The Generalised Substitution Language extended to probabilistic programs

Carroll Morgan

Programming Research Group, Oxford University, UK.
carroll@comlab.ox.ac.uk,
http://www.comlab.ox.ac.uk/oucl/groups/probs.
The work is supported by the EPSRC.

Abstract. Let predicate P be converted from Boolean to numeric type by writing $\langle P \rangle$, with $\langle \mathsf{false} \rangle$ being 0 and $\langle \mathsf{true} \rangle$ being 1, so that in a degenerate sense $\langle P \rangle$ can be regarded as 'the probability that P holds in the current state'. Then add explicit numbers and arithmetic operators, to give a richer language of arithmetic formulae into which predicates are embedded by $\langle \cdot \rangle$.
Abrial's generalised substitution language *GSL* can be applied to arithmetic rather than Boolean formulae with little extra effort. If we add a new operator ${}_p\!\oplus$ for probabilistic choice, it then becomes '*pGSL*': a smooth extension of *GSL* that includes random algorithms within its scope.

Keywords: Probability, program correctness, generalised substitutions, weakest preconditions, B, *GSL*.

1 Introduction

Abrial's Generalised Substitution Language *GSL* [1] is a weakest-precondition based method of describing computations and their meaning; it is complemented by the structures of Abstract Machines, together with which it provides a framework for the development of correct systems. In this paper we extend it to probabilistic programs, those that implement random algorithms.

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

$$\text{IF } \textit{test} \text{ THEN } \textit{this} \text{ ELSE } \textit{that} \text{ END} \tag{1}$$

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

$$this \;[\!]\; that$$

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.

With the invention of 'miracles' [12, 16, 18] the two forms of choice were unified, showing demonic choice to be the more fundamental: that innovation is exploited in *GSL* whenever one writes

$$test \to this \quad [\!] \quad \neg test \to that \tag{2}$$

instead of the more conventional (1) above.

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 [8, 4, 3, 7, 6]. 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 [5, 15] how to bring the two topics back together, taking the more natural approach of *adding* probabilistic choice, retaining demonic choice and seeing deterministic choice again as at (2). Probabilistic choice too is a special case of demonic choice: both deterministic and probabilistic choice refine demonic choice, but neither refines the other.

Because the probabilistic/demonic semantics is an extension of predicate transformers, it is possible to present its main ideas in the *GSL* idiom, which is what we do in this paper. The result could be called '*pGSL*'.

In Sec. 2 we give a brief and shallow overview of *pGSL*, somewhat informal and concentrating on simple examples. Sec. 3 sets out the definitions and properties of *pGSL* systematically, and Sec. 4 treats an example of reasoning about probabilistic loops.

An impression of *pGSL* 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 Sec. 3.3 can be skipped on first reading.

2 An impression of *pGSL*

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

For our first example, we consider the simple program

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

over variables $x, y{:}\,\mathbb{Z}$, 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 that for any predicate P over *final* states, and a standard[1] substitution S, the predicate $[S]P$ acts over *initial* states: it holds in those initial states from which S is guaranteed to reach P. Now suppose S is probabilistic, as Program (3) is; what can we say about the *probability* that $[S]P$ holds in some initial state?

It turns out that the answer is just $[S]\,\langle P\rangle$, once we generalise $[S]$ to expectations instead of predicates. We begin with the two definitions

$$[x{:=}\ E]R \quad \widehat{=} \quad \text{'}R \text{ with } x \text{ replaced everywhere'}^2 \text{ by } E \tag{4}$$

$$[S\ {}_p\!\oplus\ T]R \quad \widehat{=} \quad p*[S]R\ +\ (1{-}p)*[T]R\ , \tag{5}$$

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

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?

To find out, we calculate $[S]\,\langle P\rangle$ in this case; that is

$$[x{:=}\ -y \;{}_{\frac{1}{3}}\!\oplus\; x{:=}\ +y]\,\langle x \geq 0\rangle$$

$$\begin{array}{lll}
\equiv^{3} & (1/3) * [x{:=}\ -y]\,\langle x \geq 0\rangle & \text{using (5)} \\
& + (2/3) * [x{:=}\ +y]\,\langle x \geq 0\rangle & \\
& & \\
\equiv & (1/3)\,\langle -y \geq 0\rangle + (2/3)\,\langle +y \geq 0\rangle & \text{using (4)} \\
\equiv & \langle y < 0\rangle\,/3 \;+\; \langle y = 0\rangle \;+\; 2\,\langle y > 0\rangle\,/3\ . & \text{arithmetic}
\end{array}$$

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

> when y is negative, $\quad 1/3 + 0 + 2(0)/3 = 1/3$
> when y is zero, $\quad 0/3 + 1 + 2(0)/3 = \quad 1$
> when y is positive, $\quad 0/3 + 0 + 2(1)/3 = 2/3\,.$

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 [8], but until recently was restricted to programs that did not contain demonic choice $[\!]$. When He *et al.* [5] 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 (3) is much

[2] In the usual way, we take account of free and bound variables, and if necessary rename to avoid variable capture.
[3] Later we explain the use of '$\equiv$' rather than '$=$'.

more realistic in that we set its probabilistic parameters only within some tolerance. We say informally (but still with precision) that

$$\left.\begin{array}{l} \bullet\ x{:=}\ -y \text{ is to be executed with probability } \textit{at least } 1/3, \\ \bullet\ x{:=}\ +y \text{ is to be executed with probability } \textit{at least } 1/4 \\ \quad \text{and} \\ \bullet\ \text{it is certain that one or the other will be executed.} \end{array}\right\} \quad (6)$$

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 (6) as

$$(x{:=}\ -y \ {}_{\frac{1}{3}}\!\oplus\ x{:=}\ +y) \quad [\!] \quad (x{:=}\ -y \ {}_{\frac{3}{4}}\!\oplus\ x{:=}\ +y)\ , \qquad (7)$$

because we do not know or care whether the left or right alternative of $[\!]$ 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.[4]

To examine Program (7), we define the generalised substitution

$$[S\ [\!]\ T]R \quad \widehat{=} \quad [S]R\ \ \mathsf{min}\ \ [T]R\ , \qquad (8)$$

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

$$\begin{array}{lll} & [\text{Program (7)}]\,\langle x \geq 0\rangle & \\ \equiv & \begin{array}{ll} & \langle y \leq 0\rangle\,/3 + 2\,\langle y \geq 0\rangle\,/3 \\ \mathsf{min} & 3\,\langle y \leq 0\rangle\,/4 + \langle y \geq 0\rangle\,/4 \end{array} & \text{using (4), (5), (8)} \\ \equiv & \langle y < 0\rangle\,/3 \ +\ \langle y = 0\rangle \ +\ \langle y > 0\rangle\,/4\ . & \text{arithmetic} \end{array}$$

Our interpretation is now

[4] A convenient notation for (7) would be based on the abbreviation

$$S\ {}_{[p,q]}\!\oplus\ T \quad \widehat{=} \quad S\ {}_{p}\!\oplus T\ [\!]\ S\ {}_{q}\!\oplus T\ ;$$

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

- When y is initially negative, the demon chooses the left branch of $[\!]$ 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 $[\!]$ as abstraction. Suppose Program (7) represents some mass-produced physical device and, by examining the production method, we have determined the tolerance as above (6) 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

$$S \sqsubseteq T \quad \widehat{=} \quad [S]R \Rrightarrow [T]R \quad \text{for all } R, \tag{9}$$

where we write $\Rrightarrow$ for 'everywhere no more than' (which ensures $\langle \mathsf{false} \rangle \Rrightarrow \langle \mathsf{true} \rangle$ as expected). From (9) we see that in the special case when R is an embedded predicate $\langle P \rangle$, 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 $[S]\langle P \rangle$ is either 0 or 1, and whenever S is certain to establish P (whenever $[S]\langle P \rangle \equiv 1$) we know that T also is certain to do so (because then $1 \Rrightarrow [T]\langle P \rangle$).

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

$$x{:=}\ -y \quad {}_{\frac{1}{2}}\!\oplus \quad x{:=}\ +y\ , \tag{10}$$

which clearly satisfies Specification (6); thus it should refine Program (7). With Definition (9), we find for any R that

$$\begin{array}{lll} & [\text{Program (10)}]R & \\ \equiv & ([x{:=}\ -y]R)/2 & + \quad ([x{:=}\ +y]R)/2 \end{array}$$

$$
\begin{array}{lll}
\equiv & R^-/2 + R^+/2 & \text{introduce abbreviations} \\
\\
\equiv & (3/5)(R^-/3 + 2R^+/3) & \text{arithmetic} \\
+ & (2/5)(3R^-/4 + R^+/4) & \\
\\
\Leftarrow & R^-/3 + 2R^+/3 & \text{any linear combination exceeds } \mathsf{min} \\
\mathsf{min} & 3R^-/4 + R^+/4 & \\
\\
\equiv & [\text{Program (7)}]R\ . &
\end{array}
$$

The refinement relation (9) 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 $[\!]$ can be implement 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

$$
\begin{array}{lll}
 & \text{Program (10)} & \\
= & x{:=}\,-y \quad \frac{1}{2}\oplus \quad x{:=}\,+y & \\
\\
= & (x{:=}\,-y \quad \frac{1}{3}\oplus \quad x{:=}\,+y) & \text{arithmetic} \\
\frac{3}{5}\oplus & (x{:=}\,-y \quad \frac{3}{4}\oplus \quad x{:=}\,+y) & \\
\\
\sqsupseteq & x{:=}\,-y \quad \frac{1}{3}\oplus \quad x{:=}\,+y & ([\!]) \sqsubseteq ({}_p\oplus) \text{ for any } p \\
[\!] & x{:=}\,-y \quad \frac{3}{4}\oplus \quad x{:=}\,+y & \\
\\
\equiv & \text{Program (7)}\ . &
\end{array}
$$

3 Presentation of probabilistic *GSL*

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

3.1 Definitions of *pGSL* substitutions

In *pGSL*, substitutions act between 'expectations' rather than predicates, where an *expectation* is an expression over (program or state)

variables that takes its value in the non-negative reals extended with ∞.[5] To retain the use of predicates, we allow expectations of the form $\langle P \rangle$ when P is Boolean-valued, defining $\langle \mathsf{false} \rangle$ to be 0 and $\langle \mathsf{true} \rangle$ to be 1.

Implication-like relations between expectations are

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

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

The definitions of the substitutions in *pGSL* are given in Fig. 1.

3.2 Interpretation of *pGSL* 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 $\langle P \rangle$ 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 $\langle P \rangle$, 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

[5] This domain, more general than the $[0, 1]$ of the previous section, makes the definitions easier... but perhaps makes intuition harder. In any case, the healthiness conditions of Sec. 3.3 show that we can restrict attention to $[0, 1]$ if we wish, as indeed we do again in Sec. 4.

The probabilistic generalised substitution language *pGSL* acts over 'expectations' rather than predicates: *expectations* take values in $\mathbb{R}_{\geq 0} \cup \{\infty\}$.

$[x{:=}\ E]R$	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.
$[P \mid S]R$	$\langle P\rangle * [S]R$, where $0 * \infty \mathrel{\widehat{=}} 0$.
$[S \,[\!]\, T]R$	$[S]R \,\mathsf{min}\, [T]R$
$[P \Longrightarrow S]R$	$1 / \langle P\rangle * [S]R$, where $\infty * 0 \mathrel{\widehat{=}} \infty$.
$[\mathsf{skip}]R$	R
$[@z \cdot S]R$	$\mathsf{min}\ z \cdot ([S]R)$, where z does not occur free in R.
$[S\ {}_p\!\oplus\ T]R$	$p * [S]R\ +\ (1{-}p) * [T]R$
$S \sqsubseteq T$	$[S]R \Rrightarrow [T]R$ for all R

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

Fig. 1. *pGSL* — the probabilistic Generalised Substitution Language

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

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

$$p \quad \equiv \quad [S]\,\langle P\rangle$$

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

1. the expected worth $\langle P\rangle$ of the final state is at least[6] 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

$$\left[\begin{array}{l} x{:=}\ H \ {}_{\frac{1}{2}}\!\oplus\, x{:=}\ T\,; \\ y{:=}\ H \ {}_{\frac{1}{2}}\!\oplus\, y{:=}\ T \end{array}\right] \langle x = y\rangle$$

$$
\begin{array}{lll}
\equiv & & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad {}_{\frac{1}{2}}\!\oplus,\ :=\ \text{and sequential composition} \\
& [x{:=}\ H \ {}_{\frac{1}{2}}\!\oplus\, x{:=}\ T](\langle x = H\rangle\,/2 + \langle x = T\rangle\,/2) & \\
\equiv & & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad {}_{\frac{1}{2}}\!\oplus \text{ and } := \\
& (1/2)(\langle H = H\rangle\,/2 + \langle H = T\rangle\,/2) & \\
& + (1/2)(\langle T = H\rangle\,/2 + \langle T = T\rangle\,/2) &
\end{array}
$$

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

$$\begin{array}{lll} \equiv & (1/2)(1/2+0/2)+(1/2)(0/2+1/2) & \text{definition } \langle\cdot\rangle \\ \equiv & 1/2\ . & \text{arithmetic} \end{array}$$

We can then use the second interpretation above to conclude that the faces are the same with probability (at least[7]) 1/2.

But part of the above calculation involves the more general expression

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

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

> the expected value of $\langle x=H\rangle/2+\langle x=T\rangle/2$ 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 *pGSL*

Recall that all *GSL* constructs satisfy the property of conjunctivity[8] [1, Sec. 6.2] — that is, for any *GSL* substitution S and post-conditions P, P' we have

$$[S](P\wedge P') \quad = \quad [S]P\wedge[S]P'\ .$$

That 'healthiness property' [2] is used to prove general properties of programs.

[7] Knowing there is no demonic choice in the program, we can in fact say it is exact.
[8] They satisfy monotonicity too, which is implied by conjunctivity.

In *pGSL* the healthiness condition becomes 'sublinearity' [15], a generalisation of conjunctivity:

> Let a, b, c be non-negative finite reals, and R, R' expectations; then all *pGSL* constructs S satisfy
>
> $$[S](aR + bR' \ominus c) \quad \Leftarrow \quad a[S]R + b[S]R' \ominus c \ ,$$
>
> which property of S is called *sublinearity*.
> We have written aR for $a * R$ *etc.*, and truncated subtraction $\ominus$ is defined
>
> $$x \ominus y \quad \widehat{=} \quad (x - y) \,\mathsf{max}\, 0 \ ,$$
>
> 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.[9]

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

$$\begin{array}{lll} & [S]R' & \\ \equiv & [S](R + (R' - R)) & \\ \Leftarrow & [S]R + [S](R' - R) & \text{sublinearity with } a, b, c \mathrel{\widehat{=}} 1, 1, 0 \\ \Leftarrow & [S]R \ . & R' - R \text{ well defined, hence } 0 \Rrightarrow [S](R' - R) \end{array}$$

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

$$\begin{array}{lll} & [S]0 & \\ \equiv & [S](2 * 0) & \\ \Leftarrow & 2 * [S]0 \ , & \text{sublinearity with } a, b, c \mathrel{\widehat{=}} 2, 0, 0 \end{array}$$

so that $[S]0$ must be either 0 or ∞. We say that S is *feasible* if $[S]0 \equiv 0$.

Now write $\mathsf{max}R$ for the maximum of R over all its variables' values, and assume that S is feasible; then

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

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

where we assume for the moment that $\mathsf{max}R$ is finite so that it can be used in sublinearity. Now from $0 \Leftarrow [S]R \ominus \mathsf{max}R$ we have trivially that

$$[S]R \quad \Rrightarrow \quad \mathsf{max}\, R\ , \tag{11}$$

which we have proved under the assumption that S is feasible and $\mathsf{max}R$ is finite — but if $\mathsf{max}R$ is not finite, then (11) holds anyway.

Thus we have shown in any case that feasibility implies (11); but since (11) implies feasibility (take $R \mathrel{\widehat{=}} 0$), it could itself be taken as the definition.[10]

scaling: multiplication by a non-negative constant distributes through feasible[11] substitution. Note first that $[S](aR) \Leftarrow a[S]R$ directly from sublinearity. For $\Rrightarrow$ we have two cases: when a is 0, trivially from feasibility

$$[S](0 * R) \ \equiv\ [S]0 \ \equiv\ 0 \ \equiv\ 0 * [S]R\ ;$$

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

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

thus establishing $[S](aR) \equiv a[S]R$ 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

$$0 \,\&\, 0 = 0, \quad 0 \,\&\, 1 = 0, \quad 1 \,\&\, 0 = 0 \quad \text{and} \quad 1 \,\&\, 1 = 1 \tag{12}$$

[10] We can define $\mathsf{fis}\,(S)$ to be $\langle [S]0 = 0 \rangle$.

[11] The feasibility restriction is because when $[S](0 * R)$ is infinite, it does not equal $0 * [S]R$.

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

$$R \,\&\, R' \quad \widehat{=} \quad R + R' \ominus 1 \ , \tag{13}$$

whose right-hand side is inspired by sublinearity when $a, b, c \,\widehat{=}\, 1, 1, 1$. We now establish a (sub-) distribution property for it.

While discussing conjunction we restrict our expectations to the unit interval $[0, 1]$, as we did earlier, and assume all our substitutions are feasible.[12] In that case infinities do not intrude, and we have from feasibility that $R \Rrightarrow 1$ implies

$$[S]R \quad \Rrightarrow \quad \mathsf{max}\, R \quad \Rrightarrow \quad 1 \ ,$$

showing that the restricted domain $[0, 1]$ is closed under (feasible) substitutions.

The distribution property is then a direct consequence of sublinearity.

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

$$[S](R \,\&\, R') \quad \Lleftarrow \quad [S]R \ \ \& \ \ [S]R'$$

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 (12).

The second reason is that sub-conjunctivity implies 'full' conjunctivity for standard programs. Standard programs, containing no probabilistic choices, take standard $\langle P \rangle$-style post-expectations to

[12] We could avoid the feasibility assumption by using the domain $[0, 1] \cup \infty$, but in this presentation it is simpler not to introduce ∞.

standard pre-expectations: they are the embedding of *GSL* in *pGSL*, and for standard S we now show that

$$[S](\langle P\rangle \,\&\, \langle P'\rangle) \quad \equiv \quad [S]\,\langle P\rangle \;\; \& \;\; [S]\,\langle P'\rangle \;\; . \tag{14}$$

First note that '$\Leftarrow$' comes directly from sub-conjunctivity above, taking R, R' to be $\langle P\rangle\,, \langle P'\rangle$.

Then '$\Rrightarrow$' comes from monotonicity, for $\langle P\rangle \& \langle P'\rangle \Rrightarrow \langle P\rangle$ whence $[S](\langle P\rangle \,\&\, \langle P'\rangle) \Rrightarrow [S]\,\langle P\rangle$, and similarly for P'. Putting those together gives

$$[S](\langle P\rangle \,\&\, \langle P'\rangle) \quad \Rrightarrow \quad [S]\,\langle P\rangle \;\; \mathsf{min} \;\; [S]\,\langle P'\rangle \;\; ,$$

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

> 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
> $A \cap B$ in general is that it has probability at least $p \,\&\, q$
> [19].

The $\&$ operator also plays a crucial role in the proof (not given in this paper) of the probabilistic loop rule presented and used in the next section [13].

4 Probabilistic invariants for loops

To show *pGSL* 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.

We continue with the restriction to feasible programs, and expectations in $[0, 1]$.

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 the loop will establish the postcondition, if termination occurs.

For a probabilistic loop we have a post-expectation rather than a postcondition; but if that post-expectation is some $\langle P \rangle$ 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 'from here'. Often that invariant will have the form

$$p \quad * \quad \langle I \rangle \tag{15}$$

with p a probability and I a predicate, both expressions over the state. From the definition of $\langle \cdot \rangle$ we know that the interpretation of (15) 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 $\langle \mathsf{true} \rangle \equiv 1$ we define

$$\mathsf{trm}\,(S) \quad \widehat{=} \quad [S]1 \ . \tag{16}$$

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

$$S \quad \widehat{=} \quad S \,{}_p\!\oplus\, \mathsf{skip} \ , \tag{17}$$

in which we assume that p is some constant strictly less than 1: elementary probability theory shows that S terminates with probability 1 (after an expected $p/(1-p)$ recursive calls). And by calculation based on (16) we confirm that

$$\begin{array}{ll} & \mathsf{trm}\,(S) \\ \equiv & [S]1 \\ \equiv & p * ([S]1) + (1-p) * ([\mathsf{skip}]1) \\ \equiv & p * \mathsf{trm}\,(S) + (1-p) \ , \end{array}$$

so that $(1-p) * \mathsf{trm}\,(S) \equiv 1-p$. Since p is not 1, we can divide by $1-p$ to see that $\mathsf{trm}\,(S) \equiv 1$. That agrees with the elementary theory above, that 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.

4.3 Probabilistic correctness of loops

A loop is a least fixed point, as in the standard case [1, Sec. 9.2], which gives easily that if $\langle P\rangle * I \Rrightarrow [S]I$ then

$$I \Rrightarrow [\textsc{while}\ P\ \textsc{do}\ S\ \textsc{end}](\langle\neg P\rangle * I)$$

provided[13] the loop terminates; indeed, for the proof one simply carries out the standard reasoning almost without noticing that expectations rather than predicates are being manipulated. Thus the notion of invariant carries over smoothly from the standard to the probabilistic case.

When termination is taken into account we get the following rule [13].

> For convenience write T for the termination probability of the loop, so that
>
> $$T \quad \widehat{=} \quad \mathsf{trm}\,(\textsc{while}\ P\ \textsc{do}\ S\ \textsc{end})\ .$$
>
> Then partial loop correctness — preservation of a loop invariant I — implies total loop correctness if that invariant I nowhere[14] exceeds T:
>
> If $\quad \langle P\rangle * I \Rrightarrow [S]I$
> and $\quad I \Rrightarrow T$
> then $\quad I \Rrightarrow [\textsc{while}\ P\ \textsc{do}\ S\ \textsc{end}](\langle\neg P\rangle * I)\ .$

We illustrate the loop rule with a simple example. Suppose we have a machine that is supposed to sum the elements of a sequence

[13] The precise treatment of 'provided' uses weakest *liberal* pre-expectations [13, 10].

[14] Note that is not the same as 'implies total correctness in those states where I does not exceed T': in fact I must not exceed T in *any* state, and the weaker alternative is not sound.

[1, Sec. 10.4.2], except that the mechanism for moving along the sequence occasionally sticks, and for that moment does not move. A program for the machine is given in Fig. 2, where the unreliable component

$$k{:=}\ k+1 \quad {}_c\!\oplus \quad \mathsf{skip}$$

sticks (fails to move along) with probability $1{-}c$. With what prob-

```
PRE
  s ∈ seq(u)
THEN
  VAR k IN
    r, k := 0, 1;
    WHILE k ≤ size(s) DO
      r := r + s(k);
      k := k + 1  c⊕  skip        ← failure possible here
    END
  END
END
```

Fig. 2. An unreliable sequence-accumulator

ability does the machine accurately sum the sequence, establishing

$$r \quad = \quad \mathsf{sum}(s) \tag{18}$$

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 (18)?

The answer is in the form (15) — take p to be $c^{\mathsf{size}(s)+1-k}$, and let I be the standard invariant [1, p.459]

$$k \in 1..\mathsf{size}(s){+}1 \quad \wedge \quad r = \mathsf{sum}(s \uparrow (k{-}1)) \ .$$

Then our probabilistic invariant — call it J — is just $p * \langle I \rangle$, which is to say it is

if the standard invariant holds then $c^{\mathsf{size}(s)+1-k}$, 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

$$\left[\begin{array}{l} r{:=}\ r+s(k); \\ k{:=}\ k+1 \end{array}\ {}_c\!\oplus\ \mathsf{skip}\right] \quad J$$

$$\equiv \quad [r{:=}\ r+s(k)] \quad J \ +\ (1{-}c)*J \qquad \text{; and } {}_c\!\oplus$$

$$\Leftarrow \qquad \text{drop } (1{-}c)*J\text{, and } :=$$

$$[r{:=}\ r+s(k)]\ \ c^{\mathsf{size}(s)+1-k} \ * \ \left\langle \begin{array}{c} k\in 0..\mathsf{size}(s) \\ r=\mathsf{sum}(s\uparrow k) \end{array} \right\rangle$$

$$\equiv \quad c^{\mathsf{size}(s)+1-k} \ * \ \left\langle \begin{array}{c} k\in 0..\mathsf{size}(s) \\ r+s(k)=\mathsf{sum}(s\uparrow k) \end{array} \right\rangle \qquad :=$$

$$\Leftarrow \quad \langle k\le \mathsf{size}(s)\rangle * J\ ,$$

where in the last step the guard $k \le \mathsf{size}(s)$, and $k \ge 1$ from the invariant, allow the removal of $+s(k)$ from both sides of the lower equality.

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

$$c^{\mathsf{size}(s)+1-k} \ * \ \langle k\in 1..\mathsf{size}(s){+}1\rangle\ ,$$

which is just the probability of $\mathsf{size}(s)+1-k$ correct executions of $k{:=}\ k+1$, given that k is in the proper range to start with; hence trivially $J \Rrightarrow \mathsf{trm}\,(\mathit{while})$, 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 $\langle k > \mathsf{size}(s)\rangle * J$, which indeed 'implies' (is in the relation $\Rrightarrow$ to) the post-expectation $\langle r = \mathsf{sum}(s)\rangle$ as required.

Turning finally to the initialisation we finish off with

$$
\begin{array}{lll}
 & [r, k{:=}\ 0, 1]J & \\
\equiv & c^{\mathsf{size}(s)} \quad * & \left\langle \begin{array}{l} 1 \in 1..\mathsf{size}(s){+}1 \\ 0 = \mathsf{sum}(s \uparrow 0) \end{array} \right\rangle \\
\equiv & c^{\mathsf{size}(s)} \quad * & \langle \mathsf{true} \rangle \\
\equiv & c^{\mathsf{size}(s)} \ , &
\end{array}
$$

and our overall conclusion is therefore

$$c^{\mathsf{size}(s)} \quad \Rrightarrow \quad [\mathit{sequence\text{-}accumulator}]\, \langle r = \mathsf{sum}(s) \rangle \ ,$$

just as we had hoped: the probability that the sequence is correctly summed is at least $c^{\mathsf{size}(s)}$.

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 $c^{\mathsf{size}(s)}$ in general. For it is certainly possible that r is correctly calculated in spite of the occasional malfunction of $k{:=}\ k{+}1$; but the exact probability, should we try to calculate it, would depend intricately on the contents of s. (It would be 1, for example, if s contained only zeroes, and could be very involved if s 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, and a discussion of probabilistic *variant* arguments, are given elsewhere [13].

5 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 [8] is all that is needed for a simple probabilistic semantics, albeit one lacking abstraction. Then He's *sets* of distributions [5] and our min for demonic choice together with the fundamental property of sublinearity [15] take us the rest of the way, allowing allowing abstraction and refinement to resume their central role — this time in a probabilistic context. And as Sec. 4 illustrates, 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.

Finally, there is the larger issue of probabilistic abstract machines, or 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 [11], and that equivalence or refinement between such modules depends subtly on the power of demonic choice and its interaction with probability.

Acknowledgements

This paper reports work carried out with Annabelle McIver, 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. Jifeng He, K. Seidel, and A. K. McIver. Probabilistic models for the guarded command language. *Science of Computer Programming*, 28:171–192, 1997.
6. C. Jones. Probabilistic nondeterminism. Monograph ECS-LFCS-90-105, Edinburgh Univ. Edinburgh, U.K., 1990. (PhD Thesis).
7. C. Jones and G. Plotkin. A probabilistic powerdomain of evaluations. In *Proceedings of the IEEE 4th Annual Symposium on Logic in Computer Science*, pages 186–195, Los Alamitos, Calif., 1989. Computer Society Press.
8. D. Kozen. A probabilistic PDL. In *Proceedings of the 15th ACM Symposium on Theory of Computing*, New York, 1983. ACM.

9. Annabelle McIver and Carroll Morgan. Probabilistic predicate transformers: part 2. Technical Report PRG-TR-5-96, Programming Research Group, March 1996. Revised version to be submitted for publication under the title *Demonic, angelic and unbounded probabilistic choices in sequential programs*.
10. Annabelle McIver and Carroll Morgan. Partial correctness for probabilistic demonic programs. Technical Report PRG-TR-35-97, Programming Research Group, 1997.
11. Annabelle McIver, Carroll Morgan, and Elena Troubitsyna. The probabilistic steam boiler: a case study in probabilistic data refinement. In preparation, 1997.
12. C. C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3), July 1988. Reprinted in [14].
13. 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.
14. C. C. Morgan and T. N. Vickers, editors. *On the Refinement Calculus*. FACIT Series in Computer Science. Springer-Verlag, Berlin, 1994.
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. J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.
17. Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
18. G. Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, October 1989.
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. Revised version to be submitted for publication under the title *Probabilistic Imperative Programming: a Rigorous Approach*.
20. M. Sharir, A. Pnueli, and S. Hart. Verification of probabilistic programs. *SIAM Journal on Computing*, 13(2):292–314, May 1984.