Correctness by Construction for Probabilistic Programs

Annabelle McIver[1](✉) and Carroll Morgan[2](✉)

[1] Macquarie University, Sydney, Australia
annabelle.mciver@mq.edu.au
[2] University of New South Wales and Trustworthy Systems, Data61, CSIRO,
Sydney, Australia
carroll.morgan@unsw.edu.au

Abstract. The "correct by construction" paradigm is an important component of modern Formal Methods, and here we use the probabilistic Guarded-Command Language *pGCL* to illustrate its application to *probabilistic* programming.

pGCL extends Dijkstra's guarded-command language *GCL* with probabilistic choice, and is equipped with a correctness-preserving refinement relation ($\sqsubseteq$) that enables compact, abstract specifications of probabilistic properties to be transformed gradually to concrete, executable code by applying mathematical insights in a systematic and layered way.

Characteristically for correctness by construction, as far as possible the reasoning in each refinement-step layer does not depend on earlier layers, and does not affect later ones.

We demonstrate the technique by deriving a fair-coin implementation of any given discrete probability distribution. In the special case of simulating a fair die, our correct-by-construction algorithm turns out to be "within spitting distance" of Knuth and Yao's optimal solution.

1 Testing Probabilistic Programs?

Edsger Dijkstra argued [1, p. 3] that the construction of *correct* programs requires mathematical proof, since "...program testing can be used very effectively to show the presence of bugs but never to show their absence." But for programs that are constructed to exhibit some form of randomisation, regular testing can't even establish that *presence*: surprising, unexpected program traces are bound to turn up even in *correctly* operating probabilistic systems.

Thus evidence of quantitative errors in probabilistic systems could require many, many traces to be subjected to detailed statistical analysis—yet even then debugging probabilistic programs is a challenge once that evidence has been assembled. Unlike standard (non-probabilistic) programs, where a single failed test can often pinpoint the source of the offending error in the code, it's not easy

We are grateful for the support of the Australian Research Council.

T. Margaria and B. Steffen (Eds.): ISoLA 2020, LNCS 12476, pp. 216–239, 2020.
https://doi.org/10.1007/978-3-030-61362-4_12

to figure out what to change in the implementation of probabilistic programs in order to move closer towards "correctness" rather than further away.

Without that unambiguous relationship between failed tests and the coding errors that cause them, Dijkstra's caution regarding proofs of programs is even more apposite. In this paper we describe such a proof method for probability: *correctness by construction.* In a sentence, to apply "*CbC*" one constructs the program and its proof at the same time, letting the requirement that there *be* a proof guide the design decisions taken while constructing the program.

Like standard programs, probabilistic programs incorporate mathematical insights into algorithms, and a correctness-by-construction method should allow a program developer to refer rigorously to those insights by applying development steps that preserve "probabilistic correctness". Probabilistic correctness is however notoriously unintuitive. For example, the solution of the infamous Monty Hall problem caused such a ruckus in the mathematical community that even Paul Erdös questioned the correct analysis [14].[1] Yet once coded up as a program [10, p. 22], the Monty Hall problem is only four lines long! More generally though, many widely relied-upon probabilistic programs in security are quite short, and yet still pose significant challenges for correctness.

We describe correctness by construction in the context of *pGCL*, a small programming language which restores demonic choice to Kozen's landmark (purely) probabilistic semantics [7,8] while using the syntax of Dijkstra's *GCL* [2]. Its basic principles are that correctness for programs can be described by a generalisation of Hoare logic that includes *quantitative* analysis; and it has a definition of refinement that allows programs to be developed in such a way that both functional and probabilistic properties are preserved.[2]

2 Enabling Correctness by Construction—*pGCL*

The setting for correctness by construction of probabilistic programs is provided by *pGCL* –the probabilistic Guarded-Command Language– which contains both abstraction and (stepwise) refinement [10]. We begin by reviewing its origins, then its treatment of probabilistic choice and demonic choice, and finally its realisation of *CbC*.

(This section can be skimmed on first reading: just collect *pGCL* syntax from Figs. 2, 3, and 4, and then skip directly to Sect. 3.)

As we will not be treating non-terminating programs, we can base our description here on quite simple models for sequential (non-reactive) programs. The

[1] A game-show host, Monty Hall, exhibits three curtains, behind one of which sits a Cadillac; the other two curtains conceal goats. The contestant guesses which curtain hides the prize, and Monty then opens another, making sure however that it reveals a goat. The contestant is allowed to change his mind. Should he?

[2] If the program is a mathematical object, then as Andrew Vazonyi [14] pointed out: "I'm not interested in *ad hoc* solutions invented by clever people. I want a method that works for lots of problems... One that mere mortals can use. Which is what a correctness-by-construction method should be.".

state space is some set S and, in its simplest terms, a program takes an initial state to a final state: it (its semantics) therefore has type $S \to S$.

The three subsections that follow describe logics based on successive enrichments of this, the simplest model, and even the youngest of those logics is by now almost 25 years old: thus we will be "reviewing" rather than inventing.

The first enrichment, Sect. 2.1, is based on the model $S \to \mathbb{P}S$ that allows demonic nondeterminism,[3] so facilitating abstraction; then in Sect. 2.2 the model $S \to \mathbb{D}S$ replaces demonic nondeterminism by probabilistic choice, losing abstraction (temporarily) but in its place gaining the ability to describe probabilistic outcomes; and finally in Sect. 2.3 the model $S \to \mathbb{P}\mathbb{D}S$ restores demonic nondeterminism, allowing programs that can abstract from precise probabilities. Using syntax we will make more precise in those sections, we give here some simple examples of the three increments in expressivity:

(1) $\mathtt{x{:}=\ H}$ Set variable $\mathtt{x}$ to $\mathtt{H}$ (as in any sequential language);
(2) $\mathtt{x}{:}\in \{\mathtt{H,T}\}$ Set $\mathtt{x}$'s value demonically from the set $\{\mathtt{H},\mathtt{T}\}$;
(3) $\mathtt{x}{:}\in \mathtt{H}\ {}_{2/3}{\oplus}\ \mathtt{T}$ Set $\mathtt{x}$'s value from the set $\{\mathtt{H},\mathtt{T}\}$ with probability $^2/_3$ for $\mathtt{H}$ and $^1/_3$ for $\mathtt{T}$, a "biased coin"; and
(4) $\mathtt{x}{:}\in \mathtt{H}\ {}_{1/3}{\oplus}_{1/3}\ \mathtt{T}$ Set $\mathtt{x}$ from the set $\{\mathtt{H},\mathtt{T}\}$ with probability *at least* $^1/_3$ each way, a "capricious coin".

The last example of those (4) is the most general: for (3) is $\mathtt{x}{:}\in \mathtt{H}\ {}_{2/3}{\oplus}_{1/3}\ \mathtt{T}$; and (2) is $\mathtt{x}{:}\in \mathtt{H}\ {}_{0}{\oplus}_{0}\ \mathtt{T}$; and finally (1) is $\mathtt{x}{:}\in \mathtt{H}\ {}_{1}{\oplus}_{0}\ \mathtt{T}$.

2.1 Floyd/Hoare/DijKstra: Pre- and Postconditions: (1, 2) Above

We assume a typical sequential programming language with variables, expressions over those variables, assignment (of expressions to variables), sequential composition (semicolon or line break), conditionals and loops. It is more or less Dijkstra's *guarded command language* [2], and is based on the model $S \to \mathbb{P}S$, where $\mathbb{P}S$ is the set of all subsets of S.

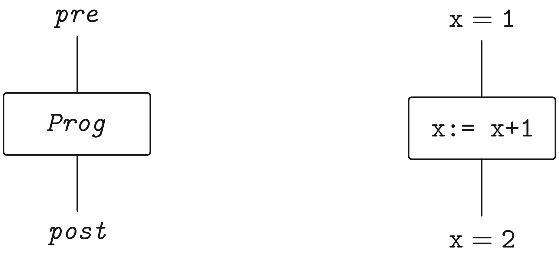
The *weakest precondition* of program Prog in such a language, with respect to a postcondition post given as a first-order formula over the program variables, is written $\mathsf{wp}(\mathit{Prog}, \mathit{post})$ and means

> the weakest formula (again on the program variables) that must hold *before* Prog executes in order to ensure that post holds *after* Prog executes [2].

In a typical compositional style, the wp of a whole program is determined by the wp of its components.

We group Dijkstra, Hoare and Floyd together because the Dijkstra-style implication $\mathit{pre} \Rightarrow \mathsf{wp}(\mathit{Prog}, \mathit{post})$ has the same meaning as the Hoare-style triple $\{\mathit{pre}\}\ \mathit{Prog}\ \{\mathit{post}\}$ which in turn has the same meaning as the original Floyd-style flowchart annotation, as shown in Fig. 1 [3,4]. All three mean "If pre holds of the state before execution of Prog, then post will hold afterwards."

[3] Constructor $\mathbb{P}$ is "subsets of" and $\mathbb{D}$ is "discrete distributions on".



At left is a "generic" Floyd annotation of a flowchart containing only one program element. If the annotation *pre* holds "on the way in" to the program *Prog*, then annotation *post* will hold on the way out. At right is an example with specific annotations and a specific program.

In the Hoare style the right-hand example would be written

$$\{\mathtt{x}=1\}\ \ \mathtt{x:=\ x+1}\ \ \{\mathtt{x}=2\}\quad .$$

In the Dijsktra style it would be written `x=1` ⇒ `wp(x:= x+1, x=2)`.
They all three have the same meaning.

Fig. 1. Floyd-style annotated flowchart

Finally, a notable –but incidental– feature of Dijkstra's approach was that (demonic) nondeterminism arose naturally, as an abstraction from possible concrete implementations.[4] That is why we use $S \to \mathbb{P}S$ rather than $S \to S$ here. In later work (by others) that abstraction was made more explicit by including explicit syntax for a binary "demonic choice" between program fragments, a composition *Left* ⊓ *Right* that could behave either as the program *Left* or as the program *Right*. But that operator (⊓) was not really an extension of Dijkstra's work, because his (more verbose) conditional

```
IF   True → Left    – If True holds, then this branch may be taken.
□    True → Right   – If True holds, then also this branch may be taken.
FI                  – (Dijkstra terminated all IF's with FI's.)
```

was there in his original guarded-command language, introducing demonic choice naturally as an artefact of the program-design process—and it expressed exactly the same thing. The (⊓) merely made it explicit.

2.2 Kozen: Probabilistic Program Logic: (3) Above

Kozen extended Dijkstra-style semantics to probabilistic programs, again over a sequential programming language but now based on the model $S \to \mathbb{D}S$, where

[4] See Sect. 3.5 for a further discussion of this.

$\mathbb{D}S$ is set of all discrete distributions in S.[5] He replaced Dijkstra's demonic nondeterminism ($\sqcap$) by a "probabilistic nondeterminism" operator (${}_p\oplus$) between programs, understood so that $\mathit{Left}\ {}_p\oplus\ \mathit{Right}$ means "Execute Left with probability p and Right with probability $1{-}p$." The probability p is (very) often $1/2$ so that `coin:= Heads` ${}_{1/2}\oplus$ `coin:= Tails` means "Flip a fair coin." But probability p can more generally be any real number, and more generally still it can even be an expression in the program variables.

Kozen's corresponding extension of Floyd/Hoare/Dijkstra [7,8] replaced Dijkstra's logical formulae with real-valued expressions (still over the program variables); we give examples below. The "original" Dijkstra-style formulae remain as a special case where real number 1 represents *True* and 0 represents *False*; and Dijkstra's definitions of wp simply carry through essentially as they are... except that an extra definition is necessary, for the new construct (${}_p\oplus$), where Kozen defines that

$$\begin{array}{ll} & \mathsf{wp}(\mathit{Left}\ {}_p\oplus\ \mathit{Right}, \mathit{post}) \\ \text{is} & p\cdot\mathsf{wp}(\mathit{Left}, \mathit{post})\ +\ (1{-}p)\cdot\mathsf{wp}(\mathit{Right}, \mathit{post}). \end{array}$$

With this single elegant extension, it turns out that in general $\mathsf{wp}(\mathit{Prog}, \mathit{post})$ is the *expected value*, given as a (real valued) expression over the *initial* state, of what post will be in the *final* state, i.e. after Prog has finished executing from that initial state. (The initial/final emphasis simply reminds us that it is the same as for Dijkstra: the weakest precondition is what must be true in the *initial* state for the postcondition to be true in the *final* state.) For example we have that

$$\mathsf{wp}(\mathtt{x:=\ 1-y}\ {}_{1/3}\oplus\ \mathtt{x:=\ 3*x},\quad \mathtt{x}+3) \qquad \text{is} \qquad 1/3(1{-}\mathtt{y}+3)\ +\ 2/3(3\mathtt{x}{+}3),$$

that is the real-valued expression $3\frac{1}{3} + 2\mathtt{x} - \mathtt{y}/3$ in which both x and y refer to their values in the initial state.

More impressive though is that if we introduce the convention that brackets $[-]$ convert Booleans to numbers, i.e. that $[\mathit{True}] = 1$ and $[\mathit{False}] = 0$, we have in general for *Boolean*-valued prop the convenient idiom[6]

$$\begin{array}{ll} & \mathsf{wp}(\mathit{Prog}, [\mathit{prop}]) \\ \text{is} & \text{"the probability that } \mathit{Prog} \text{ establishes property } \mathit{prop}\text{",} \end{array} \tag{1}$$

And if –further– it happens that the "probabilistic" program Prog actually contains no probabilistic choices at all, then (1) just above has value 1 just when

[5] Kozen's work did not restrict to discrete distributions; but that is all we need here.

[6] The expected value of the characteristic function $[\mathit{prop}]$ of an event prop is equal to the probability that prop itself holds.

Prog is guaranteed to establish *post*, and is 0 otherwise: it is in that sense that the Dijkstra-style semantics "carries through" into the Kozen extension. That is, if *Prog* contains no probabilistic choice, and *post* is a conventional (Boolean valued) formula, then we have that[7]

$$
\begin{array}{lll}
 & \textit{Dijkstra style} & [\,\mathsf{wp}(\textit{Prog}, \textit{post}\,)\,] \\
\text{is the same as} & \textit{Kozen style} & \mathsf{wp}(\textit{Prog}, [\,\textit{post}\,]).
\end{array}
$$

The full power of the Kozen approach, however, starts to appear in examples like this one below: we flip two fair coins and ask for the probability that they show the same face afterwards. Using the (Dijkstra) weakest-precondition rule that $\mathsf{wp}(\textit{Prog1}\,;\textit{Prog2},\ \textit{post}\,)$ is simply $\mathsf{wp}(\textit{Prog1},\ \mathsf{wp}(\textit{Prog2},\textit{post}\,))$,[8] we can calculate

$$
\begin{array}{ll}
 & \mathsf{wp}(\texttt{c1:= H}\ {}_{1/2}\oplus\ \texttt{c1:= T; c2:= H}\ {}_{1/2}\oplus\ \texttt{c2:= T},\quad [\texttt{c1} = \texttt{c2}]) \\
= & \mathsf{wp}(\texttt{c1:= H}\ {}_{1/2}\oplus\ \texttt{c1:= T},\quad \mathsf{wp}(\texttt{c2:= H}\ {}_{1/2}\oplus\ \texttt{c2:= T}, [\texttt{c1} = \texttt{c2}])) \\
= & \mathsf{wp}(\texttt{c1:= H}\ {}_{1/2}\oplus\ \texttt{c1:= T},\quad {}^{1}\!/_{2}[\texttt{c1} = \texttt{H}] + (1 - {}^{1}\!/_{2})[\texttt{c1} = \texttt{T}]) \\
= & {}^{1}\!/_{2}({}^{1}\!/_{2}[\texttt{H} = \texttt{H}] + {}^{1}\!/_{2}[H = \texttt{T}]) + {}^{1}\!/_{2}({}^{1}\!/_{2}[\texttt{T} = \texttt{H}] + {}^{1}\!/_{2}[\texttt{T} = \texttt{T}]) \\
= & {}^{1}\!/_{2}({}^{1}\!/_{2} \cdot 1 + {}^{1}\!/_{2} \cdot 0) + {}^{1}\!/_{2}({}^{1}\!/_{2} \cdot 0 + {}^{1}\!/_{2} \cdot 1) \\
= & {}^{1}\!/_{4} + {}^{1}\!/_{4} \\
= & {}^{1}\!/_{2}\ , \quad \text{that is that the probability that } \texttt{c1} = \texttt{c2} \text{ is } {}^{1}\!/_{2}.
\end{array}
$$

A nice further exercise for seeing this probabilistic wp at work is to repeat the above calculation when one of the coins uses $({}_{p}\oplus)$ but $({}_{1/2}\oplus)$ is retained for the other, confirming that the answer is still $1/2$.

2.3 McIver/Morgan: Pre- and Post-expectations

Following Kozen's probabilistic semantics at Sect. 2.2 just above (which itself turned out later to be a special case of Jones and Plotkin's probabilistic powerdomain construction [5]) we restored demonic choice to the programming language and called it *pGCL* [10,12]. It contains both demonic ($\sqcap$) and probabilistic $({}_{p}\oplus)$ choices; its model is $S \to \mathbb{PD}S$; and it is the language we will use for the correct-by-construction program development we begin in Sect. 3 below [10]. Figures 2, 3, and 4 summarise its syntax and its wp-logic.

To illustrate demonic- vs. probabilistic choice, we'll revisit the two-coin program from above. This time, one coin will have a probability-p bias for some constant $0 \le p \le 1$ (thus acting as a fair coin just when p is $1/2$). The other choice will be purely demonic.

We start with the (two-statement) program

$$
\begin{array}{l}
\texttt{c1:= H}\ {}_{p}\oplus\ \texttt{c1:= H} \\
\texttt{c2:= H}\ \sqcap\ \texttt{c2:= T}\qquad ,
\end{array}
$$

[7] Note that if *Prog* contains $({}_{p}\oplus)$ somewhere, the above does not apply: Dijkstra semantics has no definition for $({}_{p}\oplus)$.

[8] This is particularly compelling when wp is Curried: sequential composition $\mathsf{wp}(\textit{Prog1};\ \textit{Prog2})$ is then the functional composition $\mathsf{wp}(\textit{Prog1}) \circ \mathsf{wp}(\textit{Prog2})$.

name	syntax	semantics
expectation *post*	real-valued expression over the program variables	(the usual)
expression E	expression over the program variables (of any type)	(the usual)
condition C	Boolean-valued expression over the program variables	(the usual)
substitution	$E1\,[x\backslash E2]$	Replace all free occurrences of x in $E1$ by $E2$ (with the usual caveats.)
assignment	$x := E$	Evaluate E; assign it to x. $\mathsf{wp}(x := E,\ post) = post\,[x\backslash E]$
sequential composition	$Prog1\,;Prog2$	Execute $Prog1$ then $Prog2$. $\mathsf{wp}(Prog1\,;Prog2,\ post) = \mathsf{wp}(Prog1,\ \mathsf{wp}(Prog2, post))$
conditional	$\mathtt{IF}\ C\ \mathtt{THEN}\ Prog1\ \mathtt{ELSE}\ Prog2$	Evaluate Boolean C, then execute $Prog1$ or $Prog2$ accordingly. $\mathsf{wp}(\mathtt{IF}\ C\ \mathtt{THEN}\ Prog1\ \mathtt{ELSE}\ Prog2,\ post)$ $= [C]\cdot\mathsf{wp}(Prog1, post) + [\neg C]\cdot\mathsf{wp}(Prog2, post)$
loop	$\mathtt{WHILE}\ C\ \mathtt{DO}\ Prog$	Evaluate Boolean C, then execute $Prog$ (and repeat), or exit, accordingly. The usual least fixed point, based on $\mathtt{WHILE}\ C\ \mathtt{DO}\ Prog = \mathtt{IF}\ C\ \mathtt{THEN}\ (Prog\,;\ \mathtt{WHILE}\ C\ \mathtt{DO}\ Prog)$

The above cases cover the constructs of *pGCL* without probabilistic- or demonic choice, but nevertheless defined with Kozen-style "numeric" wp's which, applied to "post-expectations" give "pre-expectations".

Fig. 2. Syntax and wp-semantics for "restricted" *pGCL*

where the first statement is probabilistic and the second is demonic, and ask, as earlier, "What is the probability that the two coins end up equal?" We calculate

$$
\begin{array}{ll}
 & \mathsf{wp}(\mathtt{c1:=\ H}\ {}_p\!\oplus\ \mathtt{c1:=\ T;\ \ c2:=\ H} \sqcap \mathtt{c2:=\ T},\quad [\mathtt{c1} = \mathtt{c2}]) \\
= & \mathsf{wp}(\mathtt{c1:=\ H}\ {}_p\!\oplus\ \mathtt{c1:=\ T},\quad \mathsf{wp}(\mathtt{c2:=\ H} \sqcap \mathtt{c2:=\ T},\ [\mathtt{c1} = \mathtt{c2}])) \\
= & \mathsf{wp}(\mathtt{c1:=\ H}\ {}_p\!\oplus\ \mathtt{c1:=\ T},\quad [\mathtt{c1} = \mathtt{H}]\ \ \mathsf{min}\ \ [\mathtt{c1} = \mathtt{T}]) \\
= & p\cdot([\mathtt{H} = \mathtt{H}]\ \ \mathsf{min}\ \ [\mathtt{H} = \mathtt{T}]) + (1{-}p)\cdot([\mathtt{T} = \mathtt{H}]\ \ \mathsf{min}\ \ [\mathtt{T} = \mathtt{T}]) \\
= & p\cdot(1\ \ \mathsf{min}\ \ 0) + (1{-}p)\cdot(0\ \ \mathsf{min}\ \ 1) \\
= & p\cdot 0 + (1{-}p)\cdot 0 \\
= & 0 \quad ,
\end{array}
$$

name	syntax	semantics
probabilistic choice	$\mathit{Prog1}\ {}_p\oplus\ \mathit{Prog2}$	Evaluate p, which must be in $[0,1]$, then execute $\mathit{Prog1}$ with that probability; otherwise execute $\mathit{Prog2}$.

$$\mathsf{wp}(\mathit{Prog1}\,{}_p\oplus\ \mathit{Prog2},\ \mathit{post}) \;=\; p\cdot\mathsf{wp}(\mathit{Prog1},\mathit{post}) + (1\text{-}p)\cdot\mathsf{wp}(\mathit{Prog2},\mathit{post})$$

name	syntax	semantics
demonic choice	$\mathit{Prog1} \sqcap \mathit{Prog2}$	Choose demonically whether to execute $\mathit{Prog1}$ or $\mathit{Prog2}$.

$$\mathsf{wp}(\mathit{Prog1} \sqcap \mathit{Prog2},\ \mathit{post}) \;=\; \mathsf{wp}(\mathit{Prog1},\mathit{post})\ \mathsf{min}\ \mathsf{wp}(\mathit{Prog2},\mathit{post})$$

These "extra" cases cover the probabilistic- and demonic choice constructs of *pGCL*.

Fig. 3. Syntax and wp-semantics for *pGCL*'s choice constructs

to reach the conclusion that the probability of the two coins' being equal finally... is zero. And that highlights the way demonic choice is usually treated: it's a worst-case outcome. The "demon" –thought of as an agent– always tries to make the outcome as bad as possible: here because our desired outcome is that the coins be equal, the demon always sets the coin c2 so they will differ. If we were to repeat the above calculation with postcondition c1≠c2 instead, the result would *again* be zero: if we change our minds, want the coins to differ, then the demon will change his mind too, and act to make them the same.[9]

Implicit in the above treatment is that the c2 demon knows the outcome of the c1 flip—which is reasonable because that flip has already happened by the time it's the demon's turn.

Now we reverse the statements, so that the demon goes first: it must set c2 without knowing beforehand what c1 will be. The program becomes

$$\begin{array}{l} \texttt{c2:= H} \ \sqcap\ \texttt{c2:= T} \\ \texttt{c1:= H}\ {}_p\oplus\ \texttt{c1:= T} \quad , \end{array}$$

and we calculate

$$\begin{array}{ll}
 & \mathsf{wp}(\texttt{c2:= H} \sqcap \texttt{c2:= T;}\ \texttt{c1:= H}\ {}_p\oplus\ \texttt{c1:= T},\quad [\texttt{c1} = \texttt{c2}]) \\
= & \mathsf{wp}(\texttt{c2:= H} \sqcap \texttt{c2:= T},\quad \mathsf{wp}(\texttt{c1:= H}\ {}_p\oplus\ \texttt{c1:= T},\ [\texttt{c1} = \texttt{c2}])) \\
= & \mathsf{wp}(\texttt{c2:= H} \sqcap \texttt{c2:= T},\quad p\cdot[\texttt{H} = \texttt{c2}] + (1{-}p)\cdot[\texttt{T} = \texttt{c2}]) \\
= & p\cdot[\texttt{H} = \texttt{H}] + (1{-}p)\cdot[\texttt{T} = \texttt{H}] \quad \mathsf{min} \quad p\cdot[\texttt{H} = \texttt{T}] + (1{-}p)\cdot[\texttt{T} = \texttt{T}] \\
= & p\cdot 1 + (1{-}p)\cdot 0 \quad \mathsf{min} \quad p\cdot 0 + (1{-}p)\cdot 1 \\
= & p\ \ \mathsf{min}\ \ (1{-}p).
\end{array}$$

[9] This is not a novelty: demonic choice is usually treated that way in semantics—that's why it's called "demonic".

name	syntax	semantics
do nothing	`SKIP`	$\mathsf{wp}(\mathtt{SKIP}, \mathit{post}) = \mathit{post}$.
fail	`ABORT`	$\mathsf{wp}(\mathtt{ABORT}, \mathit{post}) = 0$.
probabilistic assignment	$x :\in E1\ {}_p\!\oplus E2$	As for $(x := E1)\ {}_p\!\oplus\ (x := E2)$.
demonic assignment	$x :\in E1 \sqcap E2$	As for $(x := E1) \sqcap (x := E2)$.

probabilistic conditional	IF p THEN *Prog1*
	ELSE *Prog2*
		As for *Prog1* ${}_p\!\oplus$ *Prog2* .

probabilistic loop	WHILE p DO *Prog*	As for ordinary loop, but using probabilistic conditional.

The cases above introduce special abbreviations and "syntactic sugar" for *pGCL*.

Command SKIP allows an "ELSE-less" conditional, as used e.g. in Fig. 2, to be defined in the usual way as IF C THEN *Prog1* ELSE SKIP.

Command ABORT allows $\mathsf{wp}(\mathtt{WHILE}\ C\ \mathtt{DO}\ \mathit{Prog},\ \mathit{post})$, as a least fixed point, to be defined as the supremum of

$$\begin{array}{l}
\mathsf{wp}(\mathtt{ABORT},\ \mathit{post}) \\
\mathsf{wp}(\mathtt{IF}\ C\ \mathtt{THEN}\ (\mathit{Prog};\mathtt{ABORT}),\ \mathit{post}) \\
\mathsf{wp}(\mathtt{IF}\ C\ \mathtt{THEN}\ (\mathit{Prog};(\mathtt{IF}\ C\ \mathtt{THEN}\ (\mathit{Prog};\mathtt{ABORT}))),\ \mathit{post}) \\
\vdots\ \ ,
\end{array}$$

which exists (in spite of the reals' being unbounded) because it can be shown by structural induction that

$$\mathsf{wp}(\mathit{Prog}, \mathit{post}) \quad \leq \quad \mathit{post} \quad ,$$

and that $\mathsf{wp}(\mathit{Prog}, -)$ is continuous, for all programs *Prog*. The above is therefore a chain, is dominated by *post* itself, and attains the limit at ω.

Fig. 4. Syntax and wp-semantics for *pGCL*'s choice constructs

Since the demon set flip c2 *without* knowing what the c1-flip would be (because it had not happened yet), the worst it can do is to choose c2 to be the value that it is known c1 is least likely to be—which is just the result above, the lesser of p and $1-p$. If –as before– we changed our minds and decided instead that we would like the coins to be different, then the demon would adapt by choosing c2 to be the value that c1 is *most* likely to be.

Either way, the probability our postcondition will be achieved, the pre-expectation of its characteristic function, is the same p min $(1-p)$—so that only when $p = 1/2$, i.e. when $p = (1-p)$, does the demon gain no advantage.

3 Probabilistic *Correctness by Construction* in Action[10]

Our first example problem conceptually will be to achieve a binary choice of arbitrary bias using only a fair coin. With the apparatus of Sect. 2.3 however, we can immediately move from conception to precision:

> We must write a *pGCL* program that implements $\mathit{Left}\ {}_{p}{\oplus}\ \mathit{Right}$, under the constraint that the only probabilistic choice operator we are allowed to use in the final (*pGCL*) program is $({}_{1/2}{\oplus})$.

This is not a hard problem mathematically: the probabilistic calculation that solves it is elementary. Our point here is to use this simple problem to show how such solutions can be calculated within a programming-language context, while maintaining rigour (possibly machine-checkable) at every step.

The final program is given at (8) in Sect. 3.5.

3.1 Step 1—A Simplification

We'll start by simplifying the problem slightly, instantiating the programs *Left* and *Right* to `x:= 1` and `x:= 0` respectively. Our goal is thus to implement

$$\mathtt{x}{:}{\in}\ 1\,{}_{p}{\oplus}\,0, \tag{2}$$

for arbitrary p, and our first step is to create two other distributions $1\,{}_{q}{\oplus}\,0$ and $1\,{}_{r}{\oplus}\,0$ whose average is $1\,{}_{p}{\oplus}\,0$—that is

$$1/2 \times (\,(1\,{}_{q}{\oplus}\,0) + (1\,{}_{r}{\oplus}\,0)\,) \quad = \quad (1\,{}_{p}{\oplus}\,0). \tag{3}$$

A fair coin will then decide whether to carry on with $1\ {}_{q}{\oplus}\ 0$ or with $1\ {}_{r}{\oplus}\ 0$.

Trivially (3) holds just when $(q{+}r)/2 = p$, and if we represent p, q, r as variables in our program, we can achieve (3) by the double assignment[11]

```
IF $p \leq 1/2$ → q,r:= 0,2p
▯  $p \geq 1/2$ → q,r:= 2p-1,1                                   (4)
FI
{ $p = (q+r)/2$ },
```

whose postcondition indicates what the assignment has established. If we follow that with a fair-coin flip between continuing with `q` or with `r`, viz.

```
IF $p \leq 1/2$ → q,r:= 0,2p                      – Here q is 0.
▯  $p \geq 1/2$ → q,r:= 2p-1,1                    – Here r is 1.      (5)
FI
(x:∈ 1 $_{q}{\oplus}$ 0) $_{1/2}{\oplus}$ (x:∈ 1 $_{r}{\oplus}$ 0)   – The fair coin ($_{1/2}{\oplus}$) here is permitted.
```

[10] This intent of this section can be understood based on the syntax given in Figs. 2, 3, and 4.

[11] We will sometimes include Dijkstra's closing `FI`.

then we should have implemented Program (2). But what have we gained?

The gain is that, whichever branch of the conditional is taken, there is a $1/2$ probability that the problem we have *yet* to solve will be either $({}_0\oplus)$ or $({}_1\oplus)$, both of which are trivial. If we were unlucky, well... then we just try again. But how do we show rigorously that Program (2) and Program (5) are equal?

If we look back at Program (4), we find the assertion $\{\, \mathtt{p} = (\mathtt{q}+\mathtt{r})/2 \,\}$ which is easy to establish by conventional Hoare-logic or Dijkstra-wp reasoning from the conditional just before it. (We removed it from Program (5) just to reduce clutter.) Rigour is achieved by calculating

$$\begin{array}{ll}
& \mathtt{wp}((\mathtt{x}{:}{\in}\ 1\ {}_{\mathtt{q}}{\oplus}\ 0)\ {}_{1/2}{\oplus}\ (\mathtt{x}{:}{\in}\ 1\ {}_{\mathtt{r}}{\oplus}\ 0),\quad \mathit{post}) \\
= & 1/2\ \mathtt{wp}((\mathtt{x}{:}{\in}\ 1\ {}_{\mathtt{q}}{\oplus}\ 0), \mathit{post}) \ +\ 1/2\ \mathtt{wp}((\mathtt{x}{:}{\in}\ 1\ {}_{\mathtt{r}}{\oplus}\ 0), \mathit{post}) \\
= & \mathtt{q}/2 \cdot \mathit{post}\,[\mathrm{x}\backslash 1] + {}^{(1-\mathtt{q})}/2 \cdot \mathit{post}\,[\mathrm{x}\backslash 0] + \mathtt{r}/2 \cdot \mathit{post}\,[\mathrm{x}\backslash 1] + {}^{(1-\mathtt{r})}/2 \cdot \mathit{post}\,[\mathrm{x}\backslash 0] \\
= & (\mathtt{q}{+}\mathtt{r})/2 \cdot \mathit{post}\,[\mathrm{x}\backslash 1] \ +\ (1 - (\mathtt{q}{+}\mathtt{r})/2) \cdot \mathit{post}\,[\mathrm{x}\backslash 0] \\
= & \mathtt{p} \cdot \mathit{post}\,[\mathrm{x}\backslash 1] \ +\ (1{-}\mathtt{p}) \cdot \mathit{post}\,[\mathrm{x}\backslash 0] \qquad\qquad \text{“}\{\, \mathtt{p} = (\mathtt{q}+\mathtt{r})/2 \,\}\text{”} \\
= & \mathtt{wp}(\mathtt{x}{:}{\in}\ 1\ {}_{\mathtt{p}}{\oplus}\ 0,\quad \mathit{post}),
\end{array}$$

for arbitrary postcondition post where at the end we used $\{\, \mathtt{p} = (\mathtt{q}+\mathtt{r})/2 \,\}$. Thus (2) = (5) because for any post their pre-expectations agree.

3.2 Step 2—Intuition Suggests a Loop

We now return to the remark "... then we just try again." If we replace the final fair-coin flip $(\mathtt{x}{:}{\in}\ 1\ {}_{\mathtt{q}}{\oplus}\ 0)\ {}_{1/2}{\oplus}\ (\mathtt{x}{:}{\in}\ 1\ {}_{\mathtt{r}}{\oplus}\ 0)$ by $\mathtt{p}{:}{\in}\ \mathtt{q}\ {}_{1/2}{\oplus}\ \mathtt{r}$ then –intuitively– we are in a position to "try again" with $\mathtt{x}{:}{\in}\ 1\ {}_{p}{\oplus}\ 0$. Although it is the same as the statement we started with, we have made progress because variable p has been updated—and with probability $1/2$ it is either 0 or 1 and we are done. If it is not, then we arrange for a second execution of

```
IF p ≤ 1/2 → q,r:= 0,2p
▯  p ≥ 1/2 → q,r:= 2p-1,1
FI                                          (6)
p:∈ q 1/2⊕ r
```

and, if *still* p is neither 0 nor 1, then ... we need a loop.

3.3 Step 3—Introduce a Loop

We have already shown that

$$\mathit{Program}\,(2) \quad = \quad \mathit{Program}\,(6);\ \mathit{Program}\,(2).$$

A general equality for sequential programs (including probabilistic) tells us that in that case also we have[12]

$$\mathit{Program}\,(2) \quad = \quad \mathtt{WHILE}\ \mathit{C}\ \mathtt{DO}\ \mathit{Program}\,(6)\ \mathtt{OD};\ \mathit{Program}\,(2)$$

[12] As before, we usually use Dijkstra's loop-closing OD.

for any loop condition C, provided the loop terminates. Intuitively that is clear because, if Program (2) can annihilate Program (6) once from the right, then it can do so any number of times. A rigorous argument appeals to the fixed-point definition of `WHILE`, which is where termination is used. (If C were `False`, so that the loop did not terminate, the *rhs* would be `Abort`, thus providing a clear counter-example.)

For probabilistic loops, the usual "certain" termination is replaced with *almost-sure* termination, abbreviated *AST*, which means that the loop terminates with probability one: put the other way, that would be that the probability of iterating forever is zero. For example the program

```
c:= H; WHILE c=H DO c:∈ H $_{1/2}\oplus$ T OD.
```

terminates almost surely because the probability of flipping `T` forever is zero.

A reasonably good *AST* rule for probabilistic loops is that the variant is (as usual) a natural number, but must be bounded above; and instead of having to decrease on every iteration, it is sufficient to have a non-zero probability of doing so [10,13].[13] The variant for our example loop just above is `[c=H]`, which has probability 1/2 of decreasing from `[H=H]`, that is 1, to `[T=H]` on each iteration.

The loop condition C for our program will be $0 < \mathtt{p} < 1$ and the variant comes directly from there: it is `[0<p<1]`, which has probability of 1/2 of decreasing from 1 to 0 on each iteration: and when it is 0, that is $0 < \mathtt{p} < 1$ is false, the loop must exit. With that, we have established that our original Program (2) equals the looping program

```
WHILE $0 < p < 1$ DO
  IF $p \leq 1/2 \rightarrow$ q,r:= 0,2p
  ▯  $p \geq 1/2 \rightarrow$ q,r:= 2p-1,1
  FI
  p:∈ q $_{1/2}\oplus$ r
OD
{ $p = 1 \lor p = 0$ }
x:∈ 1 $_{p}\oplus$ 0,
```

where the assertion at the loop's end is the negation of the loop guard.

3.4 Step 4—Use the Loop's Postcondition

There is still the final `x:∈ 1` $_{\mathtt{p}}\oplus$ `0` to be dealt with, at the end; but the assertion $\{\, \mathtt{p} = 1 \lor \mathtt{p} = 0 \,\}$ just before it means that it executes only when `p` is zero or one. So it can be replaced by `IF p=0 THEN x:∈` $1_1\oplus 0$ `ELSE x:∈` $1_0\oplus 0$, i.e. with

[13] By "reasonably good" we mean that it deals with most loops, but not all: it is sound, but not complete. There are more complex rules for dealing with more complex situations [11]. Strictly speaking, over infinite state spaces "non-zero" must be strengthened to "bounded away from zero" [13].

just `x:= p` . Mathematically, that would be checked by showing for all post-expectations *post* that

$$\mathsf{p} = 1 \lor \mathsf{p} = 0 \quad\Rightarrow\quad \mathsf{wp}(\mathtt{x{:}\in}\ 1\,{}_{\mathsf{p}}{\oplus}\,0, \mathit{post}\,) = \mathsf{wp}(\mathtt{x{:=}}\ \mathsf{p}, \mathit{post}\,).$$

But it's a simple-enough step just to believe (unless you were using mechanical assistance, in which case it *would* be checked).

And so now the program is complete: we have implemented $\mathtt{x{:}\in}\ 1_p{\oplus}0$ by a step-by-step correctness-by-construction process that delivers the program

```
WHILE $0 < p < 1$ DO
  IF $p \le 1/2$ → q,r:= 0,2p
  □  $p \ge 1/2$ → q,r:= 2p-1,1
  FI                                                (7)
  p:∈ $q\ {}_{1/2}{\oplus}\ r$
OD
x:= p
```

in which only fair choices appear. And each step is provably correct.

3.5 Step 5—After-the-Fact Optimisation

There is still one more thing that can (provably) be done with this program, and it's typical of this process: only when the pieces are finally brought together do you notice a further opportunity. It makes little difference—but it is irresistible.

Before carrying it out, however, we should be reminded of the way in which these five steps are isolated from each other, how all the layers are independent. This is an essential part of *CbC*, that the reasoning can be carried out in small, localised areas, and that it does not matter –for correctness– where the reasoning's target came from; nor does it matter where it is going.

Thus even if we had absolutely no idea what Program (7) was supposed to be doing, still we would be able to see that if we are replacing x by p at the end, we could just as easily replace it at the beginning; and then we can remove the variable p altogether. That gives

```
– Now $p$ is again a parameter, as it was in the original specification.
x:= $p$
WHILE $0 < x < 1$ DO
  IF $x \le 1/2$ → q,r:= 0,2x            – When $x = 1/2$, these two
  □  $x \ge 1/2$ → q,r:= 2x-1,1      – branches have the same effect.      (8)
  FI
  x:∈ $q\ {}_{1/2}{\oplus}\ r$
OD,
– The above implements x:∈ $1_p{\oplus}0$ for any $0 \le p \le 1$.
```

and we are done. When p is 0 or 1, it takes no flips at all; when p is $1/2$, it takes exactly one flip; and for all other values the expected number of flips is 2.

We notice that Program (8) appears to contain demonic choice, in that when $\mathtt{x} = 1/2$ the conditional could take either branch. The nondeterminism is real—even though the *effect* is the same in either case, that `q,r:= 0,1` occurs. But genuinely different computations are carried out to get there: in the first branch $2(1/2) - 1$ is evaluated to 0; and in the second branch $2(1/2)$ is evaluated to 1.

This is not an accident: we recall from Sect. 2.1 that for Dijkstra such nondeterminism arises naturally as part of the program-construction process. Where did it come from in this case?

The specification from which this conditional $\mathtt{IF} \cdots \mathtt{FI}$ arose was set out much earlier, at (3) which given p has many possible solutions in q, r. One of them for example is $q = r = p$ which however would have later given a loop whose non-termination would prevent Step 3 at Sect. 3.3. With an eye on loop termination, therefore, we took a design decision that at least one of q, r should be "extreme", that is 0 or 1. To end up with $q = 0$, what is the largest that p could be without sending r out of range, that is strictly more than 1? It's $p = 1/2$, and so the first IF-condition is $p \leq 1/2$. The other condition $p \geq 1/2$ arises similarly, and it absolutely does not matter that they overlap: the program will be correct whichever IF-branch taken in that case.

And, in the end –in (8) just above– we see that indeed that is so.

4 Implementing *any* Discrete Choice with a Fair Coin

Suppose instead of trying to implement a biased coin (as we have been doing so far), we want to implement a general (discrete) probabilistic choice of x's value from its type, say a finite set $\mathcal{X}$, but still using only a fair coin in the implementation. An example would be choosing x uniformly from $\{x_0, x_1, x_2\}$, i.e. a three-way fair choice. But what we develop below will work for any discrete distribution on a finite set $\mathcal{X}$ of values: it does not have to be uniform.

The combination of probability *and* abstraction allows a development like the one in Sect. 3 just above to be replayed, but a greater level of generality. We begin with a variable d of type $\mathbb{D}\mathcal{X}$,[14] where we recall that $\mathcal{X}$ is the type of x; and our specification is $\mathtt{x}{:}\!\in \mathtt{d}$, that is "Set x according to distribution d."

4.1 Replaying Earlier Steps from Sect. 3

Our first step –Step 1– is to declare two more $\mathbb{D}\mathcal{X}$-typed variables d0 and d1, and –as in Sect. 3.1– specify that they must be chosen so that their average is the original distribution d; for that we use the *pGCL* nondeterministic-choice construct "assign such that" (with syntax borrowed from Dafny [9]), from Fig. 5, to write

$$\mathtt{d0,d1}{:}|\ \mathtt{d} = (\mathtt{d0}+\mathtt{d1})/2 \quad \text{– Choose d0,d1 so that their average is d.} \tag{9}$$

[14] Recall from Sect. 2.2 that $\mathbb{D}\mathcal{X}$ is the set of discrete distributions over finite set $\mathcal{X}$.

name	syntax	semantics
choose from set	$\boldsymbol{x} :\in \textit{\textbf{set}}$	
	$\mathsf{wp}(\boldsymbol{x} :\in \textit{\textbf{set}}\,,\ \textit{\textbf{post}}\,) = (\mathsf{min}\, e \,\vert\, e \in \textit{\textbf{set}}\ .\ \textit{\textbf{post}}\,[\boldsymbol{x} \backslash e])$	
assign "such that"	$\boldsymbol{x} :\vert\ \textit{\textbf{property}}\,(\boldsymbol{x})$	
	$\mathsf{wp}(\boldsymbol{x} :\vert\ \textit{\textbf{property}}\,(\boldsymbol{x}),\ \textit{\textbf{post}}\,) = (\mathsf{min}\, e \,\vert\, \textit{\textbf{property}}\,(e)\ .\ \textit{\textbf{post}}\,[\boldsymbol{x} \backslash e])$	

The above generalise to more than a single variable, and are consistent with the earlier definitions: thus

$$
\begin{array}{ll}
 & \mathtt{x{:=}\ a} \sqcap \mathtt{x{:=}\ b} \\
= & \mathtt{x}{:\in}\ \{\mathtt{a,b}\} \\
= & \mathtt{x}{:\vert}\ \mathtt{x} \in \{\mathtt{a,b}\} \quad .
\end{array}
$$

By analogy with "choose from set" (but not itself an abstraction) we have also

name	syntax	semantics
choose from distribution	$\boldsymbol{x} :\in \textit{\textbf{dist}}$	
	$\mathsf{wp}(\boldsymbol{x} :\in \textit{\textbf{dist}}\,,\ \textit{\textbf{post}}\,) = (\sum e \,\vert\, e \in \lceil \textit{\textbf{dist}}\, \rceil\ .\ \textit{\textbf{dist}}\,(e) \cdot \textit{\textbf{post}}\,[\boldsymbol{x} \backslash e]) \quad ,$	

where $\textit{\textbf{dist}}\,(e)$ is the probability that $\textit{\textbf{dist}}$ assigns to e and $\lceil \textit{\textbf{dist}}\, \rceil$ is the *support* of $\textit{\textbf{dist}}$, the set of elements to which it assigns non-zero probability.[15]

It is just the expected value of $\textit{\textbf{post}}$, considered as a function of $\boldsymbol{x}$, over the distribution $\textit{\textbf{dist}}$ on $\boldsymbol{x}$. (Since $\mathtt{E1}\,{}_p{\oplus}\,\mathtt{E2}$ is a distribution, the definition above agrees with the earlier meaning of $\mathtt{x}{:\in}\ \mathtt{E1}\,{}_p{\oplus}\,\mathtt{E2}$ that we gave in Fig. 4 as an abbreviation.)

Fig. 5. Abstraction in *pGCL*.

The analogy with our earlier development is that there the distribution $\mathtt{d}$ was specifically $1\,{}_p{\oplus}\,0$, and we assigned

$$
\begin{array}{lll}
\text{if } \mathtt{p} \leq 1/2 & \mathtt{d0, d1} = (1\ {}_{\mathtt{0}}{\oplus}\ 0), & (1\ {}_{\mathtt{2p}}{\oplus}\ 0) \\
\text{if } \mathtt{p} \geq 1/2 & \mathtt{d0, d1} = (1\ {}_{\mathtt{2p-1}}{\oplus}\ 0), & (1\ {}_{\mathtt{1}}{\oplus}\ 0),
\end{array}
$$

which is a refinement ($\sqsubseteq$) of (9).

Our second step is to re-establish the $\mathtt{x}{:\in}\ \mathtt{d}$ -annihilating property that

$$\mathtt{Program\,(9);\ \ d{:\in}\ d0}\,{}_{1/2}{\oplus}\,\mathtt{d1;\ \ x{:\in}\ d} \quad = \quad \mathtt{x{:\in}\ d}, \tag{10}$$

[15] Summing over all possible values e of $\boldsymbol{x}$ would give the same result, since the extra values have probability zero anyway. Some find this formulation more intuitive.

which is proved using wp-calculations against a general post-expectation *post*, just as before: instead of the assertion $\{\, \mathtt{p} = (\mathtt{q}+\mathtt{r})/2 \,\}$ used at the end of Step 1, we use the assertion $\{\, \mathtt{d} = (\mathtt{d0}{+}\mathtt{d1})/2 \,\}$ established by the assign-such-that.

The third step is again to introduce a loop. But we recall from Step 3 earlier that the loop must be almost-surely terminating and, to show that, we need a variant function. Here we have no q,r that might be set to 0 or 1; we have instead d0,d1. Our variant will be that the "size" of one of these distributions must decrease strictly, where we define the *size* of a discrete distribution to be the number of elements to which it assigns non-zero probability.[16] But our specification $\mathtt{d0,d1}{:}|\ \mathtt{d} = (\mathtt{d0}{+}\mathtt{d1})/2$ above does not require that decrease; and so we must backtrack in our *CbC* and make sure that it does.

And we have made an important point, that developments following *CbC* rarely proceed as they are finally presented: the dead-ends are cut off, and only the successful path is left for the audit trail. It highlights the multiple uses of *CbC*—that on the one hand, used for teaching, the dead-ends are shown in order to learn how to avoid them; used in production, the successful path remains so that it can be modified in the case that requirements change.[17]

Thus to establish *AST* of the loop –that it terminates with probability one– we strengthen the split of d achieved by $\mathtt{d0,d1}{:}|\ (\mathtt{d0}{+}\mathtt{d1})/2 = \mathtt{d}$ with the decreasing-variant requirement, that either $|\mathtt{d0}| < |\mathtt{d}|$ or $|\mathtt{d1}| < |\mathtt{d}|$, where we are writing $|{-}|$ for "size of". Then the variant $|\mathtt{d}|$ is guaranteed strictly to decrease with probabililty $1/2$ on each iteration. That is we now write

$$\mathtt{d0,d1}{:}|\quad (\mathtt{d0}{+}\mathtt{d1})/2 = \mathtt{d} \ \wedge\ (|\mathtt{d0}| < |\mathtt{d}| \vee |\mathtt{d1}| < |\mathtt{d}|), \tag{11}$$

replacing (9), for the nondeterministic choice of d0 and d1. We do not have to reprove its annihilation property, because the new statement (11) is a refinement of the (9) from before (It has a stronger postcondition.) and so preserves all its functional properties. In fact that is the definition of refinement.

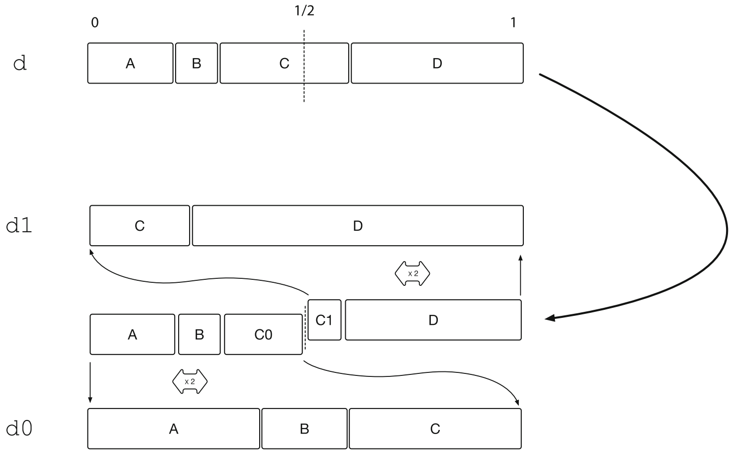
Our next step is to reduce the nondeterminism in (11) somewhat, choosing a particular way of achieving it: to "split" d into two parts d0,d1 such that the size of at least one part is smaller, we choose two subsets X_0, X_1 of $\mathcal{X}$ whose intersection contains at most one element. That is illustrated in Fig. 6, where $X_0 = \{\mathsf{A},\mathsf{B},\mathsf{C}\}$ and $X_1 = \{\mathsf{C},\mathsf{D}\}$. Further, we require that the probabilities $\mathtt{d}(X_0)$ and $\mathtt{d}(X_1)$ assigned by d to $X_0{-}X_1$ and $X_1{-}X_0$ are both no more than $1/2$.[18] Those constraints mean that we can always arrange the subsets so that the "$1/2$-line" of Fig. 6 either goes strictly through $X_0 \cap X_1$ (if they overlap) or runs between them (if they do not).[19]

[16] In probability theory this would be the cardinality of its support.

[17] And if an error was made in the *CbC* proofs, the "successful" path can be audited to see what the mistake was, why it was made, and how to fix it.

[18] Applying d to a set means the sum of the d-probabilities of the elements of the set.

[19] If for example C were much smaller, so that the dividing line went through D, the new distribution d0 would have support 4, the same as d itself. But $|\mathtt{d1}|$ would then have support 1, strictly smaller.



Suppose that $\mathcal{X}$ is $\{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\}$, and that the distribution $\mathtt{d}$ in $\mathcal{X}$ that we start with is indicated by the size of the rectangles: the size $|\mathtt{d}|$ of $\mathtt{d}$ here is therefore 4, because it contains 4 rectangles. We choose X_0 to be $\{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$ and X_1 to be $\{\mathsf{C}, \mathsf{D}\}$, so that $X_0{-}X_1$ is $\{\mathsf{A}, \mathsf{B}\}$ and $X_1{-}X_0$ is $\{\mathsf{D}\}$, and both $\mathtt{d}(X_0{-}X_1)$ and $\mathtt{d}(X_1{-}X_0)$ are no more than $1/2$. Their overlap is $\{\mathsf{C}\}$, whose probability the "$1/2$-line" splits into two pieces: one piece joins $\mathtt{d0}$ and the other piece joins $\mathtt{d1}$.

Thus by dividing the overall rectangle (representing $\mathcal{X}$ itself) exactly in the middle, at least one side must contain strictly fewer than $|\mathtt{d}|$ rectangles — and if we double the size of each small rectangle, we get our two distributions $\mathtt{d1}$ and $\mathtt{d2}$ such that $\mathtt{d} = (\mathtt{d0} + \mathtt{d1})/2$ and either $|\mathtt{d0}| < |\mathtt{d}|$ or $|\mathtt{d1}| < |\mathtt{d}|$.

Fig. 6. Dividing a discrete distribution into two pieces.

We then construct $\mathtt{d0}$ by restricting $\mathtt{d}$ to just X_0, then doubling all the probabilities in that restriction; if they sum to more than 1, we then trim any excess from the one element in $X_0 \cap X_1$ that X_0 shares with X_1. The analogous procedure is applied to generate $\mathtt{d1}$. In Fig. 6 for example we chose sizes 0.2, 0.1, 0.3 and 0.4 for the four regions, and the $1/2$ line went through the third one. On the left, the 0.2 and 0.1 and 0.3 are doubled to 0.4 and 0.2 and 0.6, summing to 1.2; thus 0.2 is trimmed from the 0.6, leaving 0.4 assigned to C. And the analogous procedure applies on the right.

4.2 "Decomposition of Data into Data Structures"

The quote is from Wirth [15]. Our program is currently

```
WHILE $|\mathtt{d}| \neq 1$ DO
    d0,d1:| $(\mathtt{d0}+\mathtt{d1})/2 = \mathtt{d} \;\wedge\; (|\mathtt{d0}| < |\mathtt{d}| \vee |\mathtt{d1}| < |\mathtt{d}|)$
    d:$\in$ d0 $_{1/2}\oplus$ d1                                                (12)
OD
x:$\in$ d // This is a trivial choice, because $|\mathtt{d}|=1$ here.
```

And it is correct: it does refine `x:∈ d`—but it is rather abstract. Our next development step will be to make it concrete by realising the distribution-typed variables and the subsets of $\mathcal{X}$ as "ordinary" datatypes using scalars and lists. In *CbC* this is done by deciding, before that translation process begins, what the realisations will be—and only then is the abstract program transformed, piece by piece. The relation between the abstract- and concrete types is called a *coupling invariant.*

Although an obvious approach is to order the type $\mathcal{X}$, say as $x_1, x_2, \ldots, x_N$ and then to realise discrete distributions as lists of length N of probabilities (summing to 1), a more concise representation is suggested by the fact that for example we represent a *two*-point distribution $x_1\ {}_p\!\oplus\ x_2$ as just *one* number p, with the $1{-}p$ implied. Thus we will represent the distribution $p_1, p_2, \ldots\ p_N$ as the list of length $N{-}1$ of "accumulated" probabilities: in this case for p we would have a list

$$p_1, \quad p_1{+}p_2, \quad \ldots, \quad \sum_{n=1}^{N-1} p_n,$$

leaving off the N^{th} element of the list since it would always be 1 anyway. Subsets of $\mathcal{X}$ will be pairs `low`,`high` of indices, meaning $\{x_{\texttt{low}}, \ldots, x_{\texttt{high}}\}$, and although that can't represent *all* subsets of $\mathcal{X}$, contiguous subsets are all we will need. Carrying out that transformation gives following concrete version of our abstract Program (12) below, where the abstract `d` is represented as the concrete `dL[low:high]` , which is the coupling invariant.[20]

And in Program (13) of Fig. 7 we have, finally, a concrete program that can actually be run. Notice that it has exactly the same *structure* as Program (12): split (the realisations of) `d` into `d0` and `d1`; overwrite `d` with one of them; exit the loop when `|d|` is one.

Neverthess, as earlier in Sect. 3.5, further development steps might still be possible now that everything is together in one place:[21] and indeed, recognising that only one of `dL0`,`dL1` will be *used*, we can rearrange Program (13)'s body so that only one of them will be *calculated*—and it can be updated as we go. That gives our really-final-this-time program (14) in Fig. 8, which will -without further intervention– use a fair coin to choose a value x_n according to *any* given discrete distribution d on finite $\mathcal{X}$. Its expected number of coin flips is no worse than $2N{-}2$, where N is the size of $\mathcal{X}$, thus agreeing with expected 2 flips for the program (8) in Sect. 3.5 that dealt with the simpler case $d = (1\ {}_p\!\oplus\ 0)$ where $\mathcal{X}$ was $\{1, 0\}$.

It's again worth emphasising –because it is the main point– that the correctness arguments for all of these steps are isolated from each other: in *CbC* every step's correctness is determined by looking at that step alone. Thus for example nothing in the translation process just above involved reasoning about the earlier steps, whether Program (12) actually implemented the `x:∈ d` that

[20] The range `low:high` is inclusive-exclusive (as in Python). A similar coupling invariant applies to `d0` and `d1`. All three invariants are applied at once.

[21] Note the necessity of keeping this as two steps: first data-refine, then (if you can) optimise algorithmically.

```
– Discrete distribution d in $\mathcal{X}$ of size $N$ is realised here as dL (for "d-list").
low,high:= 1,$N$                                – Initial support is all of $\mathcal{X}$.
WHILE low$\neq$high DO              – low$=$high means support is $\{x_{\mathtt{low}}\}$
  – Current support is $\{x_{\mathtt{low}},\ldots,x_{\mathtt{high}}\}$.

  – Find $X_0$ by examining the probabilities of $x_1,x_2,\ldots$
  n:= low                              – Determine dL0 as in lhs of Fig. 6.
  WHILE  n<high$\wedge$dL[n]<1/2  DO dL0[n]:= 2*dL[n]; n:= n+1 OD
  low0,high0:= low,n                  – Subset $X_0$ is $\{x_{\mathtt{low0}},\ldots,x_{\mathtt{high0}}\}$.
                                                                          (13)
  – Find $X_1$ by examining the probabilities of $x_N,x_{N-1},\ldots$
  n:= high-1                           – Determine dL1 as in rhs of Fig. 6.
  WHILE  low$\leq$n$\wedge$1/2<dL[n]  DO dL1[n]:= 2*dL[n]-1; n:= n-1 OD
  low1,high1:= n+1,high               – Subset $X_1$ is $\{x_{\mathtt{low1}},\ldots,x_{\mathtt{high1}}\}$.

  – Use fair coin to choose between dL0 and dL1.
  (dL,low,high):$\in$ (dL0,low0,high0) $_{1/2}\oplus$ (dL1,low1,high1)

OD
x:= $x_{\mathtt{low}}$           – Extract sole element of point distribution's support.
```

Fig. 7. Implement any discrete choice using only a fair coin.

```
– Assume discrete distribution $d$ over $\mathcal{X}=\{x_1,\ldots,x_N\}$ of size $N$
– has been represented cumulatively in list dL, as described above.

low,high:= 1,$N$                                – Initial support is all of $\mathcal{X}$.
WHILE low$\neq$high DO              – low$=$high means support is $\{x_{\mathtt{low}}\}$
  – Fair coin flipped here. (Recall Fig. 4.)
  IF 1/2 THEN                        – Then update dL as in lhs of Fig. 6.
    n:= low
    WHILE  n<high$\wedge$dL[n]<1/2 DO dL[n]:= 2*dL[n]; n:= n+1 OD          (14)
    high:= n
  ELSE                                 – Else update dL as in rhs of Fig. 6.
    n:= high-1
    WHILE  low$\leq$n$\wedge$1/2<dL[n]  DO dL[n]:= 2*dL[n]-1; n:= n-1 OD
    low:= n+1
  FI
OD
x:= $x_{\mathtt{low}}$       – Extract sole element of point distribution dL's support.
```

Fig. 8. Optimisation of Program (13)

we started with: we didn't care, and we didn't check. We just translated Program (12) into Program (13) regardless. And the subsequent rearrangement of (13) into Program (14) similarly made no use of Program (13)'s provenence.

All that is to be contrasted with the more common approach in which *only* intuition (and experience, and skill) is used, that is in which our final Program (14) might be written all at once at this concrete level, only then checking (testing, debugging, hoping) afterwards that our intuitions were correct. A transliteration of Program (14) into Python is given in Appendix A.

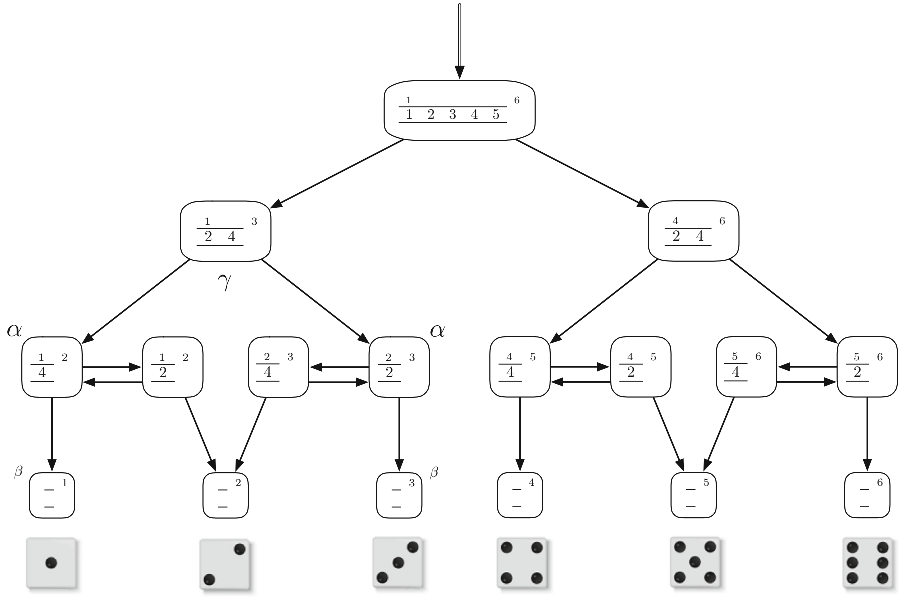
5 An Everyday Application: Simulating a Fair Die Using only a Fair Coin

Program (14) of the previous section works for any discrete distribution, without having to adapt the program in any way. However if the distribution's probabilities are not too bizarre, then the number of different values for `low` and `d` and `high` might be quite small—and then the program's behaviour for that distribution in particular can be set out as a small probabilistic state machine.

In Fig. 9 we take `d` to be the uniform distribution over the possible die-roll outcomes $\{1, 2, 3, 4, 5, 6\}$, and show the state machine that results. For that state machine in particular, we propose one last correctness-preserving step: it takes us to the optimal die-roll algorithm of Knuth and Yao [6].

6 Why Was This "Correctness by Construction"?

The programs here are not themselves remarkable in any way. (The optimality of the Knuth/Yao algorithm is not our contribution.) Even the mathematical insights used in their construction are well known, examples of elementary probability theory. *CbC* means however applying those insights in a systematic, layered way so that the reasoning in each layer does not depend on earlier layers, and does not affect later ones. The steps were specifically



Each interior node has two possible successors chosen with equal probability, and each final-die node is reached with the same probability 1/6. There are 17 nodes, and the expected number of coin flips is 4.

The nodes' origins are shown by labelling them with `low`, `d` and `high` from the states in the generating program that gave rise to them, representing the current probability distribution `d` yet to be realised over over the remaining subset $\{\texttt{low}, \ldots, \texttt{high}\}$ of possible results. With probabilities normalised out of 6 for neatness, a typical label is

$$\overset{\texttt{low} \qquad\qquad \texttt{high}}{\underline{\overline{\longleftarrow 6{\times}\texttt{d} \longrightarrow}}} \quad ,$$

where we recall that `d` gives the *sum* of the probabilities for $x_{\texttt{low}}, x_{\texttt{low}+1}, \ldots, x_{\texttt{high}-1}$ and that `d` for $x_{\texttt{high}}$ is left out, because it is always 1. Thus for example $\texttt{low} = 2$ and $\texttt{high} = 3$ and $\texttt{d} = [4]$ represents the distribution over support $\{2, 3\}$ of 4/6 for 2 and $1{-}4/6$ for 3, that is $2\ {}_{2/3}{\oplus}\ 3$.

The well-known (optimal) algorithm of Knuth and Yao for simulating a die with a fair coin has 13 states and 11/3 expected coin flips [6] — and it can be obtained from here by one last correctness-preserving step. Eliminate the choice γ, so that the two α and the two β nodes are merged; since that also merges the two die-rolls 1 and 3, restore the γ choice as a new fair choice γ' over $\{1, 3\}$, just below the merged β's. (The nodes leading to die-roll 2 are merged as well, but it makes no difference.)

Concentrating on the left (justified by symmetry), we see that the original γ choice must be done every time; but its replacement γ' is done only 2/3 of the time. That realises exactly the 1/3 efficiency advantage that Knuth/Yao optimal algorithm has over the one synthesised here by our general Program (14).

Fig. 9. Simulating a fair die with a fair coin

1. Start with the *specification* `x:∈ d` at the beginning of Sect. 4.
2. Prove a one-step annihilation property (10) for that specification.
3. Use a general loop rule to prove loop-annihilation Program (12), after Strengthening Program (9) to Program (11) to establish *AST*.
4. Propose strategy Fig. 6 for the loop body of Program (12).
5. Propose data representation of finite discrete distributions as lists, in Sect. 4.2, realising the strategy of Fig. 6 in the code of Program (13).
6. Rearrange Program (13) to produce a more efficient final program Program (14).
7. Note that **correctness by construction guarantees** that Program (14) refines `x:∈ d` for any `d`.
8. Apply Program (14) to the fair die, to produce state chart of Fig. 9.
9. Modify Fig. 9 to produce the Knuth/Yao (optimal) algorithm [6].
10. Note that **correctness by construction guarantees** that the Knuth/Yao (optimal) algorithm implements a fair die.

CbC also means that since all those steps are done explicitly and separately, they can be checked easily as you go along, and audited afterwards. But to apply *CbC* effectively, and *honestly*, one must have a rigorous semantics that justifies every single development step made. In our example here, that was supplied here by the semantics of *pGCL* [10]. But working in any "wide spectrum" language, right from the (abstract) start all the way to the (concrete) finish, means that many of those rigorous steps can be checked by theorem provers.

A Program (14) implemented in Python

```
#   Run 1,000,000 trials on a fair-die simulation.
#
#   bash-3.2$ python ISoLA.py
#   1000000
#   1 1 1 1 1 1
#   Relative frequencies
#         0.998154 1.00092  0.996474 0.998664 1.004928  1.00086
#   realised, using 4.001938 flips on average.


import sys
from random import randrange

# Number of runs, an integer on the first line by itself.
runs = int(sys.stdin.readline())

# Discrete distribution unnormalised, as many subsequent integers as needed.
# Then EOT.
d= []
for line in sys.stdin.readlines():
    for word in line.split(): d.append(int(word))
sizeX= len(d) # Size of initial distribution's support.
```

```
# Construct distribution's representation as accumulated list dL_Init.
# Note that length of dL_Init is sizeX-1,
#    because final (normalised) entry of 1 is implied.
# Do not normalise, however: makes the arithmetic clearer.
sum,dL_Init= d[0],[]
for n in range(sizeX-1): dL_Init= dL_Init+[sum]; sum= sum+d[n+1]

tallies= []
for n in range(sizeX): tallies= tallies+[0]

allFlips= 0 # For counting average number of flips.
for r in range(runs):
    flips= 0

    ### Program (14) starts here.
    low,high,dL= 0,sizeX-1,dL_Init[:] # Must clone dL_Init.
    # print "Start:", low, dL[low:high], high

    while low<high:
        flip= randrange(2) # One fair-coin flip.
        flips= flips+1

        if flip==0:
            n= low
            while n<high and 2*dL[n]<sum: dL[n]= 2*dL[n]; n= n+1
            high= n # Implied dL0[high]=1 performs trimming automatically.
            # print "Took dL0:", low, dL[low:high], high # dL0 has overwritten dL.

        else: # flip==1
            n= high-1
            while low<=n and 2*dL[n]>sum: dL[n]= 2*dL[n]-sum; n= n-1
            low= n+1 # Implied dL1[low]=0 performs trimming automatically.
            # print "Took dL1", low, dL[low:high], high # dL1 has overwritten dL.

    # print "Rolled", low, "in", flips, "flips."
    ### Program (14) ends here.

    tallies[low]= tallies[low]+1
    allFlips= allFlips+flips

print "Relative frequencies"
for n in range(sizeX): print "     ", float(tallies[n])/runs * sum
print "realised, using", float(allFlips)/runs, "flips on average."
```

References

1. Dijkstra, E.W.: On the reliability of programs (EWD303)
2. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Upper Saddle River (1976)
3. Floyd, R.W.: Assigning meanings to programs. In: Schwartz, J.T. (ed.) Mathematical Aspects of Computer Science. Proceedings of Symposium on Applied Mathematics, vol. 19, pp. 19–32. American Mathematical Society (1967)
4. Hoare, C.A.R.: An axiomatic basis for computer programming. Comm. ACM **12**(10), 576–580 (1969)
5. Jones, C.B., Plotkin, G.: A probabilistic powerdomain of evaluations. In: Proceedings of the IEEE 4th Annual Symposium on Logic in Computer Science, Los Alamitos, CA, pp. 186–195. Computer Society Press (1989)
6. Knuth, D., Yao, A.: The complexity of nonuniform random number generation. In: Algorithms and Complexity: New Directions and Recent Results. Academic Press (1976)
7. Kozen, D.: Semantics of probabilistic programs. J. Comput. Syst. Sci. **22**, 328–350 (1981)
8. Kozen, D.: A probabilistic PDL. In: Proceedings of the 15th ACM Symposium on Theory of Computing, pp. 291–297. ACM, New York (1983)
9. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
10. McIver, A.K., Morgan, C.C.: Abstraction, Refinement and Proof for Probabilistic Systems. Monographs in Computer Science. Springer, New York (2005). https://doi.org/10.1007/b138392
11. McIver, A.K., Morgan, C.C., Kaminski, B.-L., Katoen, J.-P.: A new proof rule for almost-sure termination. Proc. ACM Program. Lang. **2**(POPL), 1–28 (2017)
12. Morgan, C.C., McIver, A.K., Seidel, K.: Probabilistic predicate transformers. ACM Trans. Program. Lang. Syst. **18**(3), 325–353 (1996)
13. Morgan, C.C.: Proof rules for probabilistic loops. In: Jifeng, H., Cooke, J., Wallis, P. (eds.) Proceedings of the BCS-FACS 7th Refinement Workshop, Workshops in Computing. Springer, Heidelberg (July 1996). http://www.bcs.org/upload/pdf/ewicrw96paper10.pdf
14. Vazsonyi, A.: Which Door has the Cadillac: Adventures of a Real-Life Mathematician. Writers Club Press (2002)
15. Wirth, N.: Program development by stepwise refinement. Commun. ACM **14**(4), 221–227 (1971)