\ 0 = \sum ss[0..0) \end{array}\right] \\
\equiv & c^N \quad * \quad [\mathsf{true}] \\
\equiv & c^N\ ,
\end{array}$$

and our overall conclusion is therefore

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

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 $k{:=}\ k+1$;

---

[11] It 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.

[12] The precise treatment of 'provided' uses weakest *liberal* pre-expectations [14, 12].

[13] 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.

```
con   ss[0..N): ℤ;
var   r: ℤ;

|[  var k: ℤ;
    r, k:= 0, 0;
    do k < N →
       r:= r + ss.k;
       k:= k + 1  c⊕  k:= k − 1          ← failure possible here
    od
]|
```

Figure 2: An unreliable sequence-summer

but 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 [20]), rather than just lower bounds as we did above, this method would not be at all practical.

Further examples of loops are given elsewhere [14].

5 Case study: Rabin's choice-coordination

5.1 Introduction

Rabin's choice-coordination algorithm (explained in Secc. 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 [18] explains the algorithm he invented[14], but 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[15]: 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.

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.[16]

The termination argument is novel however, since probabilistic variant techniques [5, 14] must be used.

5.2 Informal description of Rabin's algorithm

This informal description is based on Rabin's presentation [18].

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 (nondeterministically) 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 K on his notepad (erasing k), and goes to the other place.

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.

[14] ...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 bat.

[15] "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.

[16] More precisely, replacing probabilistic choice by nondeterministic choice is an anti-refinement.

He then writes K' on the noticeboard and on his notepad (erasing k and K), and goes to the other place.[17]

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 (Fig. 3). Each tourist is represented by an instance of the number on his pad.

5.3.1 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*)).

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.

5.3.2 Notation

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

- $[\![\cdots]\!]$ — Bag (multiset) brackets.[18]
- $\Box$ — 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 $n \mathsf{\ min\ } \overline{n}$ 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 $1{-}p$.

5.3.3 Correctness criteria

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

$$\begin{array}{ll} & \#lin = M{+}N \ \wedge\ rin = \Box \\ \vee & lin = \Box \ \wedge\ \#rin = M{+}N\ . \end{array}$$

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.

5.4.1 Three invariants

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

$$\#lout + \#lin + \#rout + \#rin \quad = \quad M + N\ . \qquad (18)$$

It holds initially, and is trivially maintained.

The second invariant is

$$\begin{array}{l} lin, lout \leq R \\ rin, rout \leq L\ , \end{array} \qquad (19)$$

and expresses that a tourist's number never exceeds the number posted at the *other* place.[19] 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;

[17] For example if K is 8 or 9, first K' becomes 10 and then possibly 11.

[18] Bags are like sets except that they can have several copies of each element: the bag $[\![1,1]\!]$ contains two copies of 1, and is not the same as $[\![1]\!]$.

[19] By $b \leq K$ we mean that no element in the bag b exceeds the integer K.

$$
\begin{array}{l}
\mathit{lout}, \mathit{rout} := \ \llbracket 0 \rrbracket^{M}, \llbracket 0 \rrbracket^{N}; \\
\mathit{lin}, \mathit{rin} := \ \Box, \Box; \\
L, R := \ 0, 0; \\
\\
\textbf{do} \ \mathit{lout} \neq \Box \rightarrow \\
\quad \textbf{take} \ l \ \textbf{from} \ \mathit{lout}; \\
\quad \textbf{if} \ \mathit{lin} \neq \Box \ \textbf{then} \ \ \textbf{add} \ l \ \textbf{to} \ \mathit{lin} \ \ \textbf{else} \\
\qquad l > L \ \ \rightarrow \ \textbf{add} \ l \ \textbf{to} \ \mathit{lin} \\
\quad [\!] \quad l = L \ \ \rightarrow \ L := \ L + 2 \ {}_{\frac{1}{2}}\!\oplus \overline{(L+2)}; \quad \textbf{add} \ L \ \textbf{to} \ \mathit{rout} \\
\quad [\!] \quad l < L \ \ \rightarrow \ \textbf{add} \ L \ \textbf{to} \ \mathit{rout} \\
\quad \textbf{fi} \\
\\
[\!] \quad \mathit{rout} \neq \Box \rightarrow \\
\quad \textbf{take} \ r \ \textbf{from} \ \mathit{rout}; \\
\quad \textbf{if} \ \mathit{rin} \neq \Box \ \textbf{then} \ \ \textbf{add} \ r \ \textbf{to} \ \mathit{rin} \ \ \textbf{else} \\
\qquad r > R \ \ \rightarrow \ \textbf{add} \ r \ \textbf{to} \ \mathit{rin} \\
\quad [\!] \quad r = R \ \ \rightarrow \ R := \ R + 2 \ {}_{\frac{1}{2}}\!\oplus \overline{(R+2)}; \quad \textbf{add} \ R \ \textbf{to} \ \mathit{lout} \\
\quad [\!] \quad r < R \ \ \rightarrow \ \textbf{add} \ R \ \textbf{to} \ \mathit{lout} \\
\quad \textbf{fi} \\
\textbf{od}
\end{array}
$$

Figure 3: Rabin's choice-coordination algorithm

- Adding elements to $\mathit{lin}, \mathit{rin}$ cannot falsify it, since those elements come from $\mathit{lout}, \mathit{rout}$;
- The only commands adding elements to $\mathit{lout}, \mathit{rout}$ are

  $$\textbf{add} \ L \ \textbf{to} \ \mathit{rout} \quad \text{and} \quad \textbf{add} \ R \ \textbf{to} \ \mathit{lout} \ ,$$

  and they maintain it trivially.

  Our final invariant for partial correctness is

$$
\begin{array}{lll}
\max \mathit{lin} > L & \text{if} & \mathit{lin} \neq \Box \\
\max \mathit{rin} > R & \text{if} & \mathit{rin} \neq \Box \ ,
\end{array}
\tag{20}
$$

expressing that if any tourist has gone inside, then at least one of the tourists inside there 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 $\mathit{lin} = \Box$); 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 $\mathit{lin} = \Box$ to establish $\mathit{lin} = \llbracket l \rrbracket$ for some $l > L$.

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

5.4.2 On termination...

On termination we have $\mathit{lout} = \mathit{rout} = \Box$, and so with invariant (18) we need only

$$\mathit{lin} = \Box \quad \vee \quad \mathit{rin} = \Box \ .$$

Assuming for a contradiction that both lin and rin are non-empty, we then have from invariants (19) and (20) the inequalities

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

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

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

differing from the initial state only in the use of 4's rather than 0's. (All coin flips came up heads, and each tourist had exactly two turns.) Since the program contains no absolute comparisons[20], we are effectively back where we started — and 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 invariant whose strict decrease is guaranteed.

[20] The program checks only whether various numbers are greater than others, not what the numbers actually are.

Instead we appeal to the following rule [5, 14]:

> 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.[21]

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.

5.5.1 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 (19) those numbers are bounded above by $L \max R$.

The inner variant (increasing) is based on that idea, with some care taken however to make sure that it is bounded above and below by fixed values, independent of L and R.[22] We define $V0$ to be

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

It is trivially bounded above by $3(M+N)$, and 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 $lin \neq \Box$ 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.

5.5.2 Outer variant: changes to L and R

For the outer variant we need further invariants; the first is

$$\widetilde{L} - \widetilde{R} \quad \in \quad \{-2, 0, 2\}\ , \tag{21}$$

stating that the notice-board values can never be 'too far apart'. It holds initially; and, from invariant (19), 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:

- $\widetilde{L} = \widetilde{R} - 2 \lor \widetilde{L} = \widetilde{R} + 2$ — write $L \not\approxeq R$ for those states.
- $\overline{L} = R$ (equivalently $L = \overline{R}$) — write $L \approxeq R$.
- $L = R$.

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 $\rightarrow$.)

$$\begin{array}{lllll} \langle L \not\approxeq R \rangle & \rightarrow & \langle L \not\approxeq R \rangle & \sqcap & \langle L = R \rangle\ {}_{\frac{1}{2}}\!\oplus \langle L \approxeq R \rangle \\ \langle L = R \rangle & \rightarrow & \langle L = R \rangle & \sqcap & \langle L \not\approxeq R \rangle \\ \langle L \approxeq R \rangle & \rightarrow & \langle L \approxeq R \rangle & & \end{array}$$

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

$$\overline{L} \notin rout \quad \land \quad \overline{R} \notin lout\ . \tag{22}$$

It holds initially, and cannot be falsified by the command **add** L **to** $rout$, because $\overline{L} \neq L$. That leaves

[21] 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's no distinction.

[22] The independence of L, R is important, given our variant rule, because L and R can themselves increase without bound.

the command $L := L + 2 \;{}_{\frac{1}{2}}\!\oplus\; \overline{(L+2)}$; but in that case from (19) we have

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

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

Thus with (22) 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 $L \in lout$, impossible if $L \cong R$ and $\overline{R} \notin lout$.

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

$$\begin{array}{ll} 2, & \text{if } L = R \\ 1, & \text{if } L \not\cong R \\ 0, & \text{if } L \cong R \;, \end{array} \qquad (23)$$

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

5.5.3 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.[23] In particular it has probability at least $1/2$ of strict decrease on *every* iteration of the loop.

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

6 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 [9] is all that is needed for a simple probabilistic semantics, albeit one lacking abstraction. Then He's *sets* of distributions [6] and our min for demonic choice together with the fundamental property of sublinearity [15] take us the rest of the way, allowing abstraction and refinement to resume their central role — this time in a probabilistic context. And as Secc. 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 randomised algorithms that have been developed over the decades [17]. 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 randomised algorithm needs to exploit... which is of course just what standard predicate transformers do for conventional algorithms.

[23] Actually the inner variant increases rather than decreases — we could subtract it from $3(M+N)$ to make it decrease.

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. Using techniques like the above to answer those questions is a topic of current research [10].

Finally, 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 [13], and that equivalence or refinement between them depends subtly on the power of demonic choice and its interaction with probability.

Acknowledgements

This paper reports work carried out with Jeff Sanders and Karen Seidel, and supported by the *EPSRC*.

References

[1] J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.

[2] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall International, Englewood Cliffs, N.J., 1976.

[3] Yishai A. Feldman. A decidable propositional dynamic logic with explicit probabilities. *Information and Control*, 63:11–38, 1984.

[4] Yishai A. Feldman and David Harel. A probabilistic dynamic logic. *J. Computing and System Sciences*, 28:193–215, 1984.

[5] S. Hart, M. Sharir, and A. Pnueli. Termination of probabilistic concurrent programs. *ACM Transactions on Programming Languages and Systems*, 5:356–380, 1983.

[6] Jifeng He, K. Seidel, and A. K. McIver. Probabilistic models for the guarded command language. *Science of Computer Programming*, 28:171–192, 1997.

[7] C. Jones. Probabilistic nondeterminism. Monograph ECS-LFCS-90-105, Edinburgh Univ. Edinburgh, U.K., 1990. (PhD thesis.)

[8] C. Jones and G. Plotkin. A probabilistic powerdomain of evaluations. In *Proceedings of the 4th*

IEEE Annual Symposium on Logic in Computer Science, pages 186–195, Los Alamitos, Calif., 1989. Computer Society Press.

[9] D. Kozen. A probabilistic PDL. In *Proceedings of the 15th ACM Symposium on Theory of Computing*, New York, 1983. ACM.

[10] Annabelle McIver. Reasoning about efficiency within a probabilistic μ-calculus. In *Proceedings PROBMIV98*, June 1988.

[11] Annabelle McIver and Carroll Morgan. Probabilistic predicate transformers: part 2. Technical Report PRG-TR-5-96, Programming Research Group, March 1996.

[12] Annabelle McIver and Carroll Morgan. Partial correctness for probabilistic demonic programs. Technical Report PRG-TR-35-97, Programming Research Group, 1997. Revised version to be submitted for publication under the title *Demonic, angelic and unbounded probabilistic choices in sequential programs.*

[13] Annabelle McIver, Carroll Morgan, and Elena Troubitsyna. The probabilistic steam boiler: a case study in probabilistic data refinement. To appear in *Proceedings of the International Refinement Workshop*, Canberra 1998.

[14] 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.

[15] Carroll Morgan, Annabelle McIver, and Karen Seidel. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems*, 18(3):325–353, May 1996.

[16] Carroll Morgan. The generalised substitution language extended to probabilistic programs. *Proceedings B'98 Conference*, Montpellier, May 1998. Didier Bert Ed. Springer Verlag LNCS 1393.

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

[18] M. O. Rabin. The choice-coordination problem. *Acta Informatica*, 17(2):121–134, June 1982.

[19] K. Seidel, C. C. Morgan, and A. K. McIver. An introduction to probabilistic predicate transformers. Technical Report PRG-TR-6-96, Programming Research Group, February 1996.

[20] M. Sharir, A. Pnueli, and S. Hart. Verification of probabilistic programs. *SIAM Journal on Computing*, 13(2):292–314, May 1984.