WOFACS 98
IFIP WG2.3 and UNU/IIST
WINTER SCHOOL on PROGRAMMING METHODOLOGY
University of Cape Town, 6-17 July 1998

Chris Brink

Laboratory for Formal Aspects and Complexity in Computer Science, Department of Mathematics and Applied Mathematics, University of Cape Town

Since 1992 there has been a biennial Winter School on Formal and Applied Computer Science (WOFACS) at the University of Cape Town. Each of these have resulted in a Proceedings volume: SACJ 9, 13, 19, and the volume you hold in your hand right now. All WOFACS events have had more or less the same structure. A group of eminent academics come to Cape Town during the winter vacation, and each of them offers, over a 2-week period, a course of 10 lectures on a particular topic. These short courses are pitched at about Honours level, and have some evaluation mechanism built in: short tests, or exercises, or assignments. At a student’s request, and by arrangement with the Head of Department at his/her home institution, these courses can then be offered as part of the student’s Honours degree. In this way WOFACS makes a contribution to beginning postgraduate studies on a wide geographical front. Typically such an event would attract students and young staff members not only from across South Africa, but also from many sub-Saharan African countries. Each WOFACS was organised by the UCT Laboratory for Formal Aspects and Complexity in Computer Science (FACCSLab).

WOFACS 98 had a distinctly international flavour. The entire event was, in fact, three things at once (which explains the somewhat complicated title at the top of this page). Besides being, by our reckoning, the 4th WOFACS, it was also the third in a series of outreach offerings of IFIP Working Group 2.3 on Programming Methodology. All the speakers were from WG2.3, and the entire event was built around the topic of programming methodology. Thirdly, the event was also an offering of the International Institute for Software Technology, situated at the United Nations University in Macao. The three role players, FACCSLab, WG2.3 and UNU/IIST, shared an agenda of trying to service specifically possible participants from disadvantaged communities and other African countries, and the UNU/IIST made available some generous grants for this purpose. In keeping with the tradition of these events there were no course fees: except for a fairly modest registration charge WOFACS has always been a free service to the community.

The speakers at WOFACS 98 and their topics were:

- Prof Dines Bjørner, Technical University Denmark: Domains and Requirements, Software Architectures and Program Organisation.
- Prof David Gries, Cornell University: Logic as a Tool.
- Prof Michael Jackson: Problem Frames and Principles of Description.
- Prof Jayadev Misra, University of Texas: Toward an Applied Theory ofConcurrency.
- Dr Carroll Morgan, Oxford: Predicate Transformers and Probabilistic Programs.

Professor Gries’ course was regarded as foundational, and recommended to all participants. It was offered during the first week only, at double tempo, thus giving the other four courses the opportunity to make use of concepts and techniques introduced there. Each of the speakers made available a Course Reader of their material. These were printed and bound before the event, and were handed out to participants upon registration. WOFACS 98 was attended by more than 60 participants, inter alia from Angola, Malawi, the Congo, Gabon, Cameroon, Malawi and Uganda.

From South African Universities we had representation, besides UCT, also from the University of Stellenbosch, the Transkei, the Qwa-qwa branch of the University of the North, the Witwatersrand, Pretoria, RAU, the North-West, Vista, UNISA, the Mangosothu Technikon and the Eastern Cape Technikon. Cape Town weather can be pretty stormy in July, but there were sufficiently many beautifully clear winter days to allow participants the opportunity to do some
sightseeing and exploring, after lectures or over the weekend.

We are grateful to our financial sponsors, and pleased to acknowledge their contributions.

- UNU/IIST.
- The Foundation for Research Development.
- The Chairman's Fund of Anglo American.

We thank the University of Cape Town for the use of its premises and facilities. We are particularly grateful to the 5 speakers, who put a lot of thought and preparation into their lectures, and tackled with great success the difficult task of conveying state-of-the-art material to a heterogeneous audience. It is only fair that specific thanks should be given to Carroll Morgan, whose idea it was in the first place to have a combined event, and to Dines Bjorner, who kickstarted the fundraising campaign. Finally, I would like to add my personal thanks to my colleagues and staff who worked so hard behind the scenes to make a success of WOFACS 98.
**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

```
if test then this else that fi
```

(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.

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
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 Secs. 2 and 4, with finally a glance over Secs. 3.1 and 3.2; more thoroughly one would read Secs. 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 [·] be used to embed Boolean-valued predicates within arithmetic formulae which, for reasons explained below, we call expectations; we allow them to range over the unit interval [0, 1]. Stipulating that [false] is 0 and [true] is 1 makes [P] in a trivial sense the probability that a given predicate P holds: if false, P holds with probability 0; if true, it holds with probability 1.

For (our first) example, consider the simple program

\[ x := -y \uplus \frac{1}{3} \quad x := +y \]  

over variables x, y: Z, using a construct \( \frac{1}{3} \uplus \) 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

\[ \text{wp.}(x := E).R := \begin{cases} \text{‘R with } x \text{ replaced everywhere by } E^2 \end{cases} \]  

\[ \text{wp.}(S \uplus T).R := p \ast \text{wp.} S.R + (1-p) \ast \text{wp.} T.R, \]  

in which \( R \) is an expectation, and for our example program we ask

\[ \text{what is the probability that the predicate} \]  

\[ \text{‘the final state will satisfy } x \geq 0 \text{’ holds in} \]  

\[ \text{some given initial state of the program (2)?} \]  

\(^1\)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

\[ \text{wp.}(x := -y \uplus \frac{1}{3} \ast x := +y).[x \geq 0] \]

\[ \equiv^3 (1/3) \ast \text{wp.}(x := -y).[x \geq 0] \]  

\[ + (2/3) \ast \text{wp.}(x := +y).[x \geq 0] \]  

\[ \equiv (1/3)[-y \geq 0] + (2/3)[+y \geq 0] \]

using (4)

\[ \equiv [y < 0]/3 + [y = 0] + 2[y > 0]/3 \]

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

\[
\begin{array}{l}
\text{when } y < 0, \quad 1/3 + 0 + 2(0)/3 = 1/3 \\
\text{when } y = 0, \quad 0/3 + 1 + 2(0)/3 = 1 \\
\text{when } y > 0, \quad 0/3 + 0 + 2(1)/3 = 2/3
\end{array}
\]

Those results indeed correspond with our operational intuition about the effect of \( \frac{1}{3} \uplus \).

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 \( \Box \). 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 — an 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

\[ \bullet x := -y \] is to be executed with probability at least 1/3,

\[ \bullet x := +y \] is to be executed with probability at least 1/4 and

\[ \bullet \] it is certain that one or the other will be executed.

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

\[ x := -y \uplus \frac{1}{3} \ast x := +y \]

\[ \Box \] \[ x := -y \uplus \frac{1}{3} \ast x := +y \],

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

\(^3\)Later we explain the use of ‘\( \equiv \)’ rather than ‘\( = \)’.

SART/SACJ, No 22, 1999

15
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
\[
wp.(S \cap T).R := wp.S.R \min wp.T.R, \quad (7)
\]
using \( \min \) because we regard demonic behaviour as attempting to make the achieving of \( R \) as improbable as it can. Repeating our earlier calculation (but more briefly) gives this time
\[
wp.\( (\text{Program (6)} \).\).x \geq 0
\equiv \quad \text{using (3), (4), (7)}
\equiv \quad [y \leq 0]/3 + [y \geq 0]/3
\min 3[y \leq 0]/4 + [y \geq 0]/4
\equiv \quad \text{using arithmetic}
[y < 0]/3 + [y = 0] + [y > 0]/4.
\]

Our interpretation is now
- When \( y \) is initially negative, the demon chooses the left branch of \( \cap \) 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 \( \cap \) 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
\[
S \subseteq T \quad := \quad wp.S.R \Rightarrow wp.T.R \quad \text{for all } R, \quad (8)
\]
where we write \( \Rightarrow \) for 'everywhere no more than' (which ensures \( \text{false} \Rightarrow \text{true} \) as the notation suggests). From (8) we see that in the special case when \( R \) is an embedded predicate \([P]\), the meaning of \( \Rightarrow \) 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 \( 1 \Rightarrow wp.T.[P] \)).

For our third example we prove a refinement: consider the program
\[
x := -y \quad \frac{1}{3} \oplus \quad x := +y, \quad (9)
\]
which clearly satisfies Specification (5); thus it should refine Program (6). With Definition (8), we find for any \( R \) that
\[
wp.(\text{Program (9)}).R
\equiv \quad wp.(x := -y).R/2 + wp.(x := +y).R/2
\equiv \quad R^-/2 + R^+/2
\equiv \quad \text{using abbreviations}
\equiv \quad (3/5)(R^-/3 + 2R^+/3)
\quad + \quad (2/5)(3R^-/4 + R^+/4)
\equiv \quad \text{arithmetic}
\equiv \quad \text{any linear combination exceeds min}
3R^-/4 + R^+/4
\equiv \quad wp.(\text{Program (6)}).R.
\]
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 \( \cap \) can be implemented by any probabilistic choice whatever: in this case we used \( q \oplus \). Thus a proof of refinement at the program level might read
\[
\text{Program (9)}
= \quad x := -y \quad \frac{1}{3} \oplus \quad x := +y
\equiv \quad (x := -y \quad \frac{1}{3} \oplus \quad x := +y)
\quad \text{arithmetic}
\equiv \quad (\cap \subseteq (p \oplus)) \text{ for any } p
\equiv \quad x := -y \quad \frac{1}{3} \oplus \quad x := +y
\equiv \quad \text{Program (6)}.
\]

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

Implication-like relations between expectations are

\[ R \Rightarrow R' := R \text{ is everywhere no more than } R' \]
\[ R \equiv R' := R \text{ is everywhere equal to } R' \]
\[ R \equiv R' := R \text{ is everywhere no less than } R' \]

Note that \( \models P \Rightarrow P' \) exactly when \( [P] \Rightarrow [P'] \), 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

\[ \equiv \)
\[ \times (\text{probability } S \text{ reaches } P) \]
\[ \times (\text{worth of states satisfying } P) \]
\[ + \]
\[ \times (\text{probability } S \text{ does not reach } P) \]
\[ \times (\text{worth of states not satisfying } P) \]
\[ \equiv \]
\[ + \]
\[ \text{probability } S \text{ reaches } P \]
\[ \text{probability } S \text{ does not reach } P \]

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

\[ \wp.p.(x: H \frac{1}{2} \oplus x: T ; y: H \frac{1}{2} \oplus y: T ; x = y) \]

This is obtained by summing over all final states

\[ \wp.(x: H \frac{1}{2} \oplus x: T , y: H \frac{1}{2} \oplus y: T , x = y) \]

\[ = \]
\[ \frac{1}{2} \oplus \text{ and sequential composition} \]
\[ \wp.p.((x: H \frac{1}{2} \oplus x: T , y: H \frac{1}{2} \oplus y: T , x = y) \]

\[ = \]
\[ \frac{1}{2} \oplus \frac{1}{2} = \text{ arithmetic} \]
\[ \frac{1}{2} \]

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

\[ \wp.p.(x: H \frac{1}{2} \oplus x: T , y: H \frac{1}{2} \oplus y: T , x = y) \]

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

\[ \wp.p.(S \land P') = \wp.p.(S \land P). \wp.p.(P') \]

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 \( p \Rightarrow \wp.p.[P] \) 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 \textit{pGCL} acts over 'expectations' rather than predicates: \textit{expectations} take values in \([0, 1]\).

\[
\begin{align*}
wp.(x = E).R & \quad \text{The expectation obtained after replacing all free occurrences of } x \text{ in } R \text{ by } E, \text{ renaming bound variables in } R \text{ if necessary to avoid capture of free variables in } E. \\
wp.\text{skip}.R & \quad R \\
wp.(S; T).R & \quad \wp.S.(\wp.T.R) \\
wp.(S \cap T).R & \quad \wp.S.R \min \wp.T.R \\
wp.(S \oplus T).R & \quad p \times \wp.S.R + (1-p) \times \wp.T.R \\
S \subseteq T & \quad \wp.S.R \not\Rightarrow \wp.T.R \quad \text{for all } R
\end{align*}
\]

- \(R\) 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** \texttt{if B then S else T fi} is a special case of probabilistic choice: it is just \(S \{p\} \oplus T\). 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:** \textit{pGCL} — the probabilistic Guarded Command Language
In pGCL the healthiness condition becomes 'sublinearity' [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

\[
wp.S.(aR + bR' + c) \Leftrightarrow a(wp.S.R) + b(wp.S.R') + c, 
\]

which property of \( S \) is called sublinearity. We have written \( aR \) for \( a \ast R \) etc., and truncated subtraction \( \ominus \) is defined

\[
x \ominus y := (x - y) \max 0 ,
\]

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 \( R \Rightarrow R' \) for two expectations \( R, R' \); then

\[
wp.S.R' \equiv wp.S.(R + (R' - R)) \]

sublinearity with \( a, b, c := 1, 1, 0 \)

\[
R' - R \text{ well defined, hence } 0 \Rightarrow wp.S.(R' - R) wp.S.R .
\]

**Feasibility:** pre-expectations cannot be 'too large'.

First note that

\[
wp.S.0 \equiv wp.S.((2 \ast 0) \]

sublinearity with \( a, b, c := 2, 0, 0 \)

so that \( wp.S.0 \) must be 0.

Now write \( \max R \) for the maximum of \( R \) over all its variables' values; then

\[
0 \equiv wp.S.0 \text{ feasibility above} \]

\[
\equiv wp.S.(R \ominus \max R) \text{ feasibility above} \]

\[
\equiv wp.S.R \ominus \max R . \text{ sublinearity} \]

But from \( 0 \equiv wp.S.R \ominus (\max R) \) we have trivially that

\[
wp.S.R \Rightarrow \max R , \quad (10)
\]

which we identify as the feasibility condition\(^9\) for pGCL.

\(^8\)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 \( wp.S.0 \equiv 0 \).

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

\[
wp.S.(0 \ast R) \equiv wp.S.0 \equiv 0 \equiv 0 \ast wp.S.R ;
\]

and for the other case \( a \neq 0 \) we reason

\[
wp.S.(aR) \equiv a(1/a)wp.S.(aR) \quad a \neq 0 \\
\Rightarrow a(wp.S.((1/a)aR)) \text{ sublinearity using } 1/a \\
\equiv a(wp.S.R) ,
\]

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 \( \land \) is not defined over numbers, we have many choices for a probabilistic analogue \& of it, requiring only that

\[
\begin{align*}
0 \land 0 &= 0 \\
0 \land 1 &= 0 \\
1 \land 0 &= 0 \\
1 \land 1 &= 1
\end{align*}
\]

for consistency with embedded Booleans.

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

\[
R \land R' := R + R' \ominus 1 , \quad (12)
\]

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

\[
wp.S.(R \land R') \equiv wp.S.R \& wp.S.R' 
\]

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

\[ \text{wp}. S.([P] \land [P']) \equiv \text{wp}. S. [P] \land \text{wp}. S. [P'] . \quad (13) \]

First note that `\( \equiv \)` comes directly from sub-conjunctivity above, taking \( R, R' \) to be \([P], [P']\).

For `\( \Rightarrow \)` we appeal to monotonicity, because \([P] \land [P'] \Rightarrow [P] \) whence \( \text{wp}. S.([P] \land [P']) \Rightarrow \text{wp}. S. [P] \), and similarly for \( P' \). Putting those together gives

\[ \text{wp}. S.([P] \land [P']) \Rightarrow \text{wp}. S. [P] \land \text{wp}. S. [P'] , \]

by elementary arithmetic properties of \( \Rightarrow \). But on standard expectations — which \( \text{wp}. S. [P] \) and \( \text{wp}. S. [P'] \) are, because \( S \) is standard — the operators \( \land \) and \( \Rightarrow \) agree.

A last attribute linking \( \land \) and \( \Rightarrow \) 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 \( p \land q \) [19].

The \( \land \) 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 cannot use Boolean conjunction to combine ‘correct if terminates’ and ‘it does terminate’.

4.1 Probabilistic invariants

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

For a probabilistic loop we have a post-termination rather than a postcondition; but otherwise the situation is much the same: and if that post-termination 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

\[ p \ast [I] . \quad (14) \]

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 [true] \( \equiv 1 \) we see that a program’s probability of termination is given by

\[ \text{wp}. S.1 . \quad (15) \]

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

\[ S \; : \; = \; S \oplus \text{skip} , \quad (16) \]

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

\[ \text{wp}. S.1 \]

\[ \equiv p \ast (\text{wp}. S.1) + (1-p) \ast (\text{wp}. \text{skip}.1) \]

\[ \equiv p \ast (\text{wp}. S.1) + (1-p) , \]

so that \((1-p) \ast (\text{wp}. S.1) \equiv 1-p \). Since \( p \) is not 1, we can divide by \( 1-p \) to see that indeed \( \text{wp}. S.1 \equiv 1 \): 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.

10Elementary 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 \([P] \cdot I \Rightarrow [S]I\) then

\[
I \Rightarrow wp.(do \; P \rightarrow S \; od).([\neg P] \cdot I)
\]

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

\[
T := wp.(do \; P \rightarrow S \; od).1.
\]

Then partial loop correctness — preservation of a loop invariant \(I\) — implies total loop correctness if that invariant \(I\) nowhere\(^{13}\) exceeds \(T\): that is,

\[
\text{if } \ [P] \cdot I \Rightarrow wp.SI \\
\text{and } \ I \Rightarrow T
\]

then

\[
I \Rightarrow wp.(do \; P \rightarrow S \; od).([\neg P] \cdot I).
\]

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

\[
k := k + 1 \quad c \oplus k := k - 1
\]

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

\[
r = \sum ss
\]

on termination? \(^{(17)}\)

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

\[
0 \leq k \leq N \quad \land \quad r = \sum ss[0..k].
\]

Then our probabilistic invariant — call it \(J\) — is just \(p \cdot [I]\), 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

\[
wp. \left( \begin{array}{c}
r := r + ss.k; \\
k := k + 1 \quad c \oplus k := k - 1
\end{array} \right) \cdot J
\]

\[
\equiv wp.(r := ss.k).(c \cdot wp.(k := k + 1).J + (1-c) \cdot wp.(k := k - 1).J)
\]

\[
\equiv wp.(r := ss.k). \quad \text{drop second term, and } :=
\]

\[
c^{N-k} \cdot \left[ 0 \leq k + 1 \leq N \quad r = \sum ss[0..k] \right]
\]

\[
\equiv [k < N] \cdot J,
\]

where in the last step the guard \(k < N\), and \(k > 0\) 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

\[
c^{N-k} \cdot [0 \leq k \leq N],
\]

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 \(J \Rightarrow T\) 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 \([k \geq N] \cdot J\), which indeed ‘implies’ (is in the relation \(\Rightarrow\)) to the post-expectation \([r = \sum ss]\) as required.

Turning finally to the initialisation we finish off with

\[
wp.(r,k := 0,0) \cdot J
\]

\[
\equiv c^N \cdot \left[ 0 \leq 0 \leq N \quad 0 = \sum ss[0..0] \right]
\]

\[
\equiv c^N \cdot \left[ \text{true} \right]
\]

\[
\equiv c^N,
\]

and our overall conclusion is therefore

\[
c^N \Rightarrow wp.(\text{sequence-summer}), \quad [r = \sum ss]
\]

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

Note the importance of the inequality \(\Rightarrow\) 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 \(k := k + 1;\)
con \( ss(0..N): Z; \)
var \( r: Z; \)
\[
| \| \var k: Z;
| \ r, k: 0, 0;
| \ do k < N \rightarrow
| \ r: = r + ss.k;
| \ k: = k + 1 \land k: = k - 1 \quad \leftarrow \text{failure possible here}
| \ od
| \|
\]

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 Secs. 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 \( k < K - 1 \) — The tourist writes \( K \) on his notepad (erasing \( k \)), and goes to the other place.
- if \( k > K - 1 \) — The tourist writes “here” on the noticeboard (erasing \( K \)), and goes inside.
- if \( k = K - 1 \) — 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 $l$ (r) is the bag of numbers held by tourists waiting to look at the left (right) noticeboard; bag $l$ (r) 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 $l$ (r) is not yet empty), select one: the number he holds is $l$ ($r$). If some tourist has (already) decided on this alternative (so that $l$ (r) 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 $l$ (r)).

If this tourist’s number equals the noticeboard value, he increases the noticeboard value, copies that value and goes to the other alternative ($l$ (r)).

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.$^{18}$
- $\square$ — The empty bag.
- $[n]^N$ — A bag containing $N$ copies of value $n$.

$^{17}$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 $[1]$.

#### 5.3.3 Correctness criteria

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

$$\#l = M+N \land r = \square \lor l = \square \land \#r = M+N$$

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

### 5.4 Partial correctness

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

#### 5.4.1 Three invariants

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

$$\#l + \#l + \#r + \#r = M+N$$

It holds initially, and is trivially maintained. The second invariant is

$$l, l \leq R$$

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;

$^{19}$By $b \leq K$ we mean that no element in the bag $b$ exceeds the integer $K$. 

SART/SACJ, No 22, 1999
\[ l_{out}, r_{out} = [0]^M, [0]^N; \]
\[ l_{in}, r_{in} = \emptyset, \emptyset; \]
\[ L, R = 0, 0; \]

\[
\text{do } l_{out} \neq \emptyset \rightarrow \\
\text{take } l \text{ from } l_{out}; \\
\text{if } l_{in} \neq \emptyset \text{ then add } l \text{ to } l_{in} \text{ else} \\
\quad \text{\( l > L \rightarrow \text{add } l \text{ to } l_{in} \) if } \text{\( l = L \rightarrow L = L + 2 \frac{1}{2}(L+2); \) add } L \text{ to } r_{out} \text{;} \\
\quad \text{\( l < L \rightarrow \text{add } L \text{ to } r_{out} \) fi} \\
\text{fi} \\
\text{rout} \neq \emptyset \rightarrow \\
\text{take } r \text{ from } r_{out}; \\
\text{if } r_{in} \neq \emptyset \text{ then add } r \text{ to } r_{in} \text{ else} \\
\quad \text{\( r > R \rightarrow \text{add } r \text{ to } r_{in} \) if } \text{\( r = R \rightarrow R = R + 2 \frac{1}{2}(R+2); \) add } R \text{ to } l_{out} \text{;} \\
\quad \text{\( r < R \rightarrow \text{add } R \text{ to } l_{out} \) fi} \\
\text{od}
\]

Figure 3: Rabin's choice-coordination algorithm

- Adding elements to \( l_{in}, r_{in} \) cannot falsify it, since those elements come from \( l_{out}, r_{out} \);
- The only commands adding elements to \( l_{out}, r_{out} \) are
  \[
  \text{add } L \text{ to } r_{out} \quad \text{and} \quad \text{add } R \text{ to } l_{out},
  \]
  and they maintain it trivially.

Our final invariant for partial correctness is

\[
\max l_{in} > L \quad \text{if} \quad l_{in} \neq \emptyset \\
\max r_{in} > R \quad \text{if} \quad r_{in} \neq \emptyset,
\]
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 \( l_{in} \) case. The invariant holds on initialisation (when \( l_{in} = \emptyset \)); and inspection of the program shows that it is trivially established when the first value is added to \( l_{in} \) since the command concerned,

\[
l > L \rightarrow \text{add } l \text{ to } l_{in},
\]
is executed when \( l_{in} = \emptyset \) to establish \( l_{in} = [1] \) for some \( l > L \).

Since elements never leave \( l_{in} \), it remains non-empty and \( \max l_{in} \) can only increase; finally \( L \) cannot change when \( l_{in} \) is non-empty.

5.4.2 On termination...

On termination we have \( l_{out} = r_{out} = \emptyset \), and so with invariant (18) we need only

\[
l_{in} = \emptyset \quad \text{or} \quad r_{in} = \emptyset.
\]
Assuming for a contradiction that both \( l_{in} \) and \( r_{in} \) are non-empty, we then have from invariants (19) and (20) the inequalities

\[
L \geq \max r_{in} > R \geq \max l_{in} > L,
\]
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

\[
\begin{align*}
l_{in}, r_{in} &= \emptyset, \emptyset \\
L, R &= 4, 4,
\end{align*}
\]

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\(^2\), 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.

\(^2\)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

\[
\begin{align*}
\#[x: \text{lout}+\text{rout} \mid x \geq L] \\
+ \#[x: \text{lout}+\text{rout} \mid x \geq R] \\
+ 3 \times \#(\text{lin}+\text{rin})
\end{align*}
\]

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 \( \text{lin} \neq \emptyset \) then an element is removed from \( \text{lout} \) (\( V0 \) decreases by at most 2) and added to \( \text{lin} \) (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 \( \text{lout}+\text{rout} \) — and (before) \( l \not\in L \) but (after) \( l \geq L \).

The reasoning for \( \text{rout} \), 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

\[
\tilde{L} - \tilde{R} \in \{-2, 0, 2\}, \tag{21}
\]

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

\[
L := L + 2 \frac{1}{2} \oplus (L + 2)
\]

is executed only when \( L \leq R \), thus only when \( \tilde{L} \leq \tilde{R} \), and has the effect

\[
\tilde{L} := \tilde{L} + 2 .
\]

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

- \( \tilde{L} = \tilde{R} - 2 \lor \tilde{L} = \tilde{R} + 2 \) — write \( L \not\equiv R \) for those states.
- \( \tilde{L} = R \) (equivalently \( L = \tilde{R} \)) — write \( L \equiv R \).
- \( L = R \).

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

\[
\begin{align*}
(L \not\equiv R) & \rightarrow (L \not\equiv R) \cap (L = R) \frac{1}{2} \oplus (L \equiv R) \\
(L = R) & \rightarrow (L = R) \cap (L \not\equiv R)
\end{align*}
\]

To explain the absence of a transition leaving states (\( L \equiv R \)) we need yet another invariant:

\[
L \not\in \text{rout} \land R \not\in \text{lout} . \tag{22}
\]

It holds initially, and cannot be falsified by the command add \( L \) to \( \text{rout} \), because \( \tilde{L} \not\equiv L \). That leaves
the command \( L := L + 2 \frac{1}{2} (L + 2) \); but in that case from (19) we have

\[
\text{rout} \leq L < L + 2, \frac{(L + 2)}{(L + 2)} = (L + 2), (L + 2),
\]

so that in neither case does the command set \( L \) to the conjugate of a value already in \( \text{rout} \).

Thus with (22) we see that execution of the only alternatives that change \( L, R \) cannot occur if \( L \equiv R \), since for example selection of the guard \( l = L \) implies \( L \in \text{rout} \), impossible if \( L \equiv R \) and \( R \not\in \text{rout} \).

For the outer variant we therefore define \( V1 \) to be

\[
2, \quad \text{if } L = R \\
1, \quad \text{if } L \neq R \\
0, \quad \text{if } L \equiv R,
\]

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 Secs. 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 \( \text{per se} \) 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.

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


Notes for Contributors

The prime purpose of the journal is to publish original research papers in the fields of Computer Science and Information Systems, as well as shorter technical research notes. However, non-refereed review and exploratory articles of interest to the journal’s readers will be considered for publication under sections marked as Communications of Viewpoints. While English is the preferred language of the journal, papers in Afrikaans will also be accepted. Typed manuscripts for review should be submitted in triplicate to the editor.

Form of Manuscript

Manuscripts for review should be prepared according to the following guidelines:

- Use wide margins and 1 1/2 or double spacing.
- The first page should include:
  - the title (as brief as possible)
  - the author’s initials and surname
  - the author’s affiliation and address
  - an abstract of less than 200 words
  - an appropriate keyword list
  - a list of relevant Computing Review Categories
  - Tables and figures should be numbered and titled.
  - References should be listed at the end of the text in alphabetic order of the (first) author’s surname, and should be cited in the text according to the Harvard method.

Manuscripts accepted for publication should comply with guidelines as set out on the SACJ web page, http://www.cs.up.ac.za/sacj, which gives a number of examples.

SACJ is produced using the \LaTeX\ document preparation system, in particular \LaTeX\ 2e. Previous versions were produced using a style file for a much older version of \LaTeX, which is no longer supported. Please see the website for further information on how to produce manuscripts which have been accepted for publication.

Authors of accepted publications will be required to sign a copyright transfer form.

Charges

Charges per final page will be levied on papers accepted for publication. They will be scaled to reflect typesetting, reproduction and other costs. Currently, the minimum rate is R30.00 per final page for contributions which require no further attention. The maximum is R120.00, prices inclusive of VAT.

These charges may be waived upon request of the author and the discretion of the editor.

Proofs

Proofs of accepted papers may be sent to the author to ensure that typesetting is correct, and not for addition of new material or major amendments to the text. Corrected proofs should be returned to the production editor within three days.

Letters and Communications

Letters to the editor are welcomed. They should be signed, and should be limited to about 500 words. Announcements and communications of interest to the readership will be considered for publication in a separate section of the journal. Communications may also reflect minor research contributions. However, such communications will not be refereed and will not be deemed as fully-fledged publications for state subsidy purposes.

Book Reviews

Contributions in this regard will be welcomed. Views and opinions expressed in such reviews should, however, be regarded as those of the reviewer alone.

Advertisement

Placement of advertisements at R1000.00 per full page per issue and R500.00 per half page per issue will be considered. These charges exclude specialised production costs, which will be borne by the advertiser. Enquiries should be directed to the editor.