Probably Hoare? Hoare probably!

Carroll Morgan, Annabelle McIver
and JW Sanders

December 1999

1 Abstraction and probability

Rigorous reasoning about imperative programming [2, 3, 1] produced as a by-product an interest in the *abstraction* that links specification and implementation: one of its early appearances was as 'demonic nondeterminism' [1], unpredictable and potentially undesirable behaviour at runtime. That nondeterminism was not introduced to allow us to write tricky programs; rather it was a natural consequence of the new *design process*, and represented a controlled abstraction from implementation issues. Since then, such abstraction has assumed a central role in program development.

Rigorous reasoning about probabilistic programming was *prevented* from developing further just because it did not accommodate abstraction. The generalisation [5] of predicate-style reasoning to probabilistic programs — a beautiful idea — did not take abstraction with it: demonic choice was replaced, not augmented, by probabilistic choice. Without abstraction, the formal treatment of probabilistic programs was uncoupled from that of conventional programs and systems; and then it was left behind.

In this note we discuss that crucial interaction, of probability and abstraction, using as an example statements like

$$1/2 \;\vdash\; \{\mathit{True}\} \quad \text{`}\mathit{Flip}\ c\text{'} \quad \{c = \mathit{Heads}\}\ ,$$

meaning "with probability 1/2 the action 'Flip c' will establish the predicate $c = \mathit{Heads}$". More generally we consider 'probabilistic Hoare triples'

$$p \;\vdash\; \{\mathit{pre}\}\ \mathit{prog}\ \{\mathit{post}\}\ , \tag{1}$$

where for real $0 \leq p \leq 1$ the qualifier $p \vdash$ just means that the standard triple on the right holds with at least that probability. The *in*adequacy of statements like (1) — an inadequacy we will explain — is what we will use as a starting point.

We illustrate the issues with the iterated Monte-Carlo technique, of which primality testing is a well known example.

Nowadays it is taken for granted that interesting new conventional algorithms should always be presented with rigorous indications, if not proofs, of why they work and how they came to be designed that way. Probabilistic algorithms, however, continue to be presented operationally, with justification in natural language; and they can require considerable heroics to understand even partially.

We hope that our discussion, bringing demonic and probabilistic choice back together, will allow the presentation and justification of probabilistic algorithms to 'catch up' with that of their conventional counterparts. And we conclude that probabilistic formal reasoning can be taught alongside conventional Hoare triples or weakest preconditions in an undergraduate curriculum, although it is not quite as simple as might be hoped. Still, it is simple enough; and it is long overdue.

2 Probably Hoare

The 'probabilistic Hoare triple'

$$p \;\vdash\; \{x = C\}\ \mathit{prog}\ \{x > C\} \tag{2}$$

specifies, we say, that with probability at least p the program *prog* is guaranteed to increase the value of (integer) variable x.[1] One of many possible implementations is the probabilistic program

$$\mathit{prog} \quad — \quad x{:=}\ x+1 \quad {}_p\!\oplus \quad x{:=}\ x-1\ ,$$

where the operator ${}_p\!\oplus$ indicates that the left branch is to be taken with probability p, and the right with the complementary probability $1{-}p$. Program *prog* would also satisfy the specification

$$1{-}p \;\vdash\; \{x = C\}\ \mathit{prog}\ \{x < C\}\ , \tag{3}$$

because it can decrease x as well.

It is tempting to go on to build a nice calculus of such triples: it might have inference rules like

$$\frac{\begin{array}{l} p \;\vdash\; \{\mathit{pre}\}\ \mathit{prog}\ \{\mathit{mid}\} \\ q \;\vdash\; \{\mathit{mid}\}\ \mathit{prog}'\ \{\mathit{post}\} \end{array}}{p \times q \;\vdash\; \{\mathit{pre}\}\ \mathit{prog}; \mathit{prog}'\ \{\mathit{post}\}}\quad ,$$

though of course slightly more elaborate manoeuvres would be necessary if the second probability q were an expression containing variables altered by the first program *prog*.

But we can't go on, because there is something wrong with specifications like (2), and we will see that it is their treatment of abstraction. Abstraction arises in any development formalism able to express the twin notions of specification indifference and (its functional equivalent) concealment of implementation detail. It is therefore essential to any realistic refinement notation, where its manifestation is demonic nondeterminism. Let $\sqcap$ denote binary demonic choice, and consider the program

$$\mathit{demon} \quad — \quad (x{:=}\ x+1 \ {}_p\!\oplus\ x{:=}\ x-1) \ \sqcap\ (x{:=}\ x+1 \ {}_q\!\oplus\ x{:=}\ x-1)\ .$$

Its users cannot predict whether program *demon* will take the '${}_p\!\oplus$ branch' or the '${}_q\!\oplus$ branch': concealment of implementation detail. Equivalently, as specifiers of *demon* rather than its users, we are abstracting from whether we want the probability that x is incremented to be p or to be q: specification indifference.

The best Hoare triples we can write for *demon* in the style of (2) and (3) are

$$\begin{array}{rrl} & p \,\mathbf{min}\, q & \vdash \quad \{x = C\}\ \mathit{demon}\ \{x > C\} \\ \text{and} & (1{-}p) \,\mathbf{min}\, (1{-}q) & \vdash \quad \{x = C\}\ \mathit{demon}\ \{x < C\}\ , \end{array}$$

[1] We are using C as a 'logical constant'.

in which the **min**'s express our abstraction about how the demonic choice will be resolved.

We now see two things: first, that the probabilities we give are only lower bounds — for although the actual probability of establishing $x > C$ cannot be lower than $p \,\mathbf{min}\, q$, it could be as high as $p \,\mathbf{max}\, q$. Second, we note that the two probabilities $p \,\mathbf{min}\, q$ and $(1{-}p) \,\mathbf{min}\, (1{-}q)$ do not sum to 1 — in spite of the fact that the two postconditions $x > C$ and $x < C$ are exhaustive and disjoint for *demon*.

Those two things are *not* problems however: both phenomena are familiar from standard[2] demonic choice, where firstly we speak of *weakest* preconditions, and secondly we give up disjunctivity in the presence of demonic choice.

But now consider these two programs, which we call *fair* and *unfair*:

$$\begin{array}{rcl} \mathit{fair} & — & x{:=}\ A \quad \sqcap \quad (x{:=}\ B \,{}_{1/2}\!\oplus\, x{:=}\ C) \\ \mathit{unfair} & — & (x{:=}\ A \sqcap x{:=}\ B) \quad {}_{1/2}\!\oplus \quad (x{:=}\ A \sqcap x{:=}\ C)\ . \end{array}$$

The first one is 'fair' with respect to outcomes B and C: no matter how many B's appear in a series of runs, one would expect about the same number of C's since the choice is ${}_{1/2}\!\oplus$ between them. (There might be none of either, of course, if $\sqcap$ were resolved always to the left.) The second program however is potentially unfair: one possible resolution of its demonic choices is the refinement

$$x{:=}\ B \quad {}_{1/2}\!\oplus \quad x{:=}\ A\ .$$

In that program B results 'half the time' but C never.

Now the problem is that although *we* can distinguish *fair* and *unfair*, the probabilistic Hoare triples such as (1) cannot. Fig. 1 tabulates the two programs' probabilistic behaviour with respect to all possible postconditions.

We can distinguish those two 'probably-Hoare equivalent' programs by following each with the same third program *prog*′ [12] so that that *fair*; *prog*′ and *unfair*; *prog*′ are again distinquished by probabilistic Hoare triples. And that indicates that the 'probably-Hoare semantics' is not compositional or — which is the same thing — because some operator of the language (*e.g.* sequential composition) is not a congruence with respect to 'probably-Hoare equivalence'. Compositionality is a powerful criterion for judging the effectiveness of descriptions, whether simple or complex: for example *weight* is a useful (though limited) semantics for chemical reactions, because it is compositional. But *colour*, not being compositional, is of little use.

3 Hoare probably

We regain compositionality by having the probability not 'outside' the specification (probably Hoare) but 'inside' it (Hoare probably). That is, rather than acting over

[2] We say *standard* when we mean "in the absence of probability".

all postconditions	*fair* probability		*unfair* probability
false	0	$= 0 =$	0
$x = A$	$1 \mathbf{\ min\ } 0$	$= 0 =$	$(1/2)(1 \mathbf{\ min\ } 0) + (1/2)(1 \mathbf{\ min\ } 0)$
$x = B$	$0 \mathbf{\ min\ } 1/2$	$= 0 =$	$(1/2)(0 \mathbf{\ min\ } 1) + (1/2)(0 \mathbf{\ min\ } 0)$
$x = C$	$0 \mathbf{\ min\ } 1/2$	$= 0 =$	$(1/2)(0 \mathbf{\ min\ } 0) + (1/2)(0 \mathbf{\ min\ } 1)$
$x \neq A$	$0 \mathbf{\ min\ } 1$	$= 0 =$	$(1/2)(0 \mathbf{\ min\ } 1) + (1/2)(0 \mathbf{\ min\ } 1)$
$x \neq B$	$1 \mathbf{\ min\ } 1/2$	$= 1/2 =$	$(1/2)(1 \mathbf{\ min\ } 0) + (1/2)(1 \mathbf{\ min\ } 1)$
$x \neq C$	$1 \mathbf{\ min\ } 1/2$	$= 1/2 =$	$(1/2)(1 \mathbf{\ min\ } 1) + (1/2)(1 \mathbf{\ min\ } 0)$
true	1	$= 1 =$	1

Figure 1: Programs *fair* and *unfair* cannot be distinguished by probabilistic Hoare triples.

---

standard predicates (which are Boolean-valued state functions) the triples act over *random variables* (which are real-valued state functions).[3]

Let *preExp* and *postExp* now be real-valued expressions in the program variables, denoting random variables. (We reserve *pre* and *post* for Boolean expressions, denoting predicates.) In general we understand the triple

$$\{preExp\}\ prog\ \{postExp\} \tag{4}$$

to mean "the expression *preExp* evaluated in the initial state gives a lower bound for the *expected value* of expression *postExp* evaluated in the final state resulting from execution of *prog*".

The first nice thing about interpretation (4) is that it very neatly subsumes our earlier *probably Hoare* semantics. To see that, let a predicate *pre* be converted to a real-valued expression by writing $[pre]$ for its *characteristic function*, that has value 1 where *pre* holds and is 0 elsewhere. Then the triple

$$\{p \times [pre]\}\ prog\ \{[post]\}$$

means "from any initial state satisfying predicate *pre* the program *prog* will with probability at least p reach a final state satisfying predicate *post*". That is exactly the same meaning as our earlier $p \vdash \{pre\}\ prog\ \{post\}$.[4]

[3]The fundamental step of using random variables for sequential probabilistic-program logic was made by Kozen[5], but he did not (we believe) realise that when nondeterminism is present it is not only convenient but necessary to use them. Indeed for deterministic programs Kozen's approach is equivalent to the use of probabilistic Hoare triples — although one could argue that the former is neater.

[4]Indeed in states not satisfying *pre* the expression $p \times [pre]$ is 0, so that *prog* is required to establish something (or even to terminate) only with probability 'at least 0' — no constraint at all. But when *pre* holds initially, the expected value of *post* finally must be at least $p \times [True] = p$; and the expected value of a characteristic function $[post]$ is exactly the probability assigned to the underlying set *post*.

postExp	*fair preExp*	*unfair preExp*
$[x = A] + 2[x = B]$	1	1/2

Figure 2: Programs *fair* and *unfair* are distinguished by the probabilistic postcondition *postExp*.

The second nice thing is that the new semantics is strictly richer. Reconsider for example program *fair* but now with respect to random variable *postExp* written $[x = A] + 2[x = B]$. If the left branch of the demonic choice is taken, the expected value of *postExp* is $1 + 2 \times 0 = 1$; and if the right branch is taken it is $1/2 \times (0 + 2 \times 1) + 1/2 \times (0 + 2 \times 0) = 1$ again. Thus all together for *fair* we have $1 \,\mathbf{min}\, 1$, so that its pre-expectation is 1 as shown in Fig. 2.

But considering *unfair* with respect to the same post-expectation we have instead $1/2 \times (1 \,\mathbf{min}\, (2 \times 1)) + 1/2 \times (1 \,\mathbf{min}\, (2 \times 0)) = 1/2$, which is strictly less than the 1 calculated for *fair* above. Thus in the Kozen-style semantics, augmented with our **min**-mediated treatment of demonic choice, the programs *fair* and *unfair* are distinguished after all.

In fact we find that *unfair* delivers lower (or equal) pre-expectations in every case — so we can say that *unfair* "is refined by" *fair*. With respect to a weakest-precondition semantics [7] there is a refinement calculus exhibiting laws, as we shall see in Sec. 5, of just the type desired.

4 Interlude: structure and health

Beyond compositionality, another criterion for the suitablilty of semantics is mathematical structure. Ideally the descriptions should have some non-trivial properties that are preserved by the operations on them — for then the mathematics works 'behind the scenes', excluding automatically the phenomena one has chosen to ignore.

For standard Hoare triples such a property is the 'healthiness condition' of *conjunctivity*:

$$\begin{array}{rrcl} \text{from} & \{pre\} & prog & \{post\} \\ \text{and} & \{pre'\} & prog & \{post'\} \\ \hline \text{follows} & \{pre \wedge pre'\} & prog & \{post \wedge post'\}\ . \end{array} \tag{5}$$

Conjunctivity allows programs' properties to be established in small pieces, rather than all-at-once; and more generally it is used in the proof of algebraic laws.

For probabilistic triples we clearly cannot hope to use Boolean conjunction over real-valued expressions to manufacture an extension of standard healthiness

— although whatever we do find should have (5) as a special case. It turns out [9, p342] that the appropriate generalisation is this surprising *sublinearity* condition:

> For any reals $a, b, c \geq 0$ and expectations *preExp*, *preExp′*, *postExp* and *postExp′*,
>
> $$\begin{array}{rccc} \text{from} & \{preExp\} & prog & \{postExp\} \\ \text{and} & \{preExp'\} & prog & \{postExp'\} \end{array}$$
>
> follows
>
> $$\begin{array}{c} \{a \times preExp + b \times preExp' \ominus c\} \\ prog \\ \{a \times postExp + b \times postExp' \ominus c\} \ , \end{array}$$
>
> in which $\ominus$ is 'truncated subtraction'.[5]

In spite of its complex appearance, the condition of sublinearity is exactly what we want: for it can be shown [9, Lem. 7.3] all on its own to imply monotonicity, to specialise to various simpler forms of distribution (*e.g.* distribution of multiplication by a scalar), to give a form of the *Law of the Excluded Miracle* and finally, as we hoped above, to reproduce 'ordinary' conjunctivity when restricted to standard programs and predicates [9, 7].

Most importantly, sublinearity characterises 'real' programs *exactly* [9, Thm. 8.7], just as conjunctivity gives for standard Hoare triples an exact characterisation of the underlying relational model of standard programs in the space of predicate transformers.

5 An example: Monte-Carlo algorithms

We now show these expectations in action.

The *probabilistic primality testing* algorithm [10] establishes a number's primality, with arbitrarily high probability, by repeated failure to show that it is composite: it is an example of an iterated 'Monte-Carlo' algorithm in which the probability of error can be made arbitrarily small. We will keep primality in mind during the following discussion.

Generally speaking, suppose we want to decide some computationally expensive Boolean B (*e.g.* "a given number is prime"). A *Monte-Carlo* algorithm for that is a computationally cheap and guaranteed-to-terminate procedure which *probably* decides B — that is, it is 'fast' but it might be wrong and the probability of its being wrong has a known upper bound.

Such a Monte-Carlo procedure for B could be specified

$$b{:=}\ B \quad {}_{\geq p}\!\oplus \quad (b{:=}\ True \ \sqcap\ b{:=}\ False)\ , \tag{6}$$

[5] *Truncated subtraction* $\ominus$ is defined $x \ominus y = (x - y)\,\mathbf{max}\,0$, and has lowest syntactic precedence.

where b is some Boolean program variable, and in which ${}_{\geq p}\oplus$ indicates that the left branch is taken with probability *at least* p. Thus the above code will set b to B, as desired, with probability at least p, and the rest of the time it can set b arbitrarily: its probability of error is at most $1-p$.[6]

For probabilistic primality, an instantiation (*i.e.* refinement, with $p = 1/2$) of (6) is in fact

$$\begin{array}{l} \textbf{if } B \textbf{ then } b{:=}\ \mathit{True} \textbf{ else} \\ \quad b{:=}\ \mathit{False}\ {}_{\geq \frac{1}{2}}\oplus\ b{:=}\ \mathit{True} \\ \textbf{fi}\ . \end{array} \tag{7}$$

That is, there is a simple number-theoretic procedure that with probablity at least $1/2$ will show a given number to be composite, if in fact it is; but if the number is prime, the procedure will never show that it is composite. With the code (7) we encapsulate our use of number theory [10].

Now we have two aims in going further with this simple and well known example. The first is to discuss the significance of the (demonic) nondeterminism inherent in ${}_{\geq 1/2}\oplus$, and the second is to show that the formal treatment of an iterated algorithm based on (7) is not beyond a second-year undergraduate course on rigorous program development generally.

We consider nondeterminism first, appearing here as abstraction: 'at least $1/2$' abstracts from an exact (but unknown) probability. In general $\mathit{prog}\,{}_{\geq p}\oplus\,\mathit{prog}'$ means the nondeterministic choice of all programs $\mathit{prog}\,{}_{r}\oplus\,\mathit{prog}'$, for $r \geq p$. That turns out (by convexity, because $\sqcap$ can be refined into ${}_{q}\oplus$ for any q and because $\mathit{prog}\,{}_{q}\oplus\,(\mathit{prog}\,{}_{p}\oplus\,\mathit{prog}')$ is just $\mathit{prog}\,{}_{q+(1-q)p}\oplus\,\mathit{prog}'$) to equal

$$\mathit{prog} \quad \sqcap \quad (\mathit{prog}\,{}_{p}\oplus\,\mathit{prog}')\ .$$

Now in a rigorous treatment we *have* to use such abstractions because, in real applications, the operator ${}_{p}\oplus$ is just unimplementable: there is, for example, no coin that is *exactly* fair. Thus in general we must make do with something like ${}_{[p,q]}\oplus$ if we are to be realistic, meaning "with some probability that lies between p and q". It's a useful abbreviation for

$$(\mathit{prog}\,{}_{p}\oplus\,\mathit{prog}') \quad \sqcap \quad (\mathit{prog}\,{}_{q}\oplus\,\mathit{prog}')\ ,$$

and programs like $\mathit{prog}\,{}_{\geq p}\oplus\,\mathit{prog}'$ are similarly convenient.

Having recognised that we must accommodate abstraction we are brought immediately to a second and technical point. Most — if not all other — rigorous approaches handle demonic choice, when coupled with probability, by factoring

[6] A Las-Vegas procedure shares that property but may not terminate; however it must terminate with probability at least p in which case a correct result is obtained. Such a program, *prog*, is specified using *weakest liberal postconditions* [8, 6] with $\Rrightarrow$ to mean "is everywhere no more than"

$$\begin{array}{rcl} [\mathit{pre}] & \Rrightarrow & \mathit{wlp.prog.}[\mathit{post}] \\ p & \Rrightarrow & \mathit{wp.prog.}[\mathit{True}]\ . \end{array}$$

We leave to the reader the semantic description, for comparison, of "Monte Carlo" using *wp*.

out an 'adversary' beforehand. That's good in principle because it separates concerns: the demonic adversary is dealt with first; left behind, for a simpler analysis afterwards, is a program that is purely probabilistic.

In practice however, in spite of its conceptual simplicity, the use of adversaries imposes a high cost when employed for 'everyday' simple programs; it is a large hammer for a small nut in a program like ours above. That is why people ignore nondeterminism in such cases, although we do not; nor will we need to factor it out.

We now attempt to treast our example in the 'undergraduate style it deserves': we mimic the best undergraduate texts on the rigorous development of standard imperative programs.

Our post-expectation is $[b{=}B]$, that the program reveals in b the value of the unknown B. The 'best' pre-expectation would of course be $[\mathit{True}]$, meaning that the program works in all cases — but knowing what we do about the underlying theory, captured in (7), we will settle for less:

> we seek only a more modest program *Decide* that satisfies
>
> $$\{1 \lhd B \rhd 1 - 1/2^N\} \;\; \mathit{Decide} \;\; \{b = B\} \; .$$
>
> Naturally the program does not refer to B other than as at (7).[7]

The specification reads "if B is true then b is guaranteed to be set to true; if B is false, then b will be set to false with probability at least $1 - 1/2^N$". Thus the program can fail only by setting b to *True* when it should be *False*, and then only with probability no more than $1/2^N$.

To find an invariant our first step is to rewrite the postcondition in the style of the precondition: thus it becomes $[b] \lhd B \rhd [\neg b]$. Then in the spirit of 'replace a constant by a variable' we write it $[b] \lhd B \rhd 1 - [b]/2^0$ and suggest

$$\mathit{Inv} \;\; \widehat{=} \;\; [b] \lhd B \rhd 1 - \frac{[b]}{2^n}$$

as a possible invariant: in doing so we move towards a program of the form

$\{1 \lhd B \rhd 1 - 1/2^N\}$
$b, n := \mathit{True}, N;$
do $n \neq 0 \wedge b \rightarrow$ $\quad$ /∗ invariant *Inv* ∗/
  CheckOnce;
  $n := n - 1$
od
$\{b = B\}$,

as our next development step.

[7] We use Hoare's 'if-then-else' triangles [4] in the pre-expectation, meaning "1 if B else $1 - \frac{1}{2^N}$"; and in the post-expectation, where we should have written $\{[b = B]\}$, we dropped the brackets $[\cdot]$ to avoid clutter.

We can now carry out the usual checks, firstly that the invariant is established by the initialisation: for that we should have

$$\{1 \lhd B \rhd 1 - 1/2^N\}\ b, n := \ \mathit{True}, N\ \{\mathit{Inv}\}\ ,$$

which is checked via the substitution usual for assignment statements. That is, we want

$$1 \lhd B \rhd 1 - 1/2^N \quad \Rrightarrow \quad \mathit{Inv}[b, n := \ \mathit{True}, N]\ ,$$

where $\mathit{Inv}[b, n := \ \mathit{True}, N]$ indicates syntactic substitution and $\Rrightarrow$ means "is everywhere no more than".[8] That amounts to

$$1 \lhd B \rhd 1 - 1/2^N \quad \Rrightarrow \quad [\mathit{True}] \lhd B \rhd 1 - [\mathit{True}]/2^N\ .$$

Secondly we check that on termination we have established the postcondition: that requires showing that when $n = 0$ or $\neg b$ holds we have

$$\mathit{Inv} \quad \Rrightarrow \quad [b] \lhd B \rhd [\neg b]\ .$$

In case $n = 0$ we effectively are requiring

$$[b] \lhd B \rhd 1 - [b]/2^0 \quad \Rrightarrow \quad [b] \lhd B \rhd [\neg b]\ ,$$

and in case $\neg b$ we are requiring

$$[\mathit{False}] \lhd B \rhd 1 - [\mathit{False}]/2^n \quad \Rrightarrow \quad [\mathit{False}] \lhd B \rhd [\mathit{True}]\ .$$

Both are immediate.

That leaves the loop body where — after taking the decrement of n into account by calculating $\mathit{Inv}[n := \ n - 1]$ — we must verify

$$\{[b] \lhd B \rhd 1 - [b]/2^n\}\ \ \mathit{CheckOnce}\ \ \{[b] \lhd B \rhd 1 - [b]/2^{n-1}\}\ .$$

We may assume truth of the loop guard $n \neq 0 \wedge b$.

By inspection of the above we see that *CheckOnce* should behave like **skip** when B holds, since the left-hand alternatives $[b] \lhd B \cdots$ are identical in pre- and post-expectations; and since from the guard we know that b holds anyway, we find that the first part

$$\textbf{if}\ B\ \textbf{then}\ b := \ \mathit{True}\ \textbf{else} \cdots$$

of (7) will do.

When B does not hold, giving the other alternatives $\cdots B \rhd 1 - [b]/2^n$ (in the pre-expectation) and $\cdots B \rhd 1 - [b]/2^{n-1}$ (post-expectation), we could use

[8] Why don't we just write $\leq$ for "is everywhere no more than", as everyone else does? First we want the reasoning to 'look logical', and we have $[\mathit{False}] \Rrightarrow [\mathit{True}]$ this way; and second we would rather reserve $\leq$ for use within arithmetic expressions rather than between them. Writing $[a < b] \Rrightarrow [a \leq b]$ seems clearer than (the equally plausible) $[a < b] \leq [a \leq b]$.

$b{:=}\ \mathit{False}$, since that makes them both 1. That is indeed part of (7). But the other part is the probabilistic

$$b{:=}\ \mathit{False}\ {}_{1/2}\!\oplus\ b{:=}\ \mathit{True}\ .$$

For that we use this inference rule for probabilistic choice:

$$\frac{\begin{array}{ccc}\{\mathit{preExp}\} & \mathit{prog} & \{\mathit{postExp}\}\\ \{\mathit{preExp}'\} & \mathit{prog}' & \{\mathit{postExp}\}\end{array}}{\{p\times \mathit{preExp}+(1{-}p)\times \mathit{preExp}'\}\ \ \mathit{prog}\ {}_p\!\oplus\ \mathit{prog}'\ \ \{\mathit{postExp}\}}\quad ,$$

in this case instantiated to

$$\frac{\begin{array}{rcl}\{1\} & b{:=}\ \mathit{False} & \{1-[b]/2^{n-1}\}\\ \{1-1/2^{n-1}\} & b{:=}\ \mathit{True} & \{1-[b]/2^{n-1}\}\end{array}}{\begin{array}{c}\{(1/2)1+(1/2)(1-1/2^{n-1})\}\\ b{:=}\ \mathit{False}\ {}_{1/2}\!\oplus\ b{:=}\ \mathit{True}\\ \{1-[b]/2^{n-1}\}\ .\end{array}}$$

That leaves only the arithmetic in the pre-expectation, where we find

$$\begin{array}{ll} & (1/2)1+(1/2)(1-1/2^{n-1})\\ = & 1-1/2^n\\ = & \text{“note that } b \text{ holds”}\\ & 1-[b]/2^n\ ,\end{array}$$

which is just what we wanted: it is the right-hand side of the invariant again. Thus *CheckOnce* is implemented by (7), and we are done.

6 Conclusion

For at least as long as we have been at the Programming Research Group the authors have been aware of subscribing to a principle dear to Tony Hoare: seek *simple*, *realistic* formalisms. We believe that the work described here exemplifies that principle. As is often the case, that has resulted in a desire for algebraic reasoning in place of the all-too-often intricate reasoning offered by semantic models.

The key technical message of this presentation has been that it is possible to generalise Boolean- to real-valued ‘predicates’ (the key semantic idea), that abstraction (demonic choice) does not have to be left behind (with hindsight a crucial desideratum), that conjunctivity generalises to sublinearity (the principal technical result), and that there are many probabilistic algorithms ‘out there’ that could benefit from — or even be discovered by — such a treatment (an opportunity).

Indeed the Monte-Carlo example demonstrates that the complexity of the semantics, and even the soundness of laws, may be concealed entirely from the programmer. A consequence of bridling the tempting complexity behind probability and abstraction is the (frustrating?) observation that it all *could* have been done much earlier [5] in which case the popular textbooks on weakest-precondition or Hoare-triple style program developments would have for years now contained probabilistic and conventional algorithms together among their examples and case studies. But at least such material is now available to be enjoyed by all who share Tony's principle.

References

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

[2] R.W. Floyd. Assigning meanings to programs. In J.T. Schwartz, editor, *Mathematical Aspects of Computer Science*. American Mathematical Society, 1967.

[3] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.

[4] C.A.R. Hoare. A couple of novelties in the propositional calculus. *Zeitschr. für Math. Logik und Grundlagen der Math.*, 31(2):173–178, 1985.

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

[6] A.K. McIver and C.C. Morgan. Partial correctness for probabilistic programs. Submitted to Theoretical Computing Science; available at[11].

[7] Carroll Morgan. *pGCL*: Formal reasoning for random algorithms. *South African Computer Journal*, 22, March 1999. Also available at [11].

[8] 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. `http://www.springer.co.uk/ewic/workshops/7RW`.

[9] C.C. Morgan, A.K. McIver, and K. Seidel. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems*, 18(3):325–353, May 1996.

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

[11] PSG. Probabilistic Systems Group: Collected reports. `http://web.comlab.ox.ac.uk/oucl/research/areas/probs/bibliography.html`.

[12] K. Seidel, C.C. Morgan, and A.K. McIver. Probabilistic imperative programming: a rigorous approach. Revises [13]; available at [11].

[13] 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. Available at [11].